<https://leetcode.com/problems/coin-change-ii/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=55=2+2+15=2+1+1+15=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

**Attempt 1: 2025-03-09**

**Solution 1: Native DFS (10 min)**

**Style 1: With result.size(), MLE 14/29**

class Solution {

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

        List<List<Integer>> result = new ArrayList<List<Integer>>();

        helper(result, new ArrayList<Integer>(), amount, coins, 0);

        return result.size();

    }

    private void helper(List<List<Integer>> result, List<Integer> list, int amount, int[] coins, int index) {

        if(amount == 0) {

            result.add(new ArrayList<Integer>(list));

            return;

        }

        if(amount < 0) {

            return;

        }

        for(int i = index; i < coins.length; i++) {

            if(amount >= coins[i]) {

                list.add(coins[i]);

                helper(result, list, amount - coins[i], coins, i);

                list.remove(list.size() - 1);

            }

        }

    }

}

Time Complexity: Exponential (O(2^n)), as it explores all possible combinations.

Space Complexity: O(n) for recursion stack.

**Style 2: Without result.size(), TLE 14/29**

class Solution {

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

        return helper(amount, coins, 0);

    }

    private int helper(int amount, int[] coins, int index) {

        if(amount == 0) {

            return 1;

        }

        if(amount < 0) {

            return 0;

        }

        int result = 0;

        for(int i = index; i < coins.length; i++) {

            if(amount >= coins[i]) {

                result += helper(amount - coins[i], coins, i);

            }

        }

        return result;

    }

}

Time Complexity: Exponential (O(2^n)), as it explores all possible combinations.

Space Complexity: O(n) for recursion stack.

**Solution 2: DFS + Memoization (10 min)**

**Refer to**

<https://leetcode.com/problems/coin-change-2/discuss/162839/Another-approach-%3A-memorized-recursion-Java>

class Solution {

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

        // Test out by amount = 0, coins = []

        if(amount == 0) {

            return 1;

        }

        // Test out by amount = 7, coins = []

        if(coins.length == 0) {

            return 0;

        }

        Integer[][] memo = new Integer[coins.length][1 + amount];

        return helper(amount, coins, 0, memo);

    }

    private int helper(int amount, int[] coins, int index, Integer[][] memo) {

        if(amount == 0) {

            return 1;

        }

if(amount < 0) {

return 0;

}

        if(memo[index][amount] != null) {

            return memo[index][amount];

        }

        int result = 0;

        for(int i = index; i < coins.length; i++) {

            if(amount >= coins[i]) {

                result += helper(amount - coins[i], coins, i, memo);

            }

        }

        memo[index][amount] = result;

        return result;

    }

}

Time Complexity:

Space Complexity:

**Refer to Deepseek**

The provided memoization solution for LeetCode 518 (Coin Change II) is a top-down dynamic programming (DP) approach. Let’s analyze its time complexity and space complexity:

**Time Complexity:**

Number of Subproblems:

The memo array has dimensions [coins.length][1 + amount].

Each subproblem corresponds to a unique pair (index, amount), where:

index ranges from 0 to coins.length - 1.

amount ranges from 0 to amount.

Total subproblems = n × m, where:

n = number of coins (coins.length).

m = target amount.

Work per Subproblem:

For each subproblem (index, amount), the for loop iterates over all coins starting from index.

In the worst case, this loop runs n times (when index = 0).

Overall Time Complexity:

Since there are n × m subproblems, and each subproblem takes O(n) time, the total time complexity is:

**Space Complexity:**

Memoization Table:

The memo array has dimensions [n][m + 1], so it requires:

Recursion Stack:

The recursion depth is at most m (when the amount decreases by 1 in each call).

Each recursive call uses constant space, so the recursion stack requires:

Overall Space Complexity:

The total space complexity is:

**Optimization Opportunity:**

The current solution has a time complexity of (O(n^2 \* m)), which is less efficient than the bottom-up DP solution ((O(n \* m))). This is because the for loop inside the recursive function iterates over all coins starting from index, leading to redundant work.

To optimize:

Use bottom-up DP (as shown in the previous solution) to avoid the nested loop and achieve (O(n \* m)) time complexity.

**Conclusion:**

Time Complexity: (O(n^2 \* m))

Space Complexity: (O(n \* m))

This solution is correct but less efficient than the bottom-up DP approach. For large inputs, consider switching to the bottom-up DP solution for better performance.

**Solution 3: 2D DP (30 min)**

**Style 1: int[][] dp = new int[coins.length][1 + amount];**

**Refer to**

<https://leetcode.com/problems/coin-change-2/discuss/176706/Beginner-Mistake%3A-Why-an-inner-loop-for-coins-doensn>'t-work-Java-Soln

class Solution {

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

        // Test out by amount = 0, coins = []

        if(amount == 0) {

            return 1;

        }

        // Test out by amount = 7, coins = []

        if(coins.length == 0) {

            return 0;

        }

// dp[i][j] represents the number of ways to make amount j

// using the first i+1 coins (i.e., coins from index 0 to i)

        int[][] dp = new int[coins.length][1 + amount];

        for(int i = 0; i < coins.length; i++) {

            dp[i][0] = 1;

        }

        // Initialize the 1st row with tricky, since infinite coins supplied,

        // not require 'i == coins[0]', just 'i % coins[0] == 0' is fine

        for(int i = 0; i <= amount; i++) {

            dp[0][i] = (i % coins[0] == 0 ? 1 : 0);

        }

        for(int i = 1; i < coins.length; i++) {

            for(int j = 1; j <= amount; j++) {

// The transition is either not using the ith coin (taking the

// value from dp[i-1][j]) or using it (adding dp[i][j - coins[i]]

// since we can reuse the coin)

                dp[i][j] = dp[i - 1][j];

                if(j >= coins[i]) {

                    dp[i][j] += dp[i][j - coins[i]];

                }

            }

        }

        return dp[coins.length - 1][amount];

    }

}

Time Complexity: O(n×m), where n = number of coins, m = amount.

Space Complexity: O(n×m) for the DP table.

**Refer to Deepseek**

Okay, let's try to understand this 2D DP solution for the Coin Change II problem. The user wants to know the meaning of dp[i][j] here. Let me start by looking at the code step by step.

First, the code initializes a 2D array called dp with dimensions coins.length by (amount + 1). Then, it sets the first column (dp[i][0]) to 1 for all i. That makes sense because there's one way to make amount 0, which is by using no coins.

Next, the first row is initialized. For each amount j in the first row (i=0), dp[0][j] is 1 if j is a multiple of coins[0], otherwise 0. So, for the first coin, you can only form amounts that are exact multiples of that coin, and since you can use it multiple times, each multiple is a valid way.

Then, the nested loops start. The outer loop iterates over the coins starting from index 1, and the inner loop goes through each amount from 1 to the target amount. For each dp[i][j], the code sets it to dp[i-1][j], which means not using the current coin. Then, if the current amount j is at least the current coin's value, it adds dp[i][j - coins[i]], which represents using the current coin again.

So, putting this together, dp[i][j] seems to represent the number of ways to form amount j using the first i coins (from index 0 to i). The transition is either not using the ith coin (taking the value from dp[i-1][j]) or using it (adding dp[i][j - coins[i]] since we can reuse the coin).

The final answer is dp[coins.length - 1][amount], which would be the number of ways to form the target amount using all available coins.

Wait, the key here is that by building the table row by row, each row i incorporates the previous coins (0 to i-1) and the current coin i. So, each step allows using the ith coin multiple times, hence the combination is order-agnostic. That makes sense because the problem counts combinations, not permutations.

So, the definition of dp[i][j] here is the number of combinations to make amount j using the first (i+1) coins (since array indices start at 0). The initialization of the first row ensures that using the first coin, you can form any multiple of it. The transition ensures that each subsequent coin adds new ways by including it if possible.

Yes, this matches the standard 2D DP approach for this problem. The outer loop over coins and inner loop over amounts ensures that each combination is counted once without considering order, which aligns with the problem's requirements.

The 2D DP solution you provided correctly solves LeetCode 518 (Coin Change II). Let’s break down the meaning of dp[i][j] and how the code works:

**Definition of dp[i][j]:**

**dp[i][j] represents the number of ways to make amount j using the first i+1 coins (i.e., coins from index 0 to i).**

**Code Breakdown:**

Initialization:

dp[i][0] = 1 for all i: There’s 1 way to make amount 0 (using no coins).

dp[0][j] = 1 if j % coins[0] == 0, else 0: For the first coin, only multiples of coins[0] are valid.

DP Transition:

Exclude the current coin: dp[i][j] = dp[i - 1][j] (ways to make j using coins up to i-1).

Include the current coin (if j >= coins[i]): Add dp[i][j - coins[i]] (ways to make j - coins[i] using coins up to i, allowing reuse).

Result: dp[coins.length - 1][amount] (ways to make amount using all coins).

**Example:**

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

DP Table:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Coin Index (i) \ Amount (j) | 0 | 1 | 2 | 3 | 4 | 5 |
| 0 (Coin 1) | 1 | 1 | 1 | 1 | 1 | 1 |
| 1 (Coin 2) | 1 | 1 | 2 | 2 | 3 | 3 |
| 2 (Coin 5) | 1 | 1 | 2 | 2 | 3 | 4 |

Valid Combinations:

[1,1,1,1,1], [1,1,1,2], [1,2,2], [5] → Total 4.

**Key Insight:**

By iterating coins in the outer loop and amounts in the inner loop, the solution counts combinations (order-agnostic), ensuring no overcounting of permutations like [1,2] and [2,1].

**Conclusion:**

The code is correct, and the DP definition aligns with the problem’s requirement to count combinations (not permutations). The solution efficiently avoids overcounting by leveraging the order of loops.

**Style 2: int[][] dp = new int[1 + coins.length][1 + amount];**

class Solution {

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

// Test out by amount = 0, coins = []

if(amount == 0) {

return 1;

}

// Test out by amount = 7, coins = []

if(coins.length == 0) {

return 0;

}

int[][] dp = new int[1 + coins.length][1 + amount];

for(int i = 0; i <= coins.length; i++) {

dp[i][0] = 1;

}

for(int i = 1; i <= coins.length; i++) {

for(int j = 1; j <= amount; j++) {

dp[i][j] = dp[i - 1][j];

if(j >= coins[i - 1]) {

dp[i][j] += dp[i][j - coins[i - 1]];

}

}

}

return dp[coins.length][amount];

}

}

Time Complexity: O(n×m), where n = number of coins, m = amount.

Space Complexity: O(n×m) for the DP table.

**Solution 4: 1D DP (30 min)**

**Refer to**

<https://leetcode.com/problems/coin-change-2/discuss/99212/Knapsack-problem-Java-solution-with-thinking-process-O(nm)-Time-and-O(m)-Space>

class Solution {

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

// Test out by amount = 0, coins = []

if(amount == 0) {

return 1;

}

// Test out by amount = 7, coins = []

if(coins.length == 0) {

return 0;

}

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

dp[0] = 1;

for(int i = 1; i <= coins.length; i++) {

for(int j = 1; j <= amount; j++) {

if(j >= coins[i - 1]) {

dp[j] += dp[j - coins[i - 1]];

}

}

}

return dp[amount];

}

}

Time Complexity: O(n×m), where n = number of coins, m = amount.

Space Complexity: O(m) for the DP array.

**Why find combination requires outer for loop over coins and inner for loop over Amount ?**

**Refer to Deepseek**

// A more detail description of this problem: based on 518. Coin Changes 2

// Refer to

// https://leetcode.com/problems/coin-change-2/discuss/176706/Beginner-Mistake%3A-Why-an-inner-loop-for-coins-doensn't-work-Java-Soln

// https://leetcode.com/problems/coin-change-2/discuss/176706/Beginner-Mistake:-Why-an-inner-loop-for-coins-doensn't-work-Java-Soln/306232

// https://leetcode.com/problems/coin-change-2/discuss/141076/Logical-Thinking-with-Clear-Java-Code

i have received below comparison between a correct solution and a wrong solution, can you review if the explain

correct or not ?

Example of code giving wrong answer for outer loop iterating over amount and inner loop iterating over coins:

class Solution {

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

if(amount<=0)

return 0;

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

for(int i =0; i<amount+1; i++){

for(int coin: coins){

if(coin+i < amount+1 ){

ways[i+coin] = ways[i]+ways[i+coin];

}

if(i==0){

ways[i]=1;

}

}

}

for(int i=0; i< ways.length; i++){

System.out.print(i+":"+ways[i]+" ");

}

return ways[amount];

}

}

Correct Solution:

class Solution {

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

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

combi[0] = 1;

for(int i = 0; i < coins.length; i++){

for(int j=1; j< amount+1; j++){

if(j-coins[i]>=0)

combi[j] = combi[j]+combi[j-coins[i]];

}

}

for(int a: combi)

System.out.print(a+" ");

return combi[amount];

}

}

The two codes on superificial comparison look equal, but the Code 1 gives a higher number of solutions that the

correct answer.

The reason for this is when we create an amount array from 0...Amount, if we iterate over all the coins a solution

can be added twice. For example to create 7:

When amount is 2 and the coin value is 5, it would be counted as 1 way.

When amount is 5 and the coin value is 2, the number of ways become 2.

The set is either case remains 1 coin of 2 and 1 coin of 5. But the first method adds it twice.

So we create use an outer loop of coins so that a combination once used cannot be used again.

================================================================================================================

Well, the real reason why the answer changes because of loops is because of the change in dp definition

when you try to reduce the space complexity.If we define dp[i][j] as "number of ways to get sum 'j' using

'first i' coins", then the answer doesn't change because of loop arrangement.

So why does the answer change only when we try to reduce the space complexity?

To get the correct answer, the correct dp definition should be dp[i][j]="number of ways to get sum 'j' using

'first i' coins". Now when we try to traverse the 2-d array row-wise by keeping only previous row array(to

reduce space complexity), we preserve the above dp definition as dp[j]="number of ways to get sum 'j' using

'previous /first i coins' " but when we try to traverse the 2-d array column-wise by keeping only the

previous column, the meaning of dp array changes to dp[j]="number of ways to get sum 'j' using 'all' coins".

In the below code you can see that if we are not interested in reducing the space complexity, both the loop

arrangements yield the same answer.

Code:-

class Solution {

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

if (coins.length == 0) {

if (amount == 0) return 1;

return 0;

}

int dp[][] = new int[coins.length][amount + 1];

//Initializing first column

for (int i = 0; i < coins.length; i++) {

dp[i][0] = 1;

}

//Initializing first row.

for (int i = 1; i <= amount; i++) {

if (i % coins[0] == 0) {

dp[0][i] = 1;

} else {

dp[0][i] = 0;

}

}

//Check the difference between interchanging the for loops.

//Both of them work. You can comment it out and check.

//1st one

for (int i = 1; i < coins.length; i++) {

for (int j = 1; j <= amount; j++) {

if (j >= coins[i]) {

dp[i][j] = dp[i - 1][j] + dp[i][j - coins[i]];

} else {

dp[i][j] = dp[i - 1][j];

}

}

}

//2nd one

for (int j = 1; j <= amount; j++) {

for (int i = 1; i < coins.length; i++) {

if (j >= coins[i]) {

dp[i][j] = dp[i - 1][j] + dp[i][j - coins[i]];

} else {

dp[i][j] = dp[i - 1][j];

}

}

}

return dp[coins.length - 1][amount];

}

}

Okay, let's try to understand this problem. The user provided two solutions for the Coin Change II problem, one incorrect and one correct. They're asking if the explanation given for why the first solution is wrong is accurate.

First, I need to recall what the Coin Change II problem is about. The goal is to find the number of combinations that make up a certain amount using given coins, where each coin can be used infinitely. The order doesn't matter, so [1,2] and [2,1] count as the same combination and should not be counted twice.

Looking at the first code (the wrong one), the loops are structured with the outer loop iterating over the amount and the inner loop over the coins. The correct solution has the outer loop over coins and the inner over the amount. The explanation given is that in the wrong code, combinations are counted multiple times because the order of processing leads to considering permutations instead of combinations. For example, using coin 2 then 5 and 5 then 2 would be treated as different, thus adding duplicates.

But wait, the user's explanation says that in the wrong code, when you process coins in the inner loop, you might add the same combination in different orders. For instance, making 7 with coin 2 and 5: if you process amount 2 with coin 5 (which doesn't add anything), then later process amount 5 with coin 2 (which adds a way), but this would actually create two different paths. However, since the problem counts combinations, not permutations, this would be overcounted.

The correct approach, using coins in the outer loop, ensures that each coin is considered one after another. This way, once you move to a new coin, you're building combinations that include that coin in addition to the previous ones, but you don't reuse previous coins in a way that creates permutations. So, the order of processing coins first locks in the order, ensuring that combinations are unique.

The explanation provided by the user also mentions that when you reduce the space complexity from 2D to 1D DP, the order of loops matters. If you loop over coins first (outer loop), then the 1D array correctly represents the number of combinations. If you loop over the amount first, you end up counting permutations because each step can add coins in any order, leading to overcounting.

The user also provided a 2D DP code where both loop orders (coins first or amount first) work, which contradicts the earlier statement. Wait, but in the 2D code, both loops work? How is that possible?

Wait, in the 2D code, when you loop over coins first (i then j), each dp[i][j] depends on dp[i-1][j] (excluding the current coin) and dp[i][j - coins[i]] (including the current coin). This correctly builds combinations by considering each coin's contribution step by step.

If you loop over j (amount) first, then for each amount, you process all coins. But in this case, since the DP is 2D, each coin's contribution is tracked per amount. Wait, maybe the 2D approach doesn't have the same issue because the state is tracked per coin and per amount. So the order of loops doesn't matter in 2D because each coin's contribution is considered independently. However, when you compress to 1D, the order becomes crucial because you're overwriting the same array, and the direction of iteration affects whether you count combinations or permutations.

So the key difference is that in 1D space, the order of loops determines whether you're allowing permutations (wrong) or combinations (correct). The outer loop over coins ensures that each coin is added in a way that combinations are built without considering order, while the outer loop over amount allows multiple orders to be counted, leading to permutations being counted as separate combinations.

The user's explanation that in the first code (wrong), the same combination is counted multiple times due to different orderings is correct. The correct code with outer loop over coins avoids this by ensuring that once a coin is processed, the combinations built with that coin are not revisited in a different order, thus maintaining unique combinations.

Therefore, the original explanation provided by the user is accurate. The key issue is the order of loops when using a 1D DP array. The correct approach processes each coin sequentially and updates the DP array in a way that combinations are built without considering order, whereas the incorrect approach allows for permutations to be counted as different combinations.

**In simple:**

**Outer loop on Coins + Inner loop on Amount = Combination -> For L518**

**Outer loop on Amount + Inner loop on Coins = Permutation**

The explanation provided is partially correct but contains inaccuracies. Let’s clarify the key differences between the two solutions and why the loop order matters in the 1D DP approach.

**Key Issue: Loop Order in 1D DP**

The Coin Change II problem requires counting combinations (order-agnostic), not permutations (order-sensitive). The loop order in the 1D DP solution determines whether permutations or combinations are counted.

**Correct Solution (Outer Loop: Coins):**

for (int coin : coins) {

for (int j = coin; j <= amount; j++) {

dp[j] += dp[j - coin];

}

}

Why It Works:
By iterating over coins first, we ensure that each combination is counted exactly once. For example, the combination [1, 2] is only counted as 1 + 2, not 2 + 1. This avoids overcounting permutations.

**Incorrect Solution (Outer Loop: Amount):**

for (int i = 0; i < amount + 1; i++) {

for (int coin : coins) {

if (i + coin <= amount) {

dp[i + coin] += dp[i];

}

}

}

Why It Fails:
This counts permutations (e.g., [1, 2] and [2, 1] are treated as distinct). The outer loop over amount allows coins to be added in any order, leading to overcounting.

**Why the Explanation is Partially Correct:**

Core Issue:

The explanation correctly identifies that the loop order affects whether permutations or combinations are counted. However, the statement that "the answer changes only when reducing space complexity" is misleading. The loop order fundamentally changes the DP definition in both 1D and 2D approaches.

2D DP Clarification:

In the 2D DP code provided, both loop orders do not yield the same result! The code contains a logical error:

The second loop (outer loop over amount) recomputes dp[i][j] incorrectly, overwriting the results from the first loop.

The correct 2D DP approach requires an outer loop over coins (i) and an inner loop over amounts (j).

**Correct 2D DP Approach:**

for (int i = 1; i < coins.length; i++) {

for (int j = 1; j <= amount; j++) {

dp[i][j] = dp[i - 1][j];

if (j >= coins[i]) {

dp[i][j] += dp[i][j - coins[i]];

}

}

}

Outer Loop: Coins (ensures combinations are order-agnostic).

Inner Loop: Amount (updates dp[i][j] based on previous coins and current coin reuse).

**Why Reducing Space Complexity Changes the Answer:**

1D DP: When reducing from 2D to 1D, the loop order becomes critical.

Outer Loop (Coins): Preserves the "combinations" logic.

Outer Loop (Amount): Counts permutations instead.

2D DP: Loop order does not change the answer because the state dp[i][j] explicitly tracks the coin index i.

**Conclusion:**

Correct Explanation:

The loop order in the 1D DP solution determines whether permutations or combinations are counted. The outer loop over coins ensures combinations are counted correctly.

Incorrect Statement:

The claim that "both loop orders work in 2D DP" is false. The 2D code provided has a logical error (overwriting dp[i][j] with two loops), but the correct 2D DP requires an outer loop over coins.

Final Answer:

The original explanation correctly identifies the core issue (permutations vs. combinations) but misrepresents the 2D DP behavior. The loop order matters in 1D DP due to space optimization, not because of the DP definition itself.

**For Leetcode 518 when we upgrade 2D DP to 1D DP, why we don't need to change the inner for loop order ?**

Okay, let me try to work through this problem. So the user is asking why, when upgrading from a 2D DP solution to a 1D DP solution for Leetcode 518 (Coin Change II), the inner loop order doesn't need to change. Hmm.

First, I need to recall what the 2D DP solution does. In the 2D version, dp[i][j] represents the number of ways to make amount j using the first i+1 coins. The loops are structured such that the outer loop is over the coins, and the inner loop is over the amounts. For each coin, they iterate through each possible amount and update the dp array by considering both including and excluding the current coin.

Now, when converting this to 1D DP, the space is optimized by using a single array. But the key part is the loop order. In the 2D case, the order is coins first (outer loop), then amounts (inner loop). The 1D solution provided also uses an outer loop over coins and inner loop over amounts. But why isn't the inner loop order changed?

**Wait, maybe the confusion arises from how the 1D DP is updated. Let me think. In the 2D version, for each coin, when you process an amount j, you can use the current coin multiple times. So the 1D version must maintain the same property. If the inner loop is over amounts in increasing order, then when you compute dp[j], you're adding the value of dp[j - coin], which has already been updated in this iteration. That allows for multiple uses of the same coin, right?**

**Wait, but in the 2D case, the transition is dp[i][j] = dp[i-1][j] + dp[i][j - coin]. The dp[i][j - coin] uses the current coin again. So in the 1D version, by processing amounts in increasing order, we allow for the reuse of the same coin because when you compute dp[j], you're adding the value of dp[j - coin] which might have already been updated in the same iteration (for the same coin). This is the key point.**

**If the inner loop were reversed (from higher amounts to lower), then you wouldn't reuse the same coin multiple times, because dp[j - coin] would not have been updated yet. But in the 1D solution provided, the inner loop is in the same order as the 2D case. So the loop order (coins outer, amounts inner) remains the same to ensure that coins can be reused, and that combinations are counted instead of permutations.**

So the reason the inner loop order doesn't change is that the way the DP array is updated relies on processing amounts in a way that allows the reuse of the same coin. The order (from 1 to amount) ensures that when you compute dp[j], you have already considered the cases where the current coin is used multiple times. This preserves the combination count and avoids counting permutations.

Wait, but why can't we swap the loops? If we did, like iterating amounts first and then coins, that would lead to overcounting permutations. Because each amount would be updated by all coins in each step, leading to different orders of coins being considered as different combinations. So the order of loops in the 1D case is crucial to maintain the correct logic of building combinations by processing each coin and allowing it to contribute to all amounts in a way that doesn't count order.

Therefore, the inner loop (over amounts) remains in the same order (from 1 to amount) to allow for multiple uses of the same coin and to count combinations, not permutations. Changing the order would disrupt this and lead to incorrect results.

The loop order (outer loop over coins, inner loop over amounts) remains the same in the 1D DP solution to preserve the combination-counting logic and avoid overcounting permutations. Here's the detailed breakdown:

**Key Insight: Loop Order in 1D DP**

2D DP Logic:

dp[i][j] = dp[i-1][j] + dp[i][j - coins[i]]

Exclude the current coin: Take the value from the previous row (dp[i-1][j]).

Include the current coin: Add ways from the same row (dp[i][j - coins[i]]), allowing reuse.

1D DP Translation:

dp[j] += dp[j - coins[i]] (with dp[j] initialized from dp[i-1][j]).

The outer loop over coins ensures combinations are order-agnostic.

The inner loop over amounts (1 to amount) allows reusing the current coin multiple times.

**Why Inner Loop Order (1 to amount) Doesn’t Change:**

Preserving Combination Logic:

By iterating amounts from 1 to amount, we ensure that when computing dp[j], the value dp[j - coins[i]] includes the current coin’s contribution.

This mimics the 2D DP’s reuse of the current coin (dp[i][j - coins[i]]).

Avoiding Permutations:

If the inner loop iterated coins first (incorrect order), we would count permutations (e.g., [1,2] and [2,1] as distinct).

The current order counts combinations (order-agnostic).

**Conclusion:**

The inner loop order (1 to amount) preserves the 2D DP logic of reusing coins.

The outer loop over coins ensures combinations are counted, not permutations.

Changing the inner loop order would break the combination logic and count permutations.

**Refer to**

[L377.P16.4.Combination Sum IV(Backpack VI) (Ref.L39,L40,L322)](note://8000F8241A0B4F3AA39E5CA16B6A7BB8)