□ Coin Change II — Documentation

1. Problem Statement

Given:

- An integer amount representing the total amount of money.
- An array coins[] representing different denominations (unique integers).

You can use infinite coins of each type.
Objective: Return the number of combinations that sum to the given amount.
If the amount cannot be made with the given coins, return 0.

The solution must fit in a signed 32-bit integer.

2. Intuition

This is a classic dynamic programming problem, similar to the unbounded knapsack problem. We aim to count how many distinct combinations (not permutations) we can form using the coin denominations, where each coin can be used multiple times.

3. Key Observations

- The order of coins doesn't matter, so we use combinations (not permutations).
- A 1D DP array is sufficient, as we can build solutions incrementally.
- For a given coin c, any amount ≥ c can potentially be formed by adding c to previously computed combinations.
- Start building from smaller amounts to larger amounts to avoid overcounting.

4. **Approach**

We use a 1D dynamic programming (DP) array where dp[i] indicates the number of combinations to form amount i.

✅ Steps:

- Initialize dp array with size amount + 1, filled with 0s.
- Set dp[0] = 1 because there's one way to make amount 0 (use no coins).
- For each coin:
    - For each sub-amount i from coin to amount:
    - dp[i] += dp[i - coin]

This means we're adding the number of ways to form (i - coin) to the current dp[i].

5. **Edge Cases**

- amount = 0: Only one combination → no coins used ⇒ return 1.
- coins = []: If amount > 0, return 0 (no coins to use).
- coins with value > amount: Those coins are ignored in combinations.
- Duplicate coins: Not allowed, as per constraints.

6. **Complexity Analysis**

⏱ Time Complexity:

O(amount × len(coins))
We iterate over each coin, and for each coin, we iterate over the amount.

💾 Space Complexity:

O(amount)
We use a 1D array of size amount + 1 to store intermediate results.

7. Alternative Approaches

1. 2D DP Table:

Use a 2D array where dp[i][j] represents the number of combinations to form amount j using the first i coins.

- More intuitive but uses O(n × amount) space.
- Can be optimized to 1D as shown in our main approach.

2. Recursion + Memoization:

Use recursive calls with memoization to cache overlapping subproblems.

- Time complexity is similar, but implementation is less efficient due to recursive overhead.

8. Test Cases

Test Case ID	Input	Output	Explanation
TC1	amount = 5, coins = [1,2,5]	4	[5], [2+2+1], [2+1+1+1], [1+1+1+1+1]
TC2	amount = 3, coins = [2]	0	Cannot make 3 with only 2s
TC3	amount = 10, coins = [10]	1	Only one way: [10]
TC4	amount = 0, coins = [1,2,3]	1	Only one way: use no coins
TC5	amount = 7, coins = []	0	No coins to use

9. Final Thoughts

- This problem demonstrates the power of dynamic programming in solving combinatorial counting problems efficiently.
- The 1D DP approach is both space- and time-efficient.
- Be careful with coin ordering and indexing to ensure combinations (not permutations) are counted.
- Understanding this problem also helps in solving similar problems like minimum coin change, knapsack, and subset sum.