322. Coin Change

Medium | Breadth-First Search | Array | Dynamic Programming

Problem Description

You have an array `coins` that contains different coin denominations and an integer `amount` which represents the total amount of money you want to make with these coins. The task is to calculate the minimum number of coins needed to make up the given amount. If it's not possible to reach the `amount` with the given coin denominations, the function should return `-1`.

You can use each type of coin as many times as you want; in other words, there's an unlimited supply of each coin.

Intuition

The intuition behind the solution is based on a classic algorithmic problem, known as the Coin Change problem, which can be solved using Dynamic Programming (DP). The idea is to build up the solution by solving for smaller subproblems and then use those solutions to construct the answer for the larger problem.

The approach used is called "bottom-up" DP. We initialize an array `f` of size `amount + 1`, where each element `f[i]` will hold the minimum number of coins needed to make the sum `i`. We start with `f[0] = 0` since no coins are needed to achieve a total amount of 0.

We set all other values in `f` to `inf` (infinity) which signifies that initially, we assume it's impossible to make those amounts with any combination of the coins given.

Next, we iterate through each coin denomination, `x`. For each `x`, we go through the `f` array starting from `f[x]` to `f[amount]` trying to update the minimum number of coins needed for each amount `j` by considering the number of coins needed for `j - x` plus one more coin of denomination `x`. The inner loop uses the formula `f[j] = min(f[j], f[j - x] + 1)` to decide whether we have found a new minimum for amount `j`.

After filling up the `f` array, if `f[amount]` is still `inf`, that means it's not possible to form `amount` with the given coins, and we return `-1`. Otherwise, `f[amount]` will hold the fewest number of coins needed to make up the `amount`, and that's our answer.

Solution Approach

The solution is implemented using a dynamic programming approach, which effectively breaks down the problem of finding the minimum number of coins into smaller subproblems.

Here's a step-by-step breakdown of how the implementation works:

1. Initialize DP table: Create an array `f` with a size of `amount + 1`. The first element `f[0]` is set to 0 since no coins are needed to make an amount of 0. All other elements are set to `inf` (which represents a large number larger than any real coin count, used to indicate 'not possible' initially).

2. Algorithm loop:

   - Iterate through each of the coin denominations `x` provided in the `coins` array.
   - For each coin denomination `x`, run an inner loop from `x` to `amount` (inclusive).
   - In the inner loop, update the DP table `f` at each amount `j` (where `j` ranges from `x` to `amount`) using the formula:
     ```
     f[j] = min(f[j], f[j - x] + 1)
     ```

     What this does is check if using the current coin `x` results in a smaller coin count for the amount `j` than the one we've previously found (if any). We compare the existing number of coins for amount `j` (`f[j]`), and the number of coins for `j - x` plus one (`f[j - x] + 1` since we add one coin of denomination `x`).
3. Final decision:

   - After the DP table is filled, we check the value of `f` at index `amount` (which represents the amount we want to make).
   - If `f[amount]` is still set to `inf`, it means we could not find a combination of coins to make up the amount, hence we return `-1`.
   - If `f[amount]` has a definite number (not `inf`), it represents the minimum number of coins needed to make the amount, and we return this value as our answer.

The dynamic programming pattern used here is known as the Bottom-Up approach as we start solving for the smallest possible amount and build our way up to the desired `amount`, using previously computed values to find the next. This allows solving complex problems by combining the solutions of simpler subproblems.

The algorithm leverages the fact that reaching a smaller amount `j - x` efficiently and adding one more coin of `x` to it might be the optimal solution for a larger amount `j`.

Efficiency:

- Space complexity is `O(n)`, where `n` is the `amount`, as it only requires an array of size `amount + 1`.
- Time complexity is `O(m*n)`, where `m` is the number of coin denominations, and `n` is the `amount`, due to the nested loops iterating over each coin and each amount respectively.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach. Suppose we have `coins = [1, 3, 4]` and the `amount = 6`.

1. We initialize the DP table `f` with `amount + 1 (7)` slots: `f = [0, inf, inf, inf, inf, inf, inf]`. The first element `f[0]` is `0` because no coins are needed to make an amount of `0`.

2. Now we iterate through the coin denominations:

   a. For `x = 1`: - We iterate from `1` to `amount (6)`. - At each `j`, we update `f[j] = min(f[j], f[j - x] + 1)`. - After the loop, `f` looks like: `[0, 1, 2, 3, 4, 5, 6]`. For any `j`, at most `j` coins of denomination `1` are needed.

   b. For `x = 3`: - We iterate from `3` to `6`. - We update `f[3]` to `1`, `f[4]` to `2`, `f[5]` to `2`, and `f[6]` to `2`, because for each of these amounts, using a `3`-denomination coin is more efficient. - `f` now is: `[0, 1, 2, 1, 2, 2, 2]`.

   c. For `x = 4`: - We iterate from `4` to `6`. - We update `f[4]` to `1`, and `f[6]` to `2` (`f[5]` remains `2` as `f[5 - x]` is `inf`), which uses one coin of `4` and then utilizes previous results for remaining amount `2`. - `f` now looks like: `[0, 1, 2, 1, 1, 2, 2]`.

3. Our final DP table is `[0, 1, 2, 1, 1, 2, 2]`. Looking at the value of `f[6]`, we see `2`, which means the minimum number of coins needed to make the amount `6` is `2`. This would correspond to using coins `4` and `2`, which are both subsets of our initial `coins` array (with `2` being the sum of two `1` coins).

We return the minimum number of coins found, which is `2`. This is the least amount of coins that can make `6` with the denominations given.

Python Solution

```python
from typing import List

class Solution:
    def coinChange(self, coins: List[int], amount: int) -> int:
        # Initialize the maximum number of coins to a value greater than any possible coin number
        MAX = float('inf')

        # dp[i] will be storing the minimum number of coins required for amount i
        # dp[0] is 0 because no coins are needed for the amount 0
        dp = [0] + [MAX] * amount

        # Traverse through all the amounts from 1 to amount inclusive
        for coin in coins:  # For each coin
            for current_amount in range(coin, amount + 1):
                # Update the dp table by comparing the current value
                # with the value if we include the current coin
                dp[current_amount] = min(dp[current_amount], dp[current_amount - coin] + 1)

        # If we have not found a combination to form the amount
        # then dp[amount] will still be MAX
        return -1 if dp[amount] == MAX else dp[amount]

# Example usage:
# sol = Solution()
# print(sol.coinChange([1, 2, 5], 11))  # Output: 3 (11 can be made with three 3 coins: 5+5+1)

```

Java Solution

```java
class Solution {
    public int coinChange(int[] coins, int amount) {
        // Define a large value which would act as our "infinity" substitute.
        final int INF = 1 << 30;

        // 'dp' will hold our optimal solutions to sub-problems, dp[i] will store the minimum number of coins needed to make amount '
        int[] dp = new int[amount + 1];

        // Initialize the dp array with INF to signify that those amounts are currently not achievable with the given coins.
        Arrays.fill(dp, INF);

        // Base case initialization: No coins are needed to make an amount of 0.
        dp[0] = 0;

        // Iterate over each type of coin available.
        for (int coin : coins) {
            // For each coin, try to build up to the target amount, starting from the coin's value itself up to 'amount'.
            for (int currentAmount = coin; currentAmount <= amount; ++currentAmount) {
                // Check if the current coin can contribute to a solution for 'currentAmount'.
                // If so, update dp[currentAmount] to the minimum value between its current and the new possible number of coins usec
                dp[currentAmount] = Math.min(dp[currentAmount], dp[currentAmount - coin] + 1);
            }
        }

        # then dp[amount] will still be MAX
        return -1 if dp[amount] == MAX else dp[amount]

# Example usage:
# sol = Solution()
# print(sol.coinChange([1, 2, 5], 11))  # Output: 3 (11 can be made with three 3 coins: 5+5+1)

```


Java Solution

```java
class Solution {
    public int coinChange(int[] coins, int amount) {
        // Define a large value which would act as our "infinity" substitute.
        final int INF = 1 << 30;

        // 'dp' will hold our optimal solutions to sub-problems, dp[i] will store the minimum number of coins needed to make amount '
        int[] dp = new int[amount + 1];

        // Initialize the dp array with INF to signify that those amounts are currently not achievable with the given coins.
        Arrays.fill(dp, INF);

        // Base case initialization: No coins are needed to make an amount of 0.
        dp[0] = 0;

        // Iterate over each type of coin available.
        for (int coin : coins) {
            // For each coin, try to build up to the target amount, starting from the coin's value itself up to 'amount'.
            for (int currentAmount = coin; currentAmount <= amount; ++currentAmount) {
                // Check if the current coin can contribute to a solution for 'currentAmount'.
                // If so, update dp[currentAmount] to the minimum value between its current and the new possible number of coins usec
                dp[currentAmount] = Math.min(dp[currentAmount], dp[currentAmount - coin] + 1);
            }
        }

        # then dp[amount] will still be MAX
        return -1 if dp[amount] == MAX else dp[amount]

# Example usage:
# sol = Solution()
# print(sol.coinChange([1, 2, 5], 11))  # Output: 3 (11 can be made with three 3 coins: 5+5+1)

```


Java Solution

```java
class Solution {
    public int coinChange(int[] coins, int amount) {
        // Define a large value which would act as our "infinity" substitute.
        final int INF = 1 << 30;

        // 'dp' will hold our optimal solutions to sub-problems, dp[i] will store the minimum number of coins needed to make amount '
        int[] dp = new int[amount + 1];

        // Initialize the dp array with INF to signify that those amounts are currently not achievable with the given coins.
        Arrays.fill(dp, INF);

        // Base case initialization: No coins are needed to make an amount of 0.
        dp[0] = 0;

        // Iterate over each type of coin available.
        for (int coin : coins) {
            // For each coin, try to build up to the target amount, starting from the coin's value itself up to 'amount'.
            for (int currentAmount = coin; currentAmount <= amount; ++currentAmount) {
                // Check if the current coin can contribute to a solution for 'currentAmount'.
                // If so, update dp[currentAmount] to the minimum value between its current and the new possible number of coins usec
                dp[currentAmount] = Math.min(dp[currentAmount], dp[currentAmount - coin] + 1);
            }
        }

        # then dp[amount] will still be MAX
        # then dp[amount] will still be MAX
        return -1 if dp[amount] == MAX else dp[amount]

# Example usage:
# sol = Solution()
# print(sol.coinChange([1, 2, 5], 11))  # Output: 3 (11 can be made with three 3 coins: 5+5+1)

```

Java Solution

```java
class Solution {
    public int coinChange(int[] coins, int amount) {
        // Define a large value which would act as our "infinity" substitute.
        final int INF = 1 << 30;

        // 'dp' will hold our optimal solutions to sub-problems, dp[i] will store the minimum number of coins needed to make amount '
        int[] dp = new int[amount + 1];

        // Initialize the dp array with INF to signify that those amounts are currently not achievable with the given coins.
        Arrays.fill(dp, INF);

        // Base case initialization: No coins are needed to make an amount of 0.
        dp[0] = 0;

        // Iterate over each type of coin available.
        for (int coin : coins) {
            // For each coin, try to build up to the target amount, starting from the coin's value itself up to 'amount'.
            for (int currentAmount = coin; currentAmount <= amount; ++currentAmount) {
                // Check if the current coin can contribute to a solution for 'currentAmount'.
                // If so, update dp[currentAmount] to the minimum value between its current and the new possible number of coins usec
                dp[currentAmount] = Math.min(dp[currentAmount], dp[currentAmount - coin] + 1);
            }
        }

        // Return the answer for the target 'amount'. If dp[amount] is still INF, then it was not possible to make the amount using t
        return dp[amount] >= INF ? -1 : dp[amount];
    }
```