Topic Introduction

Today, let's explore a classic family of interview problems: **Unbounded Knapsack Dynamic Programming**. If you've ever encountered problems where you can use each item (like coins or numbers) an unlimited number of times to achieve some goal, you've met this pattern.

What is Unbounded Knapsack DP?

Unbounded knapsack dynamic programming is a technique used when you're allowed to choose any number of each item to reach a target. The classic "knapsack" problem is about picking items with weights and values to maximize value without exceeding a weight limit — and each item can be picked once. In the **unbounded** version, you can use each item as many times as you want.

How does it work?

You typically build up solutions for smaller targets using previous results, storing them in an array (often called `dp`). The idea is to figure out the answer for all values up to the target, usually by considering every way you could reach a value by adding an item.

Why is this useful in interviews?

This type of DP is common in questions involving coins, numbers, or ways to sum up to a target. It tests your understanding of dynamic programming, problem decomposition, and sometimes, how to optimize for time and space.

Simple Example (not one of the featured problems):

Suppose you have coins of 1 and 2, and you want to know in how many ways you can make 4.

You can build up an array where `dp[i]` is the number of ways to make amount `i`. For each coin, you update the ways to make every amount from that coin up to the target.

Now, let's dive into three famous problems that showcase this pattern:

- Coin Change
- Coin Change 2
- Perfect Squares

Why are these grouped together?

Each problem asks you to reach a target number using a given set of "building blocks" (coins, numbers, squares), and you can use each block as many times as you want! They all use a bottom-up DP approach with similar transitions, but each asks something slightly different: finding the minimum number of coins, counting the number of ways, or minimizing the number of perfect squares.

Problem 1: Coin Change ([LeetCode 322](https://leetcode.com/problems/coin-change/))

Problem Statement (Rephrased):

Given a list of coin denominations and a total amount, what is the minimum number of coins needed to make up that amount? If it's

not possible, return -1.

Example:

- Input: `coins = [1, 2, 5]`, `amount = 11`
- Output: `3`
- Explanation: 11 = 5 + 5 + 1 (uses three coins, which is minimal).

You Try:

What about `coins = [2]`, `amount = 3`?

Try solving this one with pen and paper before reading on!

Brute-force Approach:

Try every possible combination of coins, recursively subtracting coin values from amount until you reach zero. This is very slow (exponential time), since you may recompute the same subproblems over and over.

- **Time Complexity:** Exponential, O(S^n), where S is amount, n is number of coins.

Optimal DP Approach:

Let's use **bottom-up dynamic programming**. We want `dp[x]` to be the minimum coins needed for amount `x`. Initialize `dp[0] = 0` (zero coins needed for zero), and set all other entries to a very large value (`amount + 1` is safe, as it's impossible to need more than `amount` coins).

For each amount from 1 to `amount`, check every coin. If the coin value is less than or equal to the current amount, update `dp[amt] = min(dp[amt], dp[amt - coin] + 1)`.

Step-by-step:

- Create a DP array of size `amount + 1`, fill with `amount + 1`, set `dp[0] = 0`.
- For each amount from 1 to `amount`:
    - For each coin:
        - If coin <= amount, update `dp[amount] = min(dp[amount], dp[amount - coin] + 1)`
- If `dp[amount]` is still `amount + 1`, return -1. Otherwise, return `dp[amount]`.

Python Solution:

```
def coinChange(coins, amount):
    # dp[i] will be the minimum coins needed for amount i
    dp = [amount + 1] * (amount + 1)
    dp[0] = 0  # 0 coins to make amount 0

    for amt in range(1, amount + 1):
        for coin in coins:
            if coin <= amt:
                dp[amt] = min(dp[amt], dp[amt - coin] + 1)

    return -1 if dp[amount] == amount + 1 else dp[amount]
```

Explanation:

- We initialize the DP array with `amount + 1` as a placeholder for "infinity."
- We iterate through every amount from 1 up to the target.
- For each coin, if it fits into the current amount, we see if using it would reduce the number of coins needed.
- If we can't make the amount, we return -1.

Time Complexity: O(amount * n), where n is the number of coins.
Space Complexity: O(amount) for the DP array.

Trace Example:
Let's trace `coins = [1, 2, 5]`, `amount = 11`.

- Start with `dp = [0, inf, inf, ..., inf]` (size 12).
- At amt = 1: Try coin 1 => dp[1] = min(inf, dp[0] + 1) = 1
- At amt = 2: Try coin 1 => dp[2] = dp[1] + 1 = 2; Try coin 2 => dp[2] = min(2, dp[0] + 1) = 1
- Continue up to amt = 11, always picking the minimal solution.
- At the end, dp[11] = 3.

Try it Yourself:
What is the answer for `coins = [2, 3, 7]`, `amount = 12`?
Work out the DP array step by step.

Encouragement:
Pause and try to implement this on your own before reading further. Pen-and-paper helps!

Reflective Prompt:
Did you know this could also be solved using BFS? Try thinking about why that works!

Problem 2: Coin Change 2 ([LeetCode 518](https://leetcode.com/problems/coin-change-2/))

Problem Statement (Rephrased):

Given a list of coin denominations and a total amount, how many distinct ways can you make up that amount (using each coin as many times as you like)?

Example:

- Input: `coins = [1, 2, 5]`, `amount = 5`
- Output: `4`
- Explanation: The four combinations are:
  - 5
  - 2 + 2 + 1
  - 2 + 1 + 1 + 1
  - 1 + 1 + 1 + 1 + 1

Try This:

How many ways to make amount 3 with coins [2]?

How is this different from Problem 1?

Instead of minimizing coins, we're counting **the number of unique combinations**.
Order doesn't matter (e.g., [2,1,1] and [1,2,1] count as the same combination).

Brute-force:

Try all possible ways to sum to the amount, trying each coin any number of times. Exponential time.

Optimal DP Approach:

We use DP, but this time, `dp[i]` is the number of ways to make amount `i`.
The key difference: **the order of loops.**
We iterate coins **outside**, then amounts **inside**. This ensures we don't overcount permutations.

Step-by-step:

- Create a DP array of size `amount + 1`, initialize with 0. Set `dp[0] = 1` (1 way to make amount 0).
- For each coin:
    - For each amount from coin to amount:
        - Update `dp[amt] += dp[amt - coin]` (adding ways using this coin).
- Return `dp[amount]`.

Pseudocode:

```
function change(amount, coins):
    dp = array of size amount+1, all 0
    dp[0] = 1
    for coin in coins:
        for amt from coin to amount:
            dp[amt] += dp[amt - coin]
    return dp[amount]
```

Example Trace:

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

- Start: dp = [1,0,0,0,0,0]
- Coin 1: dp = [1,1,1,1,1,1]
- Coin 2: dp = [1,1,2,2,3,3]
- Coin 5: dp = [1,1,2,2,3,4]
- So, 4 ways to make 5.

Try it Yourself:

How many ways to make amount 7 with coins [2, 3, 5]?
Walk through the DP table step by step.

Time Complexity: O(amount * n), n = number of coins

Space Complexity: O(amount)

Problem 3: Perfect Squares ([LeetCode 279](https://leetcode.com/problems/perfect-squares/))

Problem Statement (Rephrased):

Given an integer n, find the minimum number of perfect square numbers (like 1, 4, 9, 16, ...) that sum up to n.

Example:

- Input: `n = 12`
- Output: `3`
- Explanation: 12 = 4 + 4 + 4.

Try This:
What about n = 13?

How is this similar to the previous problems?

Each perfect square is like a "coin" you can use as many times as you want to reach n, and you want the minimum number used.

Brute-force:

Try all combinations of squares, recursively, to sum to n. Exponential time.

Optimal DP Approach:

Same as Coin Change, but your "coins" are all perfect squares <= n.

Step-by-step:

- Generate all perfect squares <= n.
- Create dp[0..n], set dp[0] = 0, others to infinity.
- For amount from 1 to n:
    - For each square <= amount:
        - dp[amount] = min(dp[amount], dp[amount - square] + 1)
- Return dp[n].

Pseudocode:

```
function numSquares(n):
    squares = [k*k for k in 1..sqrt(n)]
    dp = array of size n+1, filled with inf
    dp[0] = 0
    for amt from 1 to n:
        for square in squares:
            if square <= amt:
                dp[amt] = min(dp[amt], dp[amt - square] + 1)
```

```
    return dp[n]
```

Example Trace:

n = 13

Perfect squares: [1, 4, 9]

- dp[0] = 0
- dp[1] = 1 (1)
- dp[4] = 1 (4)
- dp[9] = 1 (9)
- dp[13] = min(dp[12]+1, dp[9]+1, dp[4]+1) = min(3+1,1+1,1+1) = 2

So, 13 = 4 + 9.

Try It Yourself:

What is the answer for n = 17?

Time Complexity: O(n * sqrt(n))
Space Complexity: O(n)

Summary and Next Steps

Today you learned to spot and solve unbounded knapsack DP problems!
These three problems — Coin Change, Coin Change 2, and Perfect Squares — all use bottom-up DP to solve "use any number of these to reach a target" questions.

Key Patterns:

- **State:** What's the minimum or count for each sub-amount?
- **Transition:** For each amount, consider all ways to use each "coin" or "block."
- **Order of Loops:** For counts, using coins outside avoids counting permutations.

Common Pitfalls:

- Mixing up "minimum number" vs "number of ways" problems.
- Forgetting to initialize `dp[0]` to 0 (min) or 1 (count).
- Using the wrong order of loops in counting problems, leading to overcounting.

Action List

- Solve all three problems on your own, even if you've read the code or pseudocode.
- Try solving Coin Change 2 and Perfect Squares using a recursive+memoization (top-down) approach.
- Explore related problems like "Combination Sum" or "Word Break" that use similar DP patterns.
- Compare your code to others — look for edge-case handling and clarity.
- If you get stuck, don't worry — persistence is key! Each attempt builds mastery.

Happy coding, and see you in the next PrepLetter!