Dynamic Programming Examples

Coin Change: https://www.hackerrank.com/challenges/coin-change
Edit Distance: https://leetcode.com/problems/edit-distance/#/description

Coin Change: My Words

- You have an infinite number of coins of some finite number of denominations
- You need to make change for a particular value, how many ways are there to do it?

Coin Change: Their Words

Task

Write a program that, given

- The amount N to make change for and the number of types M of infinitely available coins
- A list of M coins - $C = \{C_1, C_2, C_3, .., C_M\}$

Prints out how many different ways you can make change from the coins to STDOUT.

The problem can be formally stated:

Given a value N, if we want to make change for N cents, and we have infinite supply of each of $C = \{C_1, C_2, \ldots, C_M\}$ valued coins, how many ways can we make the change? The order of coins doesn't matter.

Coin Change: Some Thoughts

- A vanilla recursive solution is very slow
  - Introduces redundant computation, done by each ‘branch’ of the recursive tree
- Key Concept of DP: Store information!

Problem: Make Change ’10’ given coins {6, 3, 1}:

- Chain A chooses one ‘6’ coin, no ‘3’ coins
- Chain B chooses two ‘3’ coins, no ‘6’ coins
- Both have to compute the same subproblem!

Main

```
private static int[] coins;
private static long[][] dp;

public static void main (String[] args) {
    Scanner in = new Scanner(System.in);

    int n = in.nextInt(); //n <= 250
    int m = in.nextInt();
    dp = new long[n+1][m];
    //-1 denotes unsolved subproblem
    for (int i = 0; i < n+1; i++) {
        Arrays.fill(dp[i], val: -1);
    }

    coins = new int[m];
    for (int i = 0; i < m; i++) {
        coins[i] = in.nextInt();
    }

    Arrays.sort(coins); //in practice, low-high orderings have yielded higher
                        // instances of being able to use 'dp' array
    long answer = recurse( index: 0, n);
    System.out.println(answer);
}
```

The Actual Work

```
private static long recurse(int index, int rem) {
    //base cases
    if (rem == 0) { //served exact change
        return 1;
    }
    else if (index == coins.length) { //no more coins to consider
        return 0;
    }

    //not base cases
    if (dp[rem][index] != -1) { //already precomputed subproblem
        return dp[rem][index];
    }
    else { //have to do the work
        long value = 0;
        for (int cTake = 0; cTake * coins[index] <= rem; cTake++) {
            value += recurse( index: index+1,  rem: rem-cTake*coins[index]);
        }
        dp[rem][index] = value;
        return value;
    }
}
```

Problem 2: Edit Distance

- You are given String A and String B, and three operations:
  - **Add** a character
  - **Remove** a character
  - **Replace** a character with another

- What's the minimum number of changes that will turn String A into String B?

Sound Familiar?

- Recall the Buggy Robot problem from earlier in the semester
- Same Basic Idea:
  - Branching outwards through every possibility doubles your problem size
    - redundancy, and humoring branches that won't be part of the solution
  - Solution: Like an informed pathfinding algorithm, track the cost of the best way to get to any given state

Code (Too big for slides)