ProgTeam Week 6

DP I

Recursion can be inefficient

- Look at this seemingly innocent code to calculate the n-th fibonacci number:

```
def fibonacci(n):
    if n == 0: return 0
    if n == 1: return 1
    return fibonacci(n - 1) + fibonacci(n - 2)
```

- This will take ages, for even slightly large n!

Dynamic Programming

- We know Fib(X) is always the same for the same X
- Let's memorize the answer when we calculate Fib(X) the first time

```
dp = [-1] * 100
def fibonacci_dp(n):
    if n == 0: return 0
    if n == 1: return 1
    if dp[n] != -1:
        return dp[n]
    dp[n] = fibonacci_dp(n - 1) + fibonacci_dp(n - 2)
    return dp[n]
```

- This technique is called <u>Dynamic Programming</u>

Dynamic Programming

- Key idea is we're always breaking problem into smaller problem
  - Should be some aspect of the problem that we change, that can't be changed back
- N is always decreasing when we calculate Fibonacci numbers
- Eventually we should reach "base case", where we can't divide problem further
  - N = 0, N = 1 for Fibonacci numbers; these are just given by definition

Example 1: Making Change

- What’s the minimum number of coins needed to make a value of N in an arbitrary coin system
- Greedy (take largest coin) works for real coin systems, but not in general:
    - N = 9
    - Coins = [1,4,5,7]
    - Best is {4,5}, not {7,1,1}
- Solution: use DP!

Example 1: Making Change

- MinCoins(0) = 0 <- Base Case
- MinCoins(x) = min{ MinCoins(x - c[i]) + 1}, for x - c[i] >= 0
    - This is called recursive specification
    - Has all the information we need to turn it into code
- Evaluating the runtime: There are N possible values of X, and we iterate over all coins for each of them
    - Total work ~= N * |Coins|
- In general, runtime for DP = (# of states) * (work per state)

Example 2: Longest Common Subsequence

Definitions: A Subsequence of S is a string that can be obtained by removing characters from S

“abc” is a subsequence of “adbadc” -> a~~d~~b~~ad~~c

The Longest Common Subsequence of S and T is the longest possible string that is a subsequence of both S and T

Example 2: Longest Common Subsequence

- Observation: If the last two characters of S and T match, this is always part of the LCS
- If they don’t?
  - Clearly we should discard one of the characters, but maybe not both
    - S = “ABA” and T = “ACB”. ‘B’ should be part of the LCS!
  - Using DP, we can try both options and see gets a better answer

Example 2: Longest Common Subsequence

- LCS(i,j) is the (length of the) LCS of S[0…i] and T[0…j]
- Base case: if (i < 0) or (j < 0), LCS(i,j) = 0
  - No more characters left
- If S[i] == T[j], LCS(i,j) = 1 + LCS(i - 1, j - 1)
- Otherwise, LCS(i,j) = max{LCS(i - 1,j), LCS(i, j - 1)}

Example 2: Longest Common Subsequence

- Now we know the length of the LCS. What is the actual LCS though?
- We can build it by solving the problem first, and memorizing the answer for each step
- Now let’s build the answer one character at a time, using Build(i,j):
  - If S[i] == T[j], this character goes on the back of the string
  - Otherwise, see if LCS(i - 1, j) >= LCS(i, j - 1)
    - Pick the direction that gives the larger LCS