Section 2 Overview

Agenda

- Inductive Proof
  - Inductive Proof Steps
  - Recurrence Relation
- Dynamic Programming
- Optional Problem
  - Recursive Python
  - Pascal's Triangle

Inductive Proof

Inductive Proof Steps

Steps of Induction:

1. Problem Statement (kind of optional but nice)
2. Base Case
3. Inductive Hypothesis
4. Inductive Leap of Faith (Inductive Step)
5. Conclusion

Inductive Proof

1 Money

1.1 Problem

Prove that $\sum_{i=1}^{n} [(i) \times (i+1)] = \frac{(n)(n+1)(n+2)}{3}$ where $n \geq 1$ using a beautiful inductive proof.

1.2 Solution

Base Case:

$$\text{Let } n = 1.$$

$$1 \times 2 = 2$$

$$\frac{1 \times 2 \times 3}{3} = 2$$

$$2 = 2$$

Inductive Hypothesis:

$$\text{Assume, for n=k, that } \sum_{i=1}^{k} [(i) \times (i+1)] = \frac{(k)(k+1)(k+2)}{3}$$

Inductive Step:

By the Inductive Hypothesis,

$$\sum_{i=1}^{k} [(i) \times (i+1)] = \frac{(k)(k+1)(k+2)}{3}$$

Add $((k+1) \times (k+2))$ to both sides of the equation.

$$\sum_{i=1}^{k} [(i) \times (i+1)] + ((k+1) \times (k+2)) = \frac{(k)(k+1)(k+2)}{3} + ((k+1) \times (k+2))$$

Simplify the left side of the equation.

$$\sum_{i=1}^{k+1} [(i) \times (i+1)] = \frac{(k)(k+1)(k+2)}{3} + ((k+1) \times (k+2))$$

Simplify the right side of the equation.

$$\sum_{i=1}^{k+1} [(i) \times (i+1)] = \frac{(k)(k+1)(k+2)}{3} + \frac{3((k+1)(k+2))}{3}$$

$$\sum_{i=1}^{k+1} [(i) \times (i+1)] = \frac{(k+1)(k+2)(k+3)}{3}$$

$$\sum_{i=1}^{k+1} [(i) \times (i+1)] = \frac{(k+1)((k+1)+1)((k+1)+2)}{3} \qquad \square$$

Recurrence Relation Solution

RECURRENCE

The game of Hanoi Tower is to play with a set of disks of graduated size with holes in their centers and a playing board having three spokes for holding the disks.

The object of the game is to transfer all the disks from spoke A to spoke C by moving one disk at a time without placing a larger disk on top of a smaller one. The minimal number of moves required to solve the problem with n disks can be modeled by the following recurrence relation:

$$a_n = 2a_{n-1} + 1, n \geq 1$$

$$a_1 = 1$$

Plug and Chug Solution:

$$a_1 = 1$$

$$a_2 = 2a_{2-1} + 1 = 2*1 + 1 = 3$$

$$a_3 = 2a_{3-1} + 1 = 2*3 + 1 = 7$$

$$a_4 = 2a_{4-1} + 1 = 2*7 + 1 = 15$$

$$a_n = 2a_{n-1} + 1 = 2^n - 1, n \geq 1$$

Dynamic Programming

Convert some amount of money M into a given a list of denominations (decreasing order), using the smallest possible number of coins. Return the smallest number of coins (not the denominations used)

Greedy approach:

`Input:` `An amount of money, and an array (denoms) of d denominations = (c1, c2, ..., cd), in decreasing order (c1>c2>...cd).`

`Output:` `Min number of coins to make amt`

`greedy_change(amt, denoms):t`

```
remainder = amt //remaining amount to make change for
pieces = [ ] //output array of denominations (number of each denomination)
for k = 0 to denom.length: //for each denomination (starting with largest)
     pieces[k] = remainder/denoms[k]
     remainder = remainder % denoms[k] //practice mod!
return sum(pieces)
```

Dynamic Approach:

Looks at the minimum of previous choices then adds the denomination needed.

For example, if you were making 77 cents from 1, 3, and 7 cents. 77 cents would depend on:

1. The best combination for 77-1 = 76 cents, plus 1 cent coin
2. The best combination for 77-3 = 74 cents, plus a 3 cent coin
3. The best combination for 77-7 = 70 cents, plus a 7 cent coin

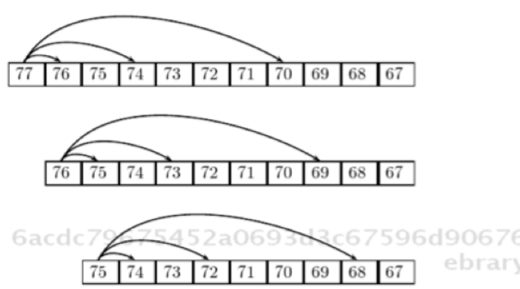

Figure 6.1 The relationships between optimal solutions in the Change problem. The smallest number of coins for 77 cents depends on the smallest number of coins for 76, 74, and 70 cents; the smallest number of coins for 76 cents depends on the smallest number of coins for 75, 73, and 69 cents, and so on.

Optional Problems

Sum All

Given a positive integer x, recursively find the sum of all numbers from 1 to x.

```
def sumAll(x):
     '''Function that recursively sums positive integers less
     than or equal to x'''
     if x==0:
          return 0
     else:
          return sumAll(x-1)+x
```

Pascal's Triangle

Write an algorithm that takes an integer n and returns the nth row of Pascal's Triangle (see here for an explanation: https://en.wikipedia.org/wiki/Pascal%27s_triangle)

```
def pascal(n):
    """
    input: an int, n, the line of pascal's triangle that you
want to return
    output: a list representing the nth line of pascal's
triangle.  If n is less than 2, then the function should return
[1]
    """
    if n < 2:
        return [1]

    prev_line = pascal(n-1)
    line = [prev_line[i]+prev_line[i+1] for i in
range(len(prev_line)-1)]
    line.insert(0,1)
    line.append(1)

    return line
```