Recursion, Induction, Dynamic Programming

CS16: Introduction to Algorithms & Data Structures
Summer 2021

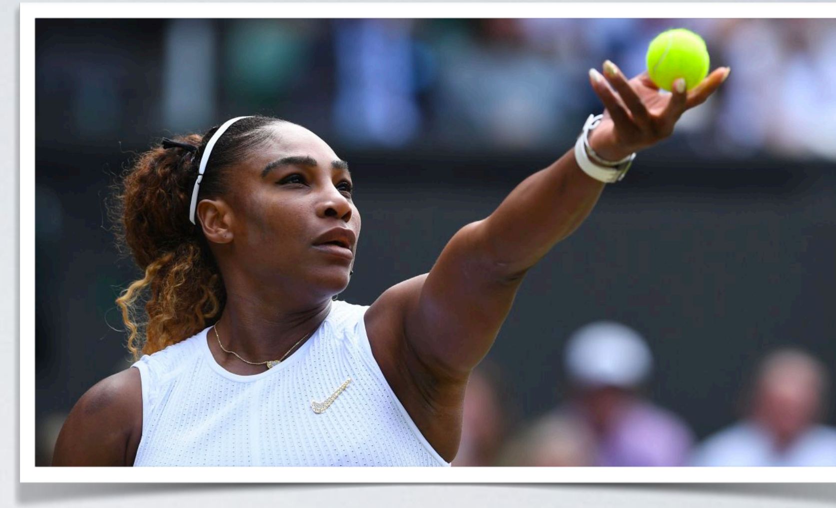
Outline

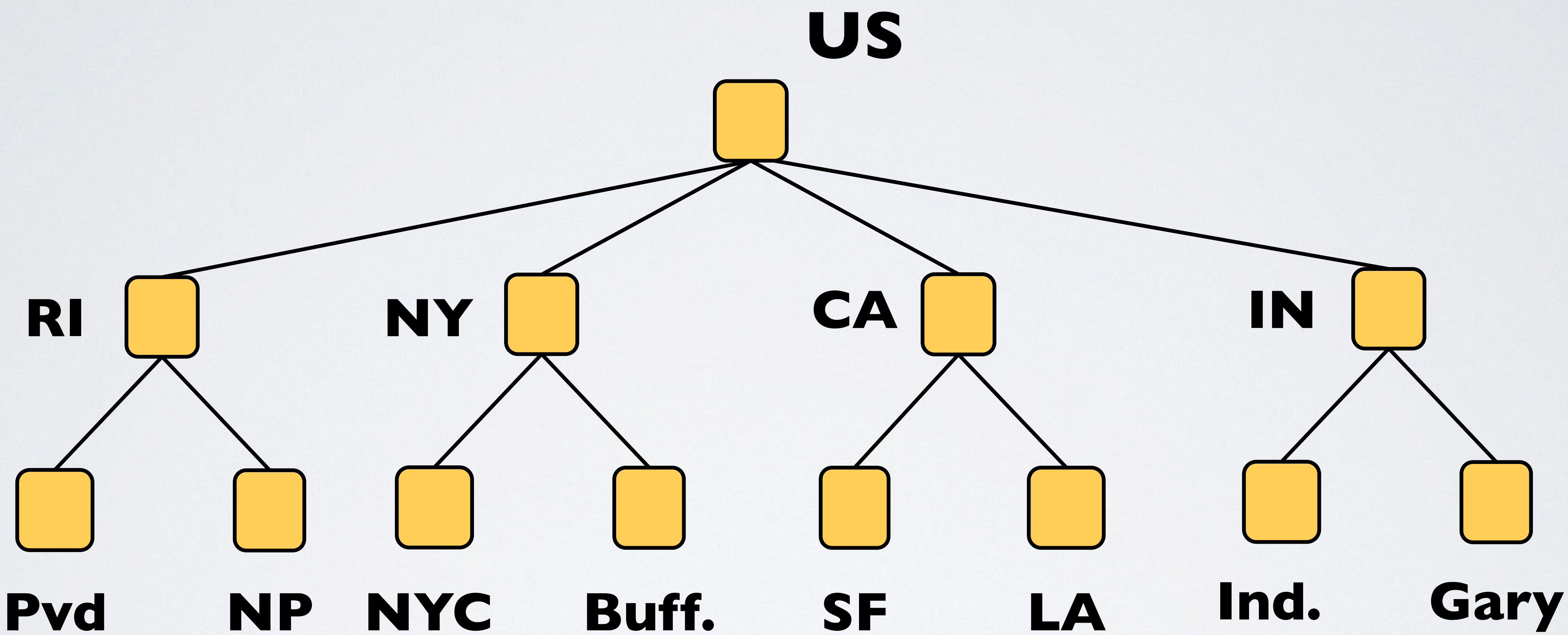
- Recursion
- Recurrence relations
- Plug & chug
- Induction
- Strong vs. weak induction

Collaboration policy

- **Can and should** discuss assignments with other students!
- Still **cannot** share code or written solutions

The Scouting Problem





recursive: defined in terms of itself

Recursion

- What is a recursive problem?
  - a problem defined in terms of itself
- What is a recursive function?
  - a function defined in terms of itself
  - example: Factorial, Fibonacci
- At each level, the problem gets easier/smaller

Recursive Algorithms

- Algorithms that call themselves
  - Call themselves on smaller inputs (sub-problems)
  - Combine the results to find solution to larger input
- Recursive algorithms
  - Can be very easy to describe & implement :-)
    - Especially for recursively-defined data structures (e.g. trees)
  - Can be hard to think about and to analyze :-(

Factorial

iterative: $$n! = \prod_{i=1}^{n} i = n \times (n-1) \times \cdots \times 1$$

recursive: $$n! = n \times (n-1)!,\ \text{with } 1! = 1$$

Recursive Factorial — Simulation

```
def factorial(n):
    if n == 1:
        return 1
    else:
        return n * factorial(n-1)
```

- call **factorial**(3)

Recursive Factorial — Simulation

```
def factorial(n):
    if n == 1:
        return 1
    else:
        return n * factorial(n-1)
```

- call **factorial**(3)
  - level #1: 3≠1 so 3 × **factorial**(2)

Recursive Factorial — Simulation

```
def factorial(n):
    if n == 1:
        return 1
    else:
        return n * factorial(n-1)
```

- call **factorial**(3)
    - level #1: $3 \neq 1$ so $3 \times$ **factorial**(2)
        - level #2: $2 \neq 1$ so $2 \times$ **factorial**(1)

Recursive Factorial — Simulation

```
def factorial(n):
    if n == 1:
        return 1
    else:
        return n * factorial(n-1)
```

- call **factorial**(3)
  - level #1: 3≠1 so 3 × **factorial**(2)
    - level #2: 2≠1 so 2 × **factorial**(1)
      - level #3: 1==1 so return 1

Recursive Factorial — Simulation

```
def factorial(n):
    if n == 1:
        return 1
    else:
        return n * factorial(n-1)
```

- call **factorial**(3)
  - level #1: 3≠1 so 3 × **factorial**(2)
    - level #2: 2≠1 so 2 × **1**
      - ~~level #3: 1==1 so return **1**~~

Recursive Factorial — Simulation

```
def factorial(n):
    if n == 1:
        return 1
    else:
        return n * factorial(n-1)
```

- call **factorial**(3)
  - level #1: 3≠1 so 3 × **2**
    - ~~level #2: 2≠1 so 2 × **1**~~
      - ~~level #3: 1==1 so return **1**~~

Recursive Factorial — Simulation

```
def factorial(n):
    if n == 1:
        return 1
    else:
        return n * factorial(n-1)
```

- call **factorial**(3) = **6**
  - ~~fact(3): 3≠1 so 3 × 2~~
    - ~~level #2: 2≠1 so 2 × 1~~
      - ~~level #3: 1==1 so return 1~~

Wait a minute!!

you keep *calling* factorial but never actually *implemented it*

Recursive Factorial — Simulation

```
def factorial(n):
    if n == 1:
        return 1
    else:
        return n * factorial(n-1)
```

Recursive Factorial — Simulation

```
def factorial(n):
    if n == 1:
        return 1
    else:
        return n * factorial(n-1)
```

Example: recursive `array_max`

```
def array_max(array, n):
    if n == 1:
        return array[0]
    else:
        return max(array[n-1], array_max(array, n-1))
```

Example: recursive `array_max`

```
def array_max(array, n):
    if n == 1:
        return array[0]
    else:
        return max(array[n-1], array_max(array, n-1))
```

```
array_max([5,1,9,2], 4) = [ 9 ]
  max( 2 , array_max([5,1,9,2],3) = [ 9 ] )
        max( 9 , array_max([5,1,9,2],2) = [ 5 ] )
              max( 1 , array_max([5,1,9,2],1) = [ 5 ] )
```

Running Time of Recursive Algos

- Difficult to analyze :-(
- With iterative algorithms
  - we can count # of ops per loop
- How can we count # ops in a recursive step?
  - We can't…

```
def factorial(n):
    out = 1
    for i in range(1, n+1):
        out = i * out
    return out
```

```
def factorial(n):
 if n == 1:
  return 1
 else:
  return n * factorial(n-1)
```

Recurrence Relations

- Functions that express run time recursively

$$\underbrace{T(n) = 2 \cdot T(n-1) + 10}_{\text{general case}}, \quad \text{with} \;\; \underbrace{T(1) = 8}_{\text{base case}}$$

  - part 1: # of operations in *general* case
  - part 2: # of operations in *base* case

Example: recursive `array_max`

```
def array_max(array, n):
    if n == 1:
        return array[0]
    else:
        return max(array[n-1], array_max(array, n-1))
```

$$\underbrace{T(n) = T(n-1) + c_1,}_{\text{general case}} \quad \text{with} \quad \underbrace{T(1) = c_0}_{\text{base case}}$$

What about Big-Oh?

- general: constant # ops for comp & max + cost of recursive call
- base: constant # ops for comp and return

Big-O from Recurrence Relation

- Step #1: Plug & Chug
  - algebraic manipulations to guess a Big-O expression
- Step #2: Induction
  - prove that Big-O expression is correct

Example: recursive `array_max`

$$\underbrace{T(n) = T(n-1) + c_1}_{\text{general case}}, \quad \text{with} \quad \underbrace{T(1) = c_0}_{\text{base case}}$$

Plug & Chug

$$\underbrace{T(n) = T(n-1) + c_1}_{\text{general case}}, \quad \text{with} \quad \underbrace{T(1) = c_0}_{\text{base case}}$$

$$\begin{aligned}
T(1) &= c_0 \\
T(2) &= c_1 + T(1) = c_1 + c_0 \\
T(3) &= c_1 + T(2) = c_1 + c_1 + c_0 = 2c_1 + c_0 \\
T(4) &= c_1 + T(3) = c_1 + 2c_1 + c_0 = 3c_1 + c_0 \\
T(5) &= c_1 + T(4) = c_1 + 3c_1 + c_0 = 4c_1 + c_0 \\
&\vdots \\
T(n) &= c_1 + T(n-1) = \ldots = \ldots = (n-1)c_1 + c_0
\end{aligned}$$

- Closed form expression

$$\boxed{T(n) = (n-1) \cdot c_1 + c_0 = O(n)}$$

Are we done?

- That was just a guess…not a proof!
  - plugged & chugged to find a pattern
  - and then we guessed at a Big-O
- How can we be sure?
- We prove it using Induction

Induction

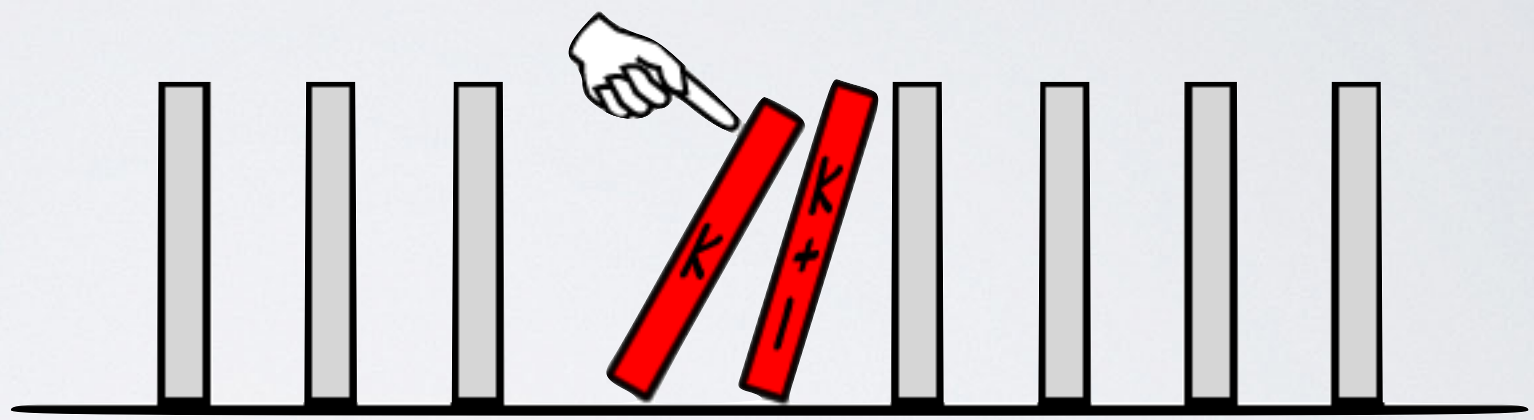
- Proof technique to prove statements about infinite sets of natural numbers
  - Can also be used for recursively-defined structures like trees
- To prove that a statement **P** is true for all positive numbers `n=1,2,3,4,…`
  - prove that a statement **P** is true for `n=1`
  - prove that **if** **P** is true for `n=k` then **P** is true for `n=k+1`

Steps to an Inductive Proof

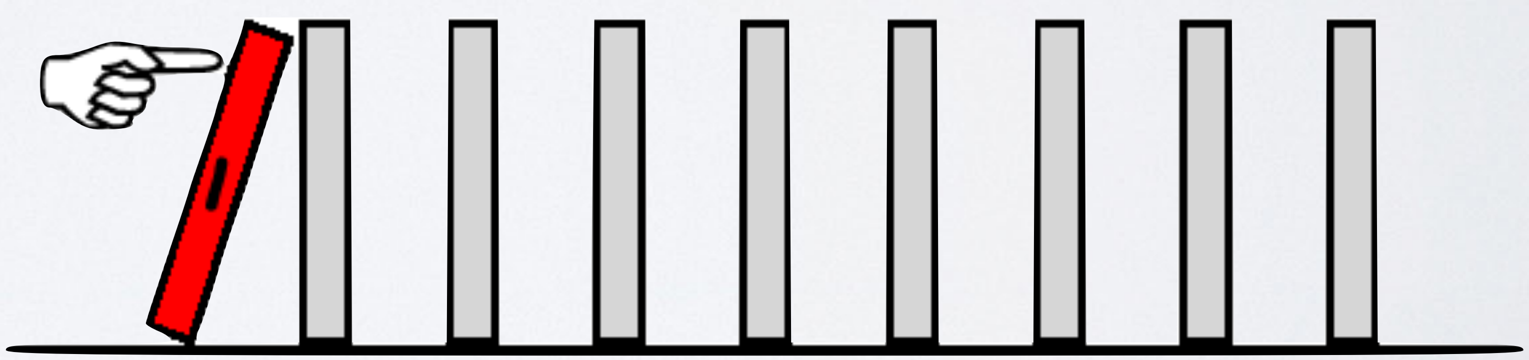
- Base case
  - prove that statement **P** is true for base case
- Inductive hypothesis
  - assume that **P** is true for some case `n = k`
- Inductive step
  - prove that if **P** is true for `n = k` then **P** is true for `n = k+1`
- Conclusion
  - Then **P** must be true for all `n`

Induction

Inductive step:



Base case:



Induction for `array_max`

- **P**(n): $T(n) = T(n-1) + c_1, \text{ w/ } T(1) = c_0$ is equal to

$$f(n) = (n-1) \cdot c_1 + c_0$$

- Prove for base case: **`n=1`**
  - $T(1) = c_0$ and $f(1) = (1-1) \cdot c_1 + c_0 = c_0$
- Inductive *assumption*: **`n=k`**
  - assume $T(k) = f(k)$
- Inductive step: $T(k+1) = T(k) + c_1$

$$= (k-1) \cdot c_1 + c_0 + c_1$$

$$= k \cdot c_1 + c_0 \ = f(k+1)$$

Induction Example #2

P(n): $A(n) = \sum_{i=1}^{n} 2i$ is equal to $f(n) = n \cdot (n+1)$

- Base case: **`n = 1`**
  - $A(1) = 2$ and $f(1) = 1 \cdot (1+1) = 2$
- Inductive assumption: **`n=k`**
  - $\sum_{i=1}^{k} 2i = k \cdot (k+1)$
- Inductive step

$$A(k+1) = \sum_{i=1}^{k+1} 2i$$

$$= \sum_{i=1}^{k} 2i + 2 \cdot (k+1)$$

$$\ldots = k \cdot (k+1) + 2 \cdot (k+1)$$

$$= (k+1) \cdot (k+2)$$

$$= f(k+1)$$

Another Induction Example

P(n): $A(n) = \sum_{i=1}^{n} i$ is equal to $f(n) = \frac{n \cdot (n+1)}{2}$

- Prove base case: **`n=1`**
  - $A(1) = 1$ and $f(1) = \frac{1 \cdot (1+1)}{2} = 1$
- Induction *assumption*: **`n=k`**
  - $A(k) = f(k)$ which means $\sum_{i=1}^{k} i = \frac{k \cdot (k+1)}{2}$
- Prove induction step!

Another Induction Example

- Prove induction step

$$
\begin{aligned}
A(k+1) &= \sum_{i=1}^{k+1} i \\
&= \sum_{i=1}^{k} i + (k+1) \\
&= \frac{k \cdot (k+1)}{2} + (k+1) \\
&= \frac{k \cdot (k+1)}{2} + \frac{2 \cdot (k+1)}{2} \qquad \times \frac{2}{2} \\
&= \frac{(k+1) \cdot (k+2)}{2} \qquad \text{factor out } (k+1) \\
&= f(k+1)
\end{aligned}
$$

<u>Induction assumption</u>

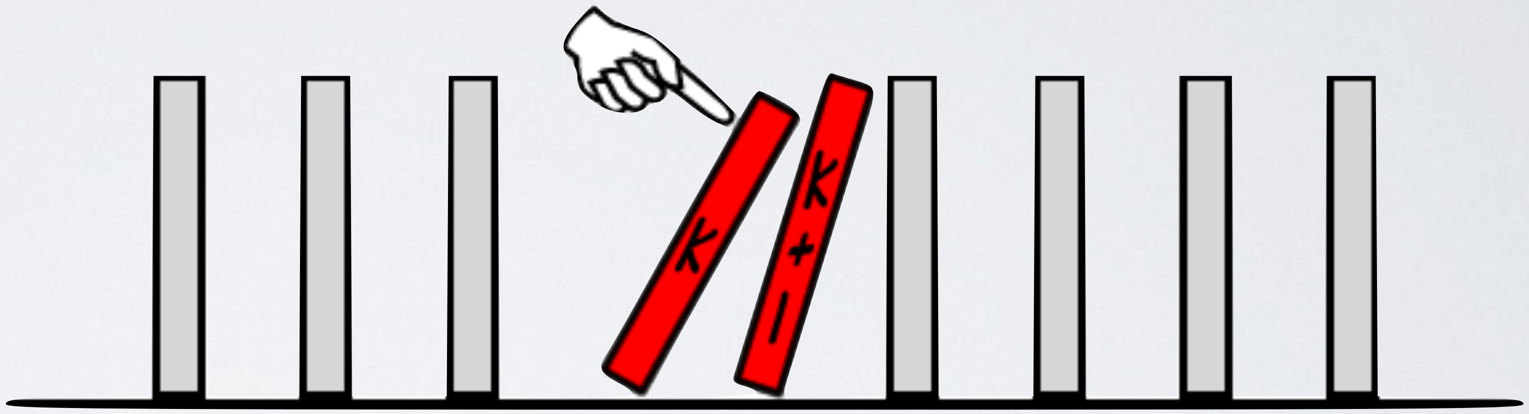
$$\sum_{i=1}^{k} i = \frac{k \cdot (k+1)}{2}$$

Strong vs. Weak Induction

- Weak induction
  - induction step assumes statement is true for `n=k` and
  - proves statement is true for `n=k+1`
- Strong induction
  - induction step assumes statement is true for `n=1,2,…,k`
  - and proves true for `n=k+1`
- Strong vs. weak refers to *assumption*
  - not strength of proof

Strong vs. Weak Induction

Weak:



Strong:

?

Dynamic programming

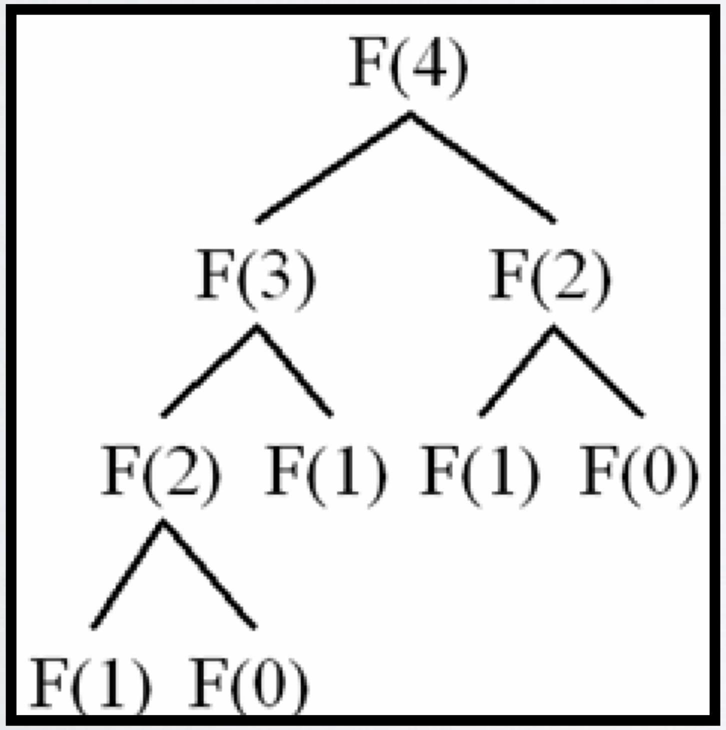
Factorial, again

```
def factorial(n):
    if n == 1:
        return 1
    else:
        return n * factorial(n-1)
```

- $T(1) = c0$
- $T(n) = c1 + T(n-1)$
- What's the big-O runtime? $\boxed{O(n)}$

Fibonacci

- Defined recursively
  - $F_0 = 0,\ F_1 = 1$
  - $F_n = F_{n-1}+F_{n-2}$



0,1,1,2,3,5,8,13,21,34,…

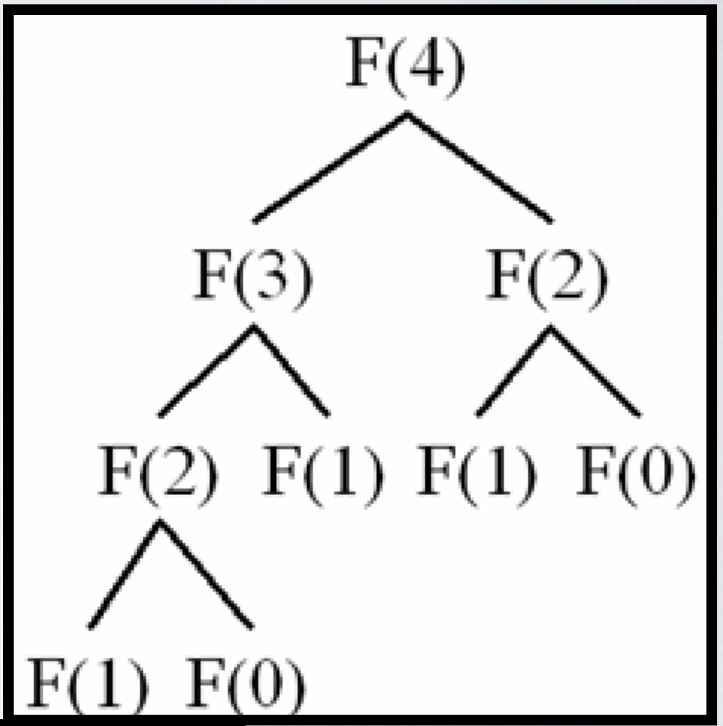
Fibonacci (Recursive)

```
function fib(n):
    if n = 0:
        return 0
    if n = 1:
        return 1
    return fib(n-1) + fib(n-2)
```

- $T(0) = c0$
- $T(1) = c1$
- $T(n) = c2 + T(n-1) + T(n-2)$
- What's the big-O runtime?

Fibonacci (Recursive)

```
function fib(n):
   if n = 0:
      return 0
   if n = 1:
      return 1
   return fib(n-1) + fib(n-2)
```



- How many time
  - 8 times
- At each level it r
  - For `fib(n)`
  - Algorithm is $O(2^n)$

On my computer, computing the 60th Fibonacci number takes ~2 days

Computing 60! is ~instantaneous

Dynamic programming to the rescue!

What is Dynamic Programming?

- Algorithm design paradigm/framework
  - Design efficient algorithms for optimization problems
- Optimization problems
  - "find the *best* solution to problem **`X`**"
  - "what is the *shortest* path between **`u`** and **`v`** in **`G`**"
  - "what is the *minimum* spanning tree in **`G`**"
- Can also be used for non-optimization problems

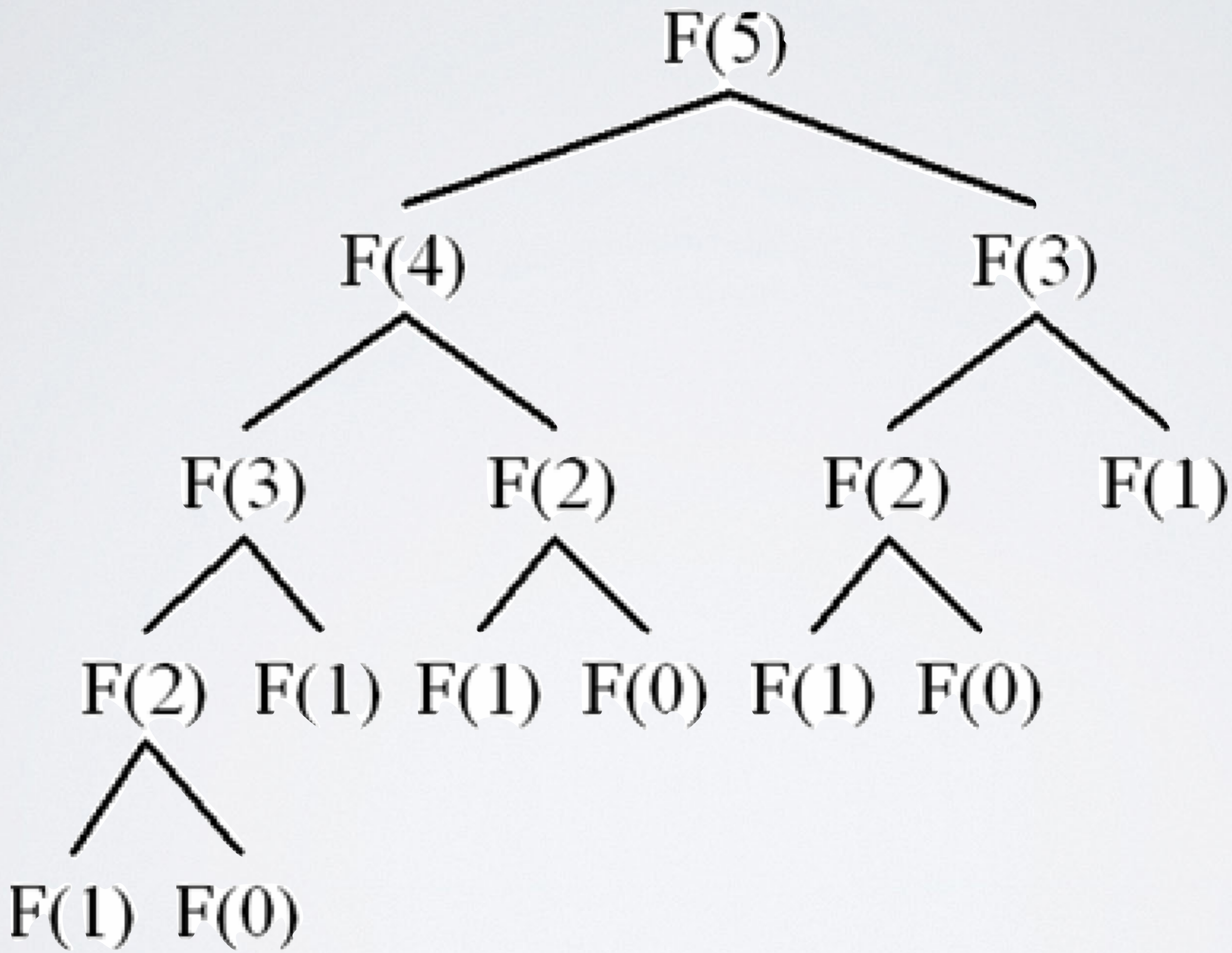
When is Dynamic Programming Applicable?

- Condition **#1**: sub-problems
  - The problem can be solved recursively
  - Can be solved by solving sub-problems
- Condition **#2**: *overlapping* sub-problems
  - Same sub-problems need to be solved many times
- Core idea
  - solve each sub-problem once and store the solution
  - use stored solution when you need to solve sub-problem again

Steps to Solving a Problem w/ DP

- What are the **sub-problems**?
- What is the "**magic**" step?
  - Given solutions to sub-problems…
  - …how do I combine them to get solution to the problem?
- In which **order** should I solve sub-problems?
  - so that solutions to sub-problems are available when I need them
- Design iterative **algorithm**
  - that solves sub-problems in right order and stores their solution

Fibonacci (Dynamic Programming)



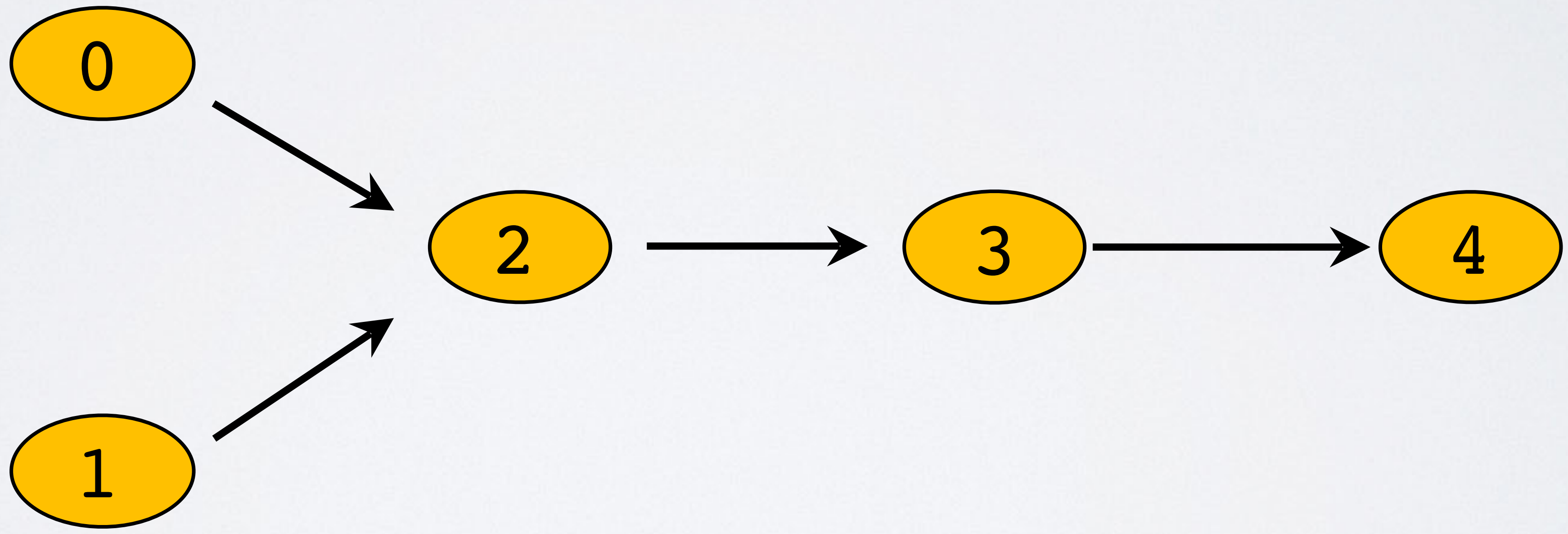
Fibonacci (Dynamic Programming)

- Given `n` compute
  - Fib(`n`) = Fib(`n-1`)+Fib(`n-2`)
  - with base cases Fib(`0`) = `0` and Fib(`1`) = `1`
- What are the **sub-problems**?
  - Fib(`n-1`), Fib(`n-2`), …, Fib(`1`), Fib(`0`)
- What is the **magic** step?
  - Fib(`n`) = Fib(`n-1`)**+**Fib(`n-2`)

Magic step is usually not provided!!

Fibonacci (Dynamic Programming)

- In which order should I solve sub-problems?
  - Fib(`0`), Fib(`1`), …,Fib(`n-1`), Fib(`n`)



Fibonacci (Dynamic Programming)

- Design iterative **algorithm**

```
function Fib(n):
   fibs = []
   fibs[0] = 0
   fibs[1] = 1

   for i from 2 to n:
       fibs[i] = fibs[i-1] + fibs[i-2]

   return fibs[n]
```

Fibonacci (Dynamic Programming)

- What's the runtime of **`dynamicFib( )`**?
  - Calculates Fibonacci numbers from **`0`** to **`n`**
  - Performs **`O(1)`** ops for each one
  - Runtime is **`O(n)`**
- We reduced runtime of algorithm
  - From exponential to linear
  - with dynamic programming!

Seams

Finding Low Importance Seams

- **Idea:** remove **seams** not columns
  - (vertical) seam is a path from top to bottom
  - that moves left or right by at most one pixel per row

Finding Low Importance Seams

- How many seams in a $c \times r$ image?
  - At each row the seam can go Left, Right or Down
  - It chooses 1 out of 3 dirs at all but last row r
  - So about 3^{r-1} seams from some starting pixel
  - There are c starting pixels so total number of seams is
    - about $c \times 3^{r-1}$
- For square $n x n$ image
  - there are about $n3^{n-1}$ possible seams

Finding Low Importance Seams

- Brute force algorithm
  - Try every possible seam & find least important one
- What is running time of brute force algorithm?
  - If image is **`nxn`** brute force takes about $n3^{n-1}$
  - So brute force is $\Omega(2^n)$ (i.e., exponential)

Seamcarve

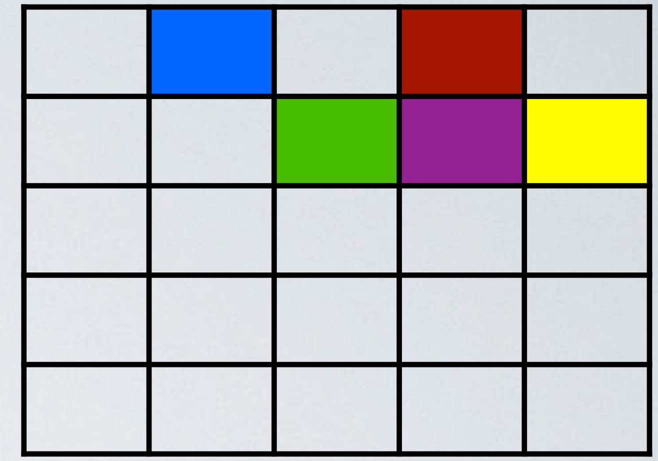
- What is the runtime of Seamcarve?
- The algorithm
  - Iterate over all pixels from bottom to top
  - Populate **`costs`** and **`dirs`** arrays
  - Create seam by choosing minimum value in top row and tracing downward
- How many operations per pixel?
  - A constant number of operations per pixel (`4`)
- Constant number of operations per pixel means algorithm is linear
  - **`O(n)`** where **`n`** is number of pixels

Seamcarve

- How can we possibly go from
  - exponential running time with brute force
  - to linear running time with Seamcarve?
  - What is the secret to this magic trick?

Dynamic Programming!

Designing Seamcarve

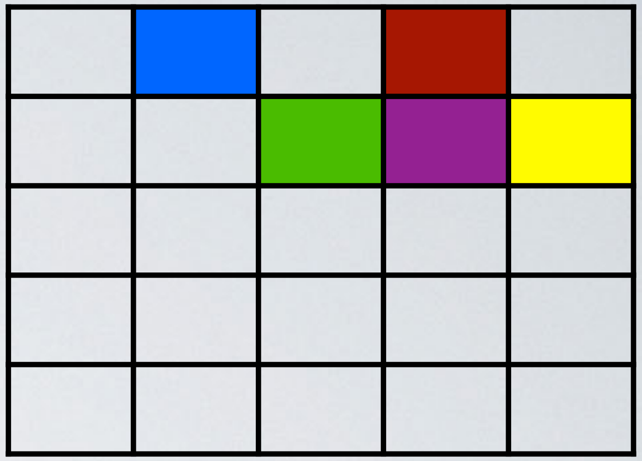


- What are the subproblems?
  - lowest cost seam (LCS) starting at is

     || min(LCS( ), LCS( ), LCS( ))

  - Are they overlapping?
    - Yes!
    - ex: LCS( ) is subproblem of LCS( ) and LCS( )

Designing Seamcarve



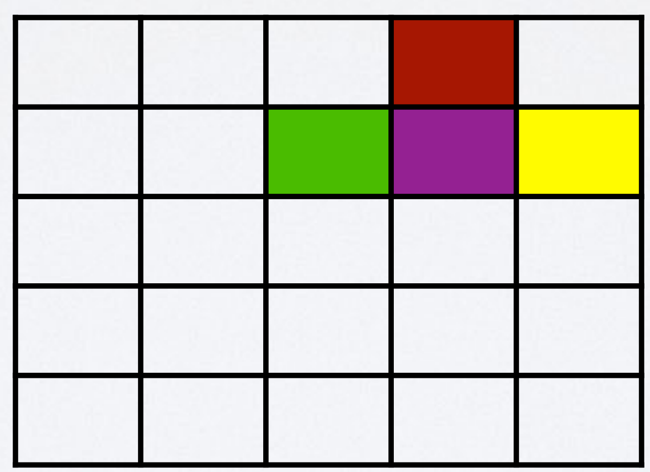
- What is the magic step?

  ‖ min(LCS( ), LCS( ), LCS( ))

- Which order should I use?
  - to solve LCS problem at cell `(i,j)`
  - we need to have solved problem at cells below

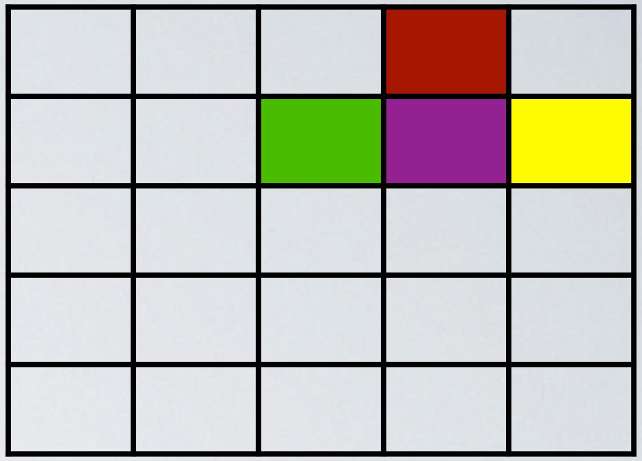
Designing Seamcarve

- Algorithm
  - compute *cost* of LCS for each cell going bottom up
  - store *cost* of LCS in an auxiliary 2D array…
  - …so we can reuse them



Cost( )=Val( )+min(Cost( ), Cost( ), Cost( ))

Designing Seamcarve



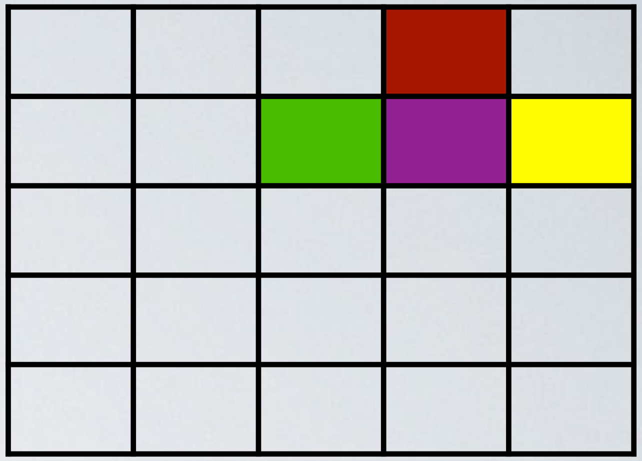
- Problem
  - Costs array only gives us *cost* of LCS at cell
  - We need the seam. What happened?
  - We used

    Cost( )=Val( )+min(Cost( ), Cost( ), Cost( ))

  - But recall that at "seam level" we had

    LCS( )= $\|$ min(LCS( ), LCS( ), LCS( ))

Designing Seamcarve



- It's OK!
  - We can keep track of minimum LCS
  - at each step in auxiliary structure Dirs

Readings

- Induction handout on course page
  - http://cs.brown.edu/courses/cs016/static/files/docs/induction.pdf