

Dynamic Programming

CS16: Introduction to Data Structures & Algorithms
Spring 2019

Outline

- ▶ Dynamic Programming
- ▶ Examples
 - ▶ Fibonacci
 - ▶ Seamcarve



This is
Hard!

**You will understand
it...**

29
Um

The element of
CONFUSION

*Struggle
is a
prelude
to
success.*



Fibonacci

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

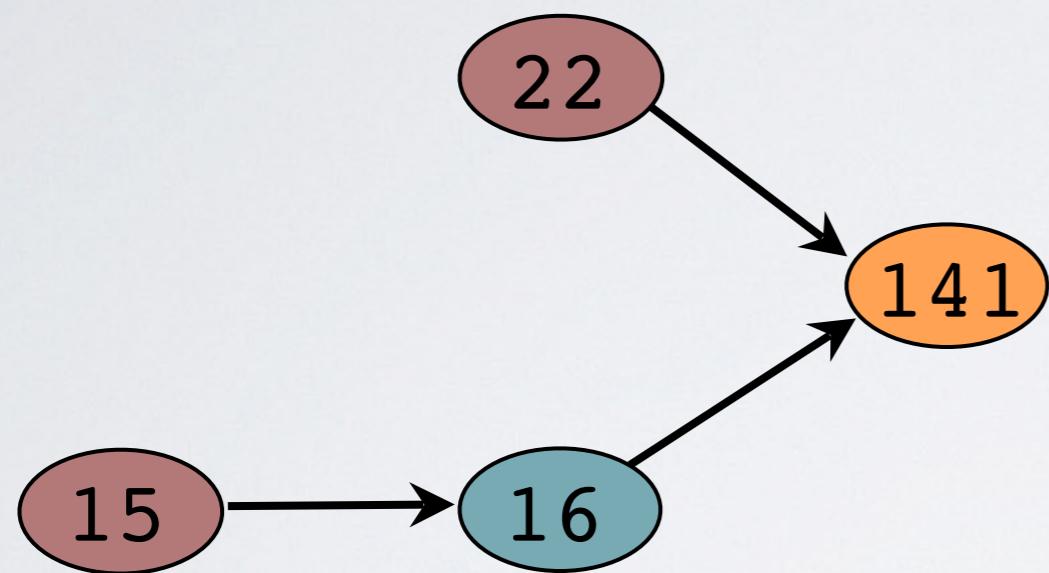
When is Dynamic Programming Applicable?

- ▶ 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

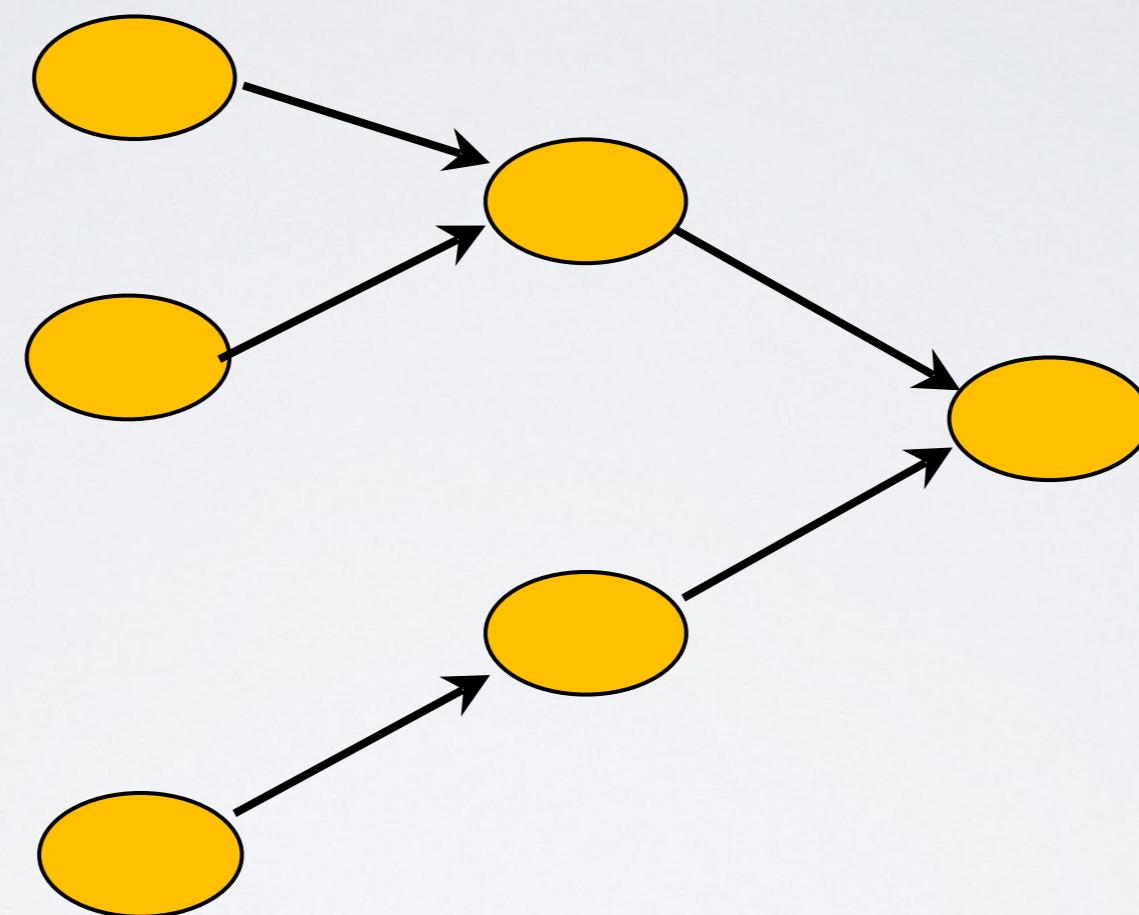
- ▶ What are the **sub-problems**?
- ▶ What is the “**magic**” step?
 - ▶ Given solution to a sub-problem...
 - ▶ ...how do I get solution to the problem?
- ▶ Which **topological order** on sub-problems can I use?
 - ▶ so that solutions to sub-problems available when needed
- ▶ Design iterative **algorithm**
 - ▶ that solves sub-problems in order and stores their solution

Topological Order?



- ▶ Valid topological orderings
 - ▶ 15, 16, 22, 141
 - ▶ 22, 15, 16, 141
 - ▶ 15, 22, 16, 141

Topological Order on Sub-Problems!



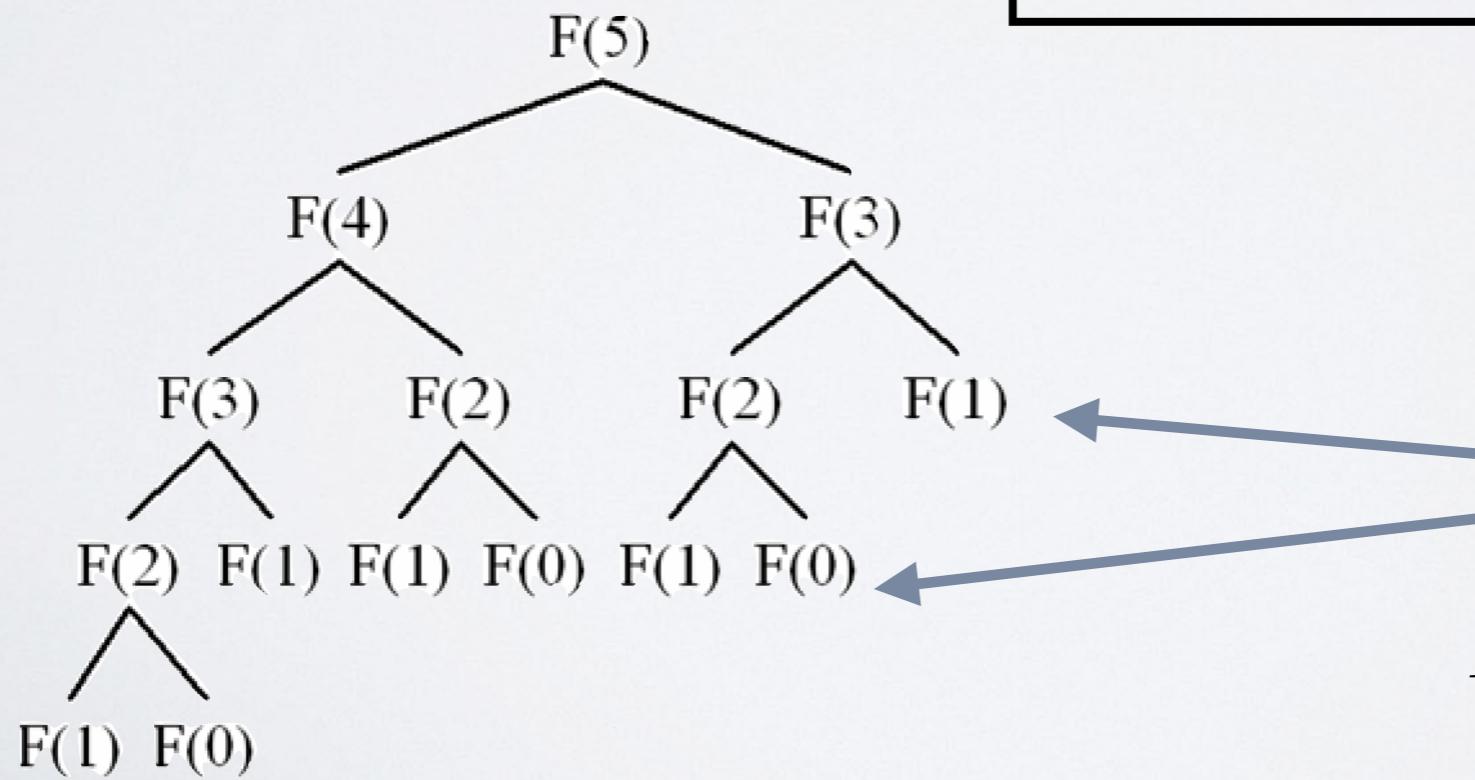
Fibonacci

Fibonacci



0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

$$F(n) = F(n - 1) + F(n - 2)$$

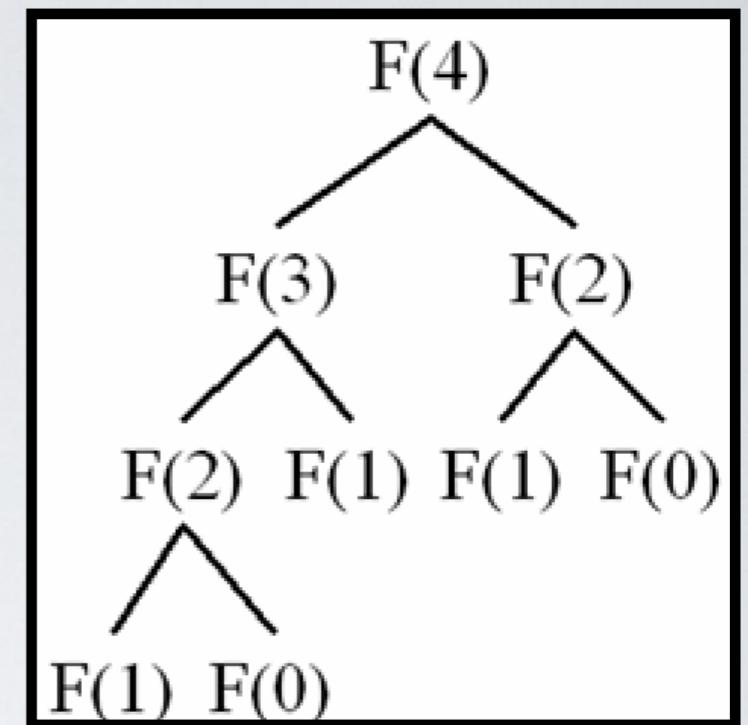


base cases:

$$F(0) = 1 \text{ & } F(1) = 1$$

Fibonacci (Recursive)

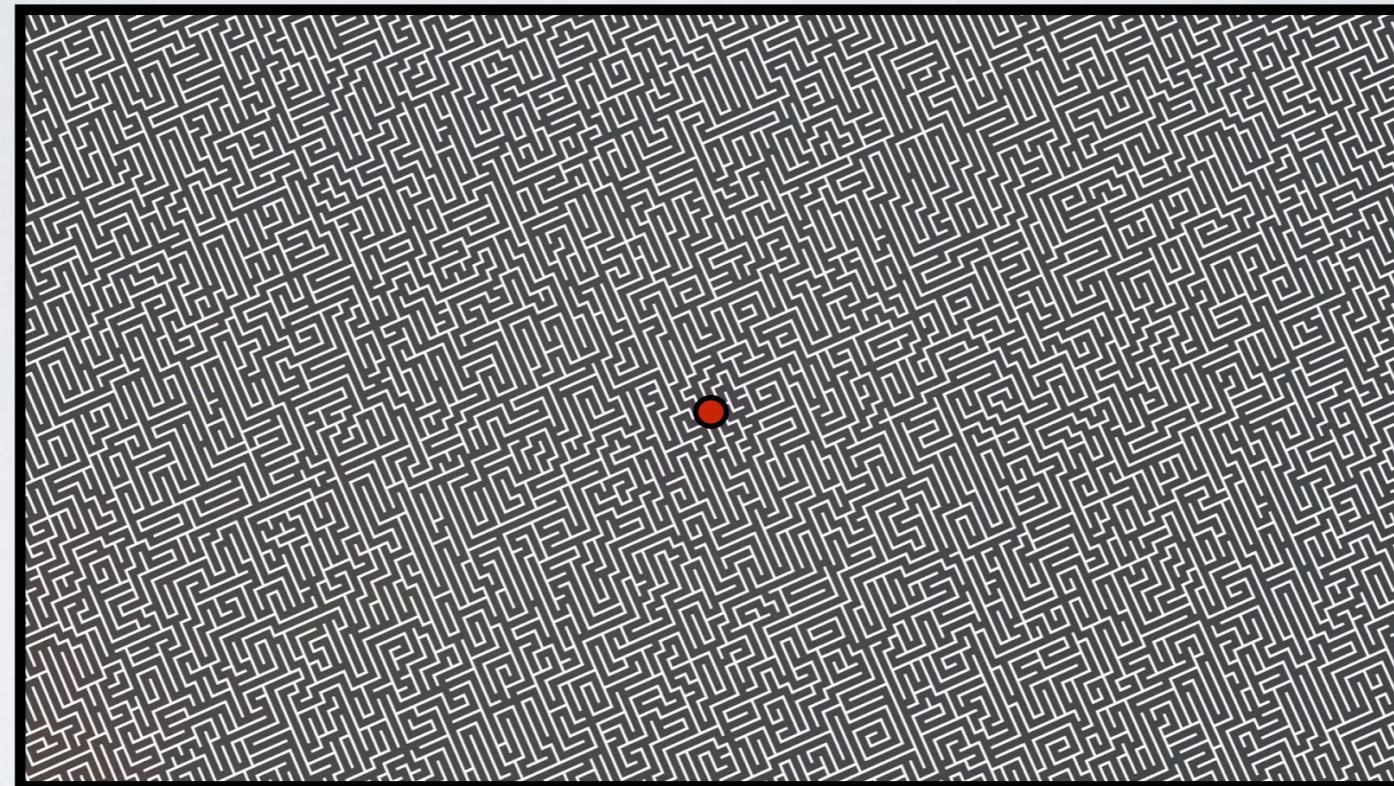
- ▶ Defined by the recursive relation
 - ▶ $F_0 = 0, F_1 = 1$
 - ▶ $F_n = F_{n-1} + F_{n-2}$
- ▶ We can implement this with a recursive function



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

Q: Is any work happening?

Recursion & Clones



- ▶ At each intersection
 - ▶ clone yourself twice and send one Left and one Right
 - ▶ wait for clones to report a path to exit (if it exists) and its length
 - ▶ pick direction that gets you to exit the fastest

Fibonacci (Recursive)

Big-O runtime of recursive `fib` function?

1 min

Activity #1

Fibonacci (Recursive)

Big-O runtime of recursive **fib** function?

1 min

Activity #1

Fibonacci (Recursive)

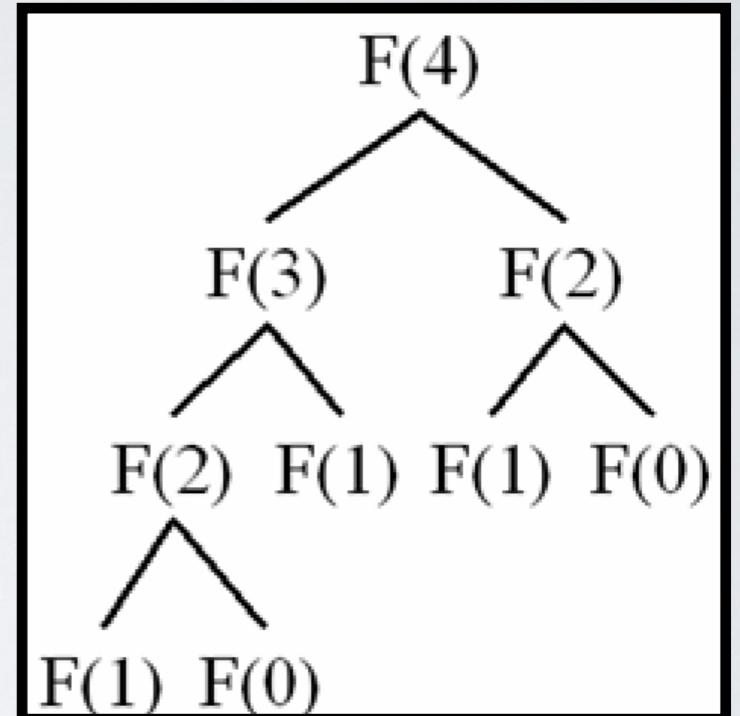
Big-O runtime of recursive **fib** function?

O min

Activity #1

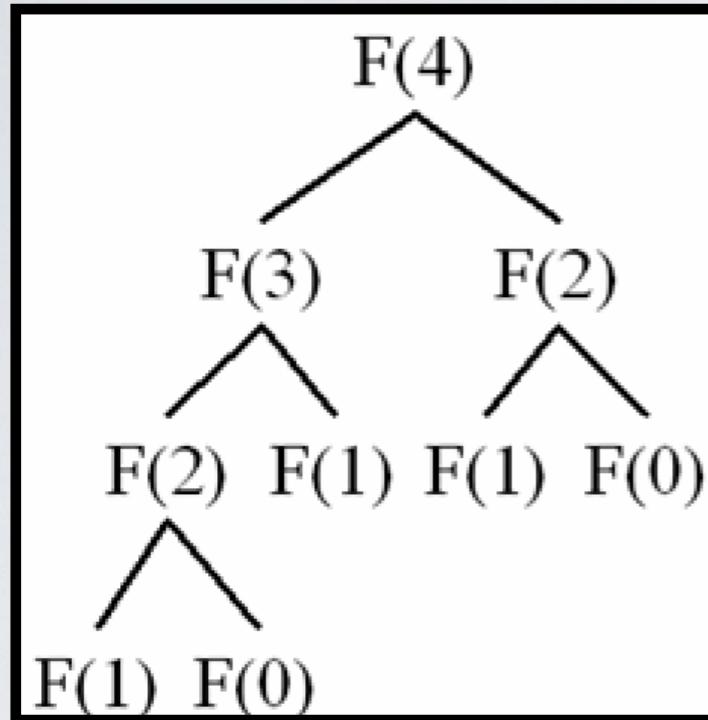
Fibonacci (Recursive)

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



- ▶ How many times does **fib** get called for **fib(4)**?
 - ▶ 8 times
- ▶ At each level it makes twice as many recursive calls as last
 - ▶ For **fib(n)** it makes approximately 2^n recursive calls
 - ▶ Algorithm is $O(2^n)$

Fibonacci (Recursive)



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

- ▶ How many times does `fib(1)` get computed?
- ▶ Instead of recomputing Fibonacci numbers over and over again
- ▶ Compute them **once** and store them for later

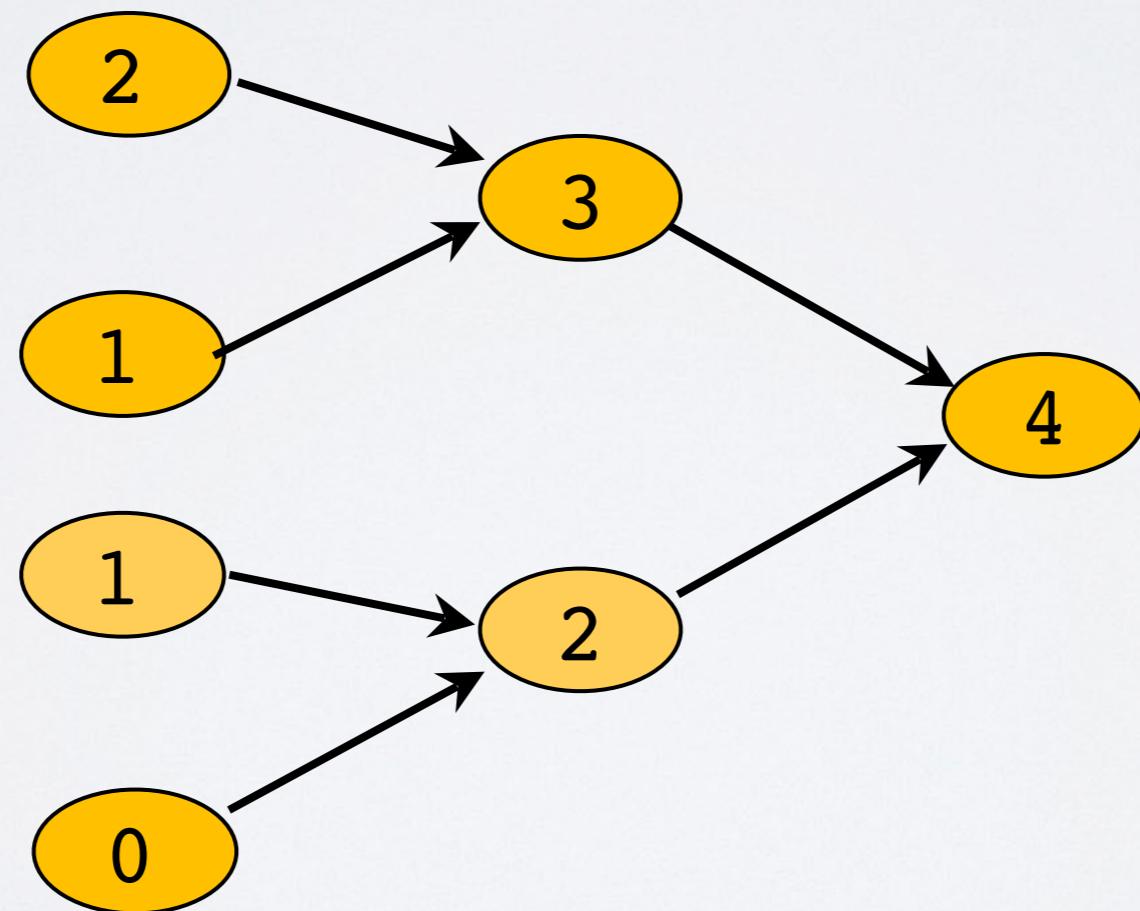
Fibonacci (Dynamic Programming)

- ▶ Given n compute
 - ▶ $\text{Fib}(n) = \text{Fib}(n-1) + \text{Fib}(n-2)$
 - ▶ with base cases $\text{Fib}(0) = 0$ and $\text{Fib}(1) = 1$
- ▶ What are the **sub-problems**?
 - ▶ $\text{Fib}(n-1), \text{Fib}(n-2), \dots, \text{Fib}(1), \text{Fib}(0)$
- ▶ What is the **magic** step?
 - ▶ $\text{Fib}(n) = \text{Fib}(n-1) + \text{Fib}(n-2)$

Magic step is
usually not
provided!!

Fibonacci (Dynamic Programming)

- ▶ Which **topological order** should I use?
- ▶ $\text{Fib}(0), \text{Fib}(1), \dots, \text{Fib}(n-1), \text{Fib}(n)$



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 clearly $O(n)$
- ▶ 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 $n \times n$ image
 - ▶ there are about $n^{3^{n-1}}$ 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 $n \times n$ brute force takes about n^{3n-1}
 - ▶ So brute force is $\Omega(2^n)$ (i.e., exponential)

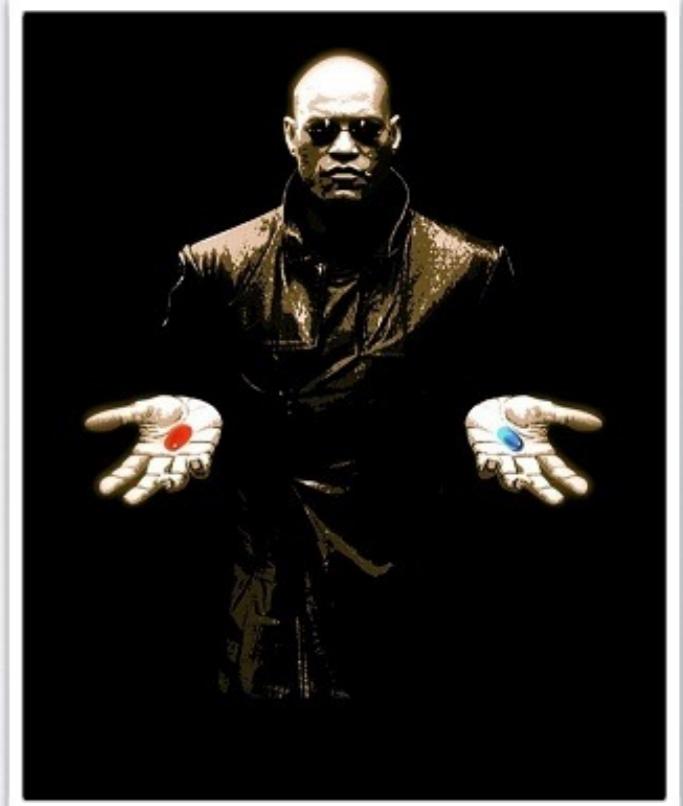
Seamcarve

- ▶ 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
- ▶ Also could have counted # of nested loops in pseudocode...

Seamcarve

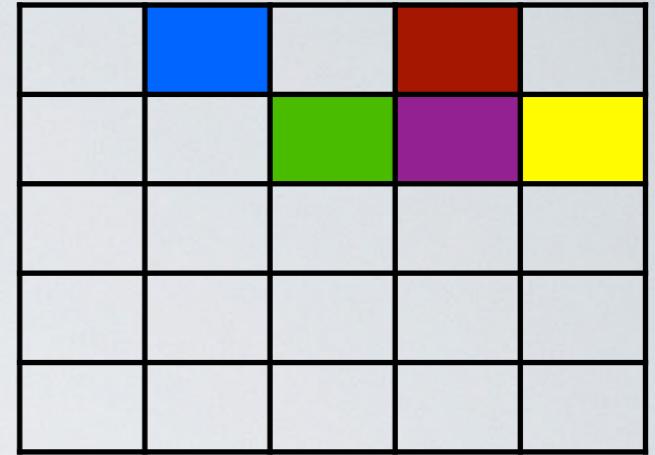
- ▶ How can we possibly go from
 - ▶ exponential running time with brute force
 - ▶ to linear running time with Seamcarve?
 - ▶ What is the secret to this magic trick?

**Dynamic
Programming!**



Designing Seamcarve

- ▶ What are the subproblems?
 - ▶ lowest cost seam (LCS) starting at  is
 || $\min(\text{LCS}(\text{green}), \text{LCS}(\text{purple}), \text{LCS}(\text{yellow}))$
- ▶ Are they overlapping?
 - ▶ Yes!
 - ▶ ex: $\text{LCS}(\text{green})$ is subproblem of $\text{LCS}(\text{blue})$ and $\text{LCS}(\text{red})$

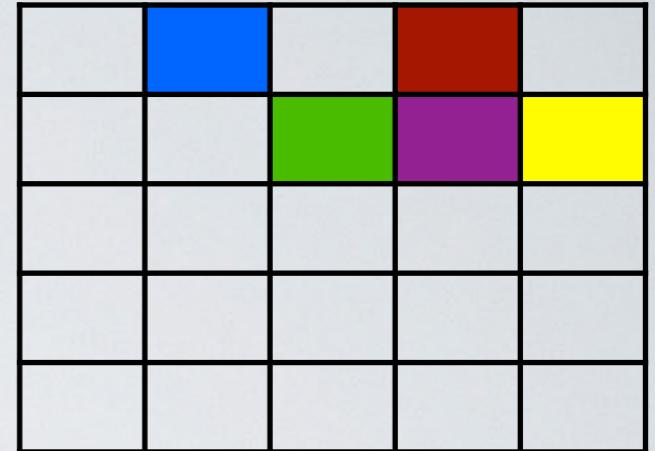


Designing Seamcarve

- ▶ What is the magic step?

 || $\min(\text{LCS}(\text{Green}), \text{LCS}(\text{Purple}), \text{LCS}(\text{Yellow}))$

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

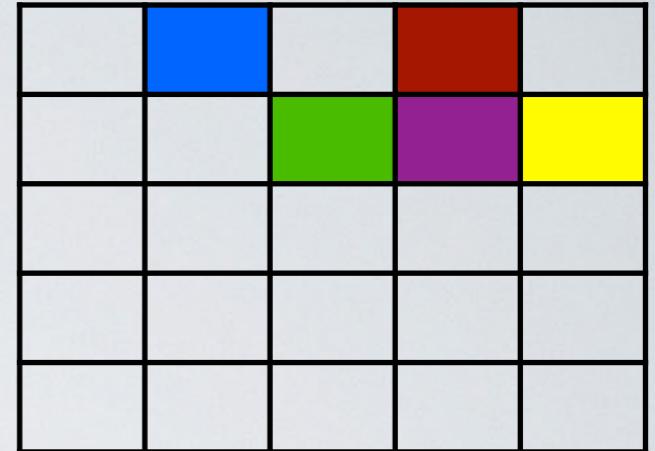


Designing Seamcarve

- ▶ What is the magic step?

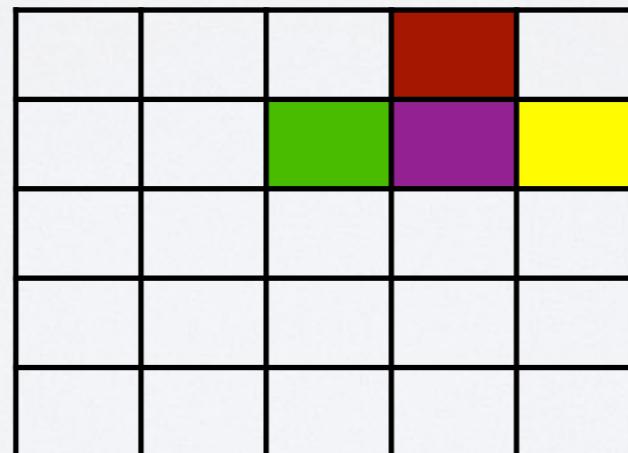
 || $\min(\text{LCS}(\text{Green}), \text{LCS}(\text{Purple}), \text{LCS}(\text{Yellow}))$

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



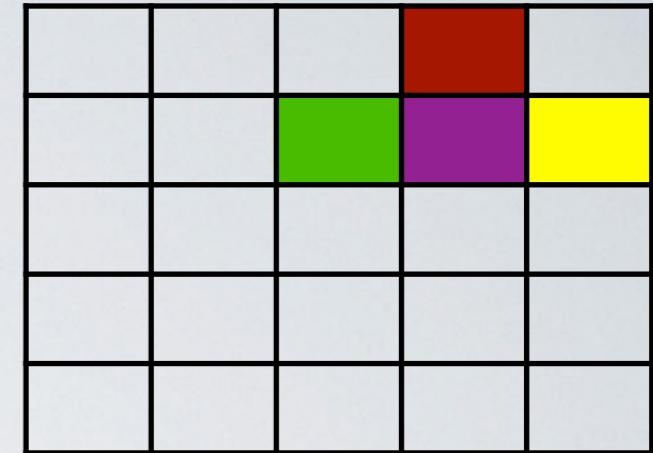
Designing Seamcarve

- ▶ 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



$$\text{Cost}(\blacksquare) = \text{Val}(\blacksquare) + \min(\text{Cost}(\blacksquare), \text{Cost}(\blacksquare), \text{Cost}(\blacksquare))$$

Designing Seamcarve



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

$$\text{Cost}(\blacksquare) = \text{Val}(\blacksquare) + \min(\text{Cost}(\blacksquare), \text{Cost}(\blacksquare), \text{Cost}(\blacksquare))$$

- ▶ But recall that at “seam level” we had

$$\text{LCS}(\blacksquare) = \blacksquare \parallel \min(\text{LCS}(\blacksquare), \text{LCS}(\blacksquare), \text{LCS}(\blacksquare))$$

Designing Seamcarve

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

| | | | | |
|--|--|-------|--------|--------|
| | | | | |
| | | | Red | |
| | | Green | Purple | Yellow |
| | | | | |
| | | | | |