Dynamic Programming

CS16: Introduction to Data Structures & Algorithms

Spring 2020

Outline

- Dynamic Programming
- Examples
  - Fibonacci
  - Seamcarve

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

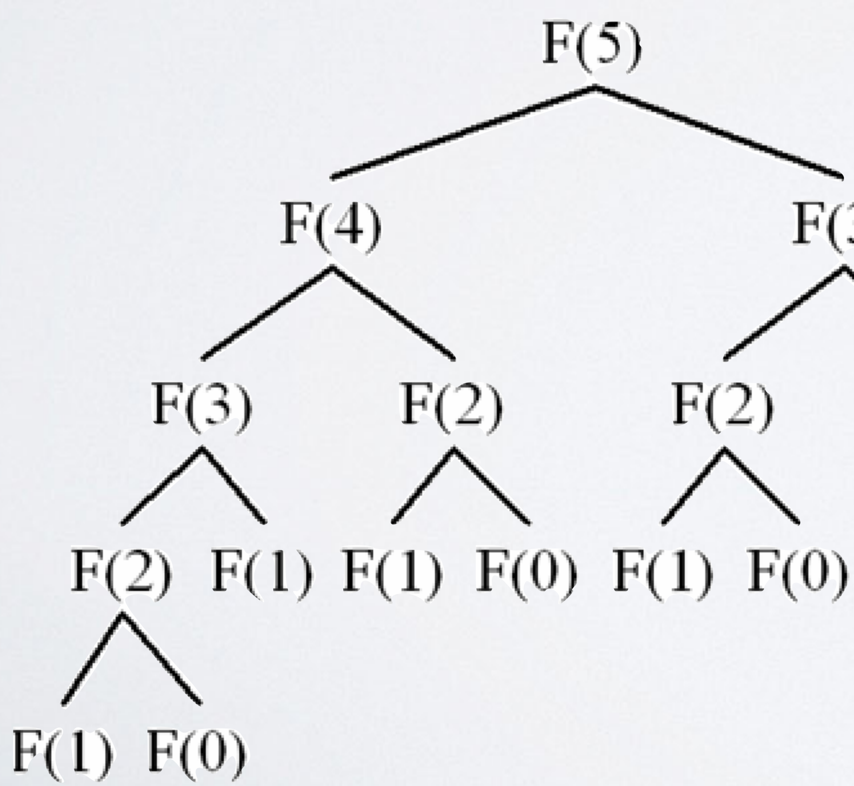
- What are the **sub-problems**?
- What is the "**magic**" step?
  - Given solutions to sub-problems…
  - …how do I combine them to get solution to the problem?
- In which **order** should I solve sub-problems?
  - so that solutions to sub-problems are available when I need them
- Design iterative **algorithm**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

Fibonacci

Fibonacci

`0,1,1,2,3,5,8,13,21,34,…`

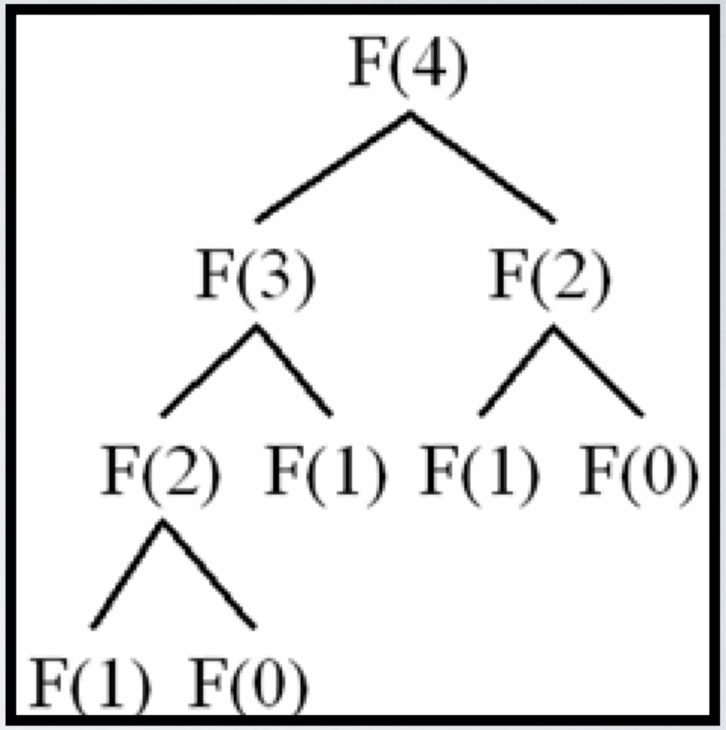
$$F(n) = F(n-1) + F(n-2)$$



base cases:

$$F(0) = 0 \ \& \ F(1) = 1$$

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 recursively

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

Fibonacci (Recursive)

Big-O runtime of recursive `fib` function?

Activity #1

1 min

Fibonacci (Recursive)

Big-O runtime of recursive `fib` function?

Activity #1

1 min

Fibonacci (Recursive)

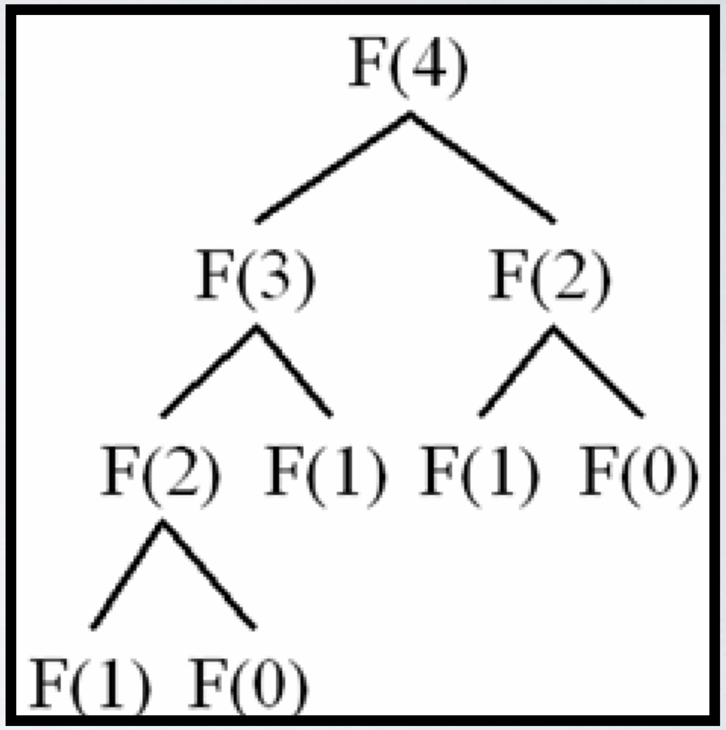
Big-O runtime of recursive `fib` function?

Activity #1

0 min

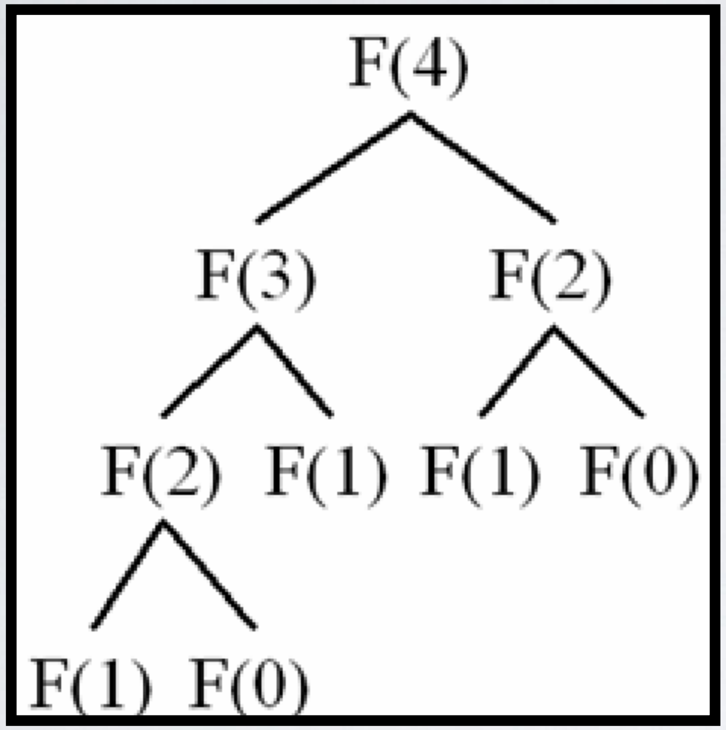
Fibonacci (Recursive)

```
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

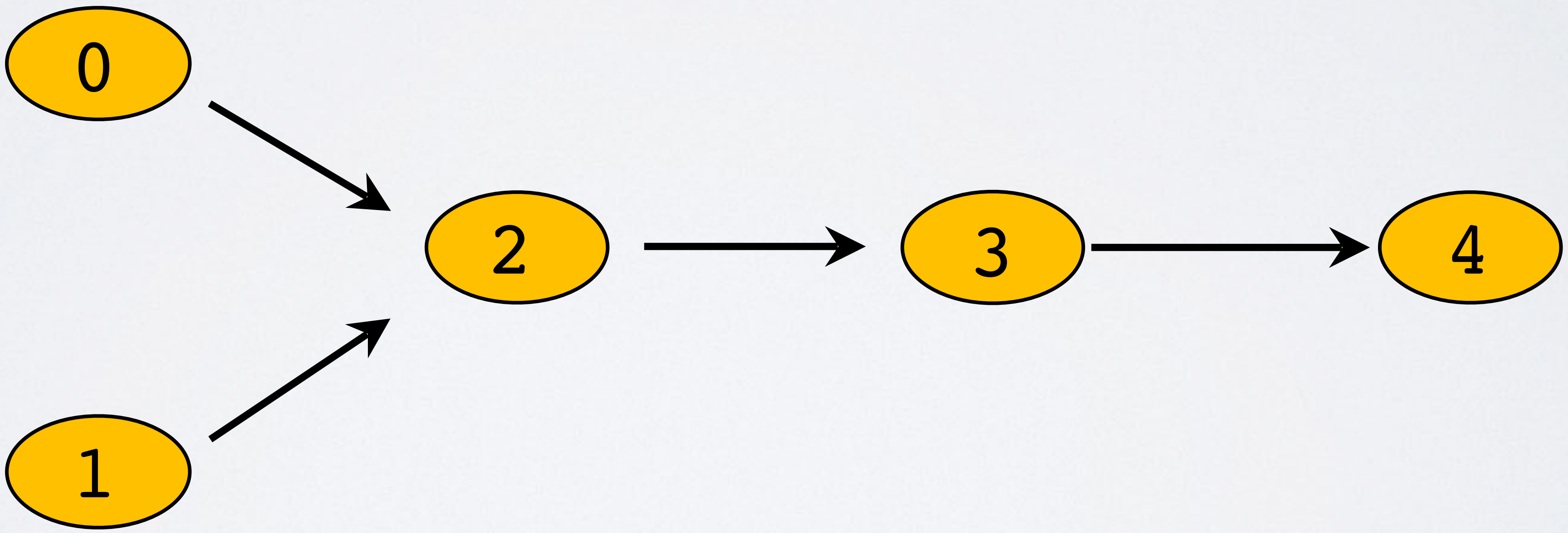
Fibonacci (Dynamic Programming)

- Given `n` compute
  - 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 **nxn** 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 $\mathbf{nxn}$ brute force takes about $\mathtt{n3^{n-1}}$
  - So brute force is $\Omega(2^n)$ (i.e., exponential)

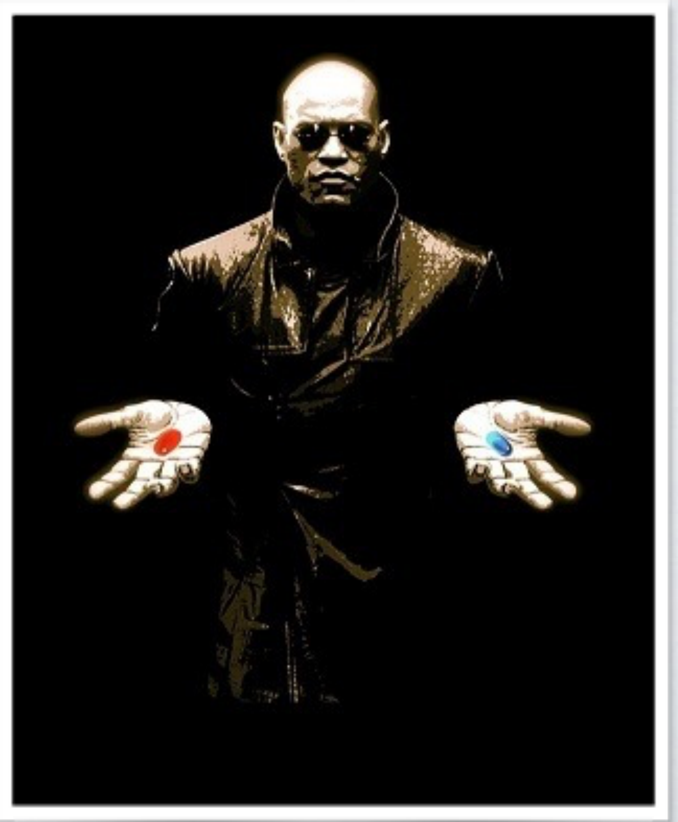
Seamcarve

- 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
- Also could have counted # of nested loops in pseudocode…

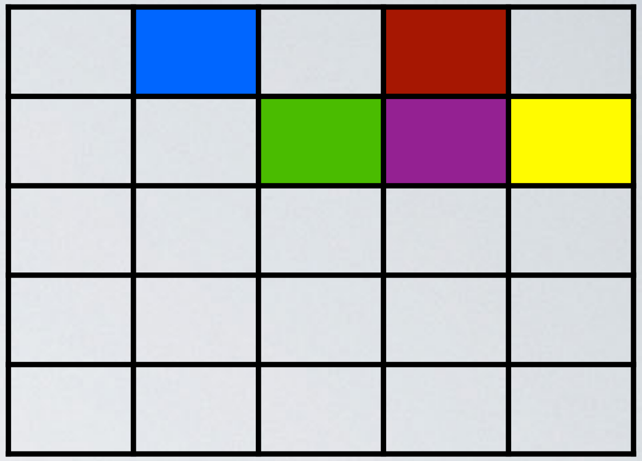
Seamcarve

- 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

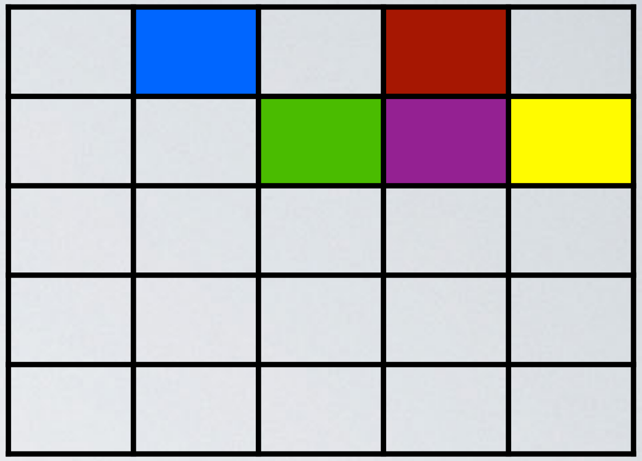


- What are the subproblems?
  - lowest cost seam (LCS) starting at is

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

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

Designing Seamcarve

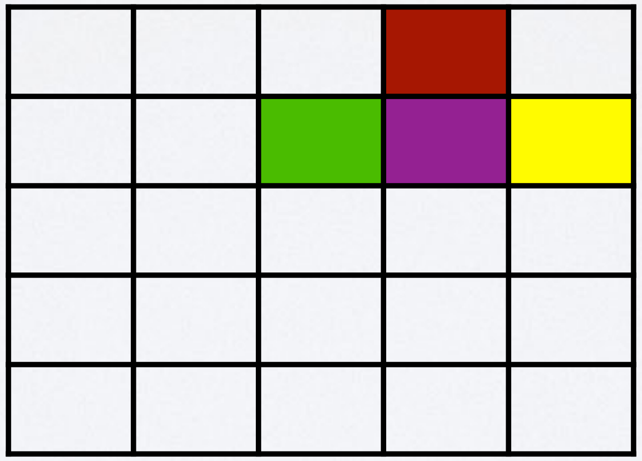


- What is the magic step?

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

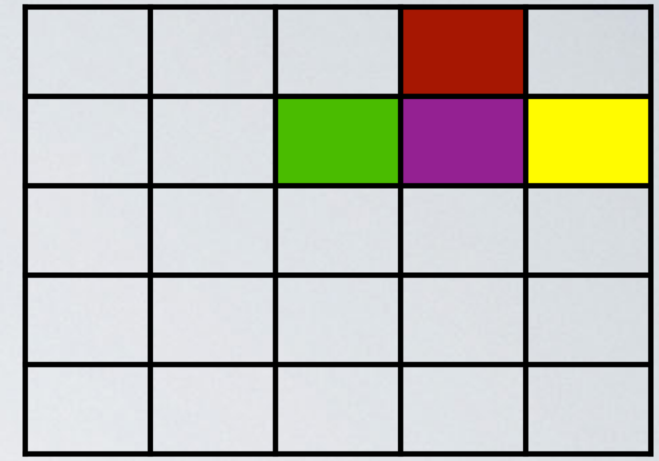
Designing Seamcarve

- 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



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

Designing Seamcarve



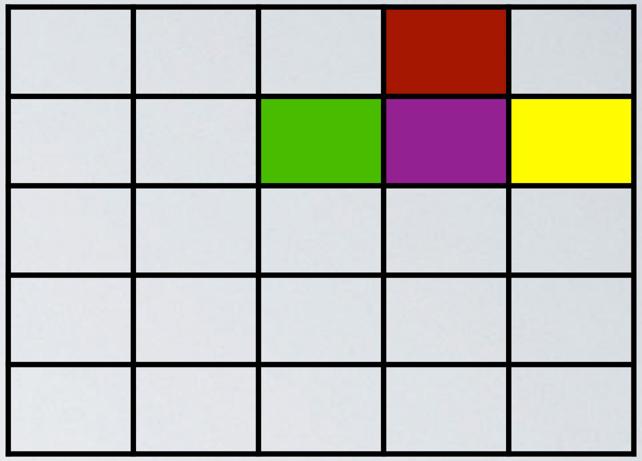
- Problem
  - Costs array only gives us *cost* of LCS at cell
  - We need the seam. What happened?
  - We used

    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