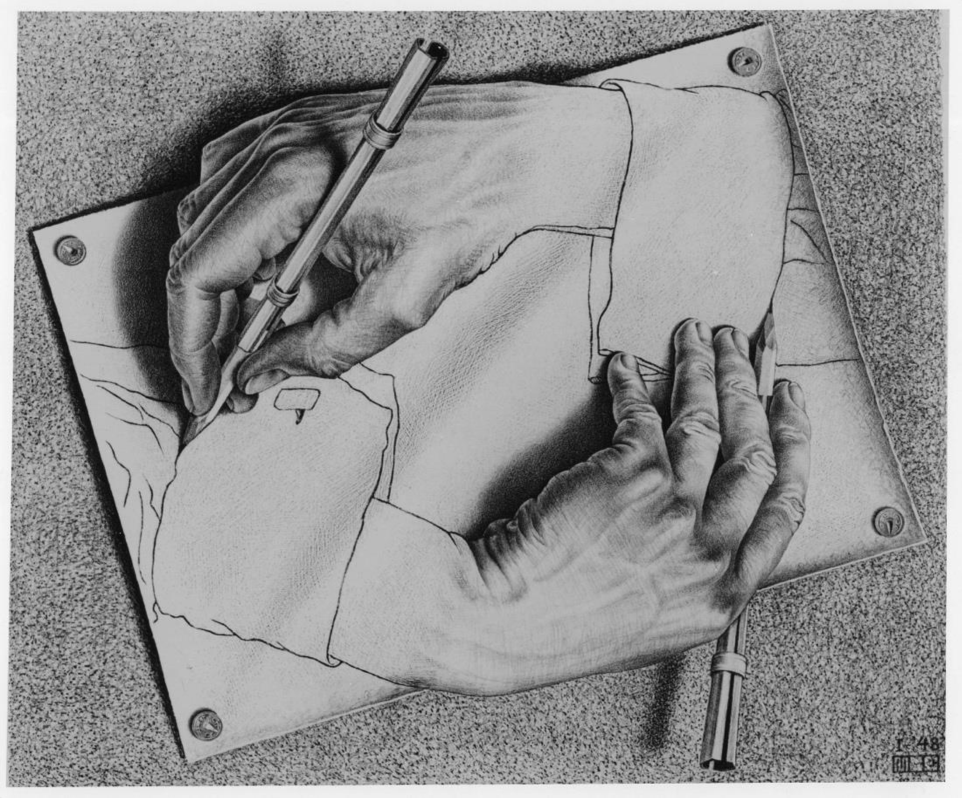
Computational Structures in Data Science

Recursion

M. C. Escher : Drawing Hands



Announcement

- Midterm on 3/11
  - Please fill out accommodations / alternates from by TONIGHT
  - https://edstem.org/us/courses/94478/discussion/7722086
- Reminders:
  - You get 1 reference sheet we provide
  - You get your own (hand written) sheet
    - We will collect these!
  - Scratch paper if necessary (lots of blank space on the exam)
- My Tea Hours will change this week (see calendar)

The Recursive Process

Recursive solutions involve two major parts:

- Base case(s), the problem is simple enough to be solved directly
- Recursive case(s). A recursive case has three components:
  - **Divide** the problem into one or more simpler or smaller parts
  - **Invoke** the function (recursively) on each part, and
  - **Combine** the solutions of the parts into a solution for the problem.

Why learn recursion?

- Recursive data is all around us!
  - Take CS61B (data structures), CS70 (discrete math), CS164 (Programming Languages), Data 101 (Data Eng) for more examples where you'll encounter recursion
- Trees (post-midterm) and Graphs are structures which are recursive in nature.
  - E.g. A social network is a graph of friends with connections to other friends, with connections to other friends.
  - Analyzing "chains" of data, can benefit from recursion
- Next Lecture: Problems that "branch" out:
  - generating subsets and permutations
  - calculating Fibonacci numbers

Computational Structures in Data Science

Palindromes

Learning Objectives

- Compare Recursion and Iteration to each other
  - Translate some simple functions from one method to another
- Write a recursive function
  - Understand the base case and a recursive case

Fun Palindromes

- C88C
- racecar
- LOL
- radar
- a man a plan a canal panama
- aibohphobia 😈
  - The fear of palindromes.
- https://czechtheworld.com/best-palindromes/#palindrome-words

Palindromes

- Palindromes are the same word forwards and backwards.
- Python has some tricks, but how could we build this?
  - `palindrome = lambda w: w == w[::-1]`
  - `[::-1]` is a slicing shortcut `[0:len(w):-1]` to reverse items.
- Let's write Reverse:

```
def reverse(s):
    result = ''
    for letter in s:
        result = letter + result
    return result
```

```
def reverse_while(s):
    """
    >>> reverse_while('hello')
    'olleh'
    """
    result = ''
    while s:
        first = s[0]
        s = s[1:] # remove the first letter
        result = first + result
    return result
```

Writing Reverse Recursively

```
def reverse(s):
    if not s:
        return ''
    return 'TODO'

def palindrome(word):
    return word == reverse(word)
```

How should reverse work?

- Our algorithm in words:
  - Take the first letter, put it at the end
  - The beginning of the string is the reverse of the rest.

```
reverse('ABC')
→ reverse('BC') + 'A'
→ reverse('C') + 'B' + 'A
→ 'C' + 'B' + 'A
→  'CBA'
```

reverse recursive

```
def reverse(s):
    if not s:
        return ''
    return
```

Recursive Case

```
def palindrome(word):
    return word == reverse(word)
```

Palindrome – Alternative Approaches

- Compare first / last letters, working our way towards the middle
- Base Case?
  - What is the *smallest* word that is a palindrome?
    - A 1-letter word!
    - A 0 letter word? Maybe?
  - We can have a recursive case:
    - If the first and last letter are the same, check the "inner word"
    - If they're not → return False

Computational Structures in Data Science

Recursion With Lists

Berkeley
UNIVERSITY OF CALIFORNIA

Another Example – Finding a Minimum

```
def first(s):
    """Return the first element in a sequence."""
    return s[0]
def rest(s):
    """Return all elements in a sequence after the first"""
    return s[1:]

def min_r(s):
    """Return minimum value in a sequence."""
    if
    
    else:
```

indexing an element of a sequence

Slicing a sequence of elements

Base Case

Recursive Case

- Recursion over sequence length

Computational Structures in Data Science

Binary Search

Searching for Items in a Sequence

- Given a sequence of sorted items, how do I find an item's position (index)
- e.g. `my_list.index(item)`
- How do we build our own?
- What if we know our list is sorted?
- We can have a clever, efficient algorithm:
  - Check the middle value → If found, return the middle index
  - If item is smaller than the middle value → search only the first half
  - If item is bigger than the middle value → search only the second half
  - Keep searching each 'half' until there's nothing left to divide.

Binary Search

```
letters = 'abcdefghijklmnopqrstuvwxyz'
def binary_search(sequence, item):
    …
binary_search(letters, 'c') → 2
```

- How do we split this sequence in half?
- We can use inner functions to control our starting and stopping of searching

Binary Search

```
letters = 'abcdefghijklmnopqrstuvwxyz'
binary_search(letters, 'c') → 2
# Step 1:
Find the midpoint
Is 'c' before or after 'm'?
step_1 = 'abcdefghijkl'
binary_search(step_1, 'c')
# Step 2
Is 'c' before or after 'f' (new middle letter)
step_2 = 'abcde'
binary_search(step_2, 'c')
```

Inner Functions

- Inner functions allow us to control our base case, without exposing it to the caller
- What might we want? This is ugly.

```
def binary_search(sequence, item, start, stop):
```

- When should we stop searching? When our 'start' is > 'stop', i.e. we've gone past the end of our sequence
- Enter inner functions!

```
def binary_search(sequence, item):
    def helper(start, stop):
        …
    return helper(0, len(sequence) - 1)
```

Binary Search – A Start

```
def binary_search(sequence, item):
    def helper(start, stop):
        if start > stop:
            return -1
        mid = (start + stop) // 2
        if sequence[mid] == item:
            return mid
        elif sequence[mid] > item:
             return ________________________
        else:
             return ________________________
    return helper(0, len(sequence) - 1)
```

Binary Search – A Start

```
def binary_search(sequence, item):
    def helper(start, stop):
        if start > stop:
            return -1
        mid = (start + stop) // 2
        if sequence[mid] == item:
            return mid
        elif sequence[mid] > item:
             return helper(start, mid - 1)
        else:
             return helper(mid + 1, stop)
    return helper(0, len(sequence) - 1)
```

Computational Structures in Data Science

Review: Order of Execution

Recall: Iteration

1. Initialize the “base” case of no iterations

2. Starting value

3. Ending value

```
def sum_of_squares(n):
    accum = 0
    for i in range(1,n+1):
        accum = accum + i*i
    return accum
```

4. New loop variable value

Recursion Key concepts – by example

1. Test for simple "base" case

2. Solution in simple "base" case

```
def sum_of_squares(n):
    if n < 1:
        return 0
    else:
        return sum_of_squares(n-1) + n**2
```

3. Assume recusive solution to simpler problem

4. "Combine" the simpler part of the solution, with the recursive case

In words

- The sum of no numbers is zero
- The sum of 1^2 through n^2 is the
  - sum of 1^2 through $(n-1)^2$
  - plus n^2

```
def sum_of_squares(n):
    if n < 1:
        return 0
    else:
        return sum_of_squares(n-1) + n**2
```

Why does it work

```
sum_of_squares(3)

# sum_of_squares(3) => sum_of_squares(2) + 3**2
#                   => sum_of_squares(1) + 2**2 + 3**2
#                   => sum_of_squares(0) + 1**2 + 2**2 + 3**2
#                   => 0 + 1**2 + 2**2 + 3**2 = 14
```

Questions

- In what order do we sum the squares ?
- How does this compare to iterative approach ?

```
def sum_of_squares(n):
    accum = 0
    for i in range(1,n+1):
        accum = accum + i*i
    return accum
```

```
def sum_of_squares(n):
    if n < 1:
        return 0
    else:
        return sum_of_squares(n-1) + n**2
```

```
def sum_of_squares(n):
    if n < 1:
        return 0
    else:
        return n**2 + sum_of_squares(n-1)
```

Trust …

- The recursive “leap of faith” works as long as we hit the base case eventually

- What happens if we don’t?

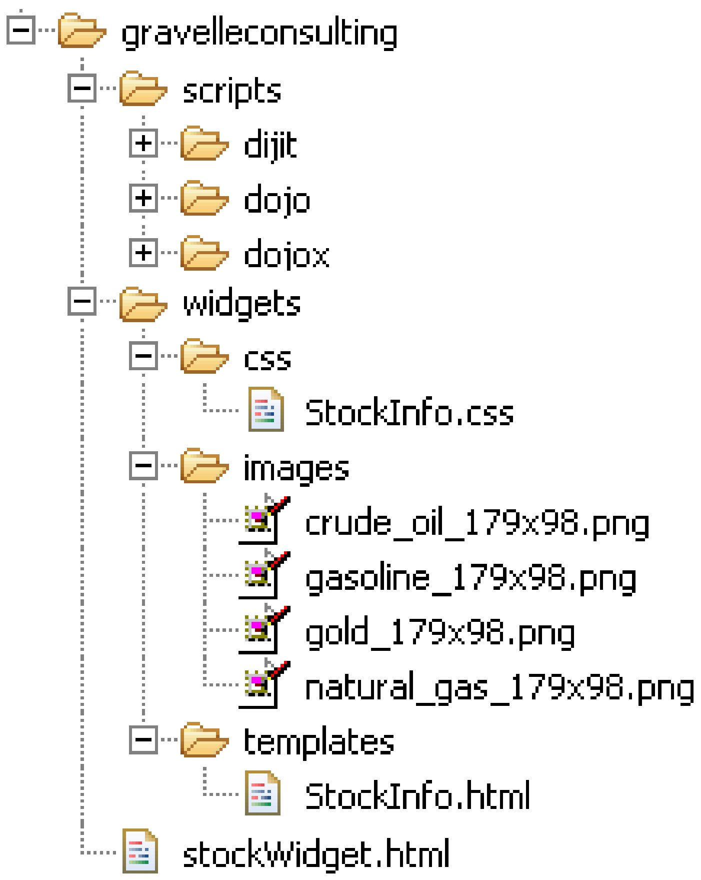
Recursion (unwanted)

Why Recursion?

- "After Abstraction, Recursion is probably the 2^{nd} biggest idea in this course"
- "It's tremendously useful when the problem is self-similar"
- "It's no more powerful than iteration, but often leads to more concise & better code"
- "It's more 'mathematical'"
- "It embodies the beauty and joy of computing"
- …

Example I

List all items on your hard disk



- Files
- Folders contain
  - Files
  - Folders

Recursion!

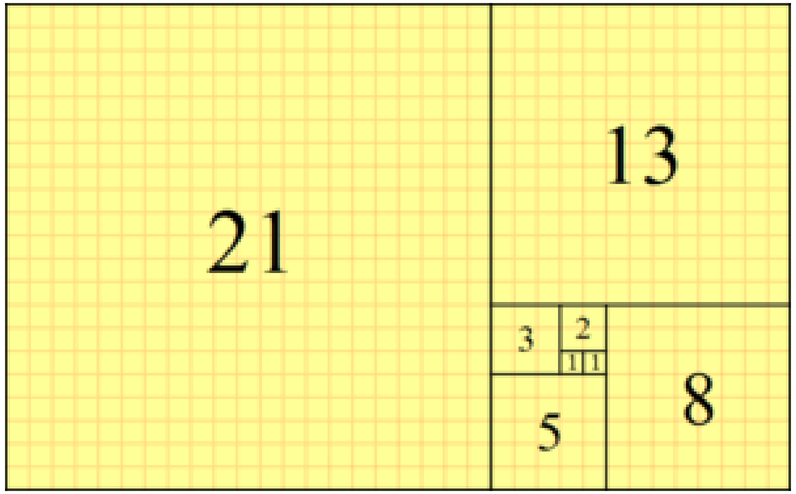
Computational Structures in Data Science

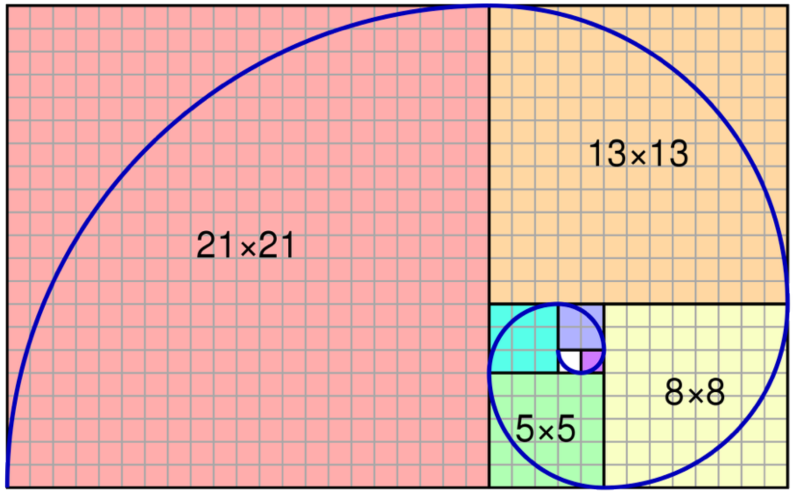
The Fibonacci Sequence

The Fibonacci Sequence

- 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89…

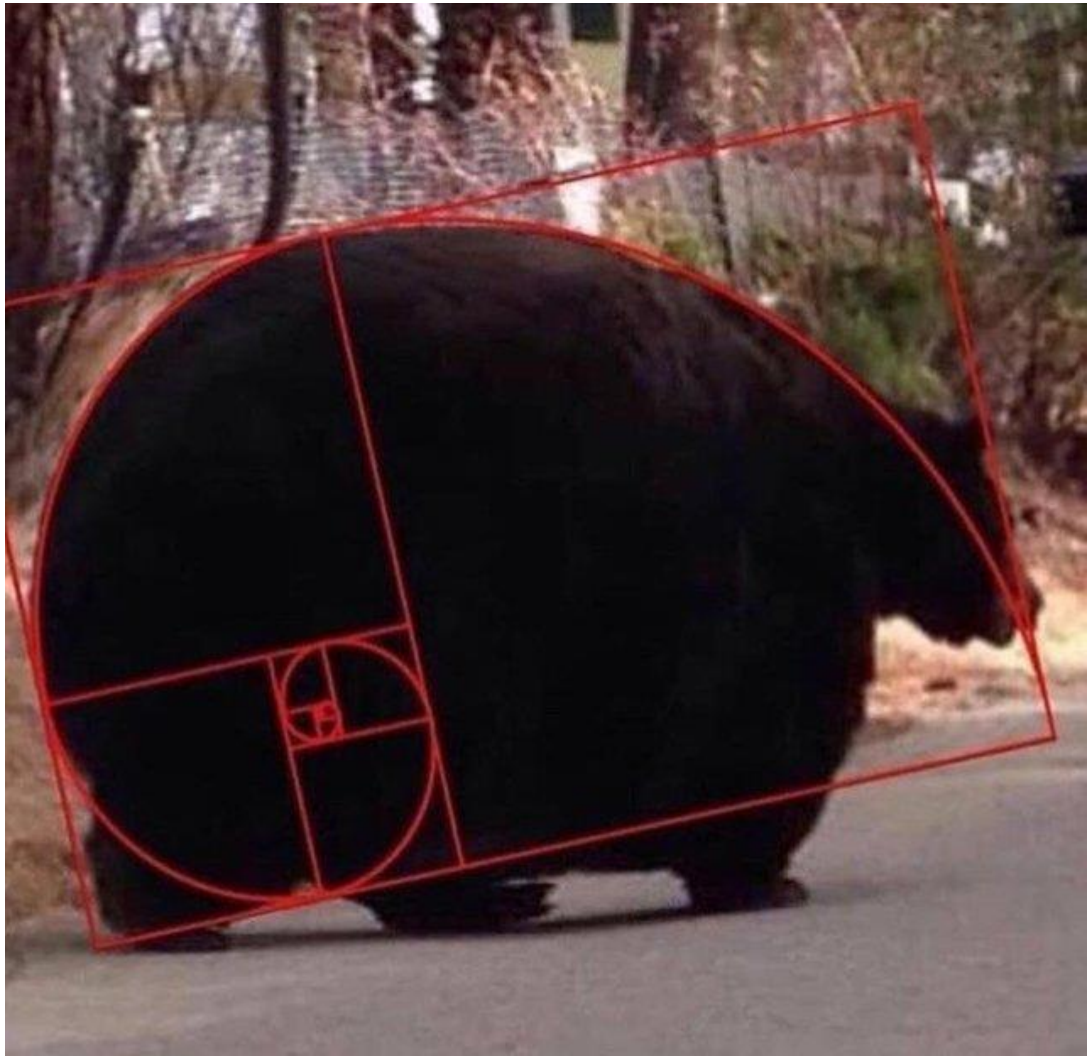
- F0 = 0, F1 = 1
- Fn = F(n-1) + F(n-2)





Golden Spirals Occur in Nature

GO BEARS



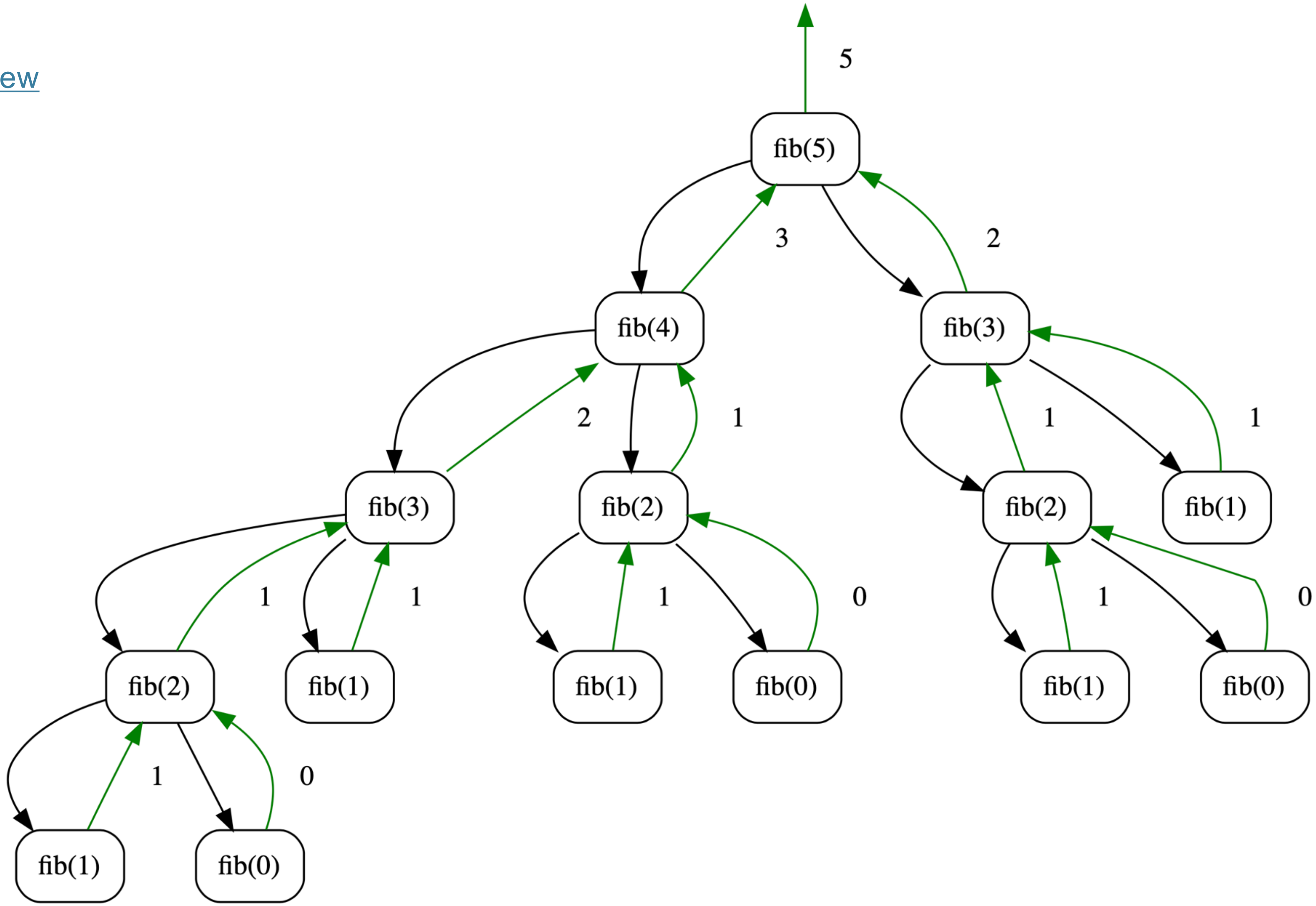
Fibonacci Code

```
fibonacci(n) = fibonacci(n-1) + fibonacci(n-2)
      where fibonacci(1) == 1 and fibonacci(0) == 0
```

```
def fib(n):
    """
    >>> fib(5)
    5
    """
    if n < 2:
        return n
    return fib(n - 1) + fib(n - 2)
```

Visualizing Fib Recursion:

Interactive View



But what about the iterative version?

- In practice, recursive fib is *slow!*
- We can write the program using a for loop.
- How do we translate this? You've done it before!
  - Technique is called "dynamic programming".

```
def iter_fib(n):
    (n_1, n_2) = (0, 1)
    for i in range(0, n):
        # This computes n_1+n_2 before updating n_1
        (n_1, n_2) = (n_2, n_1 + n_2)
    return n_1
```

What’s Similar to Fibonacci?

- Many number sequences have similar properties
  - Catalan numbers
  - Pascal's Triangle
- "Branching" Patterns in Biology:
  - (Real) Tree branches
  - Veins in leaves
  - Romanesco Broccoli
  - Population growth of animals over N generations
  - Some of these structures can be modeled recursively

Why Recursion? More Reasons

- Recursive structures exist (sometimes hidden) in nature and therefore in data!
- It's mentally and sometimes computationally more efficient to process recursive structures using recursion.