

Computational Structures in Data Science

Tree Recursion



Announcement

- Midterm on 3/11
 - <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 be on Friday (this week and next)

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

Tree Recursion



Learning Objectives

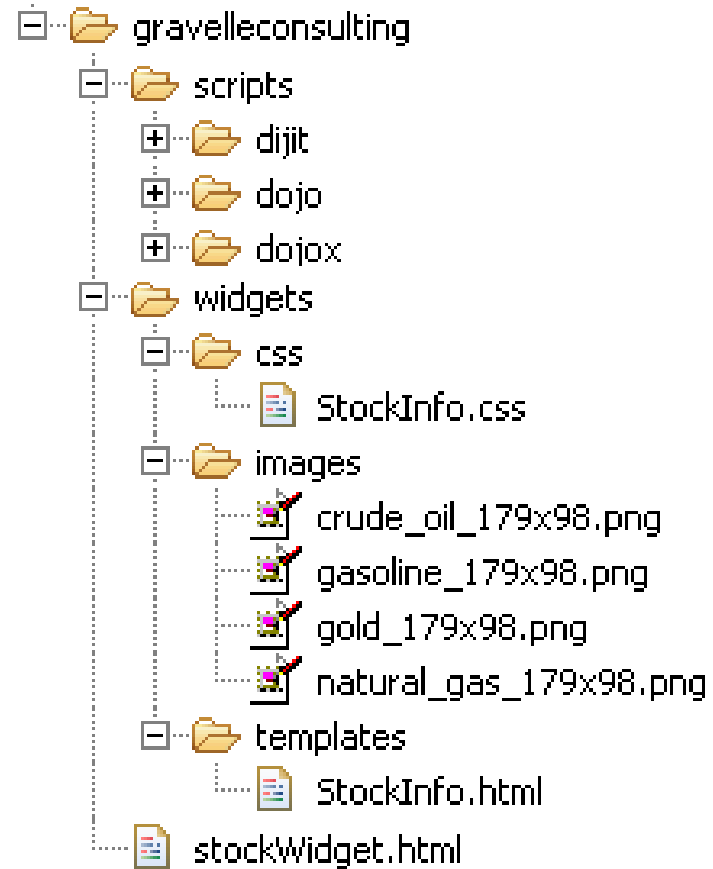
- Write Recursive functions with multiple recursive calls
- Understand Recursive Fibonacci
- Understand the the **count_change** algorithm
- Bonus: Use multiple recursive calls in to sort a list.

Tree Recursion

- Tree Recursion in which involves multiple recursive calls to solve a problem.
- Drawing out a function usually looks like an “inverted” tree.
- Revisit the "vee" program from lecture 11.
- Many of these programs *can't* be written with iteration very easily
- You *can* solve any problem with recursion or iteration, but this technique that makes some problems simpler.

Example I

List all items on your hard disk



- Files
- Folders contain
 - Files
 - Folders

```
def process_directory(directory):  
    for item in directory:  
        if is_file(item):  
            process_file(item)  
        else:  
            process_directory(item)
```

Demo – “Walking” a Directory

- This example relies on OS-level code, which we don't code.
- But the recursive template is quite simple.
- ```
>>> walk_directory('/Users/Michael/Desktop/Photography',
max_depth = 3)
```

# Computational Structures in Data Science

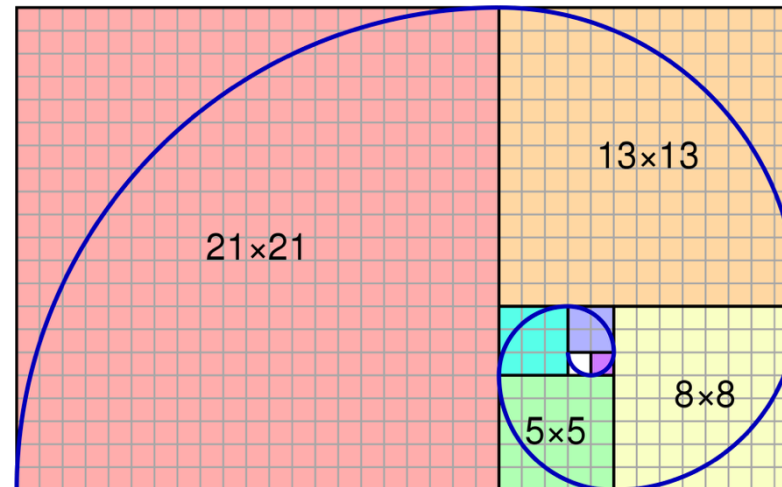
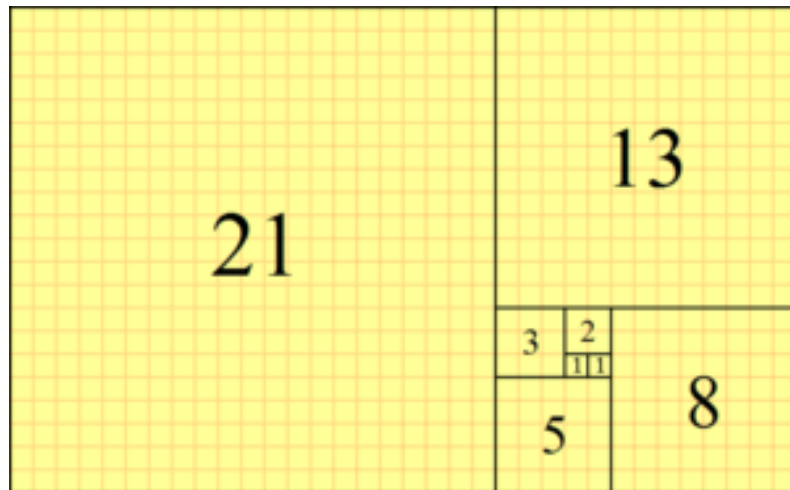
---

## The Fibonacci Sequence



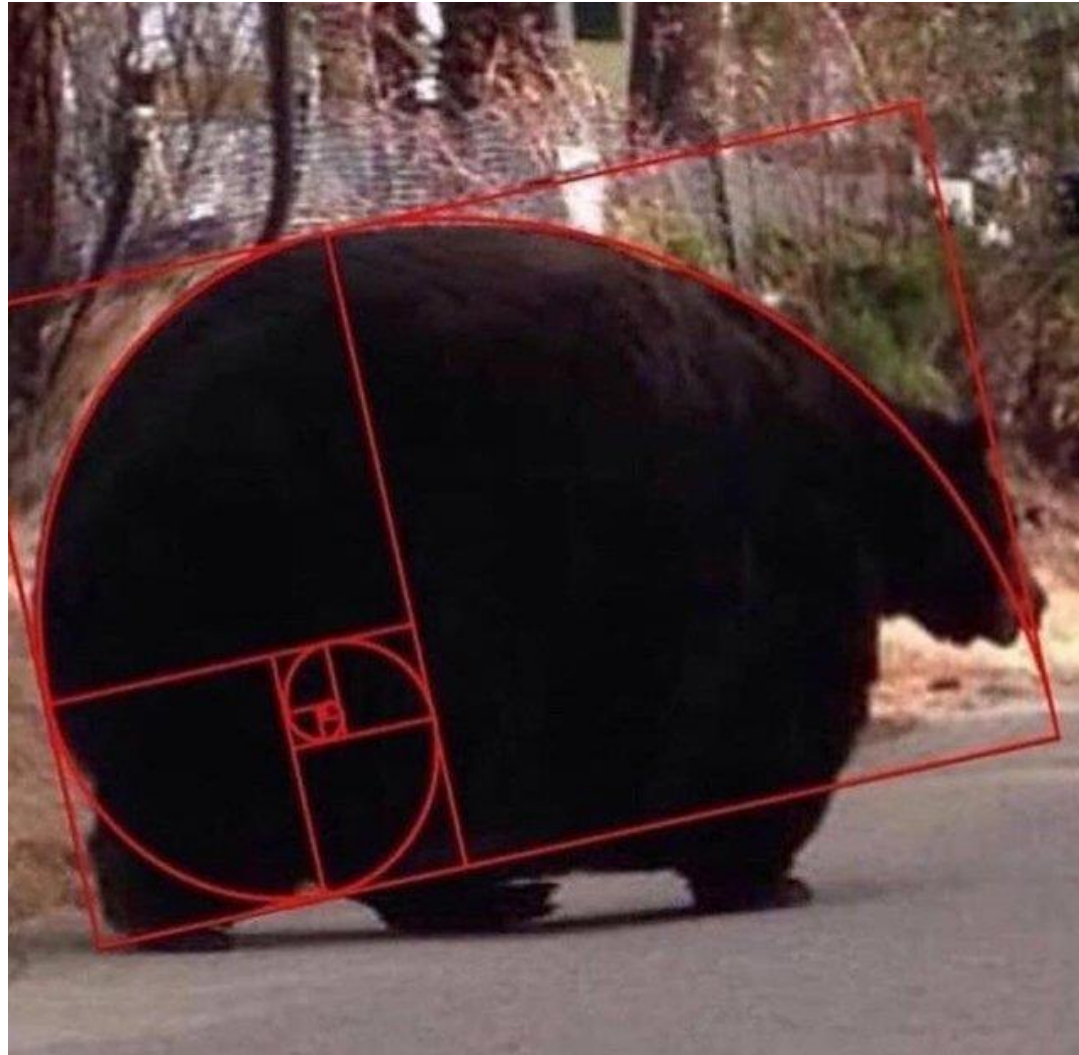
# The Fibonacci Sequence

- 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89...
- $F_0 = 0, F_1 = 1$
- $F_n = F_{n-1} + F_{n-2}$



# Golden Spirals Occur in Nature

GO BEARS



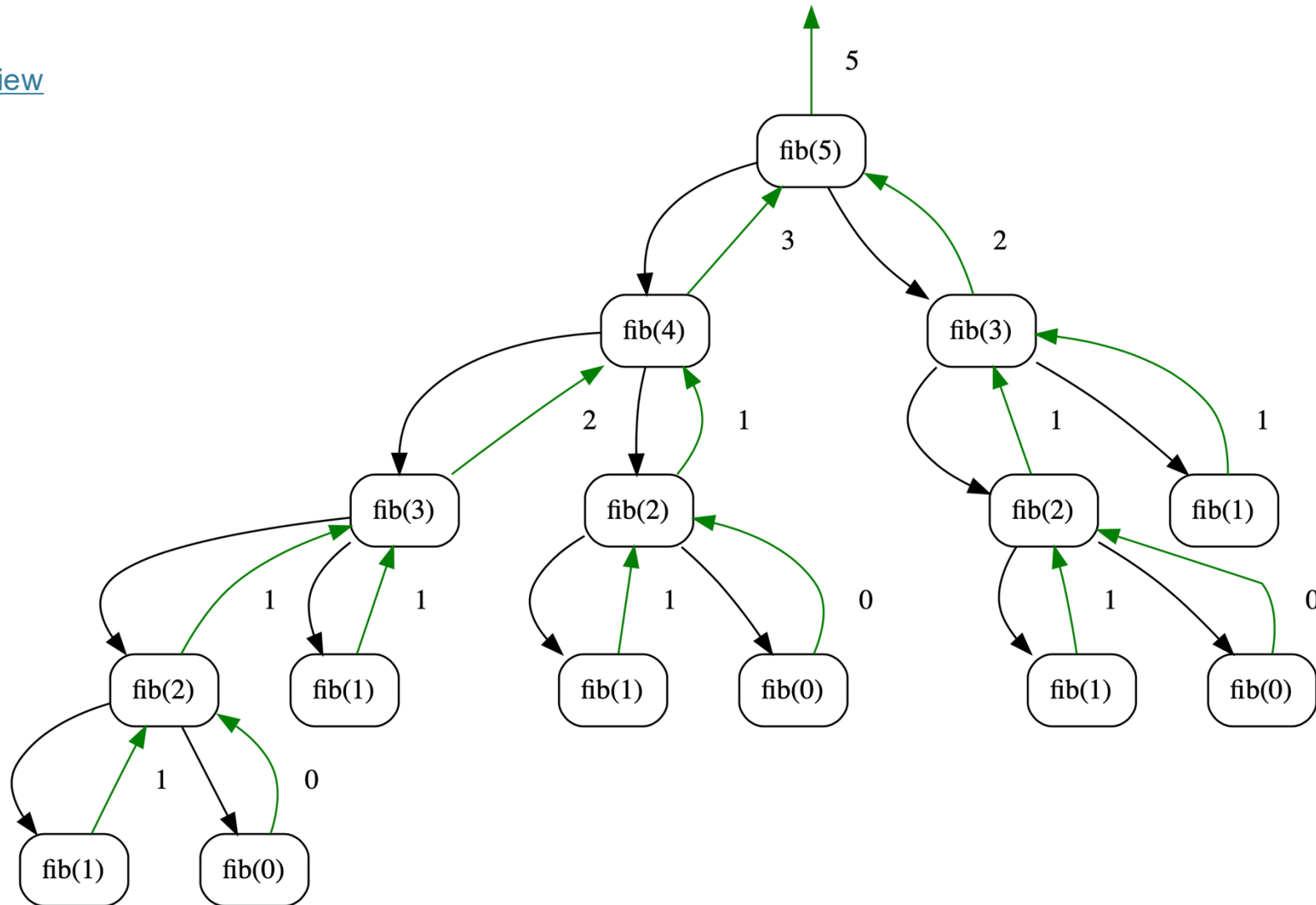
# Fibonacci Code

$\text{fibonacci}(n) = \text{fibonacci}(n-1) + \text{fibonacci}(n-2)$   
where  $\text{fibonacci}(1) == 1$  and  $\text{fibonacci}(0) == 0$

```
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

# Computational Structures in Data Science

---

Count  
Change



# Counting Change

- Problem Statement:
- Given (an infinite number of) coins, (25¢, 10¢, etc) how many different ways can I represent 10¢?
  - e.g. 5¢ can be made 2 ways: 1 nickel, or 5 pennies
  - 10¢ can be made 4 ways: [1x 10¢, 2x 5¢, 1 5¢ + 5 1¢, 10x 1¢]
  - Order doesn't matter, 5¢ + 5 1¢ is the same as 5 1¢ + 5¢
- How do we solve this?

# Why use problems like count change?

- We're partitioning coins, but these could be bills, or other currency
- Explore of problem like [count partitions](#)
- Many tree recursive questions follow a similar *recursive* step
  - Notice how instead of a conditional, we combine the results of two possible options
- We make recursive calls for all possible outcomes, then the *base case(s)* handle the conditional logic.

# Counting Change

- Problem Statement:
- Given (an infinite number of) coins, (25¢, 10¢, etc) how many different ways can I represent 10¢?
  - e.g. 5¢ can be made 2 ways: 1 nickel, or 5 pennies
  - 10¢ can be made 4 ways: [1x 10¢, 2x 5¢, 1 5¢ + 5 1¢, 10x 1¢]
  - Order doesn't matter, 5¢ + 5 1¢ is the same as 5 1¢ + 5¢
- How do we solve this?

# Counting Change

- change for 25¢ using [25, 10, 5] → 4
- What do we return?
  - 1 if valid count
  - 0 otherwise
- What are possible “smaller” problems?
  - Smaller amount of money → use coin
  - Fewer coins → “discard” coin
- What is our base case?
  - valid count: value is 0
  - invalid count: value is  $< 0$ , or no coins left
- **Recursion:**
  - Divide: split into two problems (smaller amount & fewer coins)
  - Combine: addition (# of ways)

# count\_change code

```
def count_change(value, coins):
 """
 >>> denominations = [50, 25, 10, 5, 1]
 >>> count_change(7, denominations)
 2
 """
 if value < 0 or len(coins) == 0:
 return 0
 elif value == 0:
 return 1
 using_coin = ...
 not_using_coin = ...
 return using_coin + not_using_coin
```

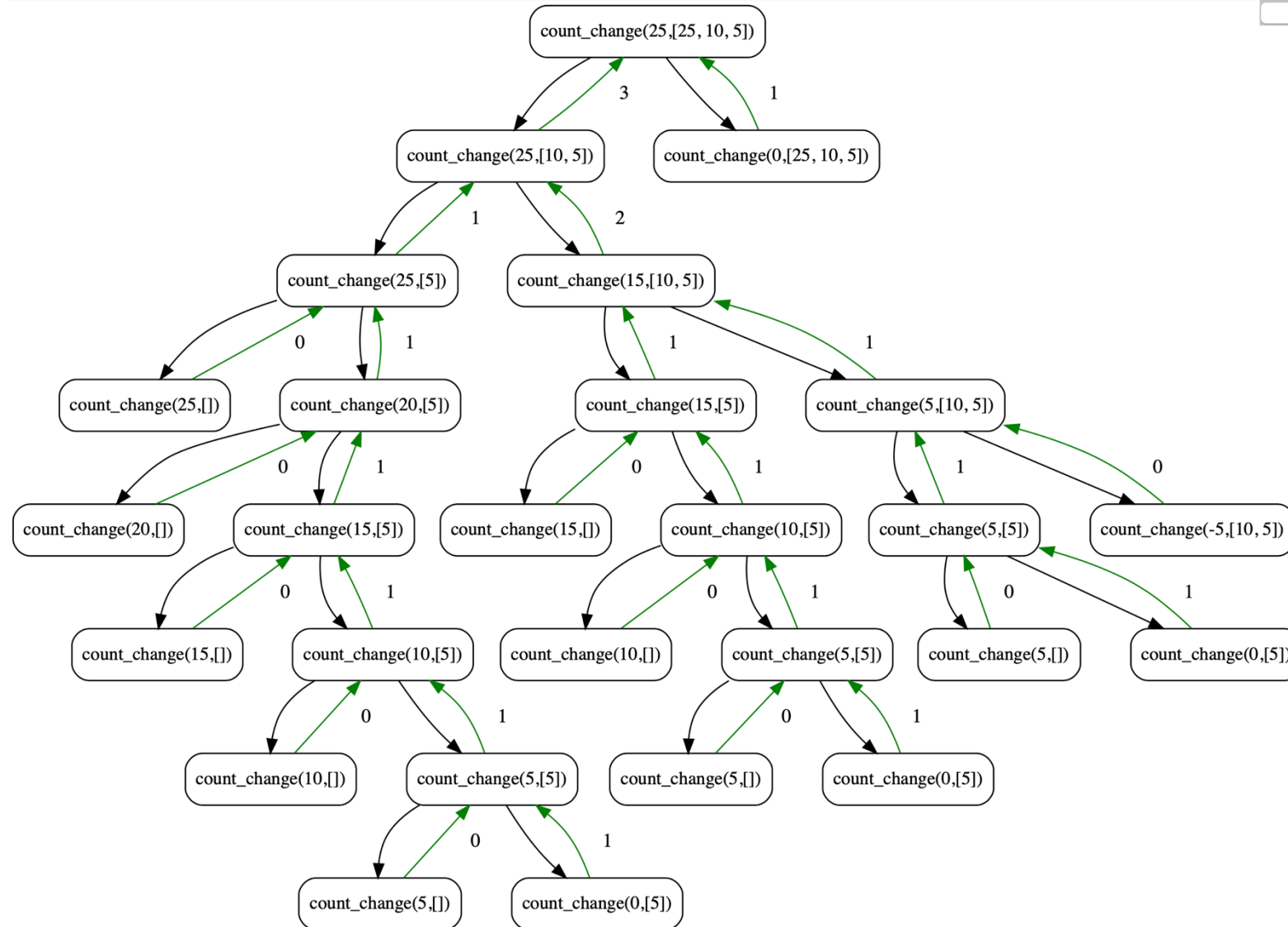
# count\_change code

```
def count_change(value, coins):
 """
 >>> denominations = [50, 25, 10, 5, 1]
 >>> count_change(7, denominations)
 2
 """
 if value < 0 or len(coins) == 0:
 return 0
 elif value == 0:
 return 1
 using_coin = count_change(value - coins[0], coins)
 not_using_coin = ...
 return using_coin + not_using_coin
```

# count\_change code

```
def count_change(value, coins):
 """
 >>> denominations = [50, 25, 10, 5, 1]
 >>> count_change(7, denominations)
 2
 """
 if value < 0 or len(coins) == 0:
 return 0
 elif value == 0:
 return 1
 using_coin = count_change(value - coins[0], coins)
 not_using_coin = count_change(value, coins[1:])
 return using_coin + not_using_coin
```

# Visualizing Count Change [\[Interactive view\]](#)



# Why use problems like count change?

- We're partitioning coins, but these could be bills, or other currency
- Explore of problem like [count\\_partitions](#)
- Many tree recursive questions follow a similar *recursive* step
  - Notice how instead of a conditional, we combine the results of two possible options
- We make recursive calls for all possible outcomes, then the *base case(s)* handle the conditional logic.

# There are many more recursive problems!

- The Knapsack Problem: Maximize the value of items thrown in a bag up to some "weight"
- Anything relating to Family Trees, Relationships, Social Networks
  - Count the Nth degree of "followers of followers" of some one
  - Many of these involve *graphs* which you'll learn in CS61B
- Subsets, Combinations, Permutations
- Longest Common Subsequence of 2 sets
  - Imagine 2 words, or 2 strings of DNA

# Computational Structures in Data Science

---

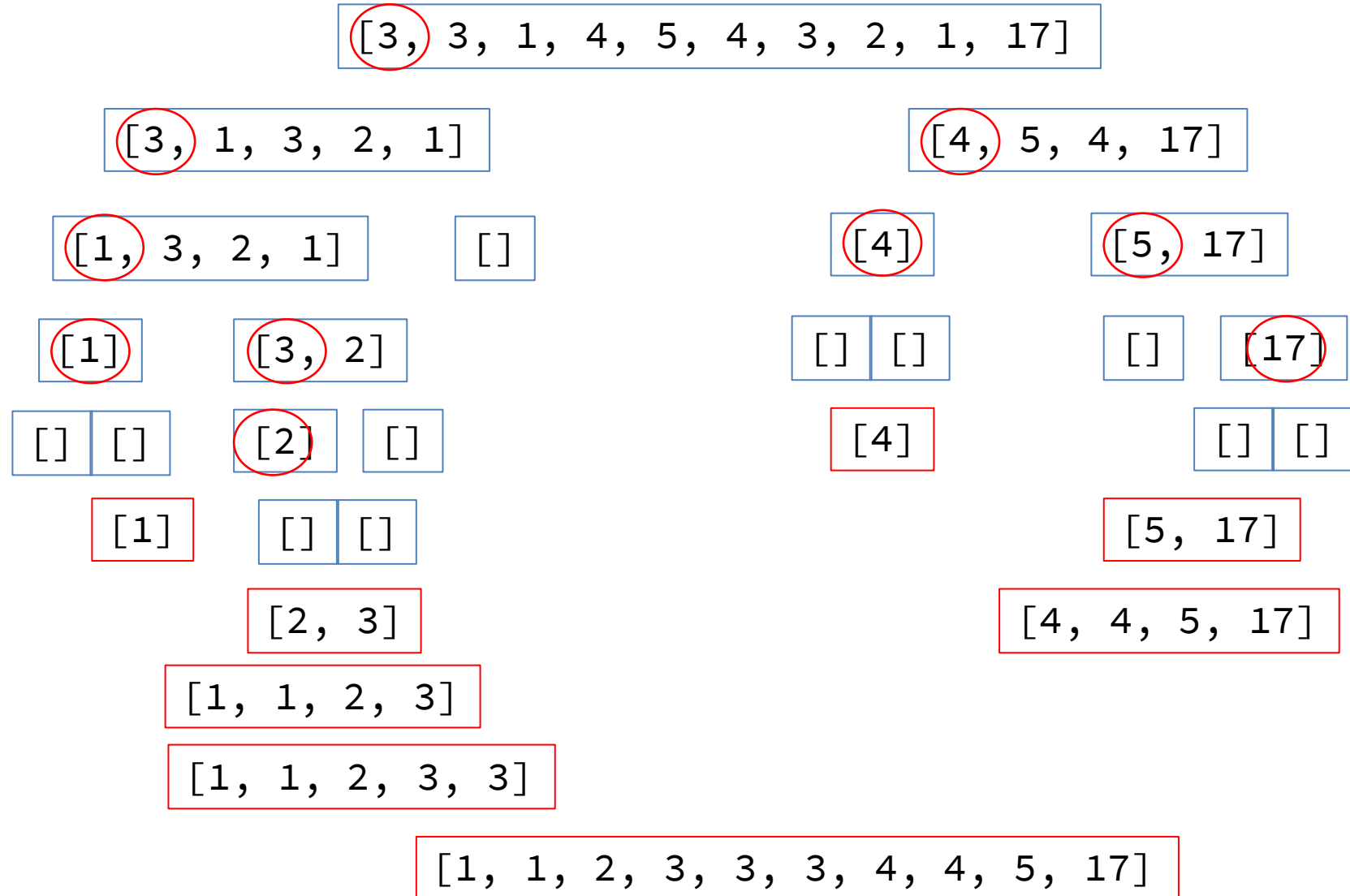
Bonus:  
Quicksort



# Quicksort

- A fairly simple to sorting algorithm
- Goal: Sort the list by breaking it into partially sorted parts
  - Pick a “pivot”, a starting item to split the list
  - Remove the pivot from your list
  - Split the list into 2 parts, a smaller part and a bigger part
  - Then recursively sort the smaller and bigger parts
  - Combine everything together: the smaller list, the pivot, then the bigger list

# QuickSort Example



# Tree Recursion

- Break the problem into multiple smaller sub-problems, and Solve them recursively

```
def split(x, s):
 return [i for i in s if i <= x], [i for i in s if i > x]

def quicksort(s):
 """Sort a sequence - split it by the first element,
 sort both parts and put them back together."""
 if not s:
 return []
 else:
 pivot = s[0]
 smaller, bigger = split(pivot, s[1:])
 return quicksort(smaller) + [pivot] + quicksort(bigger)

>>> quicksort([3,3,1,4,5,4,3,2,1,17])
[1, 1, 2, 3, 3, 3, 4, 4, 5, 17]
```

# Quicksort Visualization [\[Interactive View\]](#)

