



UC Berkeley EECS
Lecturer
Michael Ball

Computational Structures in Data Science



Tree Recursion





Announcements

- Attendance: <https://go.c88c.org/here>
- Passcode: bears
- Class Chat: <https://go.c88c.org/chat>



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.



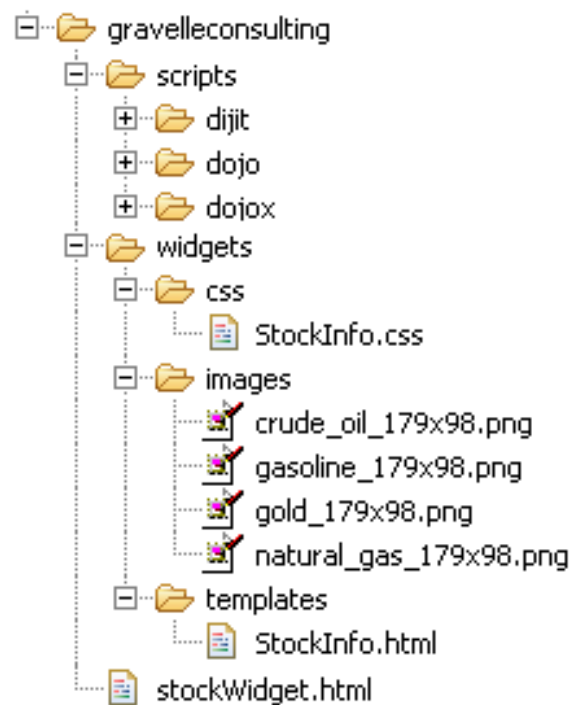
Tree Recursion

- Recursion 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



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)
```

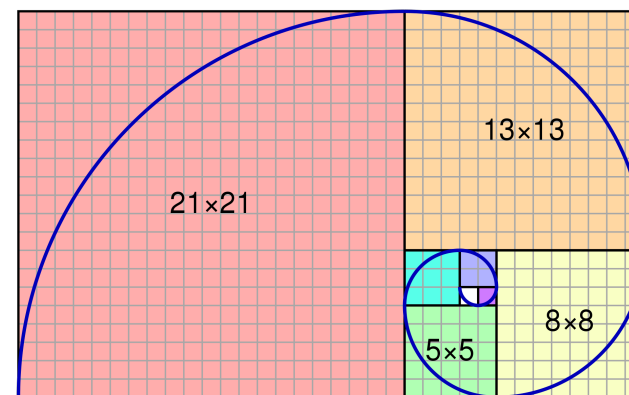
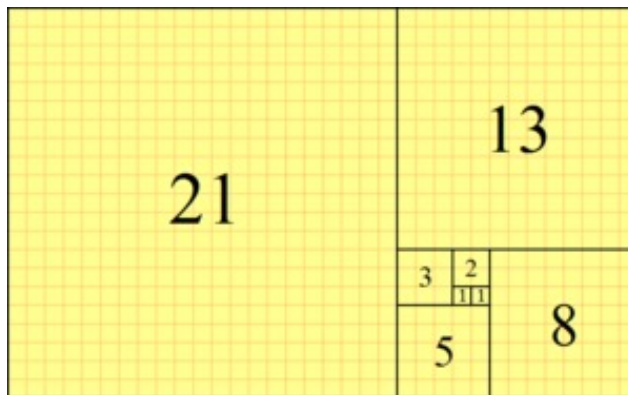


The Fibonacci Sequence

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

$$F_0 = 0, F_1 = 1$$

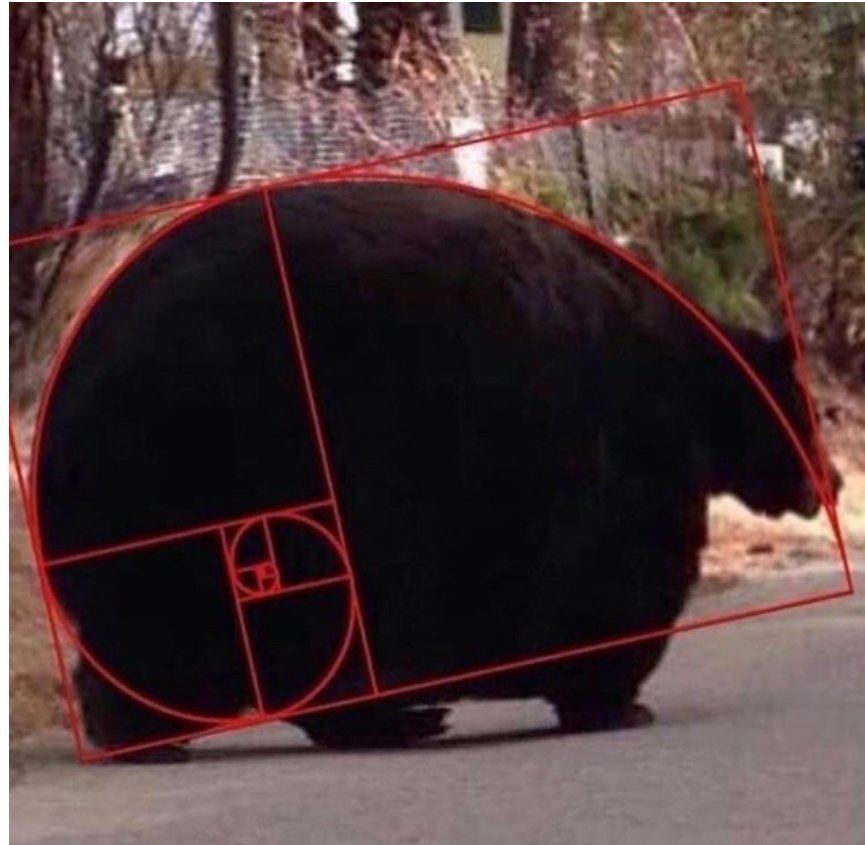
$$F_n = F_{(n-1)} + F_{(n-2)}$$



Golden Spirals Occur in Nature



GO BEARS





Fibonacci Code

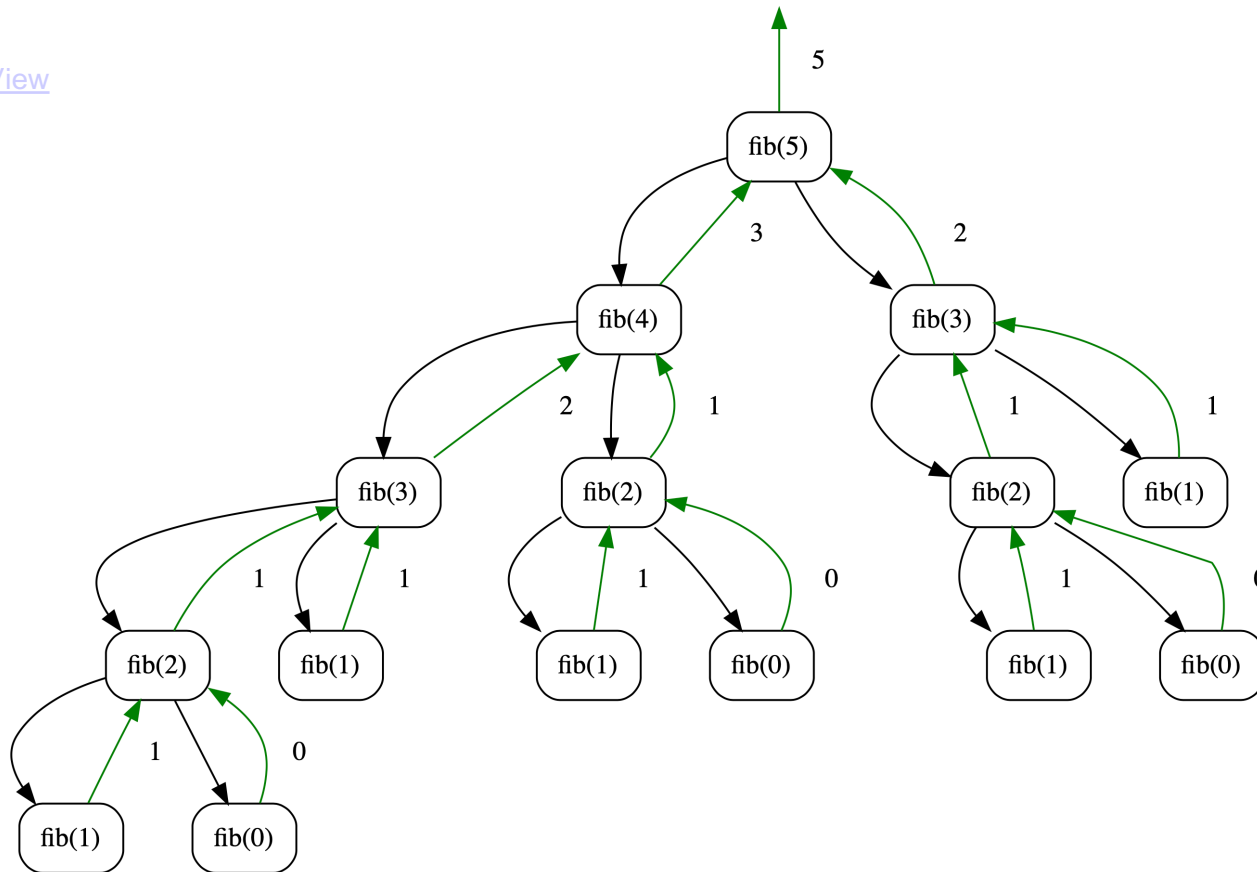
$\text{fibonacci}(n) = \text{fibonacci}(n-1) + \text{fibonacci}(n-2)$
where $\text{fibonacci}(1) == 1$ and $\text{fibonacci}(0) == 0$

```
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?
 - Technique is called "dynamic programming". Take CS70 is that sounds fun.

```
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
```



Announcements

- Attendance: <https://go.c88c.org/here>
- Passcode: bears
- Class Chat: <https://go.c88c.org/chat>



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] $\rightarrow$ 4
- What do we return?
 - 1 if valid count
 - 0 otherwise
- What are possible “smaller” problems?
 - Smaller amount of money $\rightarrow$ use coin
 - Fewer coins $\rightarrow$ “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)



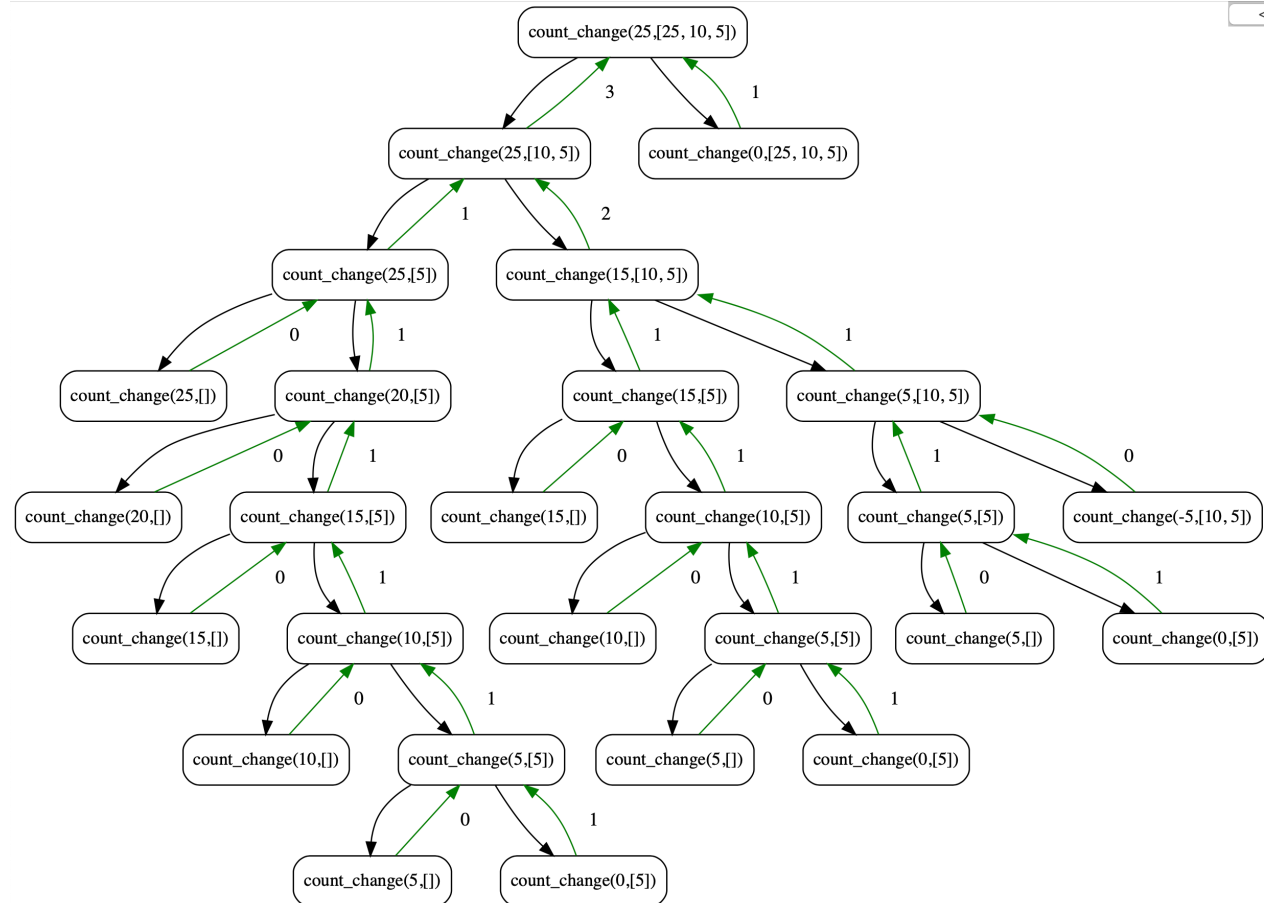
count_change code

```
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 = 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.



Announcements

- Attendance: <https://go.c88c.org/here>
- Passcode: bears
- Class Chat: <https://go.c88c.org/chat>



UC Berkeley EECS
Lecturer
Michael Ball

Computational Structures in Data Science



Bonus: Quicksort



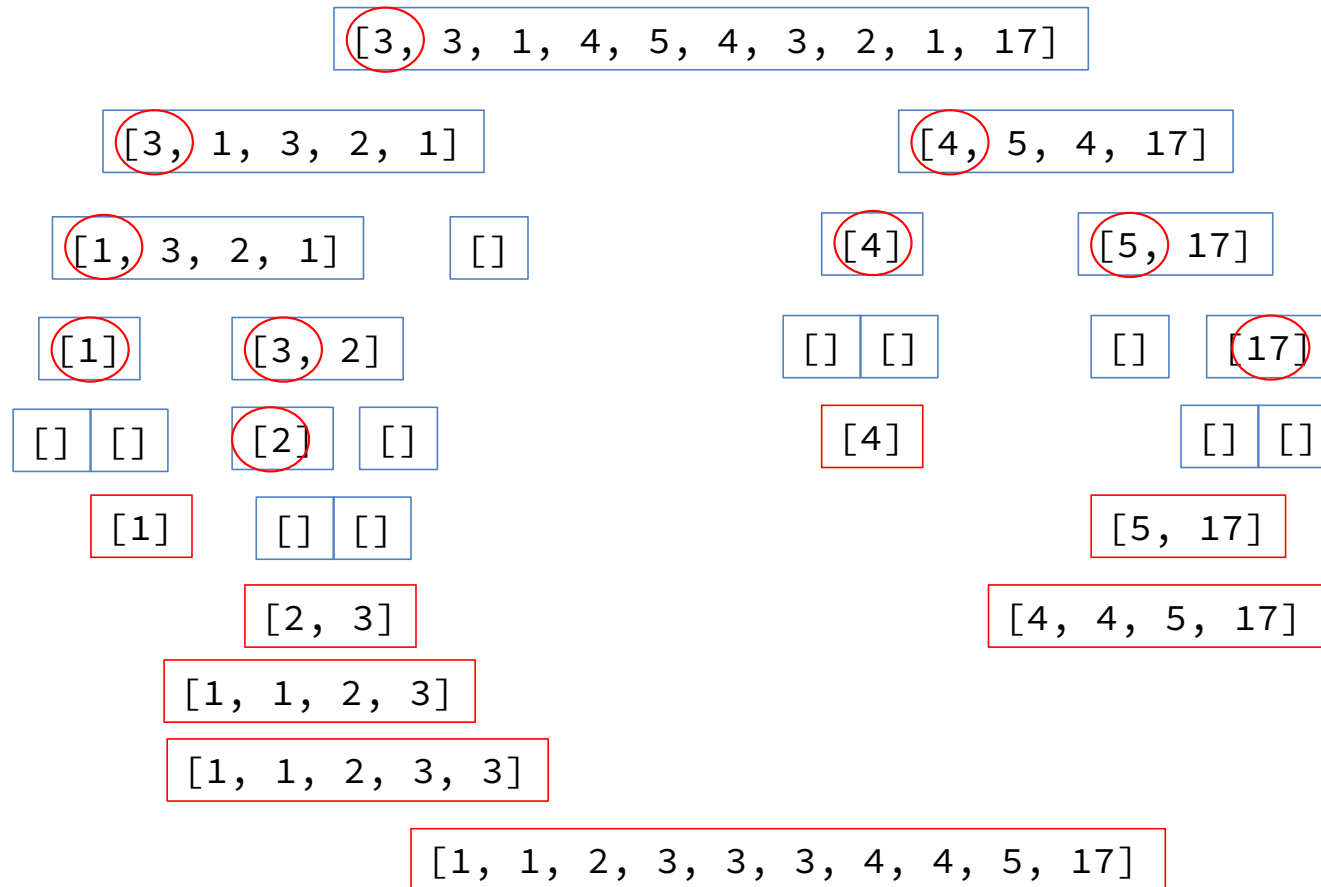


Quicksort

- 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]
```

-