Computational Structures in Data Science

Tree Recursion

Week 4, Summer 2024. 7/10 (Wed)

Lecture 13

Berkeley
UNIVERSITY OF CALIFORNIA

Announcements

- Project01 (“Maps”) Checkpoint due tonight!
- HW05, Lab05 due tonight!
- (Important) Midterm exam scheduling!

Announcements: Midterm scheduling!

- IMPORTANT: "Midterm Exam Scheduling" form was due on Gradescope last night (7/9)
- IF YOU DIDN'T FILL IT OUT: we will send ONE MORE opportunity to schedule your midterm exam
- If you miss this opportunity: we will assume you are taking the midterm at the "standard" slot: **Wednesday July 17th, 2024, 3 PM – 5 PM PST**
  - And, if you don't attend this midterm slot: you won't be taking the midterm
    - (reminder) midterm "clobber" policy: final can replace your midterm

- **(DSP students with +50% exam time)** We have sent you a Google Form to schedule your exam. Please fill this out ASAP!

Midterm content

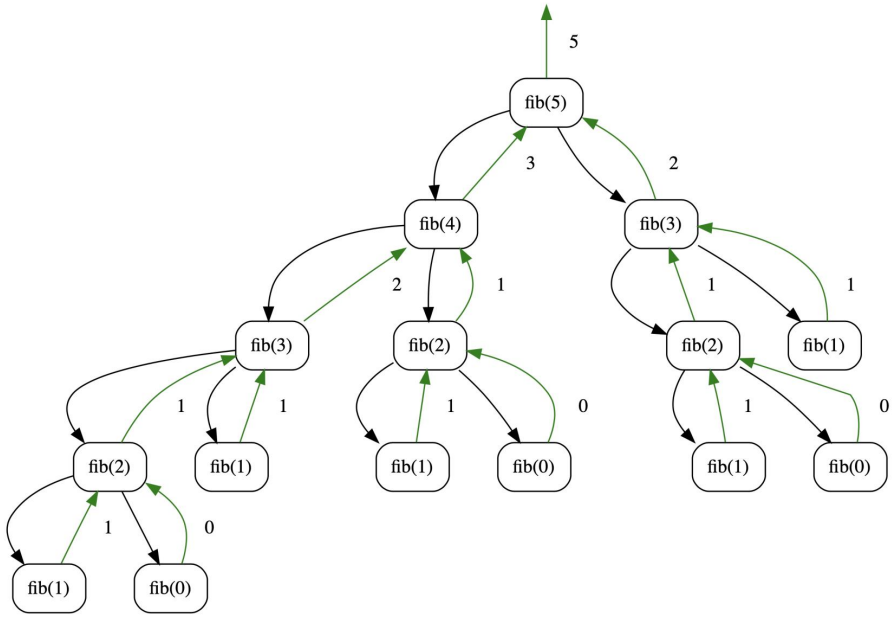
- Midterm will cover content from start of course up to (and including) OOP+Inheritance, aka:
  - Start (inclusive): Lecture 01: “Welcome & Intro” (6/17)
  - End (inclusive): Lecture 15: “OOP – Inheritance” (7/15)
- Midterm will be done through Zoom + Gradescope
- Study tip: past C88C exams can be found here: https://c88c.org/sp24/articles/resources.html#past-midterms
  - Take a look to get a sense of what C88C exams tend to look like. (I highly, highly encourage this)
    - “Be prepared” – Boy Scouts
    - “Luck is when preparation meets opportunity” – Roman philosopher Seneca

Midterm logistics

- The midterm will be held over Zoom + Gradescope
  - You must have your camera + screen sharing on during the entire exam, and we will be doing screen+camera recording.
- You must take the exam in a quiet room with no other students present
- Things to bring to the exam (and nothing else!):
  - **Photo ID**. Ideally your UCB student ID, but anything with your name + photo is fine, eg: Passport, driver's license, etc.
  - **(Optional)** Five (5) pages of handwritten (not typed!) notes
  - **(Optional, recommended)** Additional blank scratch paper, pencil/pen/eraser.
- We will provide everyone with a 1-2 page digital PDF of additional reference
- Other than the above notes, the exam will be closed book, closed notes.
- (For more info, stay tuned for an Ed post)

Today's Lecture

- Tree recursion
  - aka recursive functions that make multiple recursive calls per "level"
  - call stack looks like a "tree"
- Function domain/range

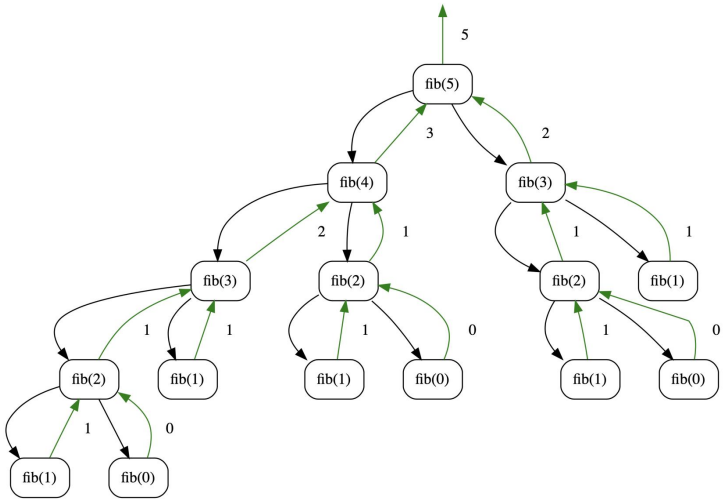


Learning Objectives

- 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.

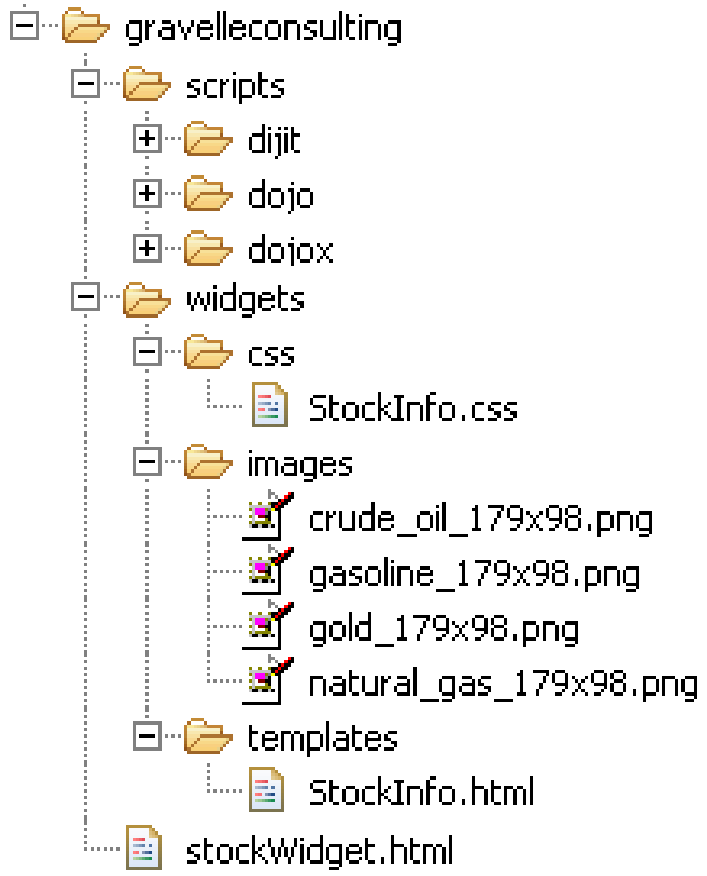
Tree Recursion

- Tree Recursion involves **multiple recursive calls** to solve a problem.
- Drawing out a function usually looks like an “inverted” tree.
- (optional) Revisit the "vee" program from lecture 11.
- Many of these programs *can't* be written with iteration very easily
- Tip: in principle, you *can* solve any problem with recursion or iteration. But, recursion can make 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)
```

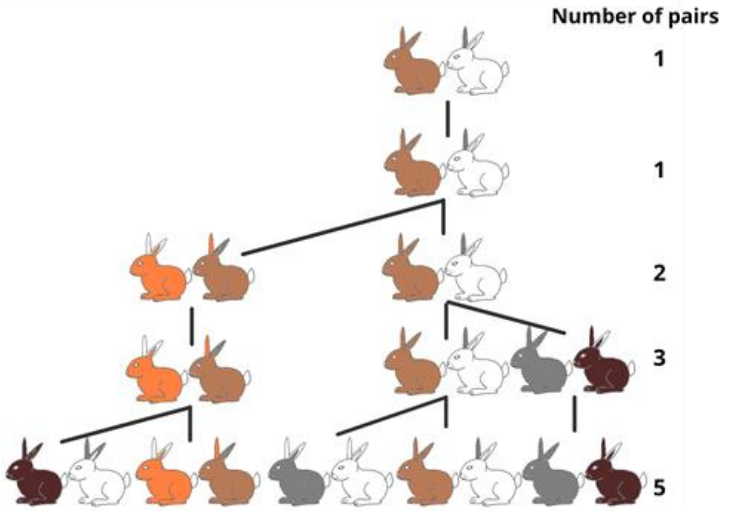
Computational Structures in Data Science

The Fibonacci Sequence

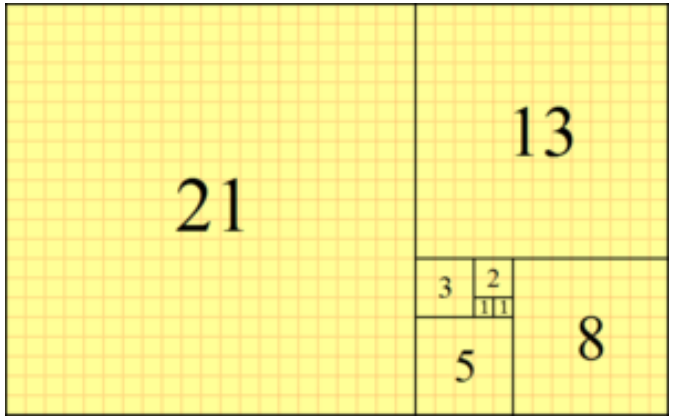
The Fibonacci Sequence

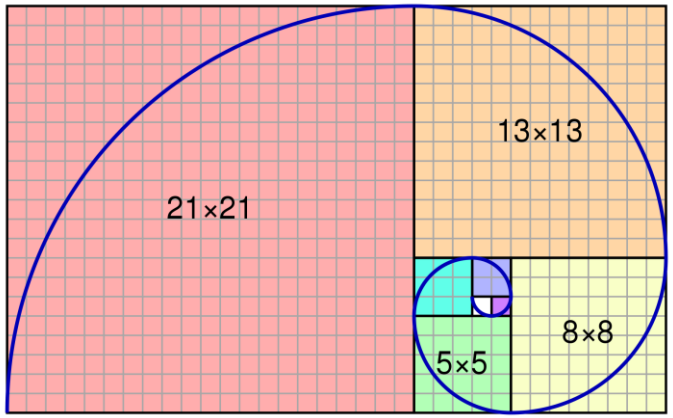
- 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89…

- Fib(0) = 0, Fib(1) = 1
- Fib(n) = Fib(n-1) + Fib(n-2)



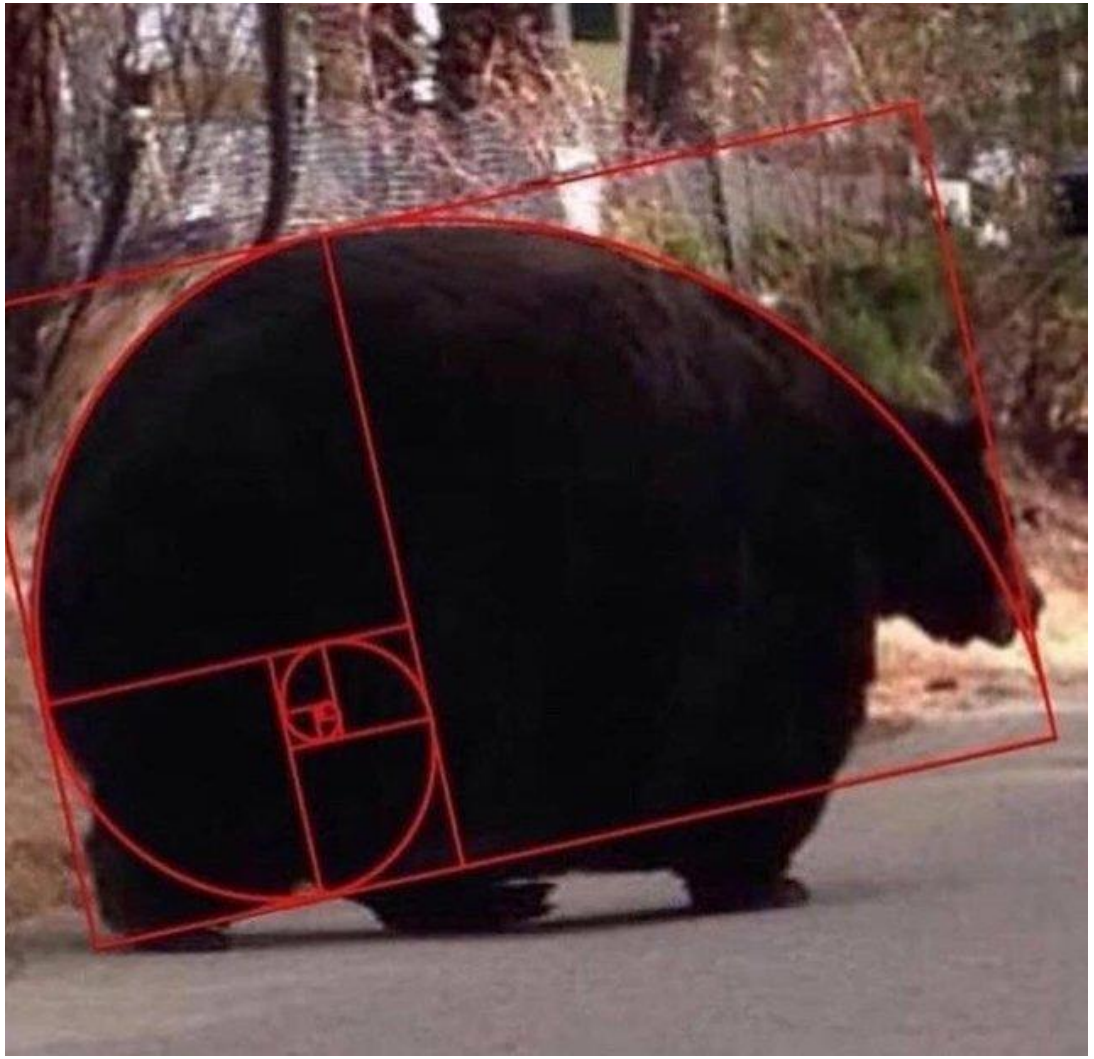
Source: https://www.imaginationstationtoledo.org/about/blog/the-fibonacci-sequence





Golden Spirals Occur in Nature

GO BEARS



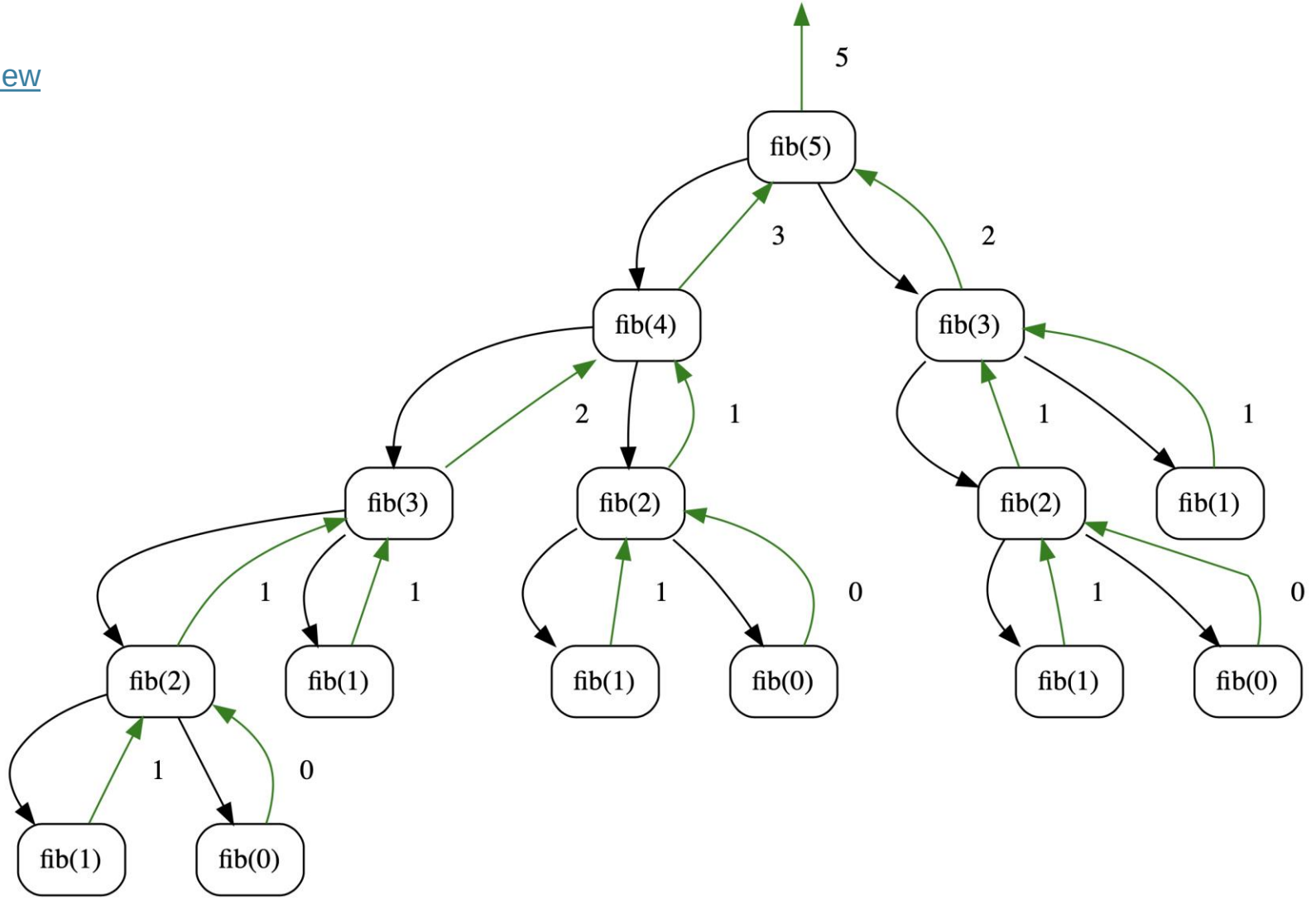
Fibonacci Code

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

```python
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". (covered in CS170)

```
def iter_fib(n):
    (n_1, n_2) = (0, 1)
    for i in range(0, n):
        fib_next = n_1 + n_2
        n_1 = n_2
        n_2 = fib_next
        # Note: below update is equivalent to above
        # 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

Figure 6 shows the compounding of some of the inflorescences. These pictures were all done with simple recursion.

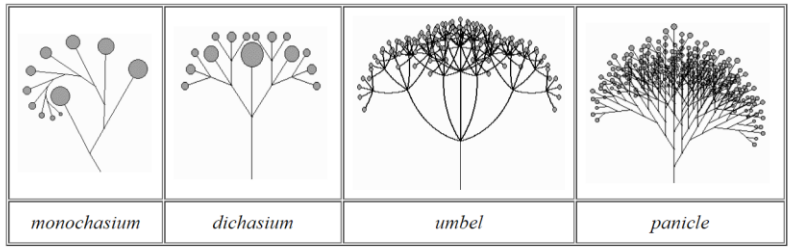


Figure 6: *Compound inflorescences*

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 calculate this?

Question: Put on our “recursion” hats, and ask:
what is the recursive structure of this problem?
(base cases, recursive cases)

Counting Change

- change for 25¢ using [25, 10, 5] → 4

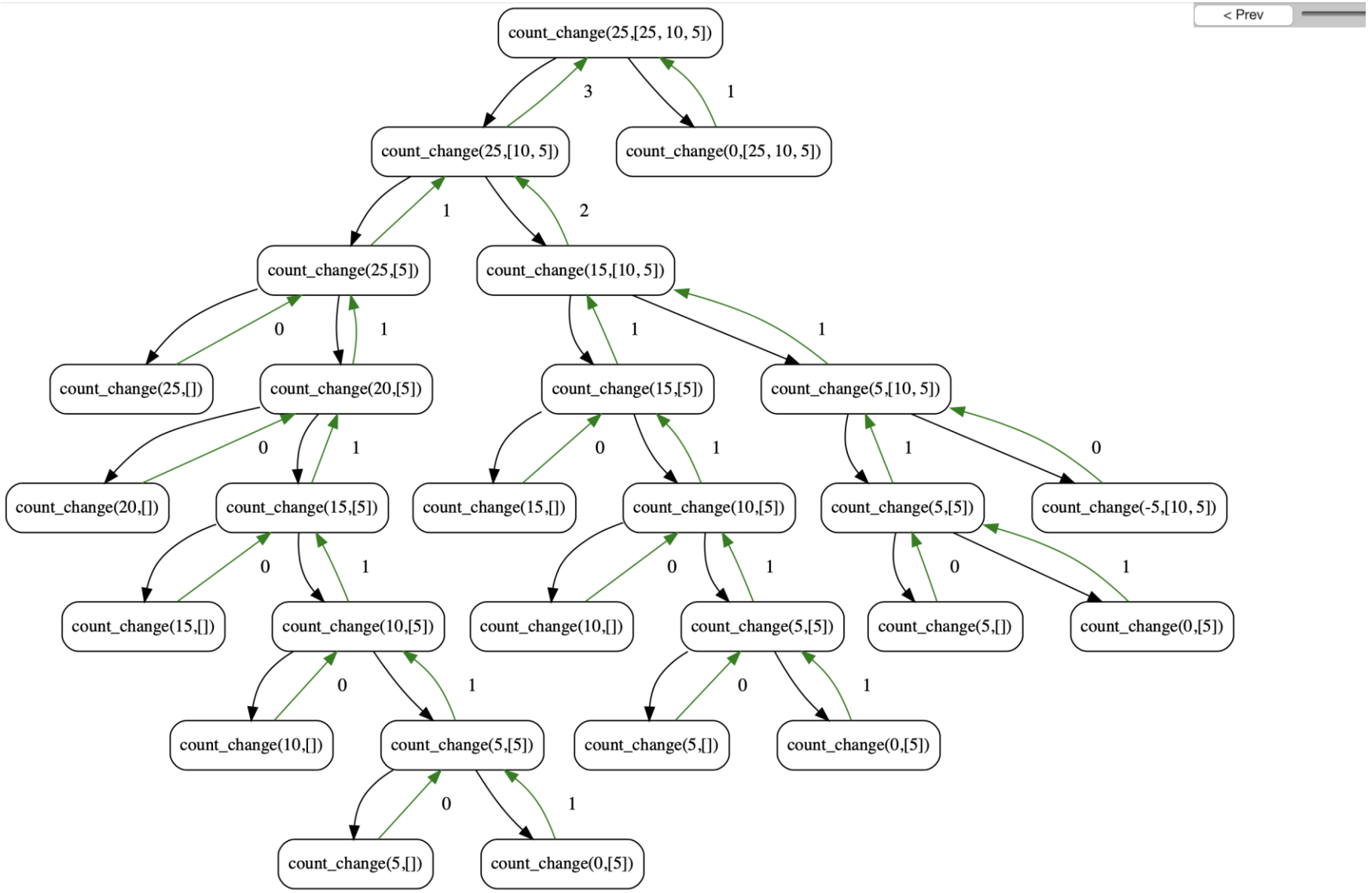
- What are possible "smaller" problems?
  - Smaller amount of money → use coin
  - Fewer coins → "discard" coin
- What is our base case?
  - valid count: value is 0 -> return 1
  - invalid count: value is < 0, or no coins left -> return 0
- **Recursion:**
  - Divide: split into two problems (smaller 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
- 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

Exercise: map

- Recall: map(fn, sequence) is a higher-order function that applies fn(item) to each item in sequence.

```
>>> map(lambda x: x * 2, [1, 2, 3])
[2, 4, 6]
```

Question: re-implement map() as a recursive function, map_recurse(). Do not use for/while loops, list comprehensions, or the built-in map().

Hint: put on our “recursion” hat. What is the recursive structure? (base cases, recursive cases)

```
def map_recurse(fn, seq):
    # FILL ME IN
```


Exercise: map

- Recall: map(fn, sequence) is a higher-order function that applies fn(item) to each item in sequence.

```
>>> map(lambda x: x * 2, [1, 2, 3])
[2, 4, 6]
```

Base case:
Map-ing an empty list -> return empty list

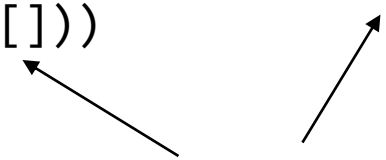
Recursive case:
Apply `fn` to the first seq element, and concatenate this to the result of map-ing the rest of the sequence.

```
def map_recurse(fn, seq):
    if not seq:
        return []
    return [fn(seq[0])] + map_recurse(fn, seq[1:])
```


Exercise: map

- (Visualization) Call stack

```
fn_square = lambda x: x * 2
map_recurse(fn_double, [1, 2, 3])
-> [fn_double(1)] + map_recurse(fn_double, [2, 3])
-> [fn_double(1)] + ([fn_double(2)] + map_recurse(fn_double, [3]))
-> [fn_double(1)] + ([fn_double(2)] + ([fn_double([3])] + map_recurse(fn_double, [])))
-> [fn_double(1)] + ([fn_double(2)] + ([fn_double([3])] + []))
-> [fn_double(1)] + ([fn_double(2)] + [6])
-> [fn_double(1)] + [4, 6]
-> [2] + [4, 6]
-> [2, 4, 6]
```



Base case! Empty list

```
def map_recurse(fn, seq):
    if not seq:
        return []
    return [fn(seq[0])] + map_recurse(fn, seq[1:])
```


Exercise: “deep” map

- Now, let’s look at `map_deep()`, where the input list can have **nested lists**. Notably, map_deep() should preserve the nested structure.

```
>>> map_deep(fn_square, [1, [2], [3, [4]]])
[2, [4], [6, [8]]]
```

Question: what is the recursive structure of this problem? (base cases, recursive cases)

Exercise: “deep” map

- Now, let’s look at `map_deep()`, where the input list can have **nested lists**. Notably, map_deep() should preserve the nested structure.

```
>>> map_deep(fn_square, [1, [2], [3, [4]]])
[2, [4], [6, [8]]]
```

Base cases:
Map_deep()-ing an empty list -> return empty list

Recursive cases:

(First val) if the first element is a list, transform it via map_deep(). Else, transform it via fn()
Then, concatenate the transformed first val to the result to map_deep()-ing the rest of the list.

Compare to “shallow” map_recurse()
Base case:
Map-ing an empty list -> return empty list

Recursive case:
Apply `fn` to the first seq element, and concatenate this to the result of map-ing the rest of the sequence.

New: in map_deep(), we may perform a recursive call when transforming the first val.
In map_recurse() we simply apply fn() to the first val.

Tip: isinstance()

- To implement map_deep(), first a tip: `isinstance(thing, type)` lets you check the type of an object/expression:

```
>>> isinstance([1, 2, 3], list)
>>> isinstance([1, 2, 3], dict)
>>> x = 42
>>> isinstance(x, int)
```


Exercise: “deep” map

```
def map_deep(fn, seq):
    if not seq:
        return []
    else:
        if isinstance(seq[0], list):
            first_val = map_deep(fn, seq[0])
        else:
            first_val = fn(seq[0])
        return [first_val] + map_deep(fn, seq[1:])
```

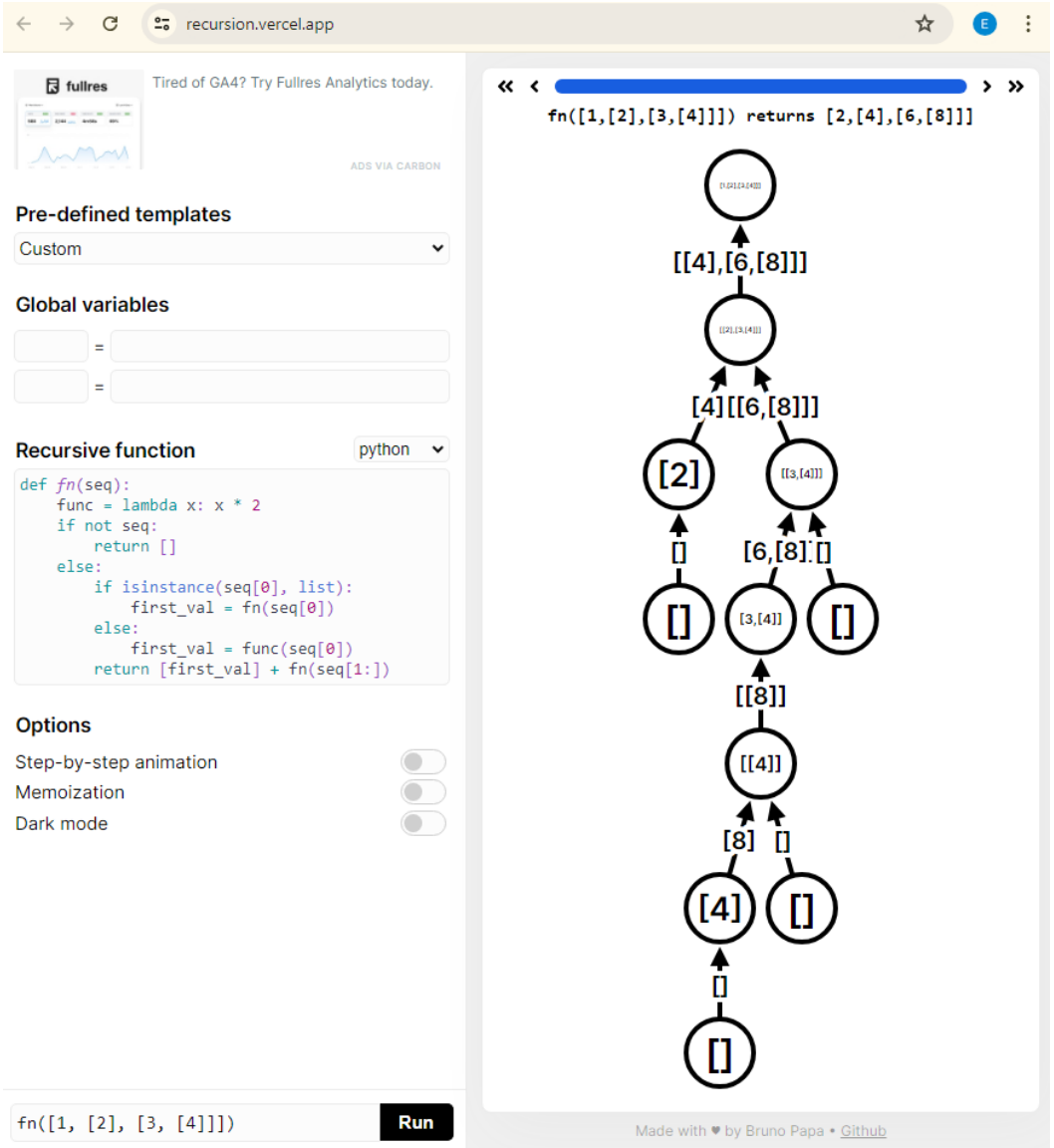

Visualization: map_deep

Link: https://recursion.vercel.app/
Tip: this website is a little finicky, limitations:

- Function must be named fn()
- Lambda arguments are not supported. So, I had to hard-code the square_fn in the function body

```
def fn(seq):
    func = lambda x: x * 2
    if not seq:
        return []
    else:
        if isinstance(seq[0], list):
            first_val = fn(seq[0])
        else:
            first_val = func(seq[0])
        return [first_val] + fn(seq[1:])

fn([1, [2], [3, [4]]])
```



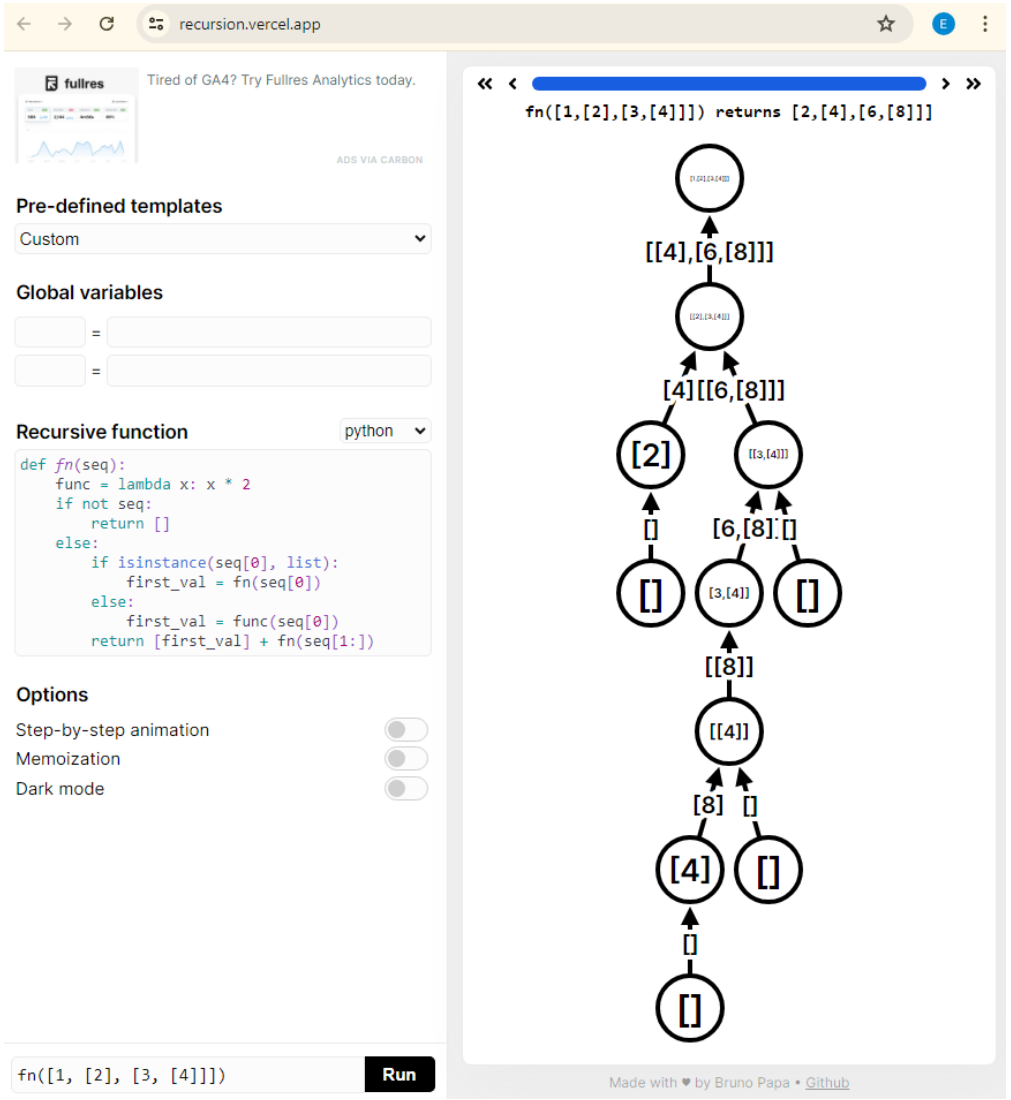
Visualization: map_deep vs shallow map

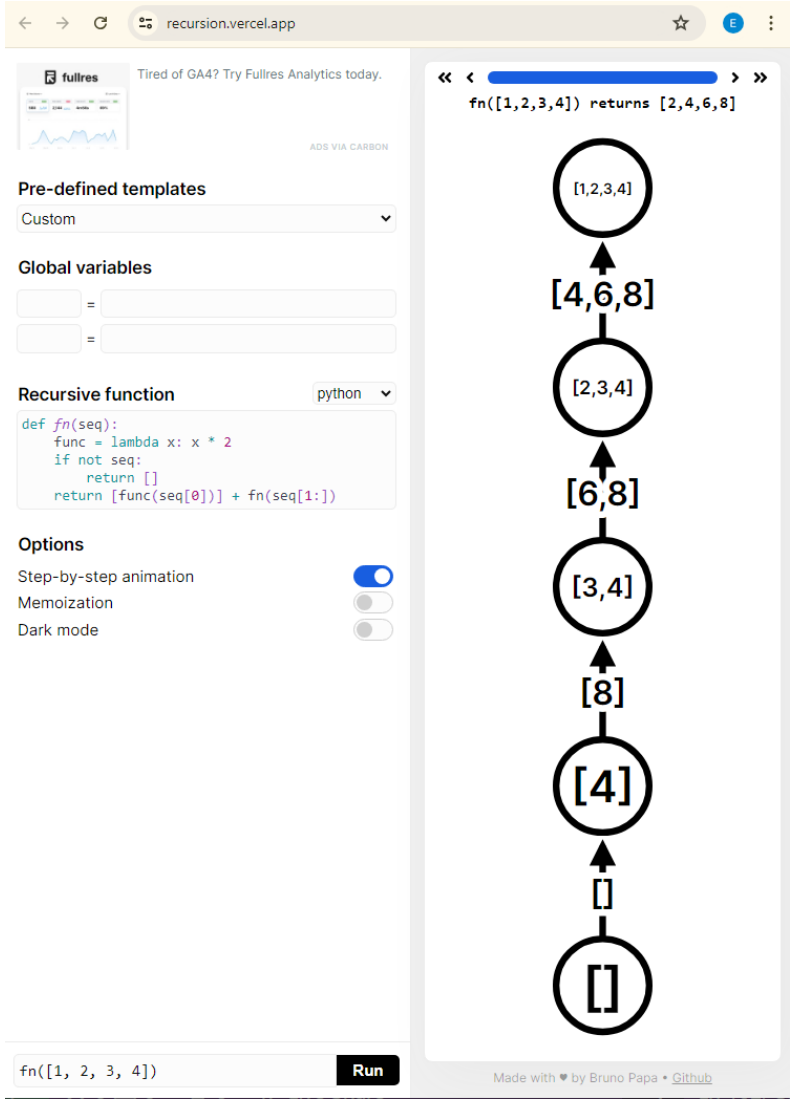
Observation:
"Deep" map (LHS) looks like a "tree", hence "Tree recursion"

"Shallow" map (RHS) has no "branches".

```
def fn(seq):
    func = lambda x: x * 2
    if not seq:
        return []
    return [func(seq[0])] + fn(seq[1:])

fn([1, 2, 3, 4])
```





Link:
https://recursion.vercel.app/

Tip: function “domain”, “range”

- A handy tip about working with functions is to keep track of their expected inputs and outputs, aka “**domain**” and “**range**”
- The **domain** of a function is the number and types a function accepts as input
- The **range** of a function is the output type(s) that the function returns

```
def double_num(num):
    """
    >>> double_num(42)
    84
    """
    # Domain: a number (eg int/float)
    # Range: a number (int/float)
    return num * 2
```

```
def sum_nums(nums):
    """
    >>> sum_nums([1, 2, 3])
    6
    """
    # Domain: a List of numbers
    # Range: a number
    out = 0
    for num in nums:
        out += num
    return out
```


Tip: function “domain”, “range”

- A handy tip about working with functions is to keep track of their expected inputs and outputs, aka “**domain**” and “**range**”
- The **domain** of a function is the number and types a function accepts as input
- The **range** of a function is the output type(s) that the function returns

```
def map_recurse(fn, seq):
    """
    >>> map_recurse(lambda x: x * 2, [1, 2, 3])
    [2, 4, 6]
    """
    # Domain: (arg1) function of one arg, (arg2) sequence
    # Range: a sequence
    if not seq:
        return []
    return [fn(seq[0])] + map_recurse(fn, seq[1:])
```

Question: what is the domain and range of this function?

Tip: function “domain”, “range”

- When writing Python code (particularly tree recursion), my advice is to **always keep track of and respect a function’s domain and range**!
- A common class of bugs (both in C88C, and in “the real world”) is:
  - (1) Calling a function with the wrong arg types, or
  - (2) Expecting a function returns type X, when it actually returns type Y

Tip: function “domain”, “range”

- Example: let’s examine a buggy implementation of map_deep()

```
def map_deep_buggy(fn, seq):
    # Domain: (arg1) function of one arg,
    #   (arg2) list of int OR nested list(s)
    # Range: list
    if not seq:
        return []
    else:
        first_val = map_deep_buggy(fn, seq[0])
        return first_val + map_deep_buggy(fn, seq[1:])
```

Possible types: number, List

Bug: the `first_val` recursive call violates `map_deep_buggy()`’s **domain**, as we may pass a NUMBER instead of a LIST as the second arg.

Question: I claim there is a bug in this code. Can you (1) identify the bug and (2) relate it to function domain and range?

This **domain violation** should ring alarm bells in your head that this implementation is wrong.

Tip: function “domain”, “range”

```
def map_deep_buggy(fn, seq):
    # Domain: (arg1) function of one arg,
    #   (arg2) list of int OR nested list(s)
    # Range: list
    if not seq:
        return []
    else:
        first_val = map_deep_buggy(fn, seq[0])
        return first_val + map_deep_buggy(fn, seq[1:])
```

Possible types: number, List

Explanation: We try to call `map_deep_buggy(fn, 1)`, which errors because we can’t subscript an integer: `1[0]` -> error.

This **domain violation** should ring alarm bells in your head that this implementation is wrong.

```
>>> map_deep_buggy(lambda x: x * 2, [1, [2, [3]]])
Traceback (most recent call last):
  File "C:\Users\Eric\c88c\lectures\lecture13\lecture13.py", line 147, in <module>
    print(map_deep_buggy(lambda x: x * 2, [1, [2, [3]]]))
  File "C:\Users\Eric\c88c\lectures\lecture13\lecture13.py", line 144, in map_deep_buggy
    first_val = map_deep_buggy(fn, seq[0])
  File "C:\Users\Eric\c88c\lectures\lecture13\lecture13.py", line 144, in map_deep_buggy
    first_val = map_deep_buggy(fn, seq[0])
TypeError: 'int' object is not subscriptable
```

Tip: learn how to read and interpret Python error messages. Useful not only in C88C, but also your future classes + career.

(Aside) Python types

- Python doesn't have a strict static system like "statically typed" languages like Java, C/C++. Instead, it's a dynamically typed language.
- **Consequence**: in Python, we can accidentally pass the wrong types into functions. This isn't possible in statically-typed languages like Java/C/C++

```
def map_shallow(fn, lst):
    # Domain: (arg1) function of one arg
    #   (arg2) list
    # Range: list
    if not lst:
        return []
    return [fn(lst[0])] + map_shallow(fn, lst[1:])

>>> map_shallow(lambda x: x * 2, 42)
…
line 144, in map_deep_buggy
    first_val = map_deep_buggy(fn, seq[0])
TypeError: 'int' object is not subscriptable
```


(Aside) Python type annotation hints

- In your future Python work, you may enjoy annotating functions with their domain/range via python3.5+'s built-in type-annotation hints (typing package)

```
from typing import List, Callable

def map_shallow_typehint(fn: Callable, lst: List[int]) -> List:
    # Domain: (arg1) function of one arg
    #   (arg2) list
    # Range: list
    if not lst:
        return []
    return [fn(lst[0])] + map_shallow_typehint(fn, lst[1:])

>>> map_shallow(lambda x: x * 2, 42)
…
line 144, in map_deep_buggy
    first_val = map_deep_buggy(fn, seq[0])
TypeError: 'int' object is not subscriptable
```

Take care: type-hints don't actually do anything, the above error will still happen. Aka a "fancy comment"

`List[int]` means "a list of integers. `List` implies "a list with elements of unknown type"

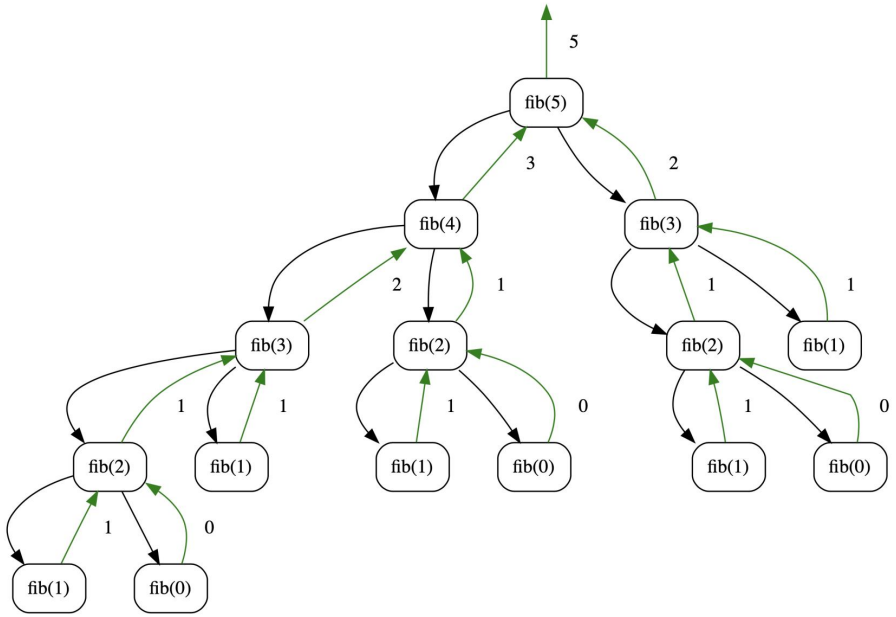
Callable: if you know the exact input/output types ahead of time, you can do:
`Callable[[input_type_1, …], out_type]`

Some IDE's (eg VSCode, PyCharm) can interpret these type-hints and warn you if you are passing in the wrong types!

Note: this slide is NOT required for this class.

Today's Lecture. Any questions?

- Tree recursion
  - aka recursive functions that make multiple recursive calls per "level"
  - call stack looks like a "tree"
- Function domain/range



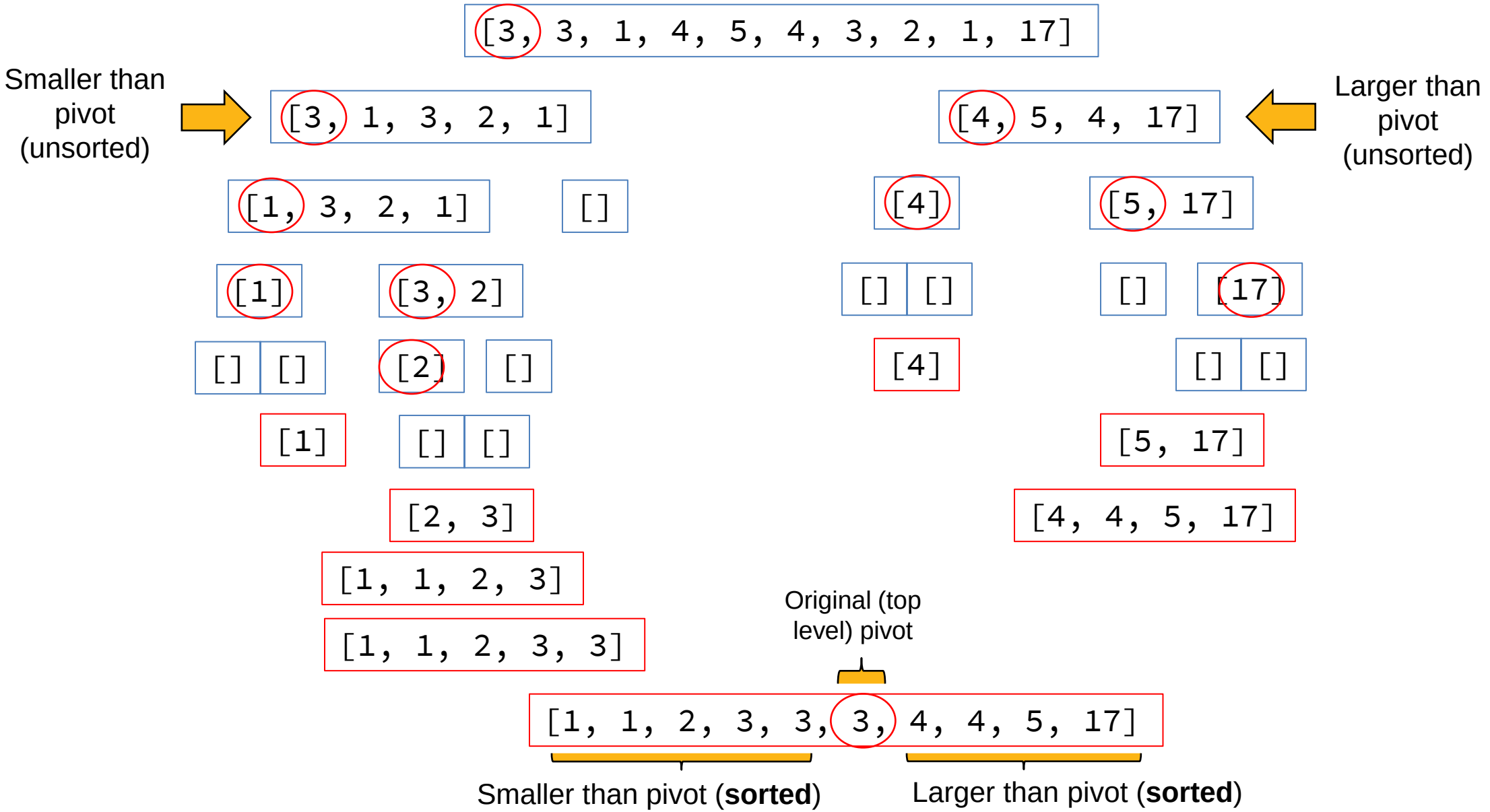
Computational Structures in Data Science

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

- https://recursion.vercel.app/