Lecture 7: Tree Recursion

Brian Hou
June 29, 2016

Announcements

- Project 1 is due tomorrow, +1 EC point if submitted today
  - Run **ok --submit** to check against hidden tests
  - Check your submission at ok.cs61a.org
  - Invite your partner (watch this video)
- Homework 2 is due today, Homework 1 solutions uploaded
- Quiz 2 is tomorrow at the beginning of lecture
  - If you have an alternate time or are not enrolled in the class, please arrive at 11:45 am
- Week 2 checkoff must be done in lab today or tomorrow
  - Talk about hw01, lab02, lab03 with a lab assistant
- Alternate Exam Request: goo.gl/forms/FDQix4I5dNXPQDgw2

Hog Contest Rules

- Up to two people submit one entry; max one entry per person
- Your score is the number of entries against which you win more than 50.00001% of the time
- All strategies must be deterministic, pure functions of the current player and opponent scores
- Top 3 entries will receive EC
- The real prize: honor and glory
  - Also: bragging rights

Ready? cs61a.org/proj/hog_contest

Roadmap

Introduction

Functions

Data

Mutability

Objects

Interpretation

Paradigms

Applications

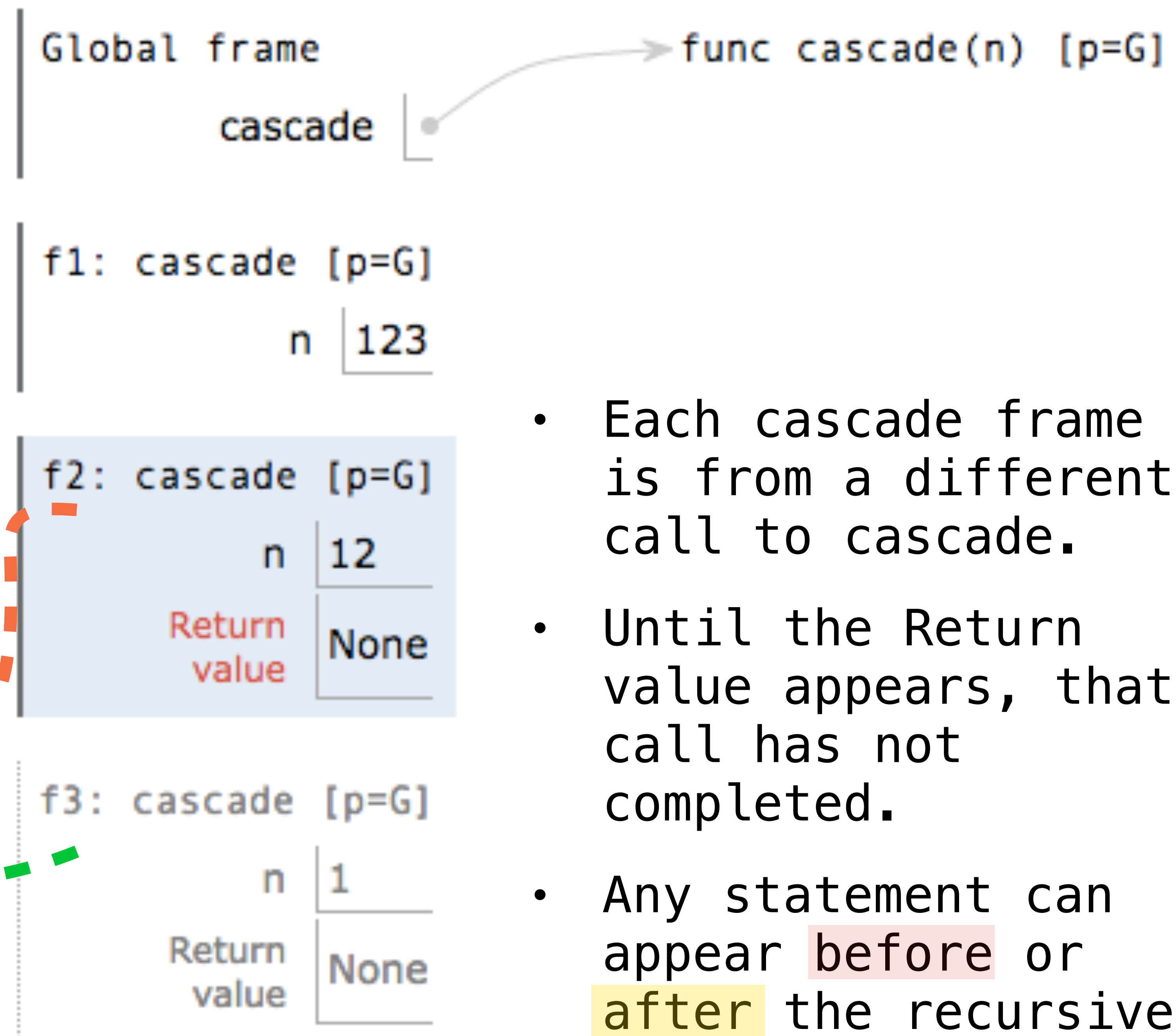
- This week (Functions), the goals are:
  - To understand the idea of *functional abstraction*
  - To study this idea through:
    - higher-order functions
    - recursion
    - orders of growth

Recursion

The Cascade Function (demo)

```
def cascade(n):
    if n < 10:
        print(n)
    else:
        print(n)
        cascade(n//10)
        print(n)

cascade(123)
```



Output

123
12
1
12

- Each cascade frame is from a different call to cascade.
- Until the Return value appears, that call has not completed.
- Any statement can appear before or after the recursive call.

Two Definitions of Cascade (demo)

```
def cascade(n):
    if n < 10:
        print(n)
    else:
        print(n)
        cascade(n // 10)
        print(n)
```

```
def cascade(n):
    print(n)
    if n >= 10:
        cascade(n // 10)
        print(n)
```

- If two implementations are equally clear, then shorter is usually better
- In this case, the longer implementation is more clear (to me)
- When learning to write recursive functions, put base cases first

Inverse Cascade

Output

```
1
12
123
1234
123
12
1
```

```python
def inverse_cascade(n):
    grow(n)
    print(n)
    shrink(n)
```

```python
def f_then_g(f, g, n):
    if n:
        f(n)
        g(n)
```

```python
grow   = lambda n: f_then_g(                    )

shrink = lambda n: f_then_g(                    )
```

Fibonacci

The Fibonacci Sequence

n:	0,	1,	2,	3,	4,	5,	6,	7,	8,	... ,	35
fib(n):	0,	1,	1,	2,	3,	5,	8,	13,	21,	... ,	9,227,465

The Fibonacci Sequence

n:	0,	1,	2,	3,	4,	5,	6,	7,	8,
fib(n):	0,	1,	1,	2,	3,	5,	8,	13,	21,

```
def fib(n):
    pred, curr = 0, 1
    k = 1
    while k < n:
        pred, curr = curr, pred + curr
        k += 1
    return curr
```

The next Fibonacci number is the sum of the two previous Fibonacci numbers

The Fibonacci Sequence

n:	0, 1, 2, 3, 4, 5, 6, 7, 8,
fib(n):	0, 1, 1, 2, 3, 5, 8, 13, 21,

```
def fib(n):
    if n == 0:
        return 0
    pred, curr = 0, 1
    k = 1
    while k < n:
        pred, curr = curr, pred + curr
        k += 1
    return curr
```

This correction was made on July 3 at 10PM

The next Fibonacci number is the sum of the two previous Fibonacci numbers

The Fibonacci Sequence

n:	0,	1,	2,	3,	4,	5,	6,	7,	8,
fib(n):	0,	1,	1,	2,	3,	5,	8,	13,	21,

```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-2) + fib(n-1)
```

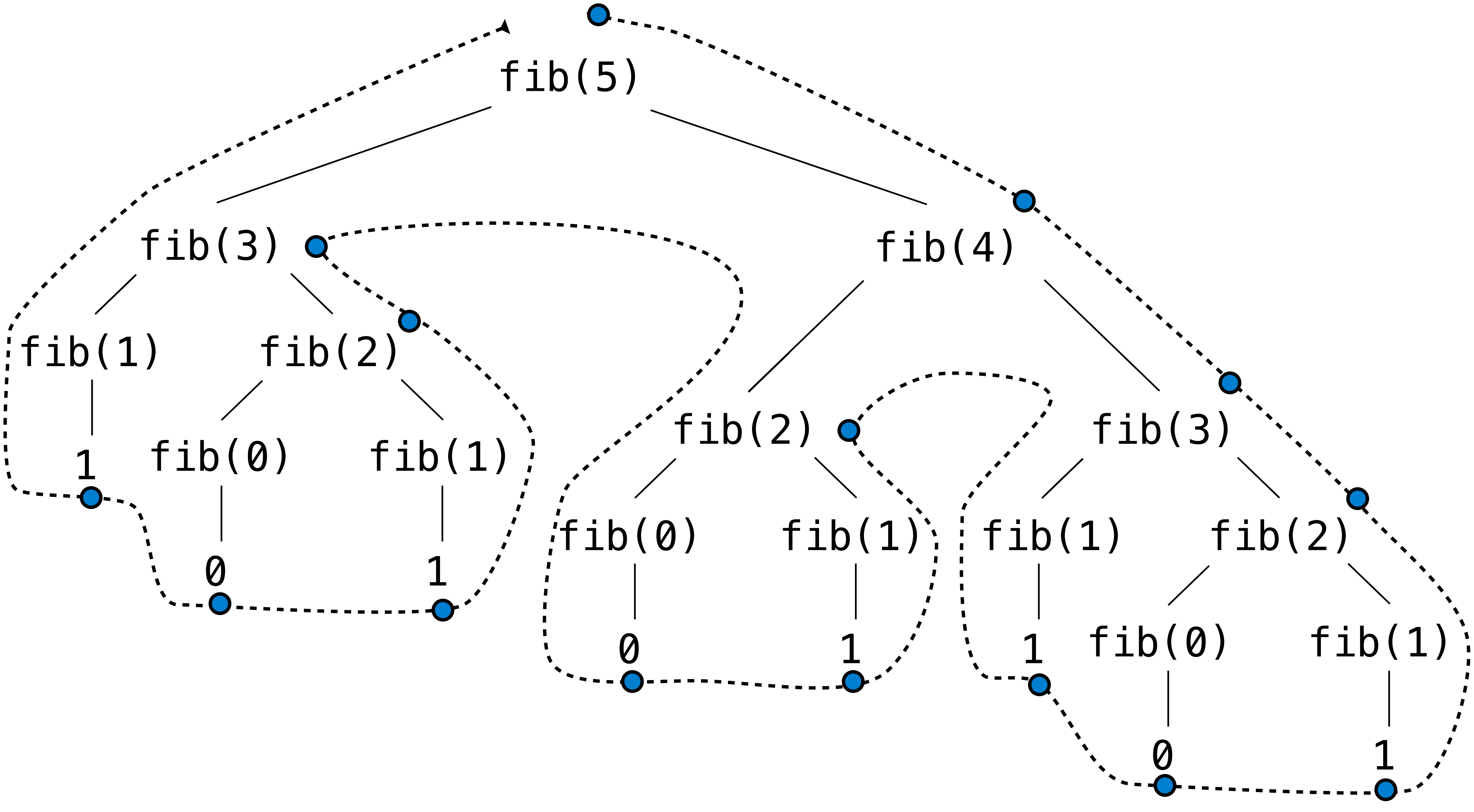
The next Fibonacci number is the sum of the two previous Fibonacci numbers

Tree Recursion

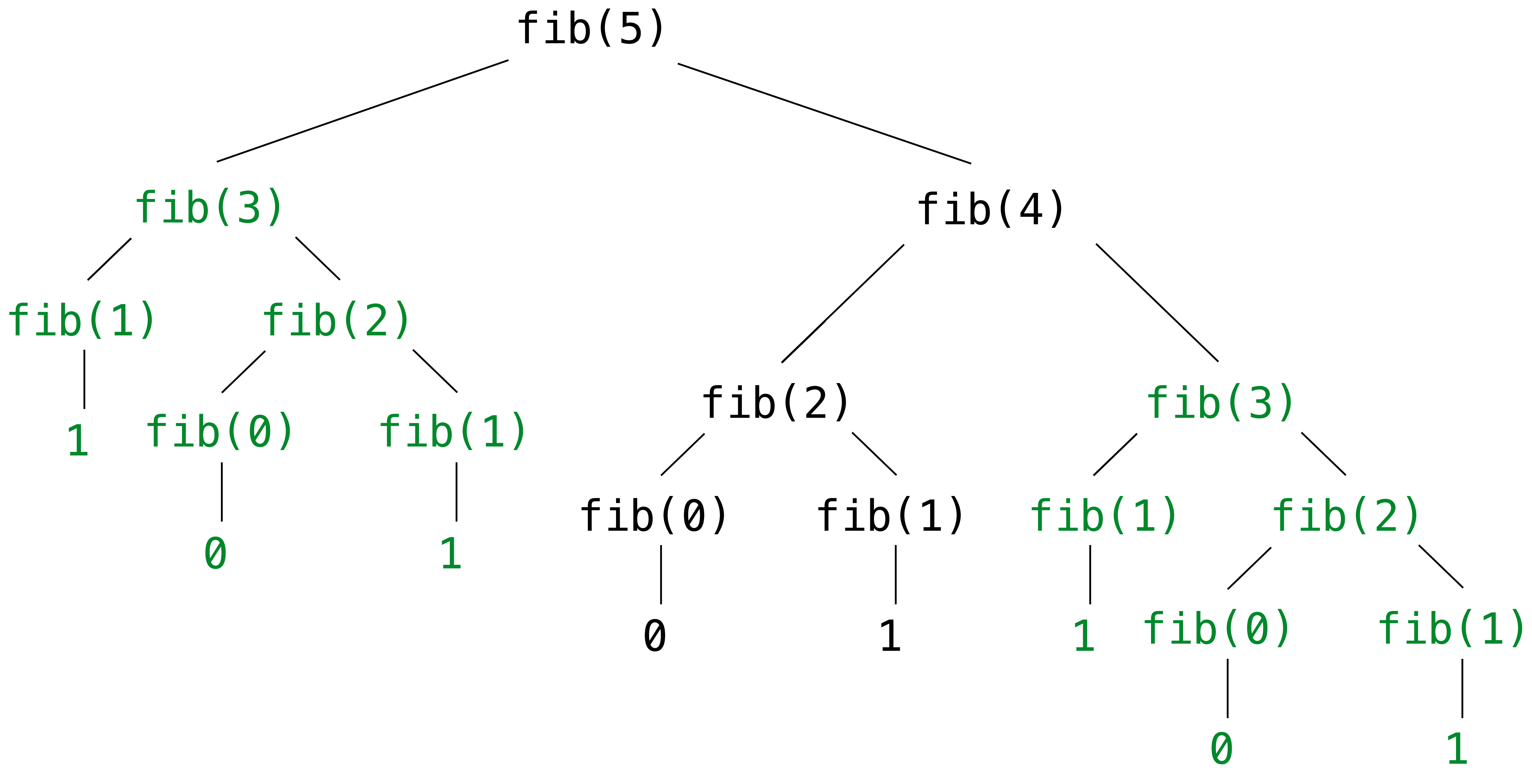
Tree-shaped processes arise whenever executing the body of a recursive function makes more than one recursive call

```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-2) + fib(n-1)
```

A Tree-Recursive Process (demo)



A Tree-Recursive Process



Break!

Counting Partitions

Counting Partitions

The number of partitions of a positive integer n, using parts up to size m, is the number of ways in which n can be expressed as the sum of positive integer parts up to m in increasing order.

count_partitions(6, 4)

How many different ways can I give out 6 pieces of chocolate if nobody can have more than 4 pieces?

2 + 4 = 6
1 + 1 + 4 = 6

3 + 3 = 6
1 + 2 + 3 = 6
1 + 1 + 1 + 3 = 6

2 + 2 + 2 = 6
1 + 1 + 2 + 2 = 6
1 + 1 + 1 + 1 + 2 = 6

1 + 1 + 1 + 1 + 1 + 1 = 6

Counting Partitions

The number of partitions of a positive integer n, using parts up to size m, is the number of ways in which n can be expressed as the sum of positive integer parts up to m in increasing order.

2 + 4 = 6

1 + 1 + 4 = 6

3 + 3 = 6

1 + 2 + 3 = 6

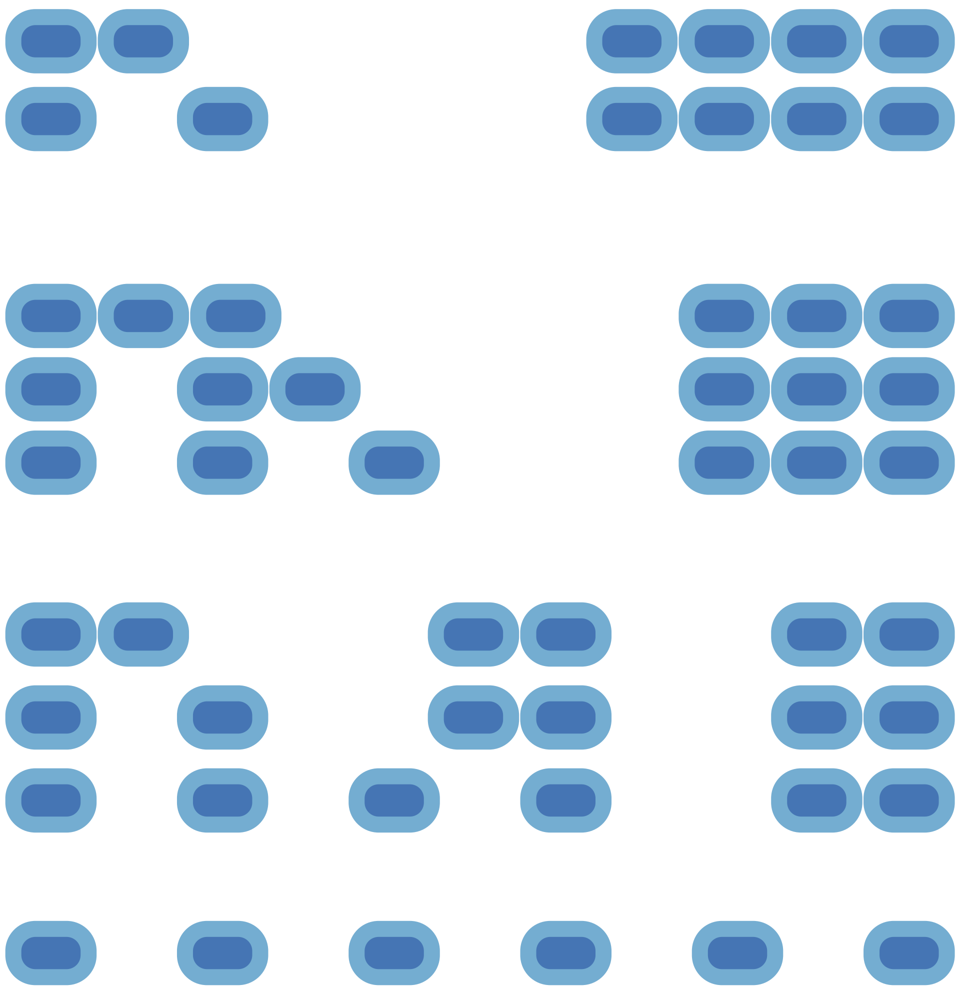
1 + 1 + 1 + 3 = 6

2 + 2 + 2 = 6

1 + 1 + 2 + 2 = 6

1 + 1 + 1 + 1 + 2 = 6

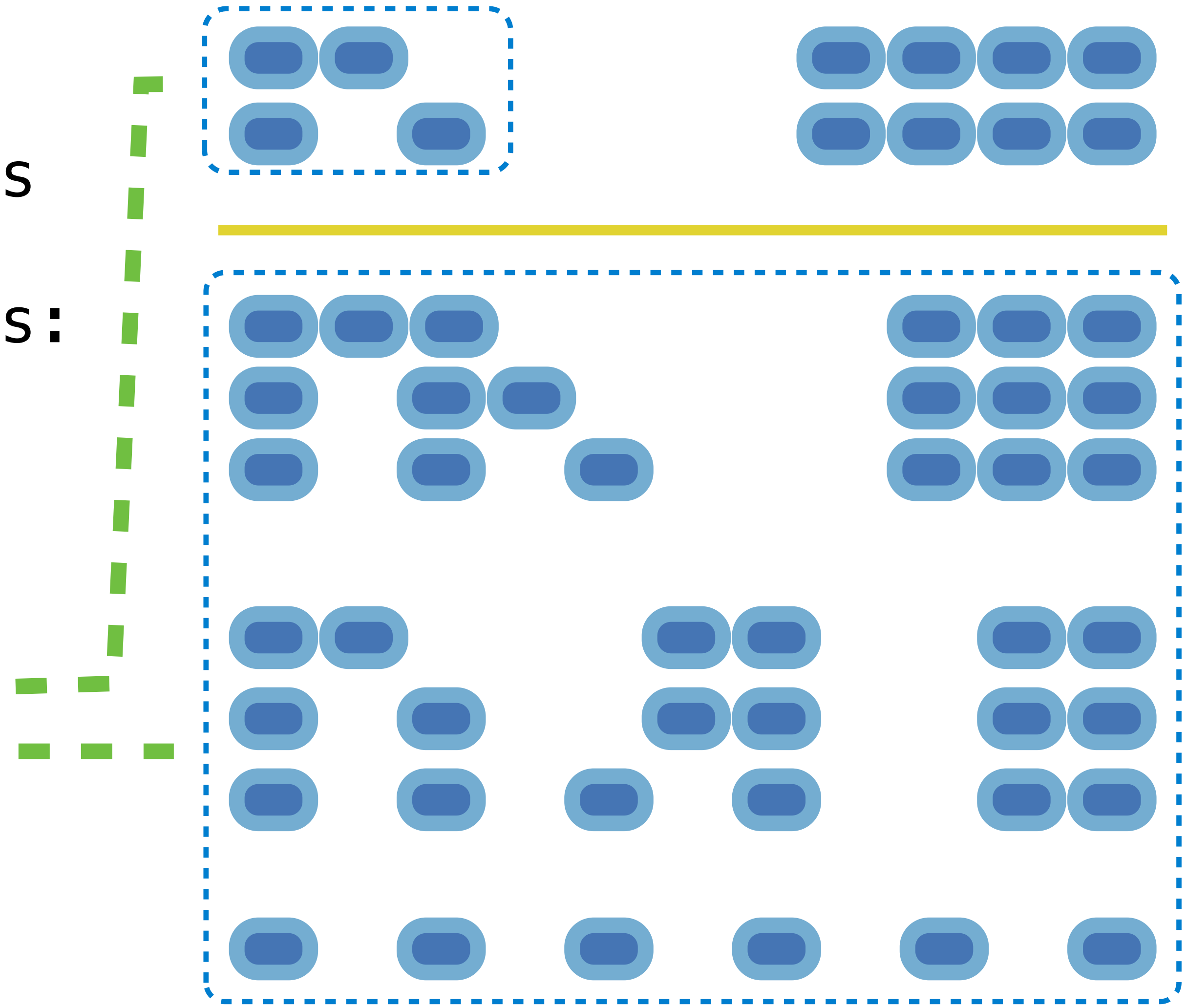
1 + 1 + 1 + 1 + 1 + 1 = 6



Counting Partitions

The number of partitions of a positive integer n, using parts up to size m, is the number of ways in which n can be expressed as the sum of positive integer parts up to m in increasing order.

- Recursive decomposition: finding simpler instances of the problem.
- Explore two possibilities:
  - Use at least one 4
  - Don't use any 4
- Solve two simpler problems:
  - count_partitions(2, 4)
  - count_partitions(6, 3)
- Tree recursion often involves exploring different choices.



Counting Partitions

The number of partitions of a positive integer n, using parts up to size m, is the number of ways in which n can be expressed as the sum of positive integer parts up to m in increasing order.

- Recursive decomposition: finding simpler instances of the problem.
- Explore two possibilities:
  - Use at least one 4
  - Don't use any 4
- Solve two simpler problems:
  - count_partitions(2, 4)
  - count_partitions(6, 3)
- Tree recursion often involves exploring different choices.

```
def count_partitions(n, m):
    if n == 0:
        return 1
    elif n < 0:
        return 0
    elif m == 0:
        return 0
    else:
        with_m = count_partitions(n-m, m)
        without_m = count_partitions(n, m-1)
        return with_m + without_m
```