Tree Recursion

Announcements

- Drop deadlines:
  - June 30th with tuition refund
  - After the deadline, you can still drop the course but will receive summer tuition credit
- Lab 2 is due today
- HW 1 is due today
  - Homework Recovery allows you to recover 1 point
- Hog Checkpoint is due tomorrow
  - Project Party from 3-5:30 pm today in Warren 101B
- Still have a few spaces in tutoring sections: tutorials.cs61a.org
- Using Resources on Ed

Order of Recursive Calls

The Cascade Function

(Demo)

```
1  def cascade(n):
2      if n < 10:
3          print(n)
4      else:
5          print(n)
6          cascade(n//10)
7          print(n)
8
9  cascade(123)
```

Global frame

cascade → func cascade(n) [parent=Global]

f1: cascade [parent=Global]

n 123

f2: cascade [parent=Global]

n 12

Return value None

f3: cascade [parent=Global]

n 1

Return value None

Program output:

```
123
12
1
12
123
```

- 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.

http://pythontutor.com/composingprograms.html#code=def%20cascade%28n%29%3A%20%20%20%20%0A%20%20%20%20if%20n%20%3C%2010%3A%20%20%20%20%20%20%20%20%0A%20%20%20%20%20%20%20%20print%28n%29%20%20%20%20%0A%20%20%20%20else%3A%20%20%20%20%20%20%20%20%0A%20%20%20%20%20%20%20%20print%28n%29%20%20%20%20%20%20%20%20%0A%20%20%20%20%20%20%20%20cascade%28n//10%29%20%20%20%20%20%20%20%20%20%0A%20%20%20%20%20%20%20%20print%28n%29%20%20%20%20%20%20%20%20%0A%20%20%20%20%20%20%20%20%0Acascade%28123%29&cumulative=true&curInstr=0&mode=display&origin=composingprograms.js&py=3&rawInputLstJSON=%5B%5D

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 (at least to me)
- When learning to write recursive functions, put the base cases first
- Both are recursive functions, even though only the first has typical structure

Tree Recursion

Tree Recursion

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

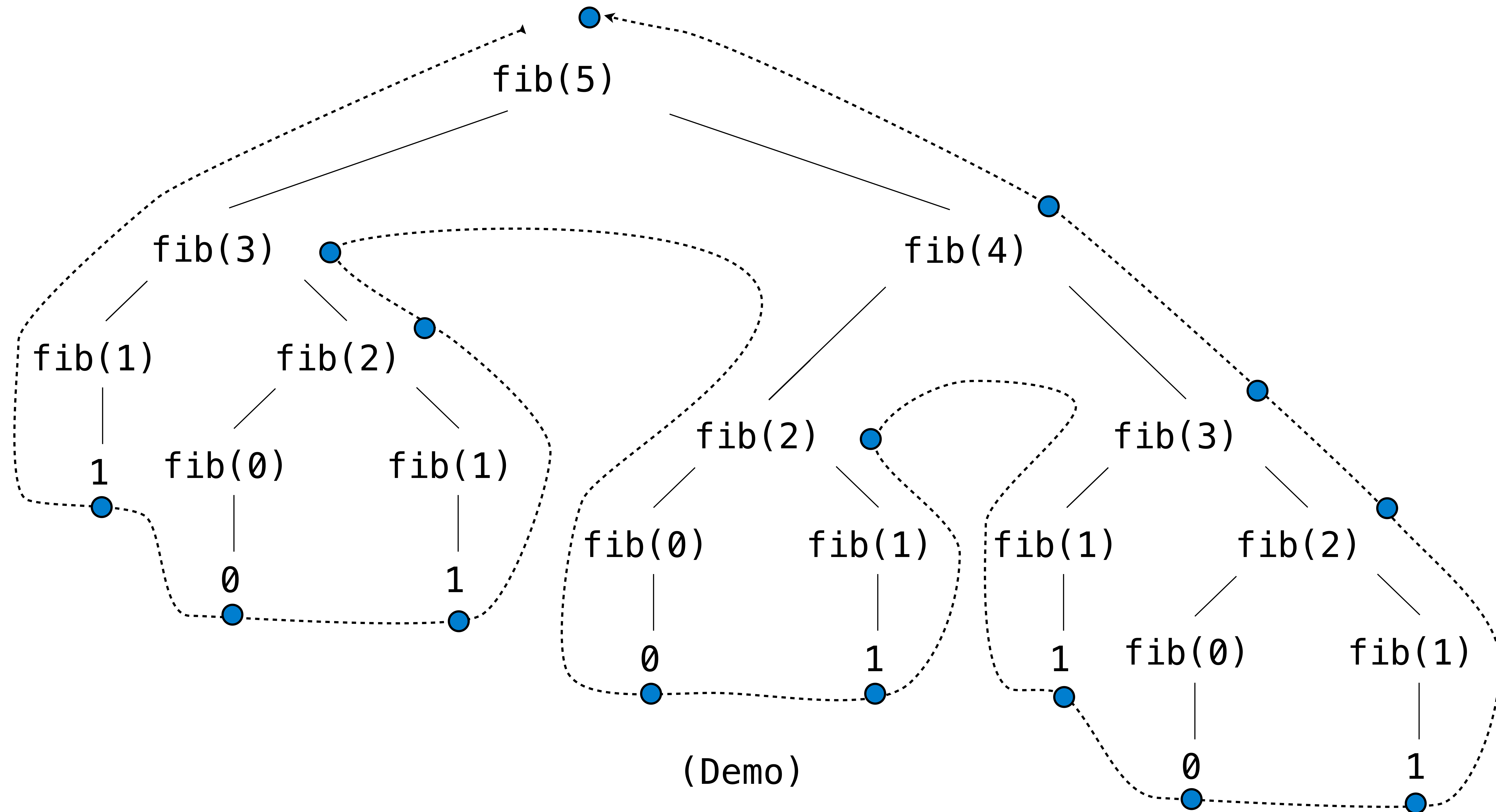
n:	0,	1,	2,	3,	4,	5,	6,	7,	8,	... ,	35
fib(n):	0,	1,	1,	2,	3,	5,	8,	13,	21,	... ,	9,227,465

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

http://en.wikipedia.org/wiki/File:Fibonacci.jpg

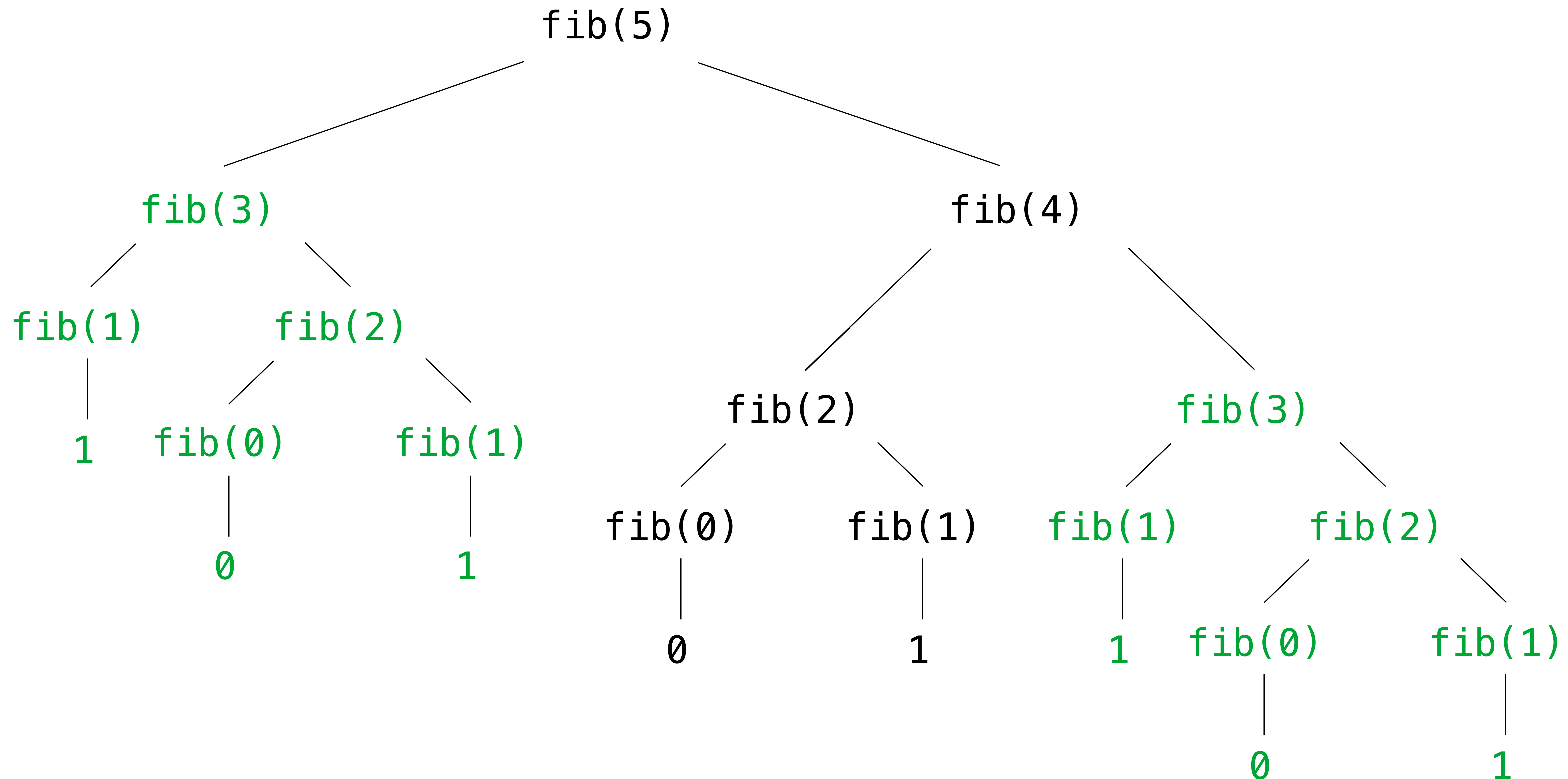
A Tree-Recursive Process

The computational process of fib evolves into a tree structure



Repetition in Tree-Recursive Computation

This process is highly repetitive; fib is called on the same argument multiple times



(We will speed up this computation dramatically next week by remembering results)

Recursive vs. Iterative

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

Recursive Approach

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

Iterative Approach

- Visually represents the recursive definition
- Computation happens in MANY frames
- Number of operations increases exponentially to N

- Easy to follow calculations
- Computation in one frame
- Number of operations is directly proportional to N

Break

Example: 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)
```

2 + 4 = 6

1 + 1 + 4 = 6

3 + 3 = 6

1 + 2 + 3 = 6

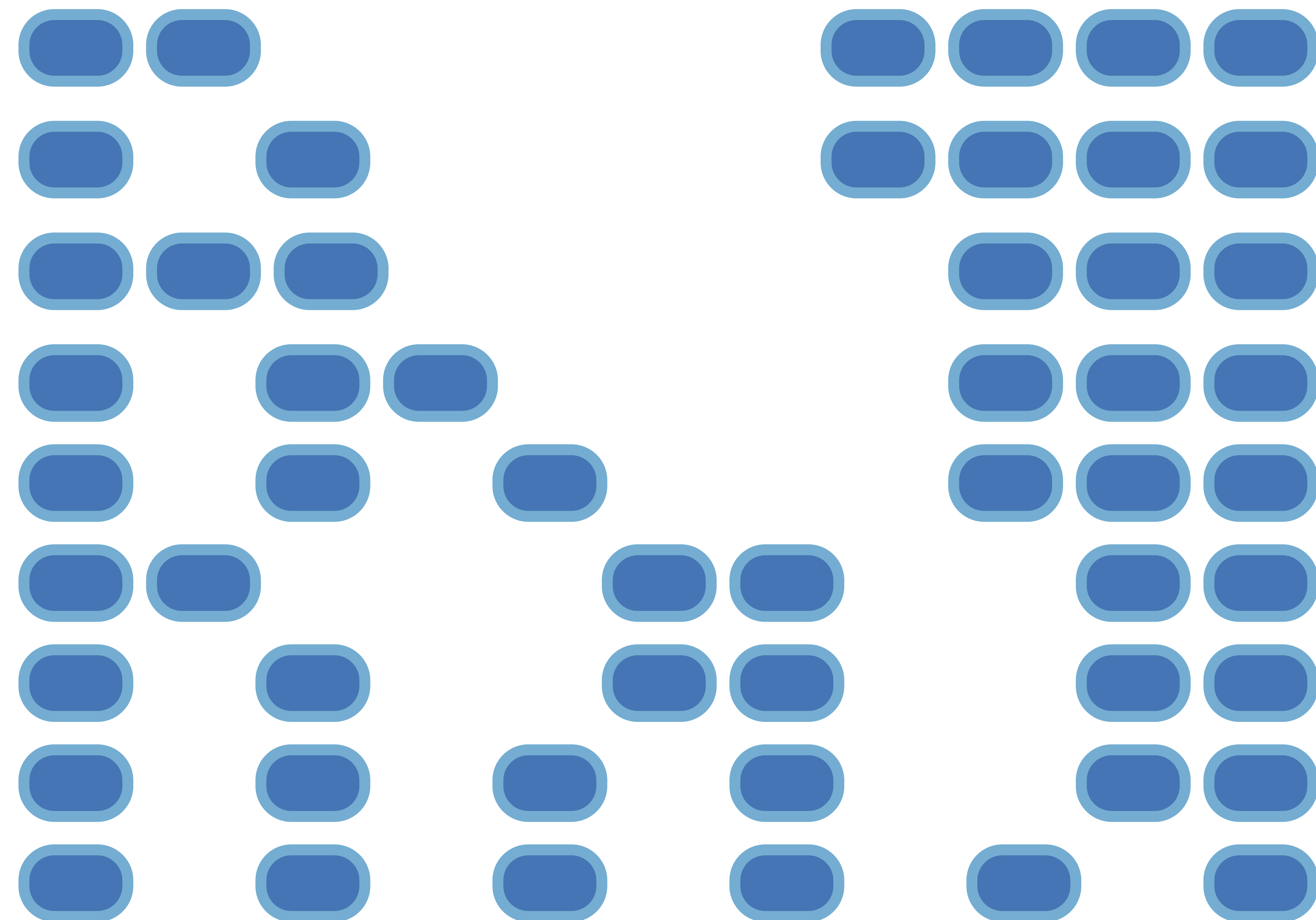
1 + 1 + 1 + 3 = 6

2 + 2 + 2 = 6

1 + 1 + 2 + 2 = 6

1 + 1 + 1 + 1 + 2 = 6

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

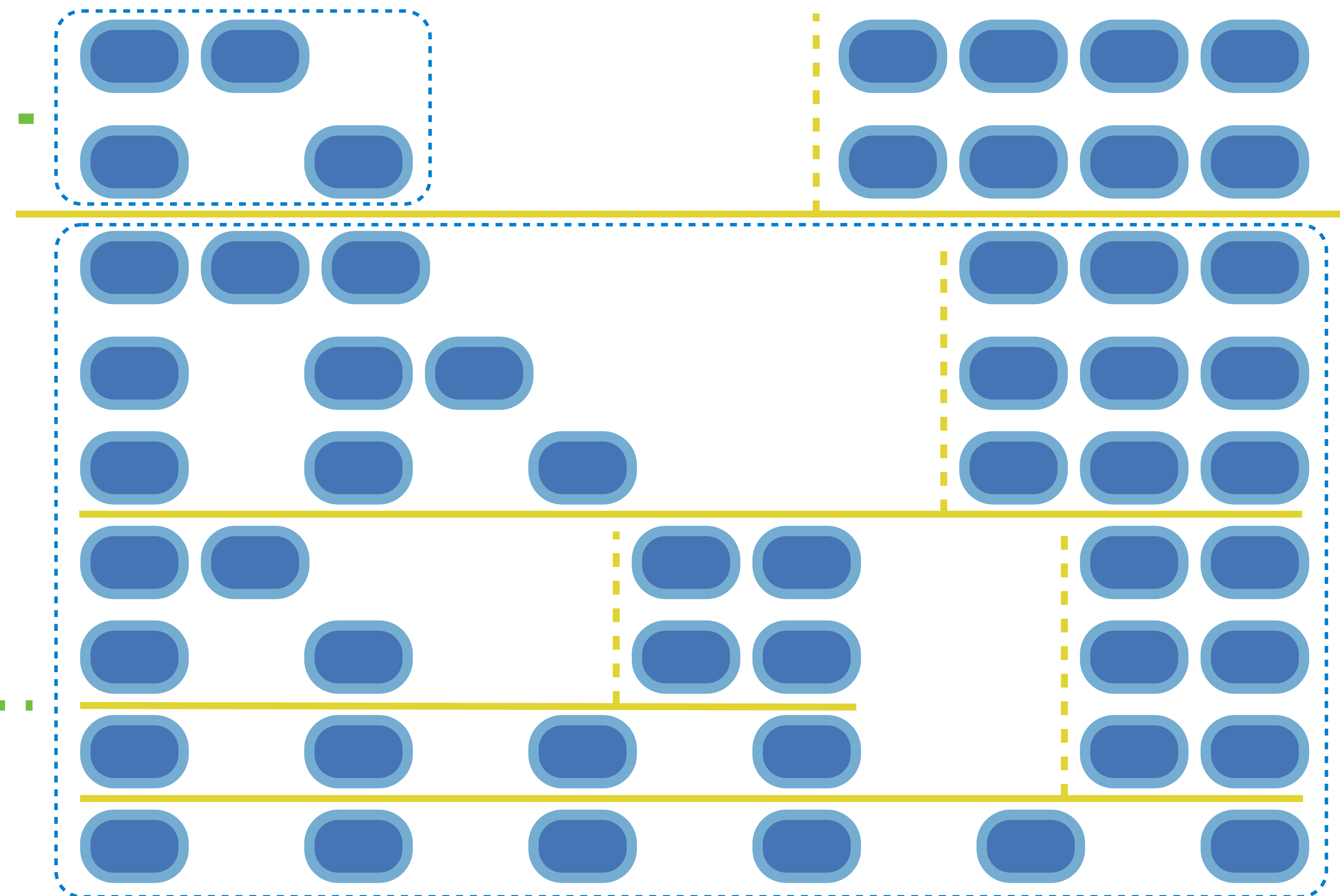


Counting Partitions

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 non-decreasing order.

count_partitions(6, 4)

- 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) → with_m = count_partitions(n-m, m)
  - count_partitions(6, 3) → without_m = count_partitions(n, m-1)
- 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
```

(Demo)

pythontutor.com/composingprograms.html#code=def%20count_partitions%28n,%20m%29%3A%0A%20%20%20%20if%20n%20%3D%3D%200%3A%0A%20%20%20%20%20%20%20%20return%201%0A%20%20%20%20elif%20n%20<%200%3A%0A%20%20%20%20%20%20%20%20return%200%0A%20%20%20%20elif%20m%20%3D%3D%200%3A%0A%20%20%20%20%20%20%20%20return%200%0A%20%20%20%20else%3A%0A%20%20%20%20%20%20%20%20with_m%20%3D%20count_partitions%28n-m,%20m%29%20%0A%20%20%20%20%20%20%20%20without_m%20%3D%20count_partitions%28n,%20m-1%29%0A%20%20%20%20%20%20%20%20return%20with_m%20%2B%20without_m%0A%20%20%20%20%20%20%20%20%0Aresult%20%3D%20count_partitions%285,%203%29%0A%0A#%201%20%2B%201%20%2B%201%20%2B%201%20%2B%201%20%3D%205%0A#%201%20%2B%201%20%2B%201%20%2B%202%20%2B%20%20%20%3D%205%0A#%201%20%2B%202%20%2B%202%20%2B%20%20%20%20%20%20%20%3D%205%0A#%201%20%2B%201%20%2B%203%20%2B%20%20%20%20%20%20%20%3D%205%0A#%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%2B%203%20%2B%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%3D%205&mode=display&origin=composingprograms.js&cumulative=false&py=3&rawInputLstJSON=[]&curInstr=0

Summary

- The order of calls matters
  - You must return from the function called before moving onto the next line of code
- A tree recursive functions contains multiple recursive calls within its body, each modeling a specific decision
- Base cases may not always be apparent, and sometimes working through recursive calls can help you figure them out
- Recursion is not efficient and in the process, the computer recomputes the same values again
  - We’ll learn next week how to overcome that