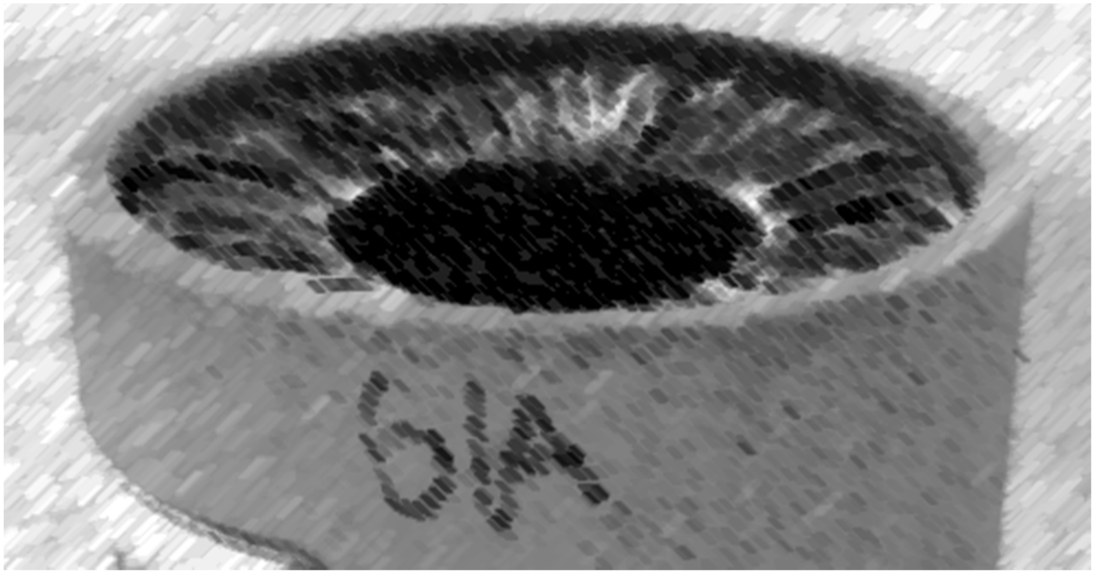


CS61A Lecture 24

Amir Kamil and Hamilton Nguyen

UC Berkeley

March 18, 2013

Announcements

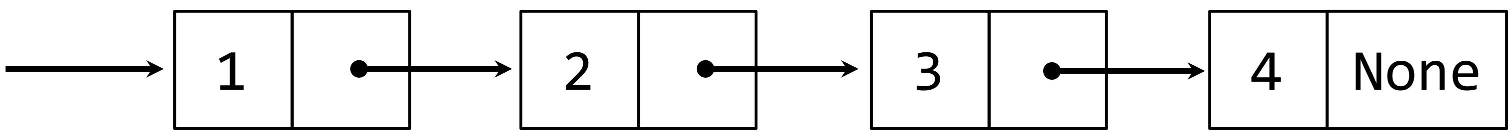
- Ants project due tonight
- HW8 due Wednesday at 7pm
- Midterm 2 Thursday at 7pm
  - See course website for more information

Closure Property of Data

A tuple can contain another tuple as an element.

Pairs are sufficient to represent sequences.

Recursive list representation of the sequence 1, 2, 3, 4:



Recursive lists are recursive: the rest of the list is a list.

Nested pairs (old): `(1, (2, (3, (4, None))))`

Rlist class (new): `Rlist(1, Rlist(2, Rlist(3, Rlist(4))))`

Recursive List Class

Methods can be recursive as well!

```
class Rlist(object):
    class EmptyList(object):
        def __len__(self):
            return 0
    empty = EmptyList()
    def __init__(self, first, rest=empty):
        self.first = first
        self.rest = rest
    def __len__(self):
        return 1 + len(self.rest)
    def __getitem__(self, i):
        if i == 0:
            return self.first
        return self.rest[i - 1]
```

There's the base case!

Yes, this call is recursive

Recursive Operations on Rlists

Recursive list processing almost always involves a recursive call on the rest of the list.

```
>>> s = Rlist(1, Rlist(2, Rlist(3)))

>>> s.rest
Rlist(2, Rlist(3))

>>> extend_rlist(s.rest, s)
Rlist(2, Rlist(3, Rlist(1, Rlist(2, Rlist(3)))))
```

```
def extend_rlist(s1, s2):
    if s1 is Rlist.empty:
        return s2
    return Rlist(s1.first, extend_rlist(s1.rest, s2))
```

Map and Filter on Rlists

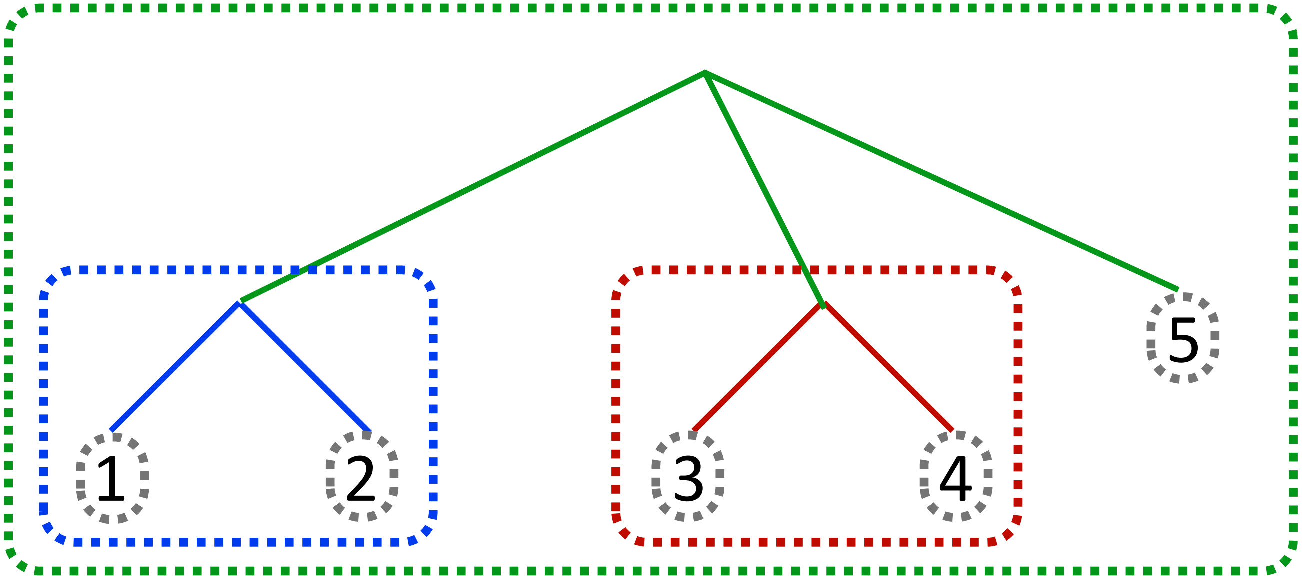
We want operations on a whole list, not an element at a time.

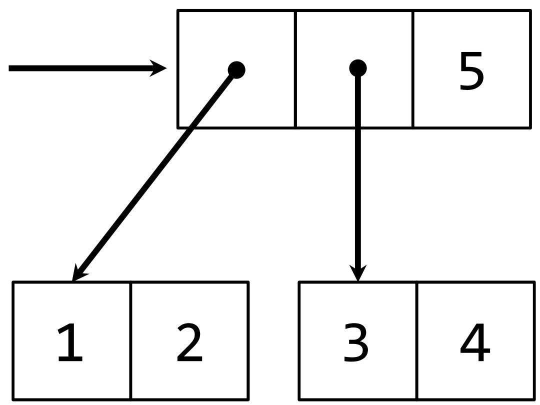
```
def map_rlist(s, fn):
    if s is Rlist.empty:
        return s
    return Rlist(fn(s.first), map_rlist(s.rest, fn))


def filter_rlist(s, fn):
    if s is Rlist.empty:
        return s
    rest = filter_rlist(s.rest, fn)
    if fn(s.first):
        return Rlist(s.first, rest)
    return rest
```

Tree Structured Data

Nested Sequences are Hierarchical Structures.

`((1, 2), (3, 4), 5)`





In every tree, a vast forest

Example: http://goo.gl/0h6n5

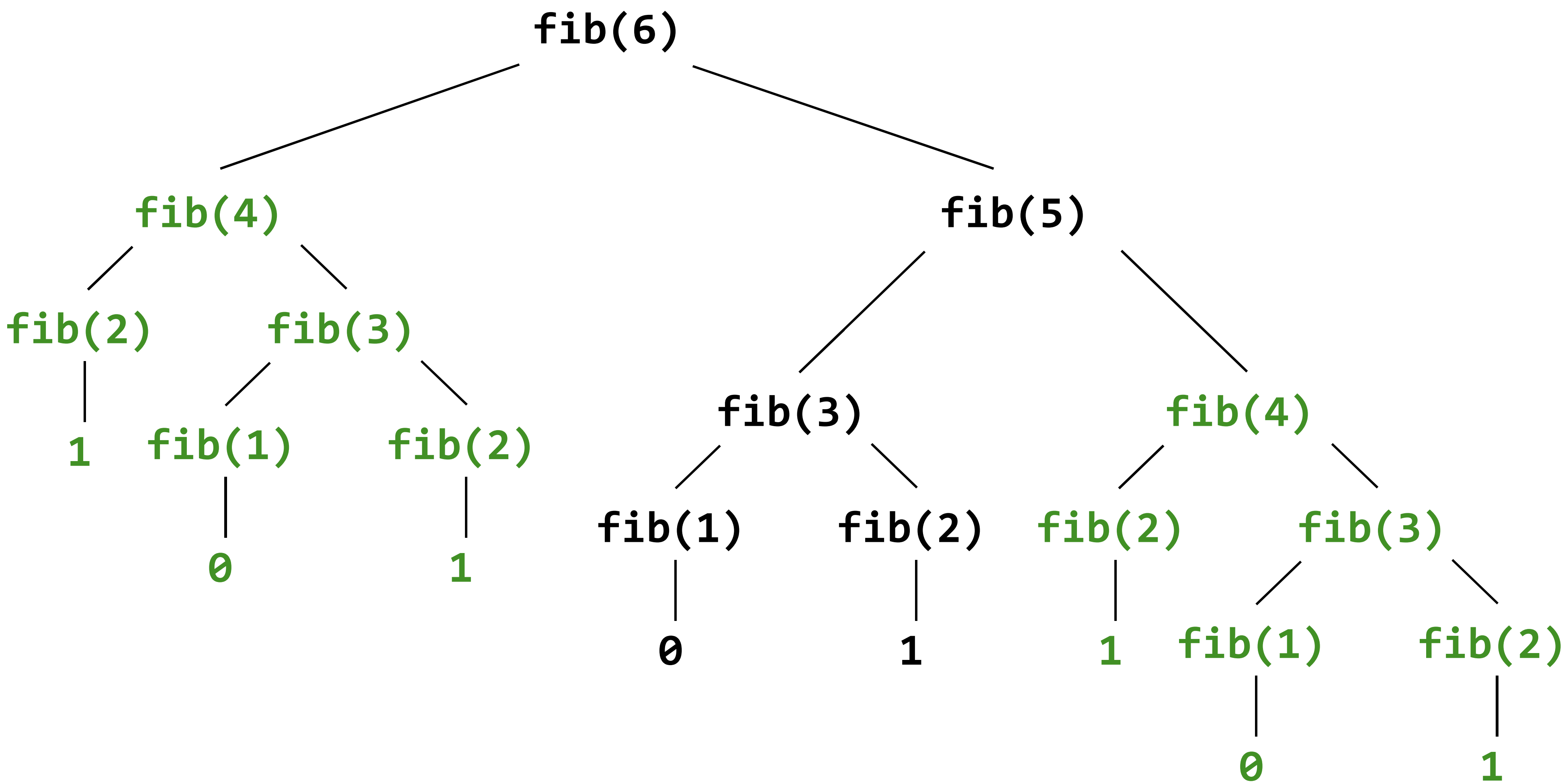
Recursive Tree Processing

Tree operations typically make recursive calls on branches

```
def count_leaves(tree):
    if type(tree) != tuple:
        return 1
    return sum(map(count_leaves, tree))


def map_tree(tree, fn):
    if type(tree) != tuple:
        return fn(tree)
    return tuple(map_tree(branch, fn)
                 for branch in tree)
```

Trees with Internal Node Values

Trees can have values at internal nodes as well as their leaves.



The Consumption of Time

Implementations of the same functional abstraction can require different amounts of time to compute their result.

```
def count_factors(n):

    factors = 0
    for k in range(1, n + 1):
        if n % k == 0:
            factors += 1
    return factors
```

Time (remainders)

n

```
    sqrt_n = sqrt(n)
    k, factors = 1, 0
    while k < sqrt_n:
        if n % k == 0:
            factors += 2
        k += 1
    if k * k == n:
        factors += 1
    return factors
```

$\lfloor \sqrt{n} \rfloor$

Order of Growth

A method for bounding the resources used by a function as the "size" of a problem increases

n: size of the problem

R(n): Measurement of some resource used (time or space)

$$R(n) = \Theta(f(n))$$

means that there are positive constants k_1 and k_2 such that

$$k_1 \cdot f(n) \leq R(n) \leq k_2 \cdot f(n)$$

for sufficiently large values of ***n***.

Constant Time: $\Theta(1)$

Time does **not** depend on input size.

```
def g(n):
    return 42

def foo(n):
    baz = 7
    if n > 5:
        baz += 5
    return baz

def is_even(n):
    return n % 2 == 0
```

Iteration vs. Tree Recursion (Time)

Iterative and recursive implementations are not the same.

Time

```
def fib_iter(n):
    prev, curr = 1, 0
    for _ in range(n - 1):
        prev, curr = curr, prev + curr
    return curr
```

$\Theta(n)$

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

$\Theta(\phi^n)$

Next time, we will see how to make recursive version faster.

The Consumption of Time

Implementations of the same functional abstraction can require different amounts of time to compute their result.

```
def count_factors(n):

    factors = 0
    for k in range(1, n + 1):
        if n % k == 0:
            factors += 1
    return factors
```

Time

$\Theta(n)$

```
    sqrt_n = sqrt(n)
    k, factors = 1, 0
    while k < sqrt_n:
        if n % k == 0:
            factors += 2
        k += 1
    if k * k == n:
        factors += 1
    return factors
```

$\Theta(\sqrt{n})$

Exponentiation

Goal: one more multiplication lets us double the problem size.

```
def exp(b, n):
    if n == 0:
        return 1
    return b * exp(b, n - 1)
```

$$b^n = \begin{cases} 1 & \text{if } n = 0 \\ b \cdot b^{n-1} & \text{otherwise} \end{cases}$$

```
def square(x):
    return x * x

def fast_exp(b, n):
    if n == 0:
        return 1
    elif n % 2 == 0:
        return square(fast_exp(b, n // 2))
    else:
        return b * fast_exp(b, n - 1)
```

$$b^n = \begin{cases} 1 & \text{if } n = 0 \\ (b^{\frac{1}{2}n})^2 & \text{if } n \text{ is even} \\ b \cdot b^{n-1} & \text{if } n \text{ is odd} \end{cases}$$

Exponentiation

Goal: one more multiplication lets us double the problem size.

```
def exp(b, n):
    if n == 0:
        return 1
    return b * exp(b, n - 1)


def square(x):
    return x * x

def fast_exp(b, n):
    if n == 0:
        return 1
    elif n % 2 == 0:
        return square(fast_exp(b, n // 2))
    else:
        return b * fast_exp(b, n - 1)
```

	Time	Space
exp	$\Theta(n)$	$\Theta(n)$
fast_exp	$\Theta(\log n)$	$\Theta(\log n)$

The Consumption of Space

Which environment frames do we need to keep during evaluation?

Each step of evaluation has a set of **active** environments.

Values and frames in active environments consume memory.

Memory used for other values and frames can be reclaimed.

Active environments:

- Environments for any statements currently being executed
- Parent environments of functions named in active environments

The Consumption of Space

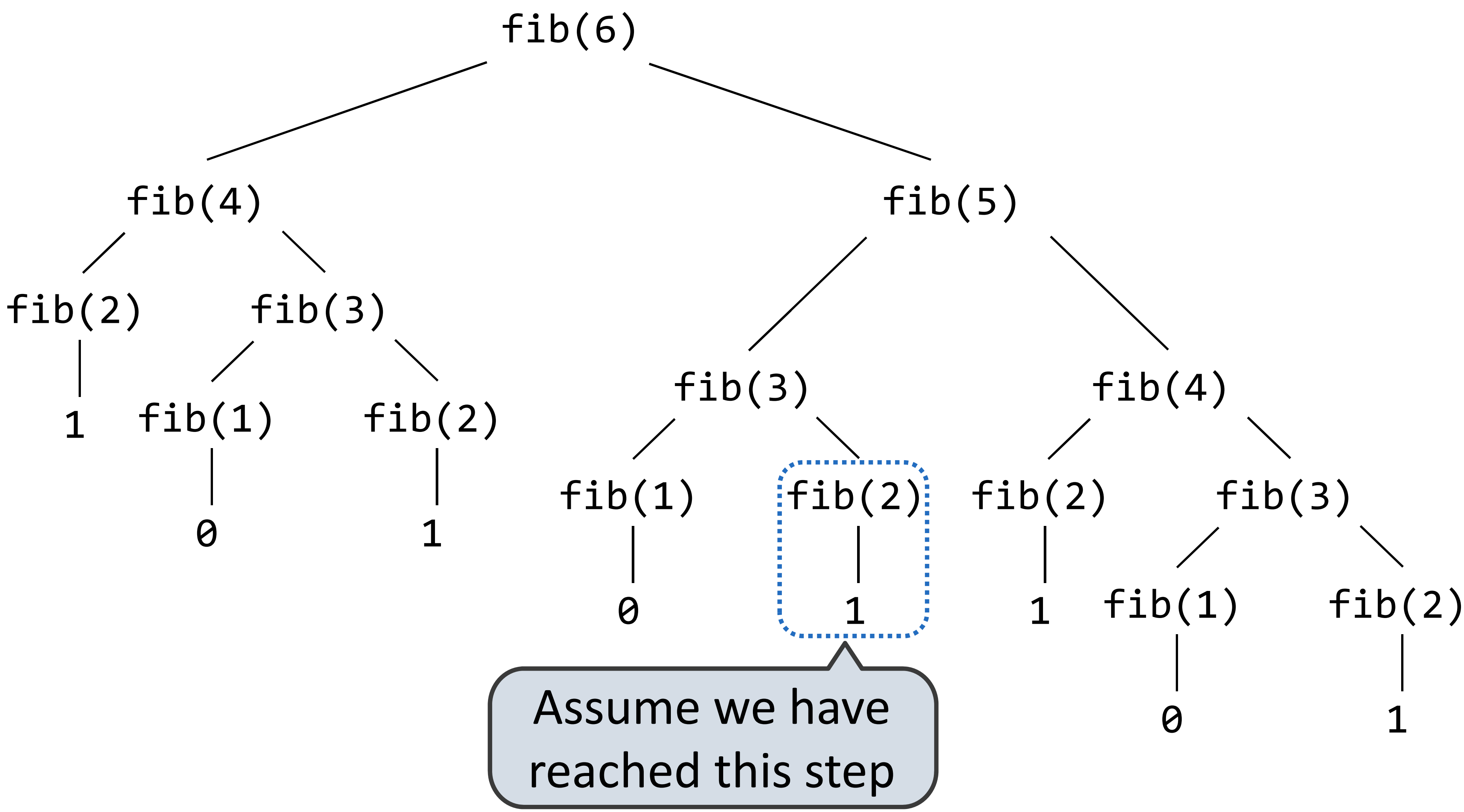
Implementations of the same functional abstraction can require different amounts of time to compute their result.

```
def count_factors(n):

    factors = 0
    for k in range(1, n + 1):
        if n % k == 0:
            factors += 1
    return factors
```

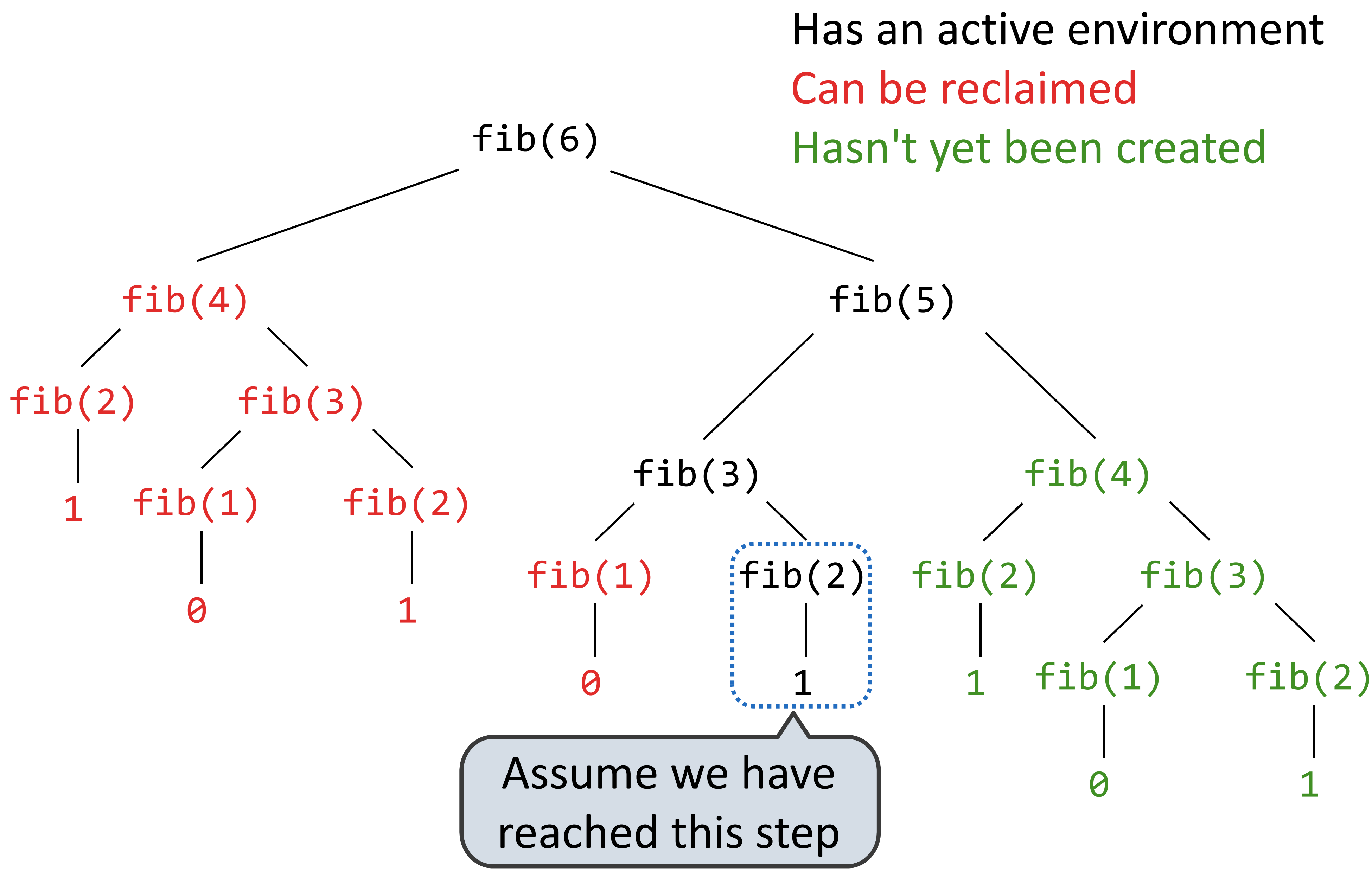
```
    sqrt_n = sqrt(n)
    k, factors = 1, 0
    while k < sqrt_n:
        if n % k == 0:
            factors += 2
        k += 1
    if k * k == n:
        factors += 1
    return factors
```

Time	Space
$\Theta(n)$	$\Theta(1)$
$\Theta(\sqrt{n})$	$\Theta(1)$

Fibonacci Memory Consumption



Fibonacci Memory Consumption



Iteration vs. Tree Recursion

Iterative and recursive implementations are not the same.

Code	Time	Space
`fib_iter`	$\Theta(n)$	$\Theta(1)$
`fib`	$\Theta(\phi^n)$	$\Theta(n)$

```
def fib_iter(n):
    prev, curr = 1, 0
    for _ in range(n - 1):
        prev, curr = curr, prev + curr
    return curr
```

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

Next time, we will see how to make recursive version faster.

Comparing Orders of Growth (*n* is problem size)

$\Theta(b^n)$ Exponential growth! Recursive fib takes

$\Theta(\phi^n)$ steps, where $\phi = \frac{1+\sqrt{5}}{2} \approx 1.61828$

$\Theta(n^6)$

Incrementing the problem scales *R(n)* by a factor.

$\Theta(n^2)$ Quadratic growth. E.g., operations on all pairs.

Incrementing *n* increases *R(n)* by the problem size *n*.

$\Theta(n)$ Linear growth. Resources scale with the problem.

$\Theta(\sqrt{n})$

$\Theta(\log n)$ Logarithmic growth. These processes scale well.

Doubling the problem only increments *R(n)*.

$\Theta(1)$ Constant. The problem size doesn't matter.