Last day of Midterm 2 Material

61A Lecture 23

Friday, October 19

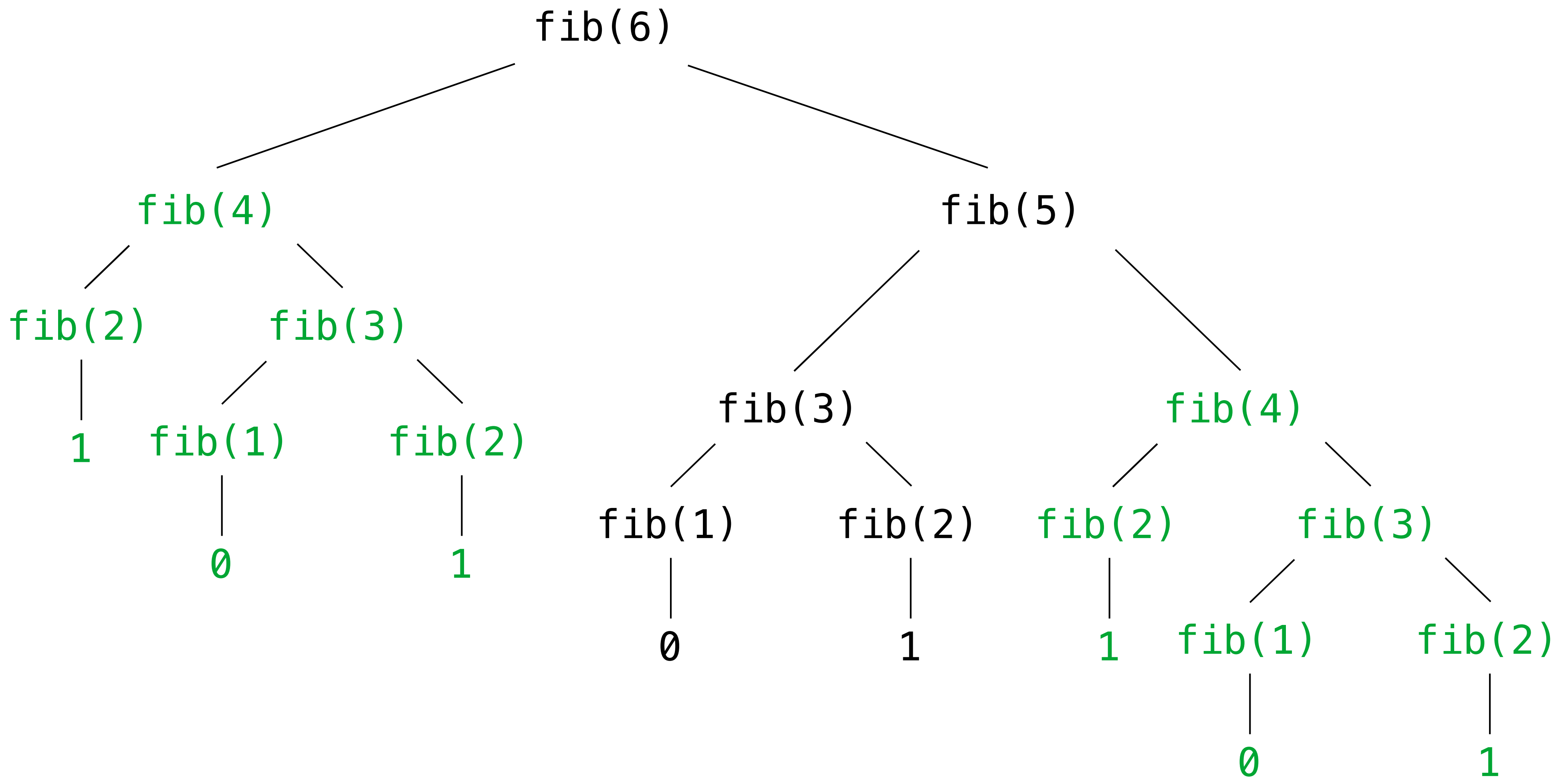
Trees with Internal Node Values

Trees with Internal Node Values

Trees can have values at their roots as well as their leaves.

Trees with Internal Node Values

Trees can have values at their roots as well as their leaves.



Trees with Internal Node Values (Entries)

Trees need not only have values at their leaves.

A valid tree cannot
be a subtree of
itself (no cycles!)

Trees with Internal Node Values (Entries)

Trees need not only have values at their leaves.

```
class Tree(object):
```

A valid tree cannot be a subtree of itself (no cycles!)

Trees with Internal Node Values (Entries)

Trees need not only have values at their leaves.

```
class Tree(object):
    def __init__(self, entry, left=None, right=None):
```

A valid tree cannot be a subtree of itself (no cycles!)

Trees with Internal Node Values (Entries)

Trees need not only have values at their leaves.

```
class Tree(object):
    def __init__(self, entry, left=None, right=None):
        self.entry = entry
```

A valid tree cannot be a subtree of itself (no cycles!)

Trees with Internal Node Values (Entries)

Trees need not only have values at their leaves.

```
class Tree(object):
    def __init__(self, entry, left=None, right=None):
        self.entry = entry
        self.left = left
```

A valid tree cannot be a subtree of itself (no cycles!)

Trees with Internal Node Values (Entries)

Trees need not only have values at their leaves.

```
class Tree(object):
    def __init__(self, entry, left=None, right=None):
        self.entry = entry
        self.left = left
        self.right = right
```

A valid tree cannot be a subtree of itself (no cycles!)

Trees with Internal Node Values (Entries)

Trees need not only have values at their leaves.

```
class Tree(object):
    def __init__(self, entry, left=None, right=None):
        self.entry = entry
        self.left = left
        self.right = right
```

Valid if left and right are each either None or a Tree instance

A valid tree cannot be a subtree of itself (no cycles!)

Trees with Internal Node Values (Entries)

Trees need not only have values at their leaves.

```
class Tree(object):
    def __init__(self, entry, left=None, right=None):
        self.entry = entry
        self.left = left
        self.right = right

def fib_tree(n):
```

Valid if left and right are each either None or a Tree instance

A valid tree cannot be a subtree of itself (no cycles!)

Trees with Internal Node Values (Entries)

Trees need not only have values at their leaves.

```
class Tree(object):
    def __init__(self, entry, left=None, right=None):
        self.entry = entry
        self.left = left
        self.right = right

def fib_tree(n):
    if n == 1:
```

Valid if left and right are each either None or a Tree instance

A valid tree cannot be a subtree of itself (no cycles!)

Trees with Internal Node Values (Entries)

Trees need not only have values at their leaves.

```
class Tree(object):
    def __init__(self, entry, left=None, right=None):
        self.entry = entry
        self.left = left
        self.right = right

def fib_tree(n):
    if n == 1:
        return Tree(0)
```

Valid if left and right are each either None or a Tree instance

A valid tree cannot be a subtree of itself (no cycles!)

Trees with Internal Node Values (Entries)

Trees need not only have values at their leaves.

```
class Tree(object):
    def __init__(self, entry, left=None, right=None):
        self.entry = entry
        self.left = left
        self.right = right
```

Valid if left and right are each either None or a Tree instance

A valid tree cannot be a subtree of itself (no cycles!)

```
def fib_tree(n):
    if n == 1:
        return Tree(0)
    if n == 2:
```

Trees with Internal Node Values (Entries)

Trees need not only have values at their leaves.

```
class Tree(object):
    def __init__(self, entry, left=None, right=None):
        self.entry = entry
        self.left = left
        self.right = right

def fib_tree(n):
    if n == 1:
        return Tree(0)
    if n == 2:
        return Tree(1)
```

Valid if left and right are each either None or a Tree instance

A valid tree cannot be a subtree of itself (no cycles!)

Trees with Internal Node Values (Entries)

Trees need not only have values at their leaves.

```
class Tree(object):
    def __init__(self, entry, left=None, right=None):
        self.entry = entry
        self.left = left
        self.right = right

def fib_tree(n):
    if n == 1:
        return Tree(0)
    if n == 2:
        return Tree(1)
    left = fib_tree(n-2)
```

Valid if left and right are each either None or a Tree instance

A valid tree cannot be a subtree of itself (no cycles!)

Trees with Internal Node Values (Entries)

Trees need not only have values at their leaves.

```
class Tree(object):
    def __init__(self, entry, left=None, right=None):
        self.entry = entry
        self.left = left
        self.right = right

def fib_tree(n):
    if n == 1:
        return Tree(0)
    if n == 2:
        return Tree(1)
    left = fib_tree(n-2)
    right = fib_tree(n-1)
```

Valid if left and right are each either None or a Tree instance

A valid tree cannot be a subtree of itself (no cycles!)

Trees with Internal Node Values (Entries)

Trees need not only have values at their leaves.

```
class Tree(object):
    def __init__(self, entry, left=None, right=None):
        self.entry = entry
        self.left = left
        self.right = right
```

Valid if left and right are each either None or a Tree instance

A valid tree cannot be a subtree of itself (no cycles!)

```
def fib_tree(n):
    if n == 1:
        return Tree(0)
    if n == 2:
        return Tree(1)
    left = fib_tree(n-2)
    right = fib_tree(n-1)
    return Tree(left.entry + right.entry, left, right)
```

Trees with Internal Node Values (Entries)

Trees need not only have values at their leaves.

```
class Tree(object):
    def __init__(self, entry, left=None, right=None):
        self.entry = entry
        self.left = left
        self.right = right
```

Valid if left and right are each either None or a Tree instance

A valid tree cannot be a subtree of itself (no cycles!)

```
def fib_tree(n):
    if n == 1:
        return Tree(0)
    if n == 2:
        return Tree(1)
    left = fib_tree(n-2)
    right = fib_tree(n-1)
    return Tree(left.entry + right.entry, left, right)
```

Demo

The Consumption of Time

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

The Consumption of Time

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

```
def count_factors(n):
```

The Consumption of Time

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

```
def count_factors(n):
```

(Demo)

The Consumption of Time

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

```
def count_factors(n):
```

(Demo)

Time (remainders)

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
```

(Demo)

Time (remainders)

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
```

(Demo)

Time (remainders)
n

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
```

(Demo)

Time (remainders)

n

The Consumption of Time

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

```
def count_factors(n):
```

(Demo)

Time (remainders)

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

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
```

The Consumption of Time

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

```
def count_factors(n):
```

(Demo)

Time (remainders)

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

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
```

$\lfloor \sqrt{n} \rfloor$

The Consumption of Space

The Consumption of Space

```
Which environment frames do we need to keep during evaluation?
```

The Consumption of Space

Which environment frames do we need to keep during evaluation?

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

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.

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.

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:

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

The Consumption of Space

Which environment frames do we need to keep during evaluation?

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

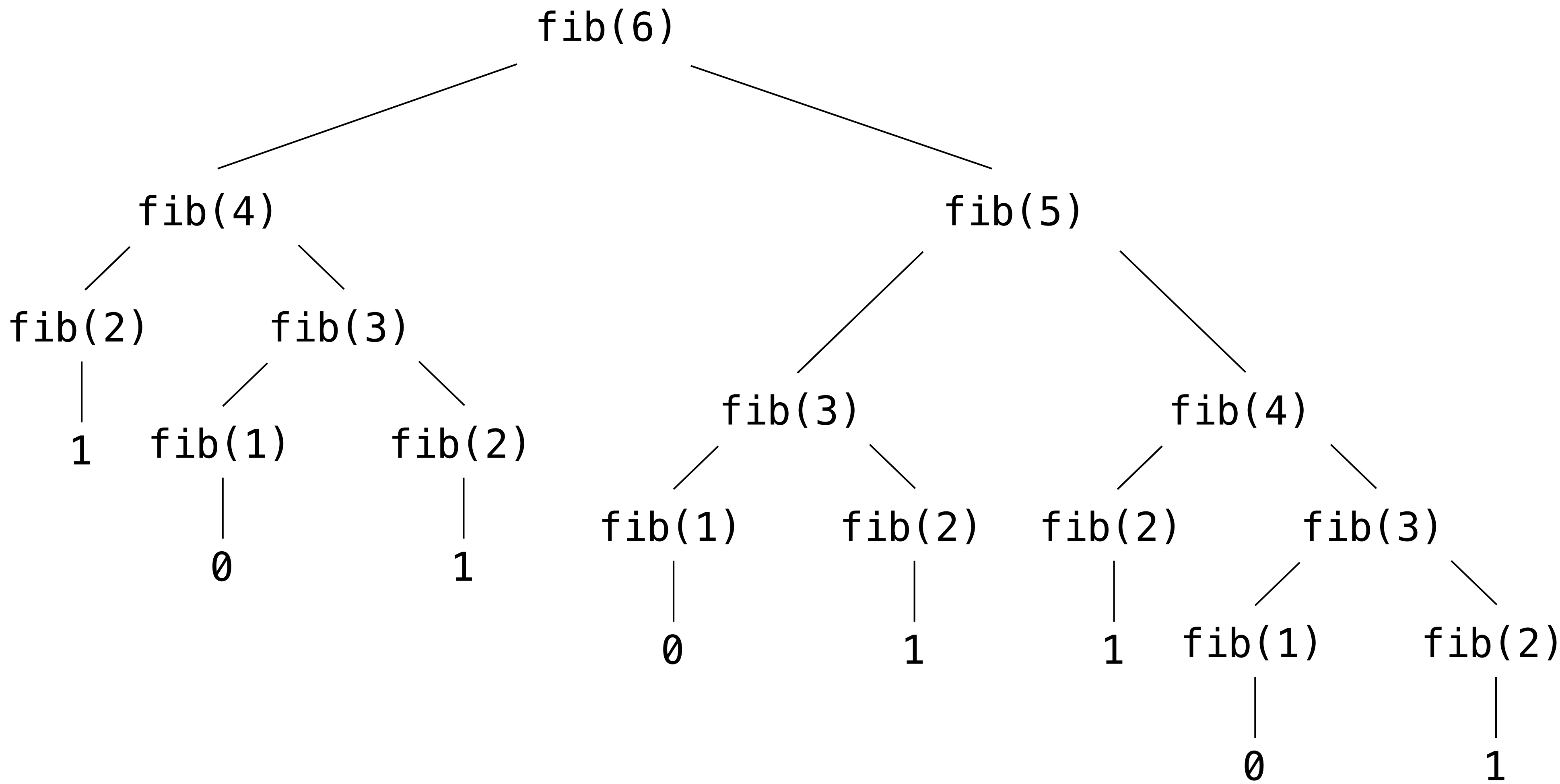
Values and frames in active environments consume memory.

Memory used for other values and frames can be reclaimed.

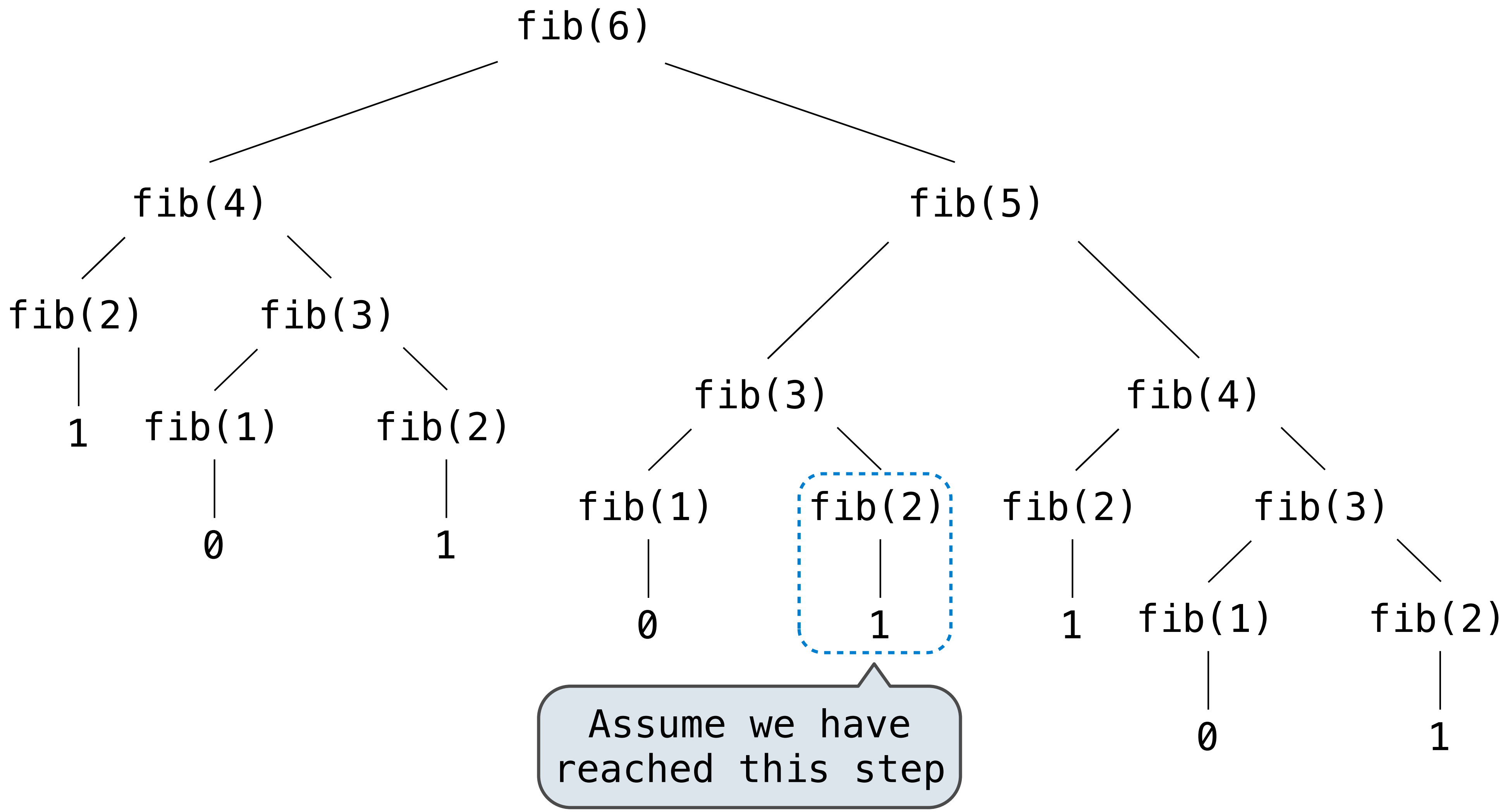
Active environments:

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

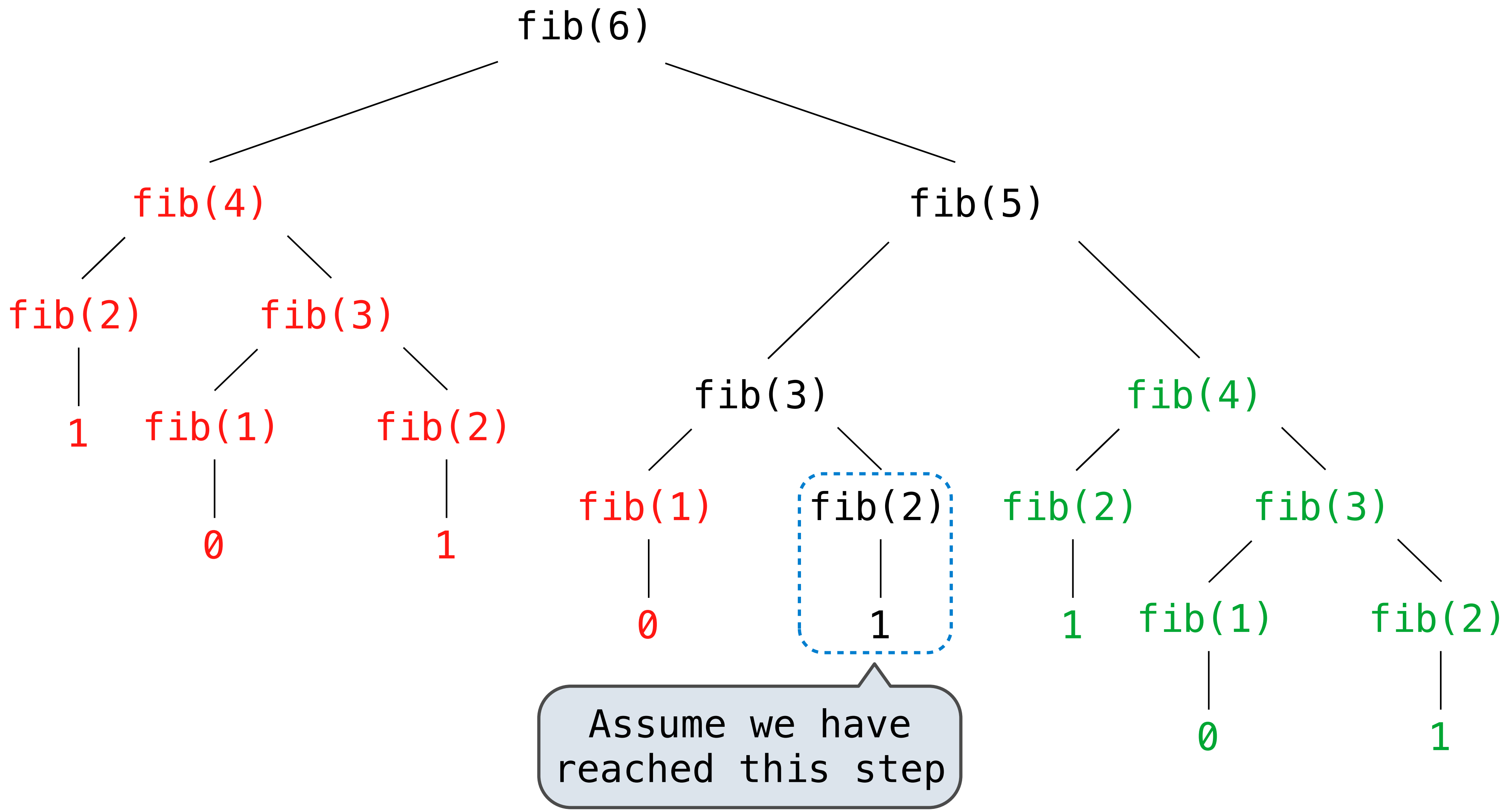
Fibonacci Memory Consumption



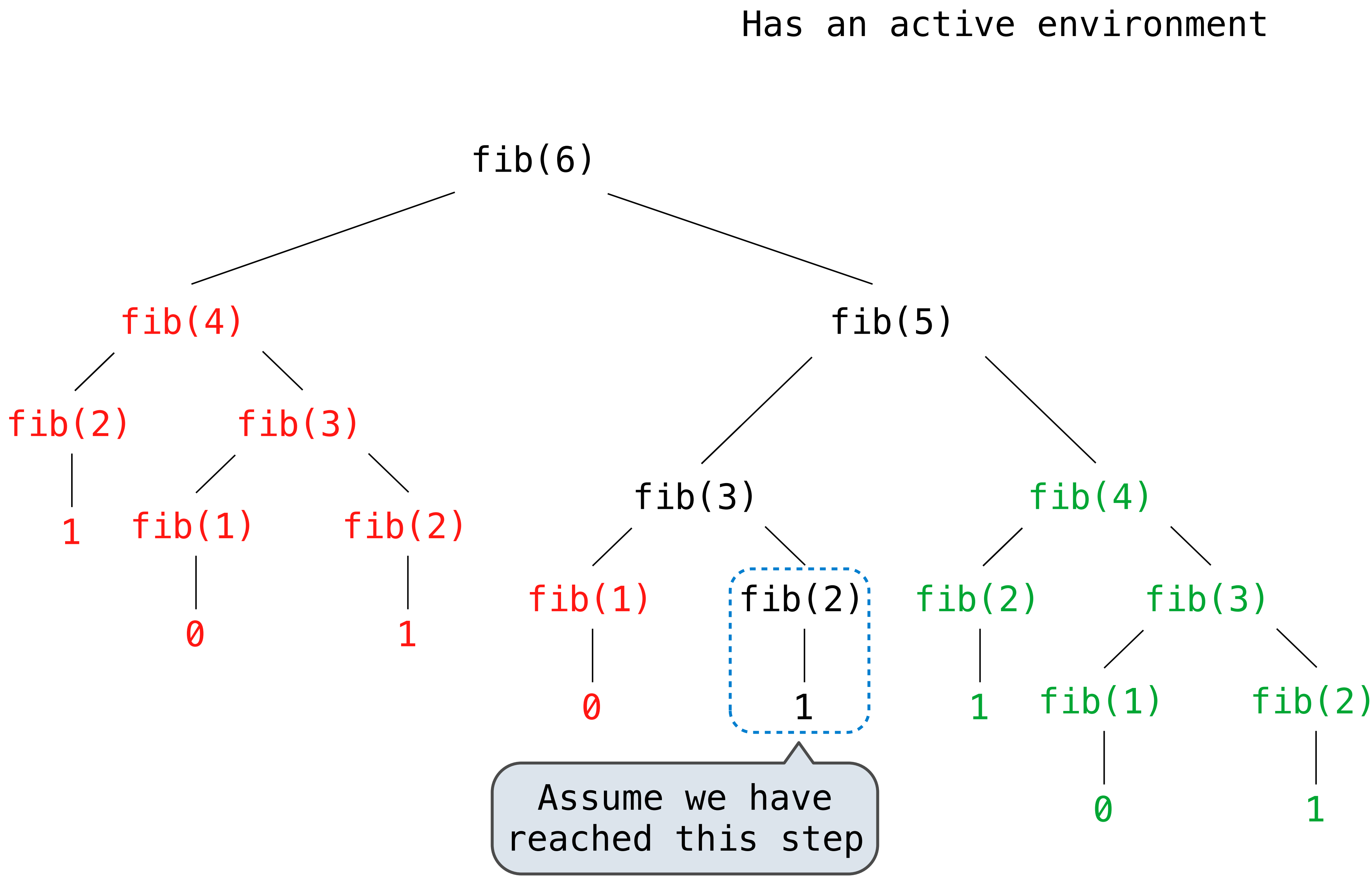
Fibonacci Memory Consumption



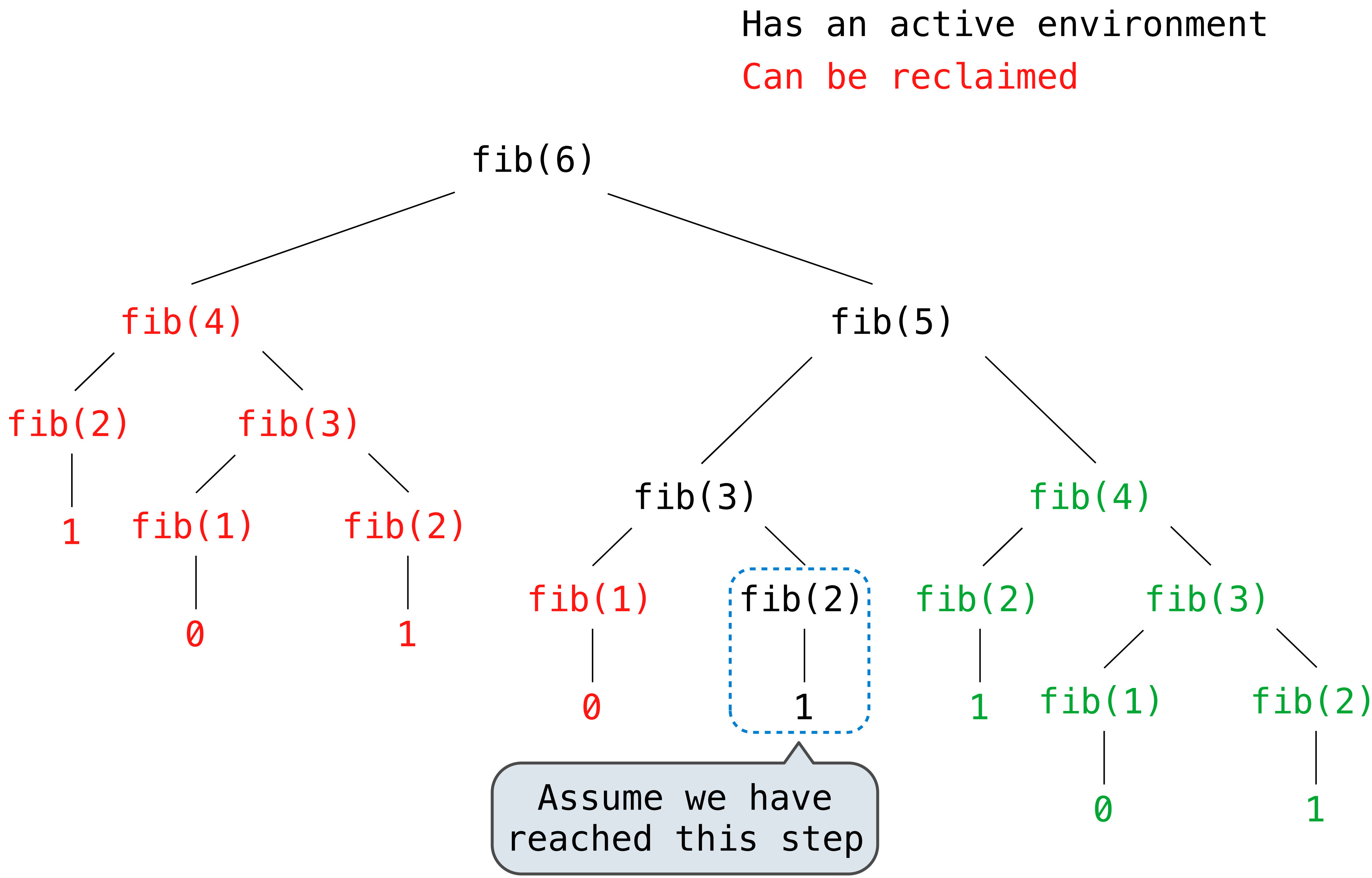
Fibonacci Memory Consumption



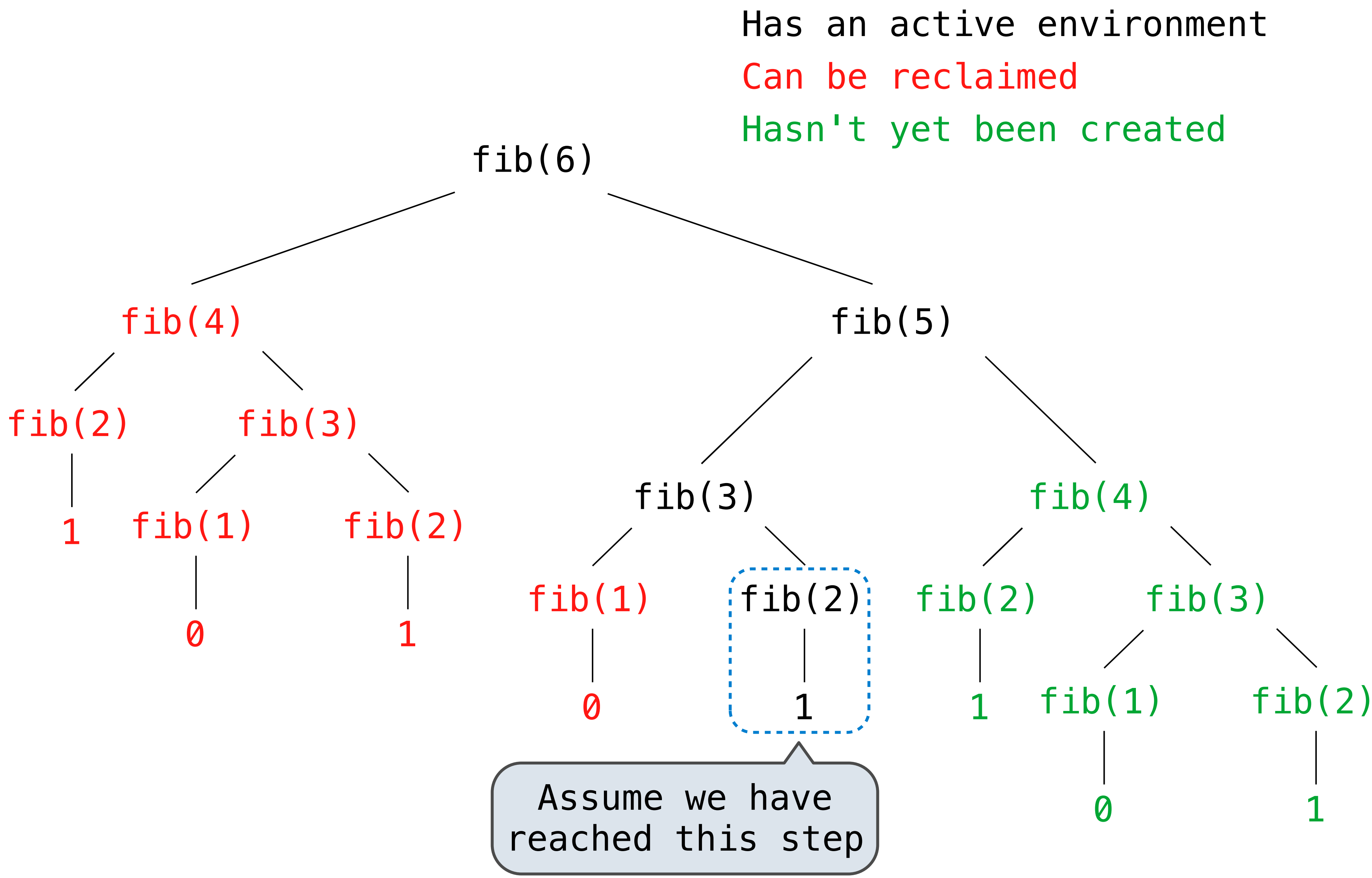
Fibonacci Memory Consumption



Fibonacci Memory Consumption



Fibonacci Memory Consumption



Order of Growth

Order of Growth

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

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

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)

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)

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

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

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

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

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*.**

Iteration vs Memoized Tree Recursion

Iterative and memoized implementations are not the same.

	Time	Space

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

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

Iteration vs Memoized Tree Recursion

Iterative and memoized implementations are not the same.

	Time	Space
fib_iter	$\Theta(n)$	
fib (memoized)		

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

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

Iteration vs Memoized Tree Recursion

Iterative and memoized implementations are not the same.

	Time	Space
`fib_iter`	$\Theta(n)$	$\Theta(1)$

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

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

Iteration vs Memoized Tree Recursion

Iterative and memoized implementations are not the same.

	Time	**Space**
`def fib_iter(n): ...`	$\Theta(n)$	$\Theta(1)$
`@memo def fib(n): ...`	$\Theta(n)$	

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

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

Iteration vs Memoized Tree Recursion

Iterative and memoized implementations are not the same.

Time	Space
$\Theta(n)$	$\Theta(1)$
$\Theta(n)$	$\Theta(n)$

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

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

The Consumption of Time

Implementations of the same functional abstraction can require different amounts of time.

```
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**

The Consumption of Time

Implementations of the same functional abstraction can require different amounts of time.

```
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
$\Theta(n)$	$\Theta(1)$

The Consumption of Time

Implementations of the same functional abstraction can require different amounts of time.

```
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
$\Theta(n)$	$\Theta(1)$
$\Theta(\sqrt{n})$	$\Theta(1)$

Exponentiation

Exponentiation

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

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)
```

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)
```

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

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)
```

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

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

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)
```

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

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

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

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)
```

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

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

def fast_exp(b, n):
```

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

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)
```

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

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

def fast_exp(b, n):
    if n == 0:
        return 1
```

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

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)
```

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

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

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

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

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)
```

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

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

def fast_exp(b, n):
    if n == 0:
        return 1
    if 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.

	Time	Space

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

Exponentiation

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

	Time	**Space**
`exp`	$\Theta(n)$	$\Theta(n)$

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

Exponentiation

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

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

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

Comparing orders of growth (n is the problem size)

Comparing orders of growth (n is the problem size)

$\Theta(b^n)$

Comparing orders of growth (n is the problem size)

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

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

Comparing orders of growth (n is the problem size)

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

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

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

Comparing orders of growth (n is the problem size)

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

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

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

$\Theta(n^2)$

Comparing orders of growth (n is the problem size)

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

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

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

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

Comparing orders of growth (n is the problem size)

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

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

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.

Comparing orders of growth (n is the problem size)

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

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

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.

$\Theta(n)$

Comparing orders of growth (n is the problem size)

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

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

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.

Comparing orders of growth (n is the problem size)

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

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

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.

$\Theta(\log n)$

Comparing orders of growth (n is the problem size)

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

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

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.

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

Comparing orders of growth (n is the problem size)

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

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

`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.`

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

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

Comparing orders of growth (n is the problem size)

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

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

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.

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

Doubling the problem only increments R(n).

$\Theta(1)$

Comparing orders of growth (n is the problem size)

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

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

`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.`

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

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

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

Comparing orders of growth (n is the problem size)

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

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

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.

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

Doubling the problem only increments R(n).

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

Comparing orders of growth (n is the 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.`

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

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

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

Comparing orders of growth (n is the 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.