Growth

Announcements

Measuring Efficiency

Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:

Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:

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

Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:

```
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

Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:

fib(5)

```
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

Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:

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

fib(5)

fib(3)

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

Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:

fib(5)

fib(3)   fib(4)

```
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

Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:

```
                    fib(5)
          /                    \
      fib(3)                  fib(4)
     /      \
 fib(1)    fib(2)
   |       /    \
   1   fib(0)  fib(1)
          |       |
          0       1
```

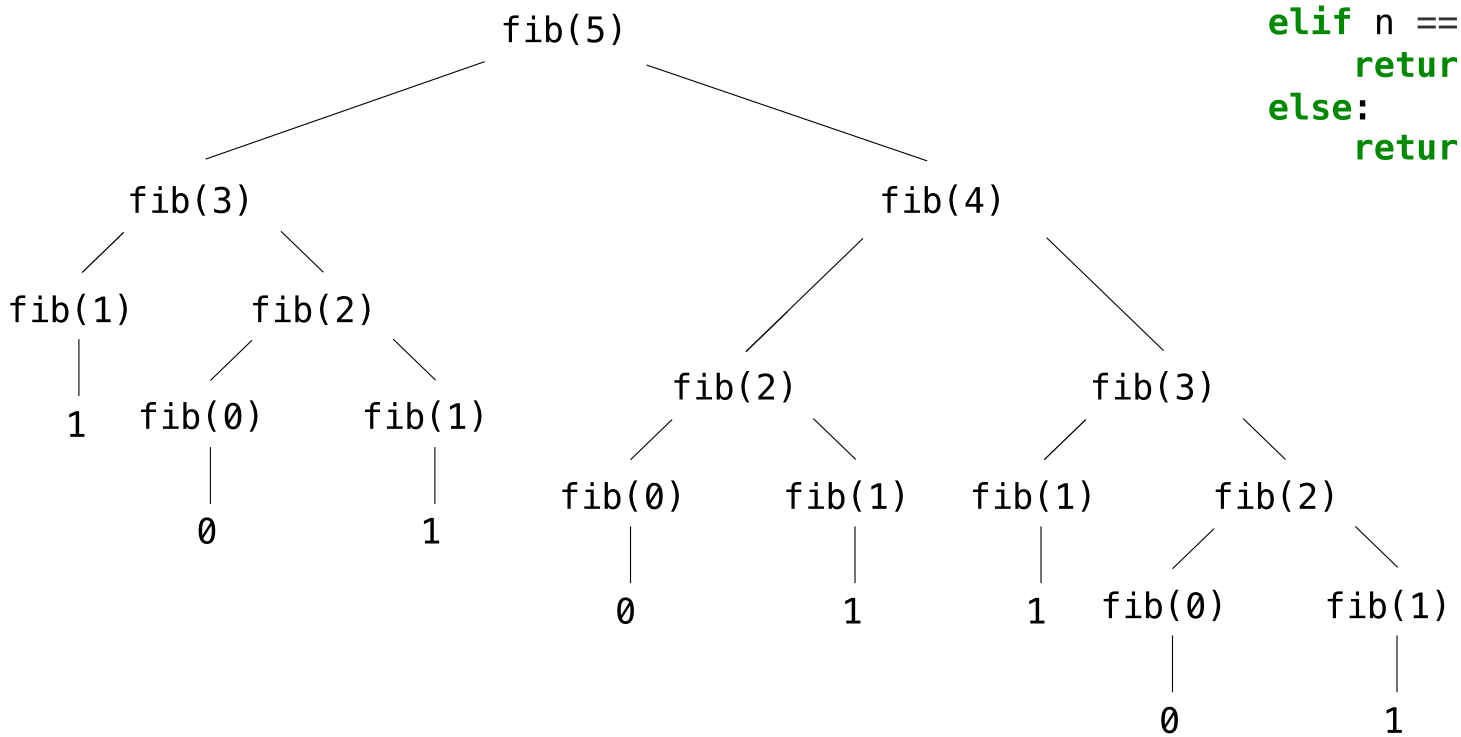
```python
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-2) + fib(n-1)
```

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

Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:

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

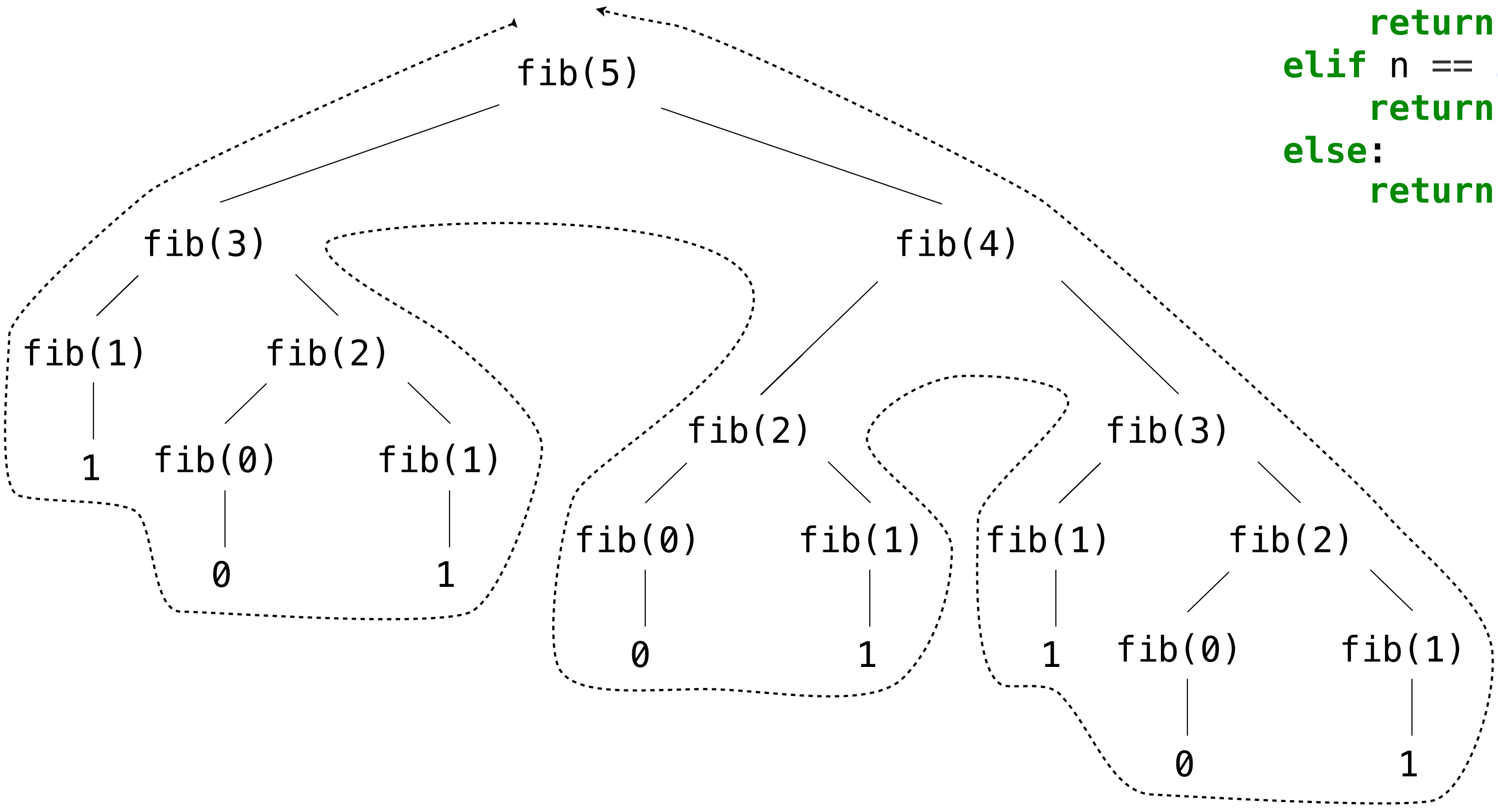


http://en.wikipedia.org/wiki/File:Fibonacci.jpg

Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:

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

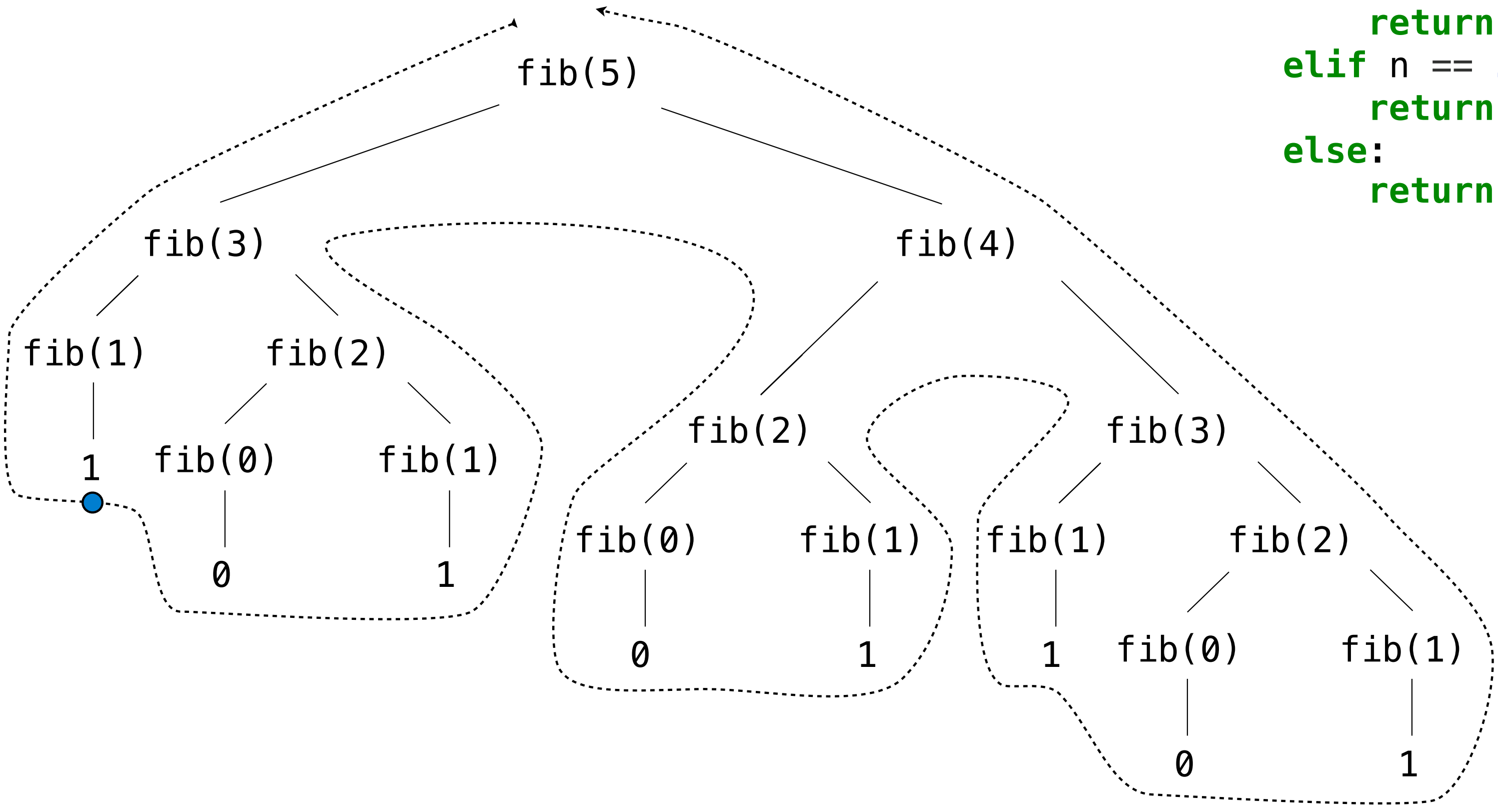


http://en.wikipedia.org/wiki/File:Fibonacci.jpg

Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:

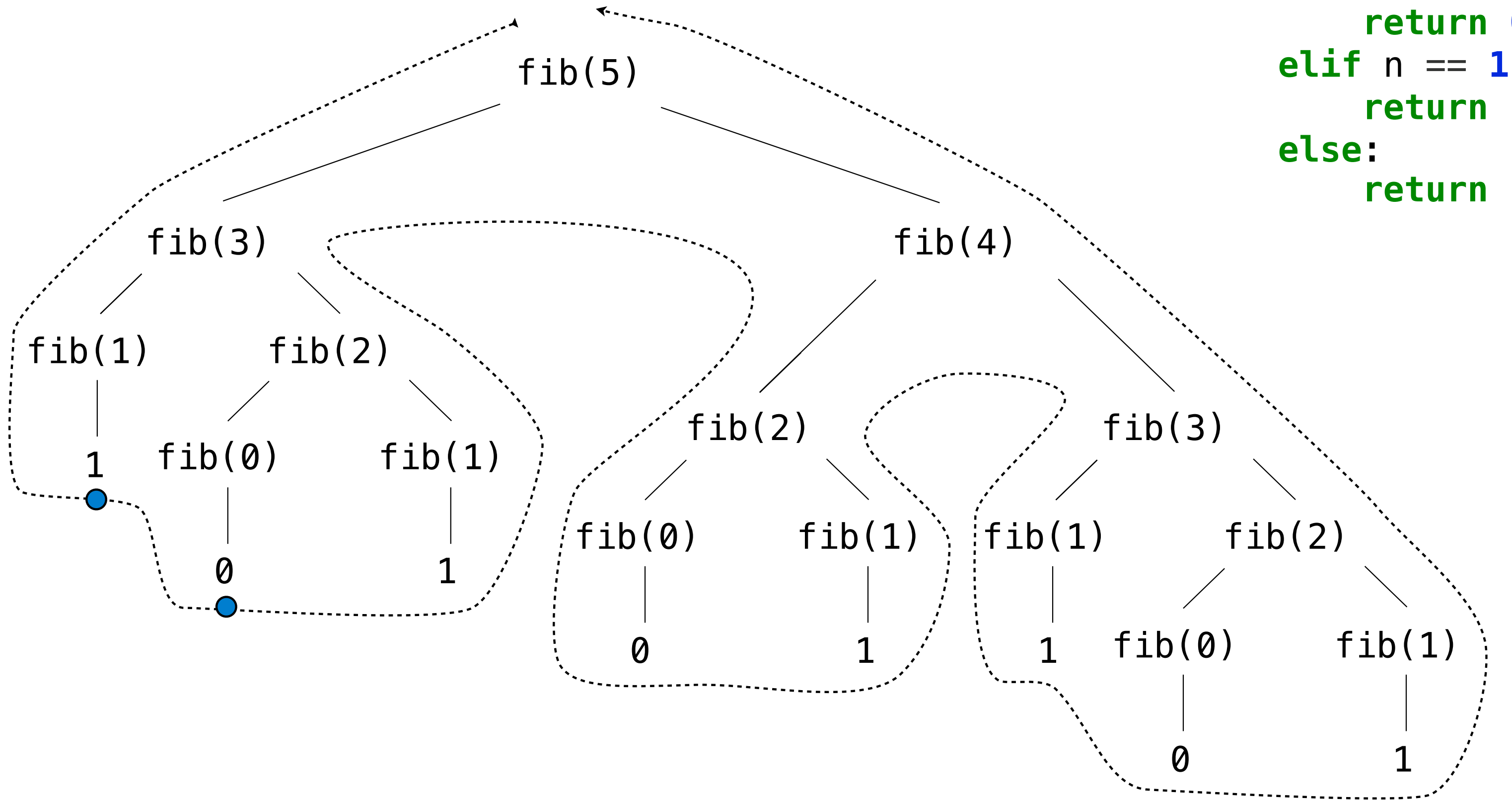
```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-2) + fib(n-1)
```



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

Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:

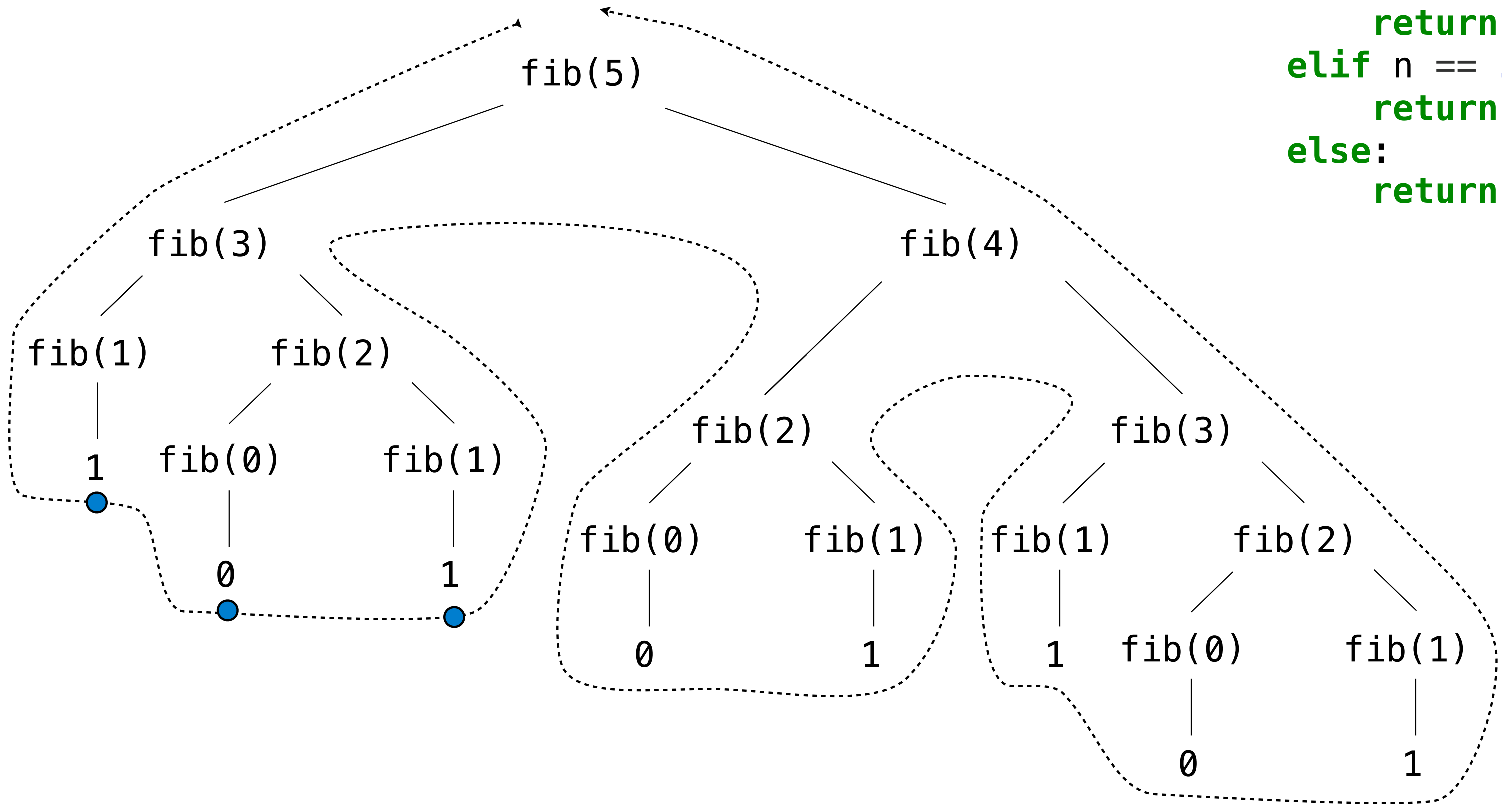



```python
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-2) + fib(n-1)
```

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

Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:

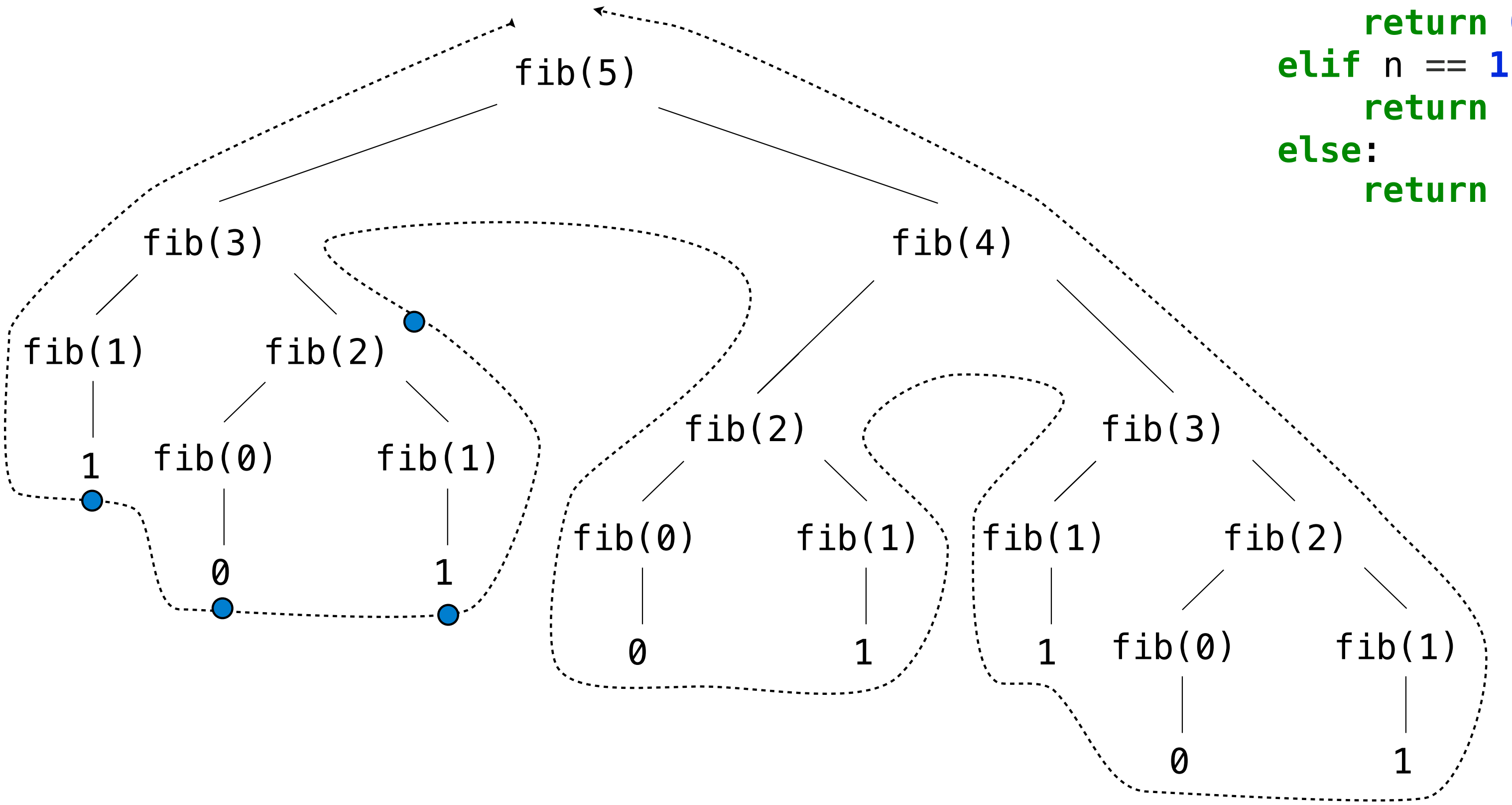



```python
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-2) + fib(n-1)
```

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

Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:



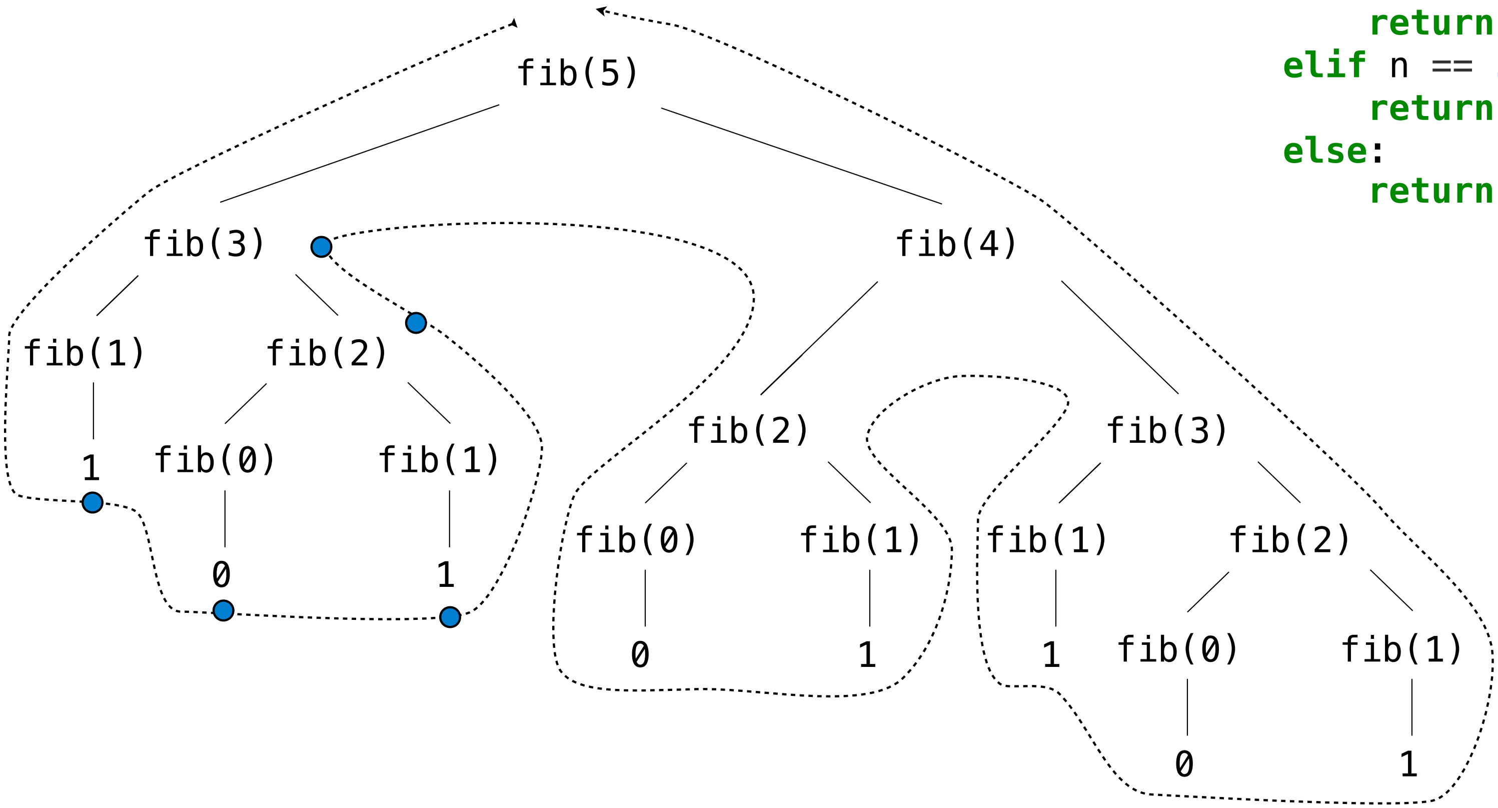

```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-2) + fib(n-1)
```

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

Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:

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

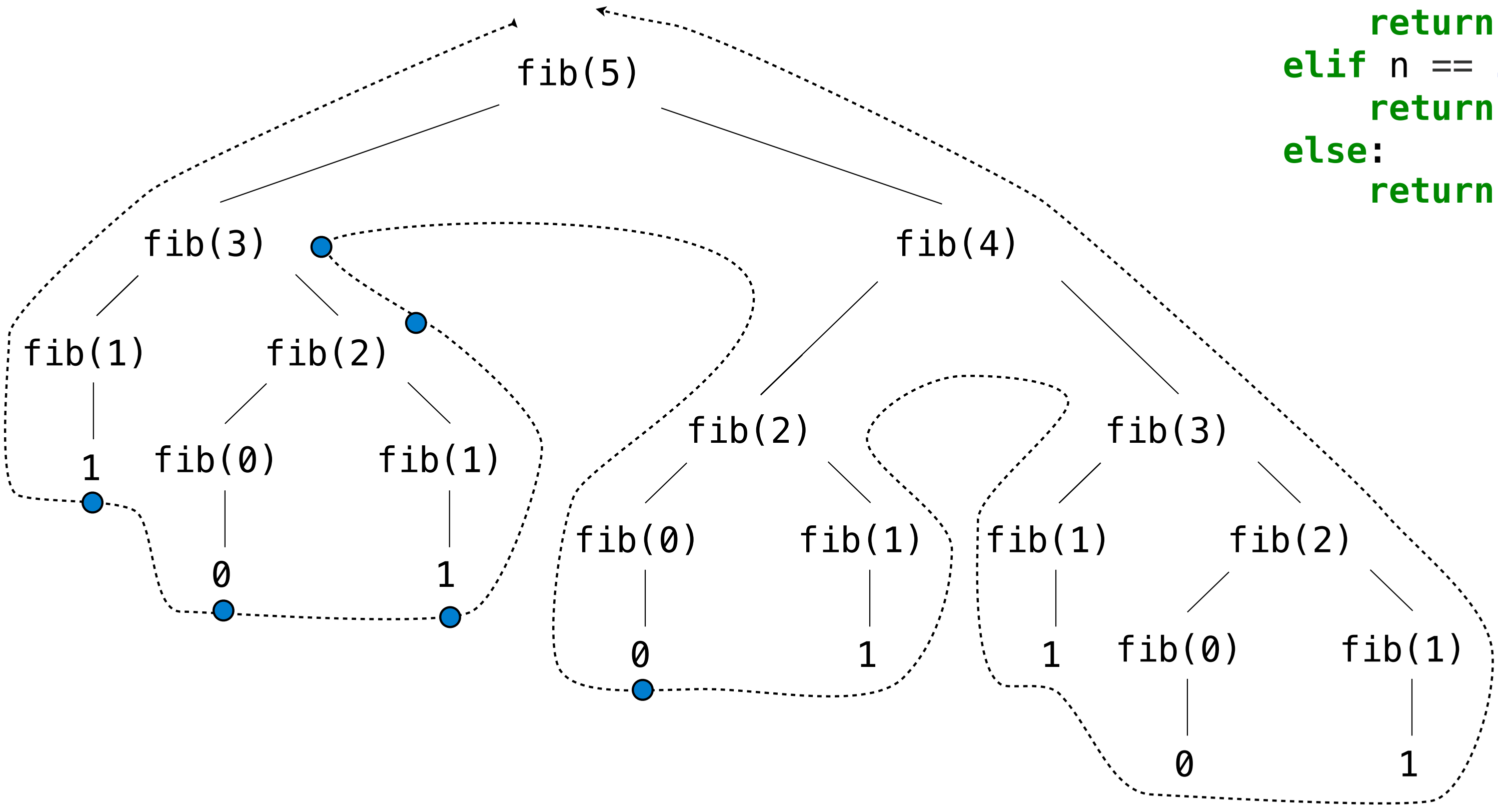


http://en.wikipedia.org/wiki/File:Fibonacci.jpg

Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:

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

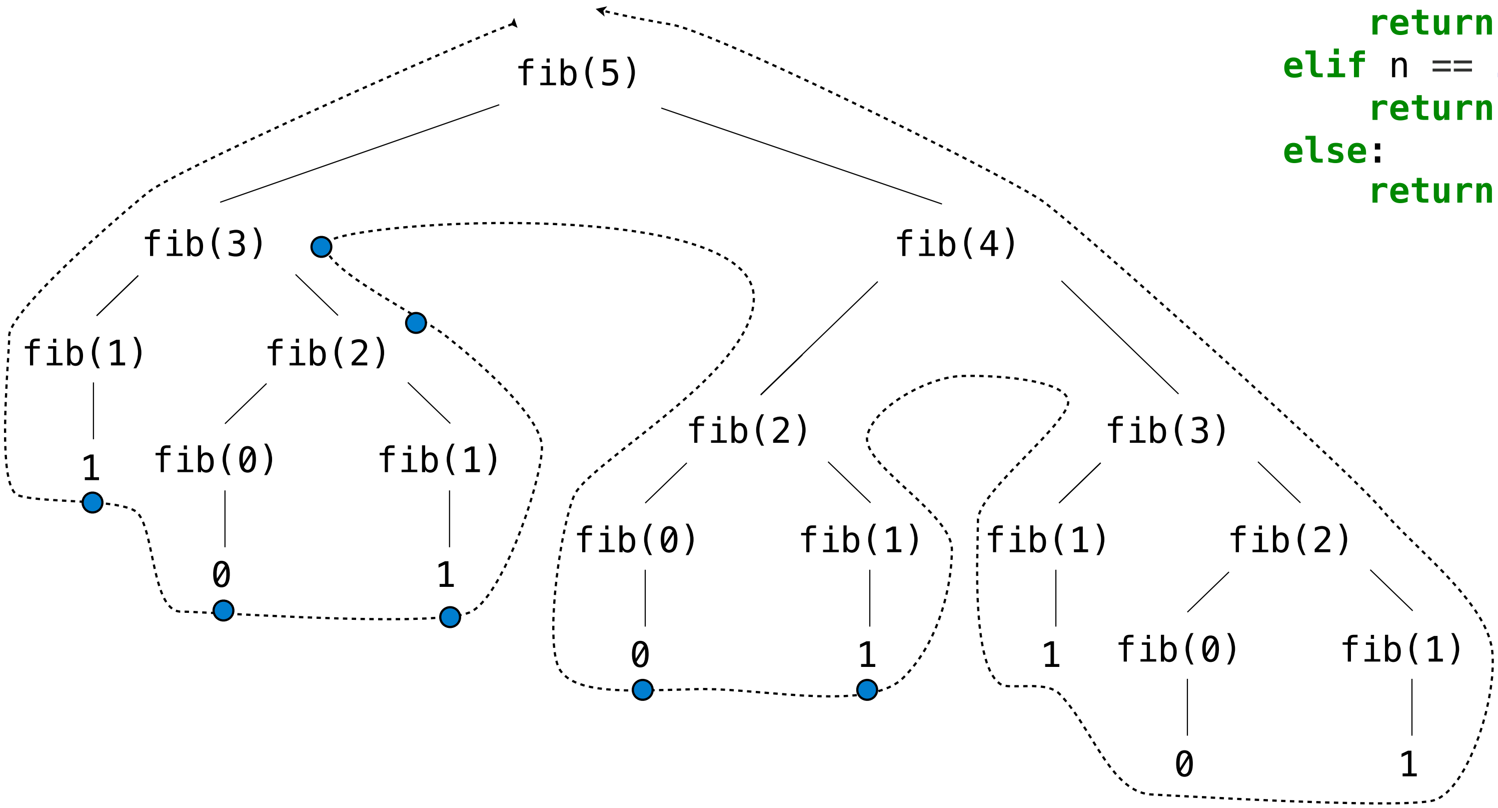


http://en.wikipedia.org/wiki/File:Fibonacci.jpg

Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:

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

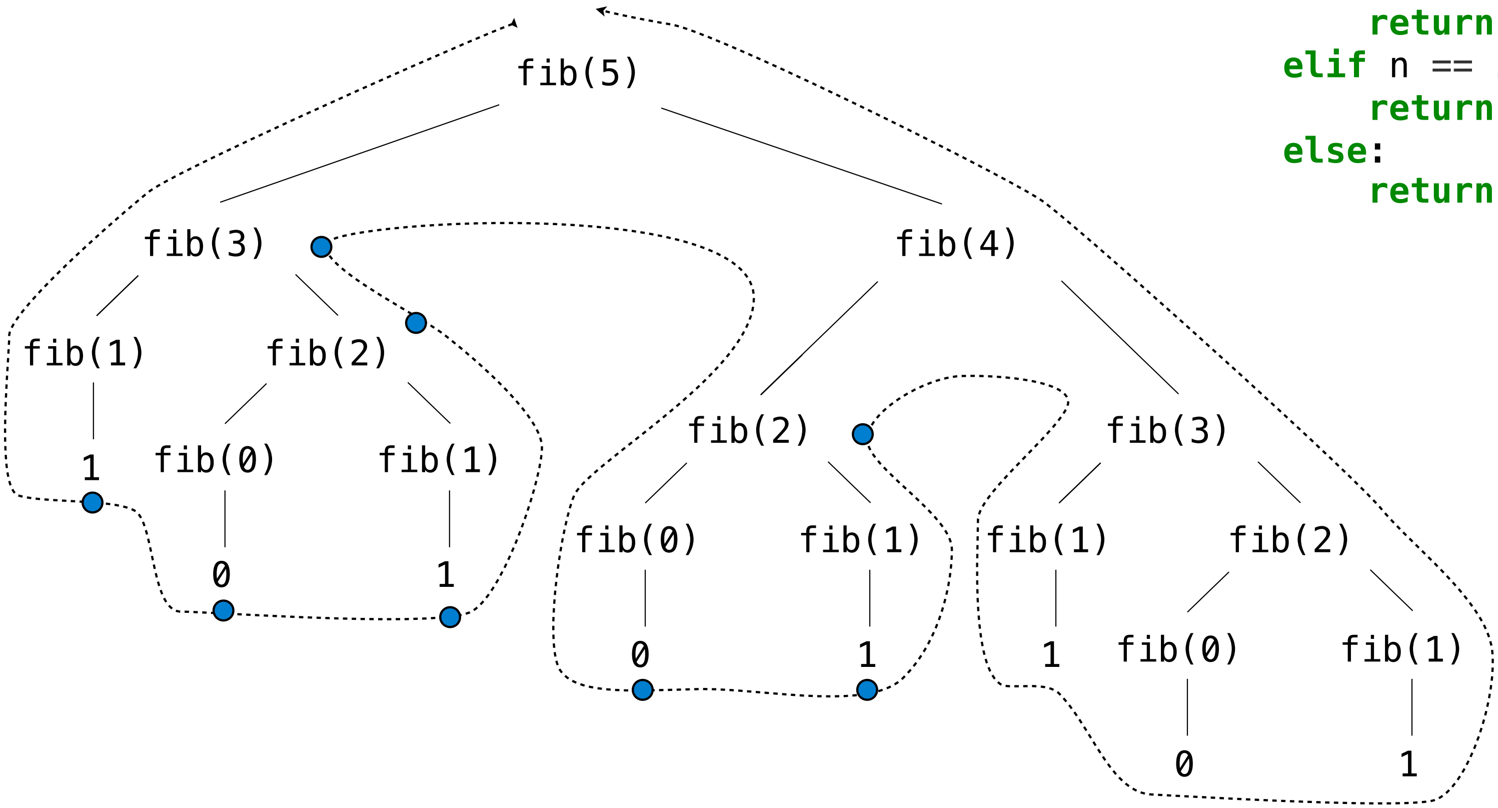


http://en.wikipedia.org/wiki/File:Fibonacci.jpg

Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:

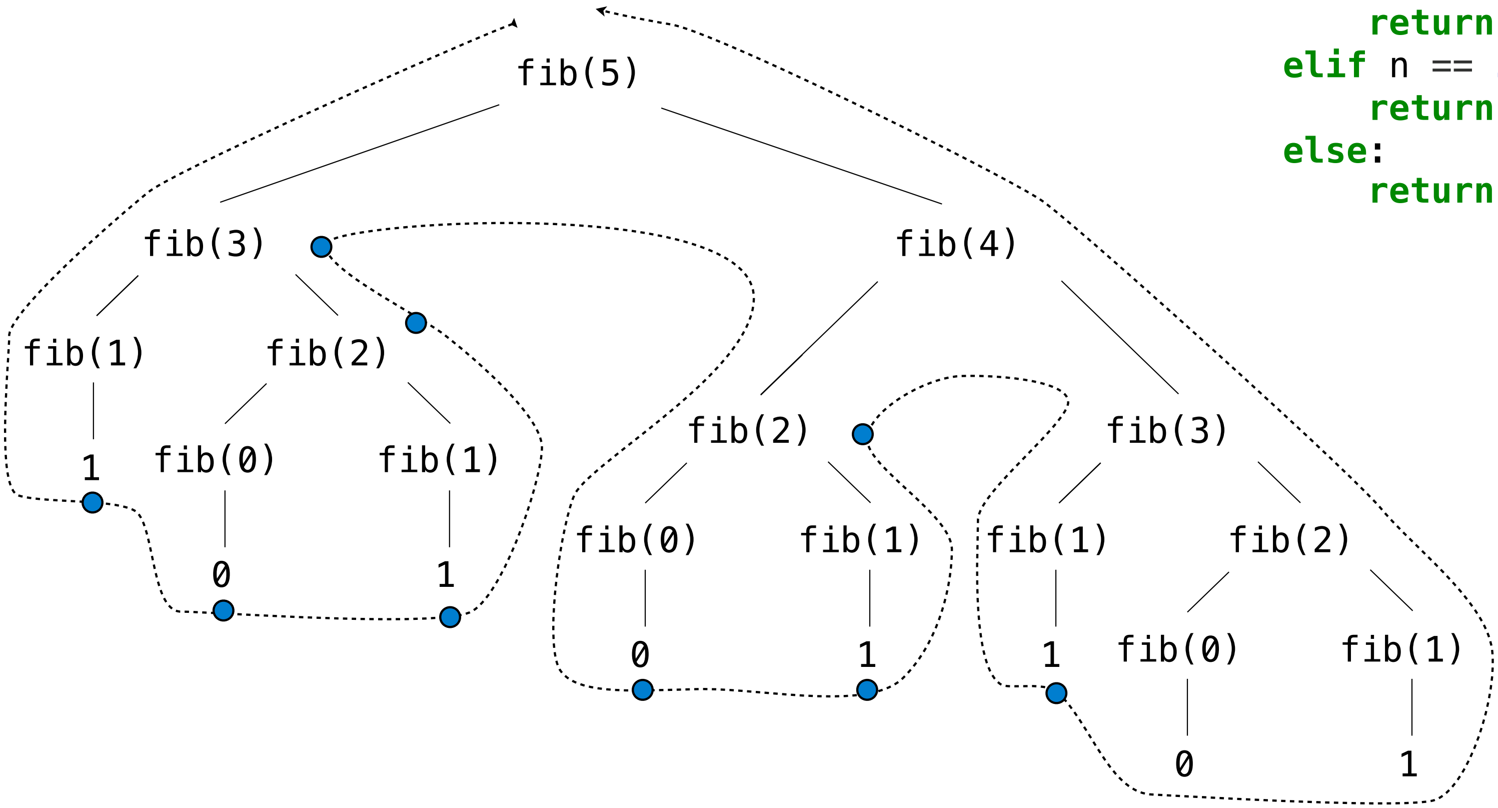
```python
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-2) + fib(n-1)
```



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

Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:



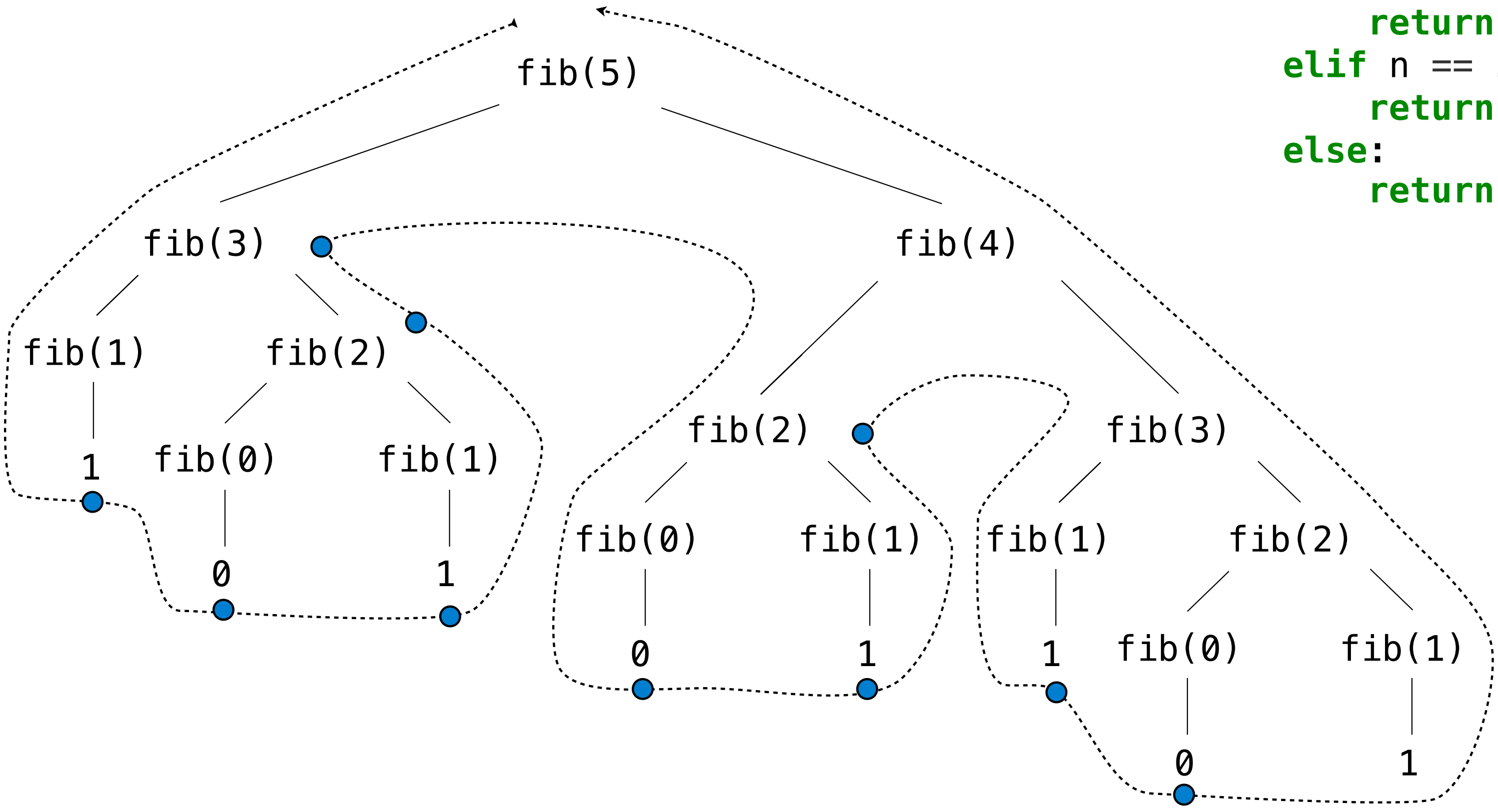

```python
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-2) + fib(n-1)
```

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

Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:

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

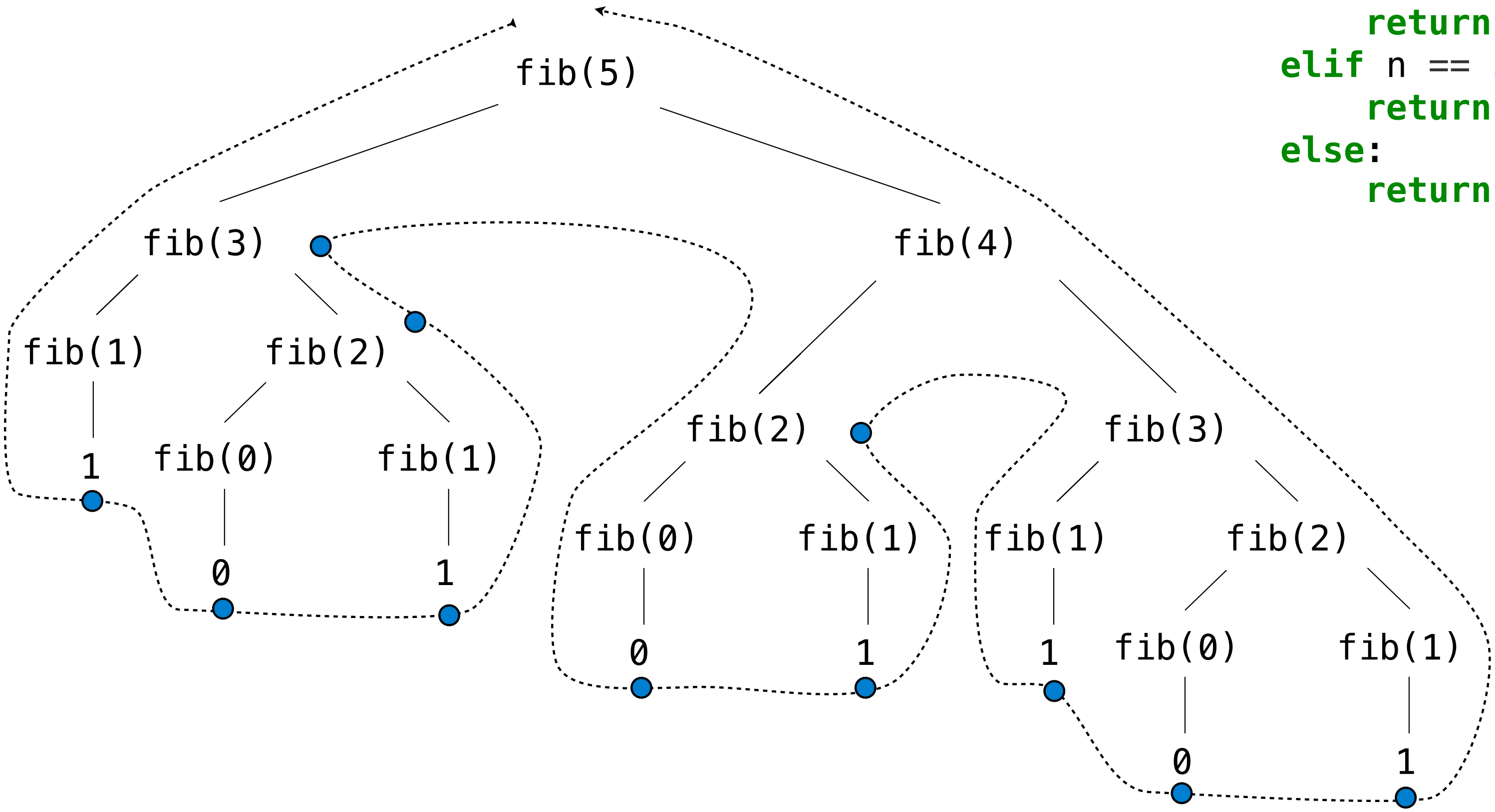


http://en.wikipedia.org/wiki/File:Fibonacci.jpg

Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:

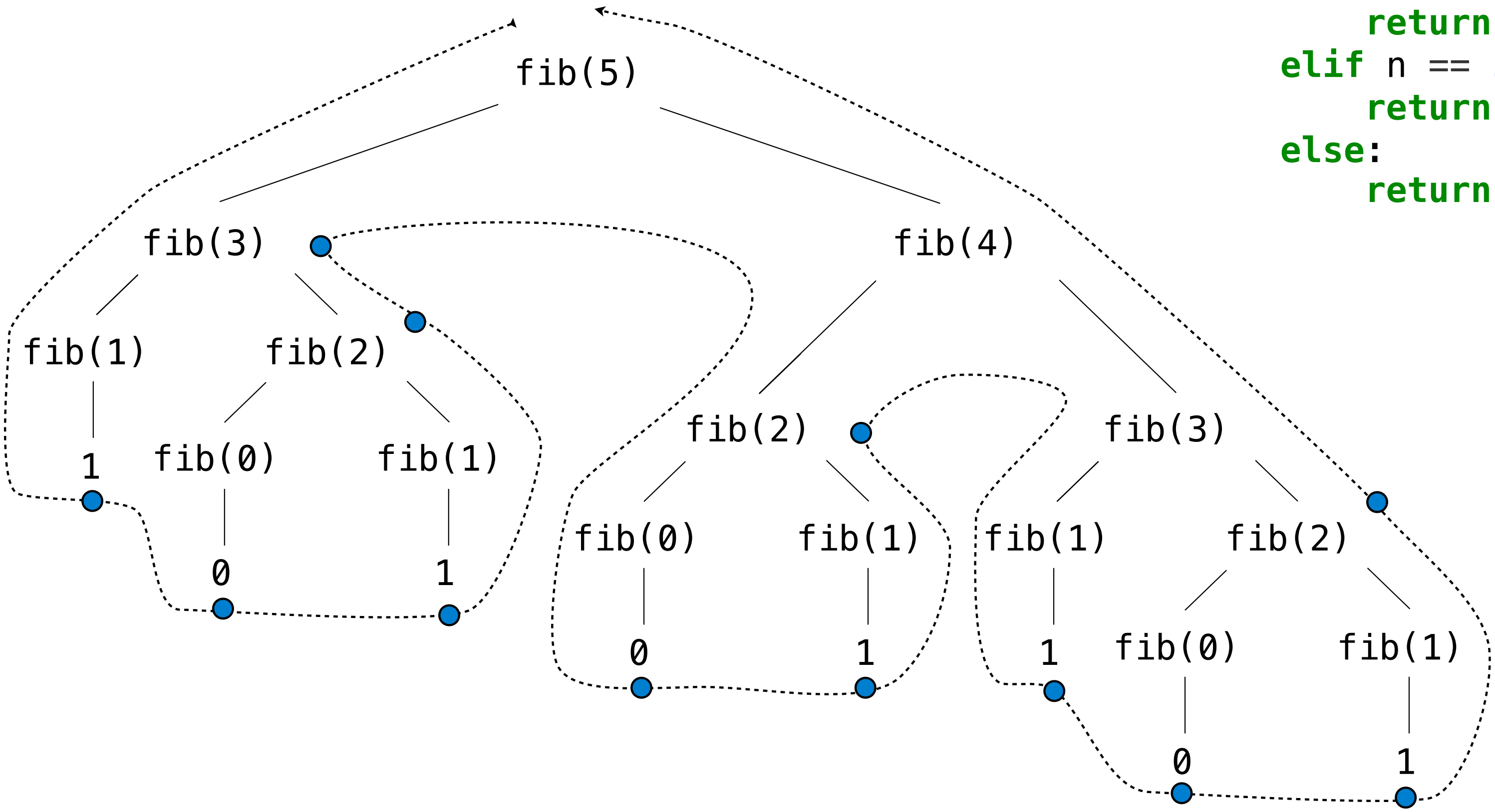
```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-2) + fib(n-1)
```



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

Recursive Computation of the Fibonacci Sequence

```
Our first example of tree recursion:
```



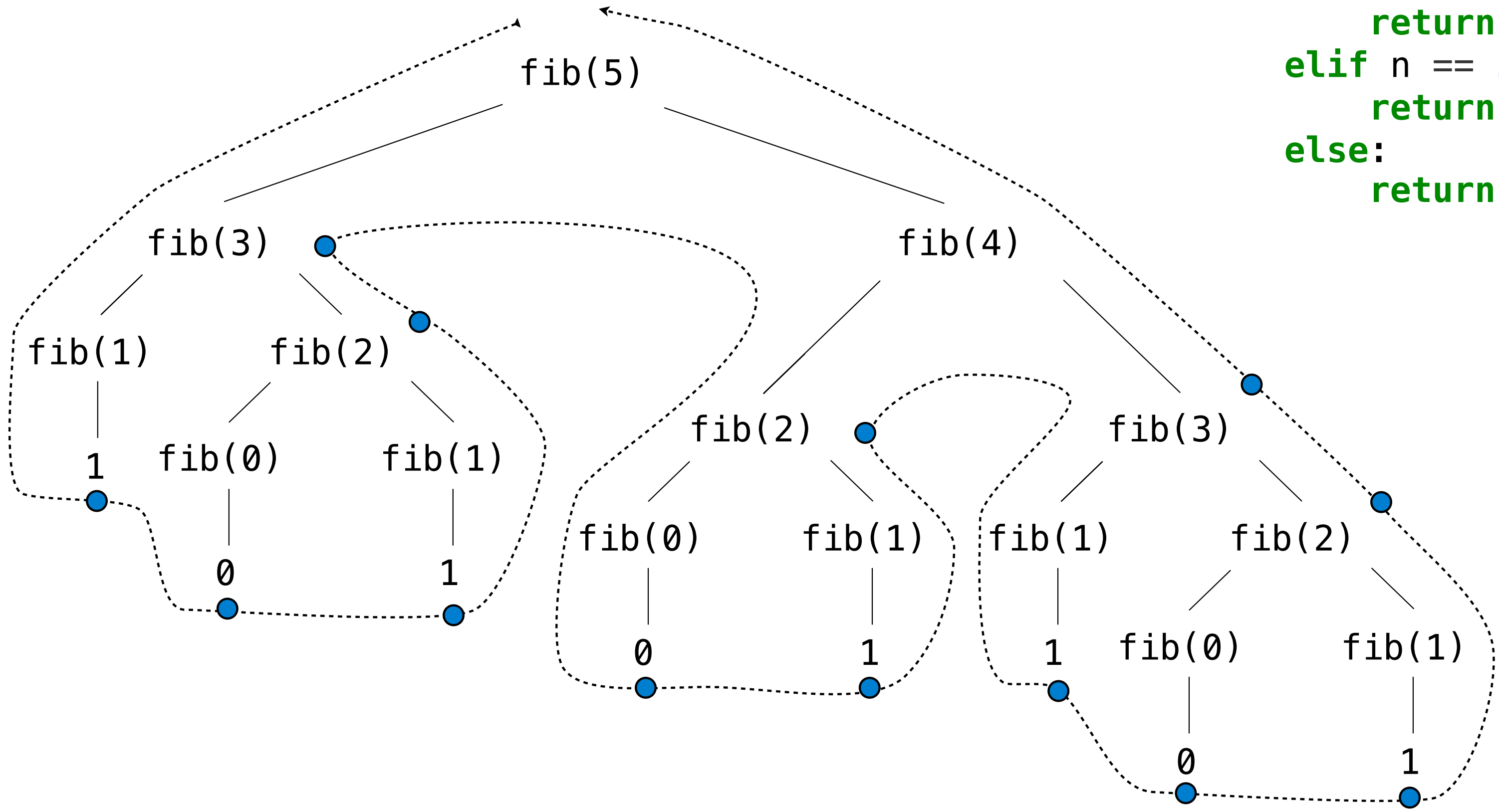

```python
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-2) + fib(n-1)
```

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

Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:

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

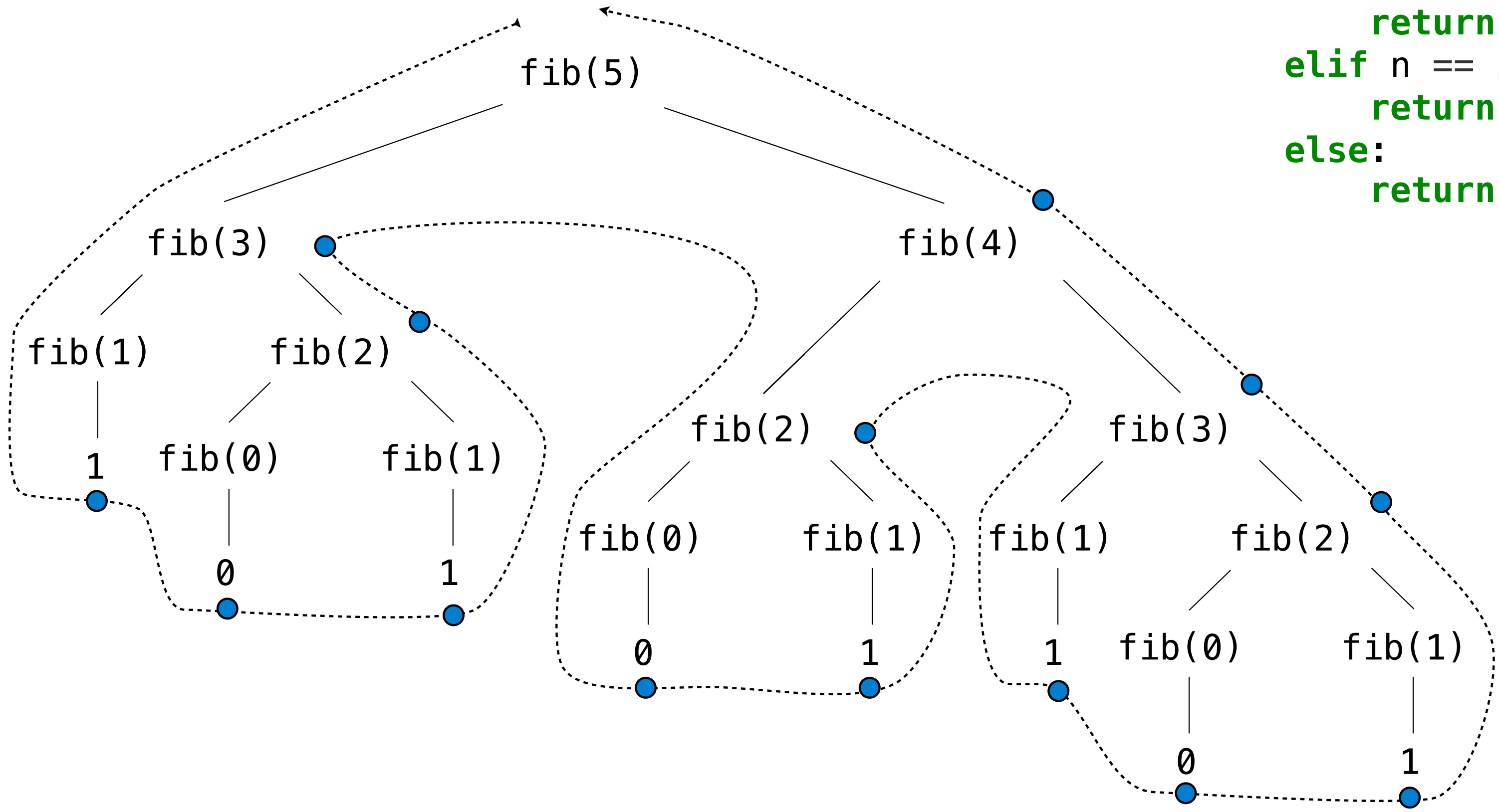


http://en.wikipedia.org/wiki/File:Fibonacci.jpg

Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:

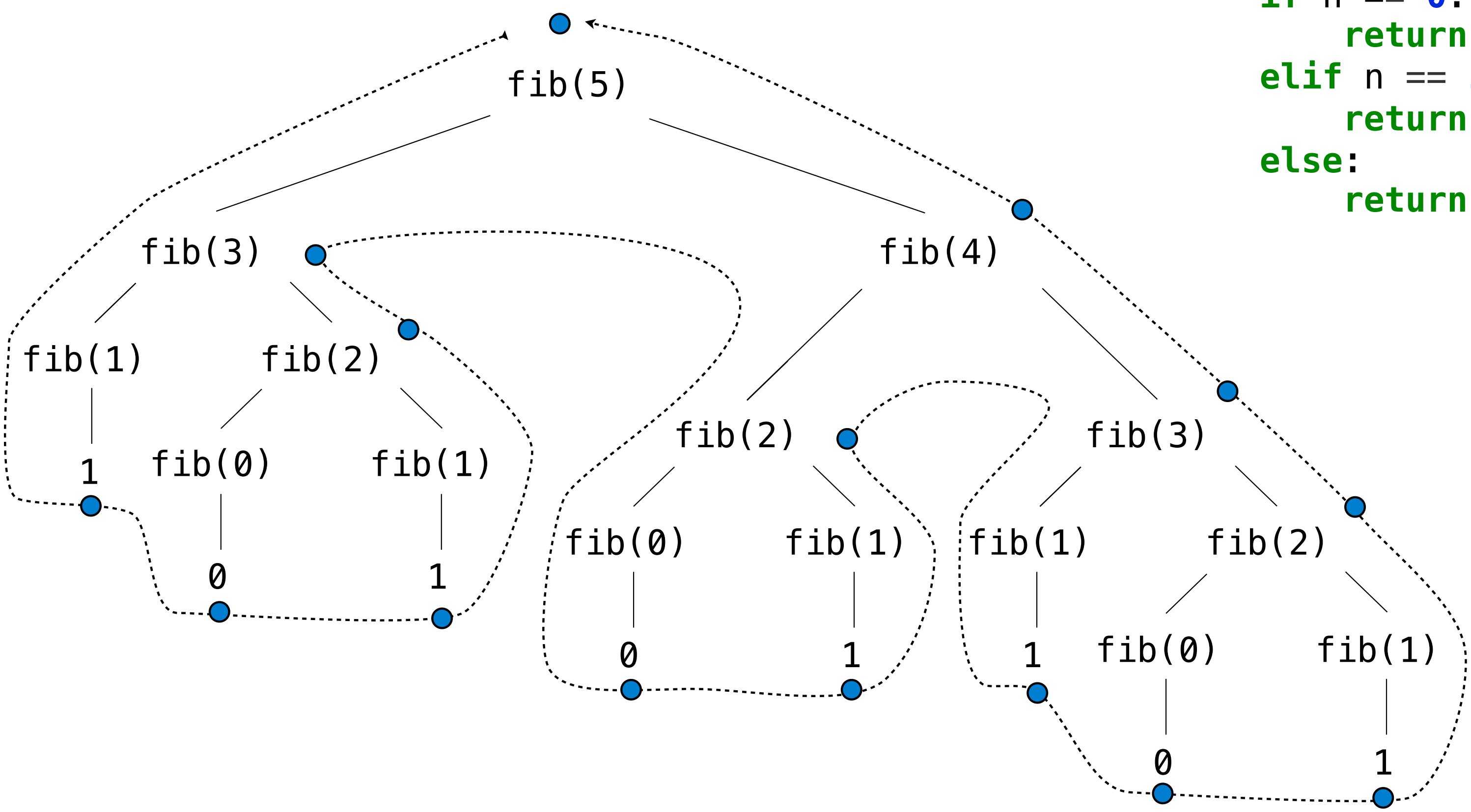
```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-2) + fib(n-1)
```



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

Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:

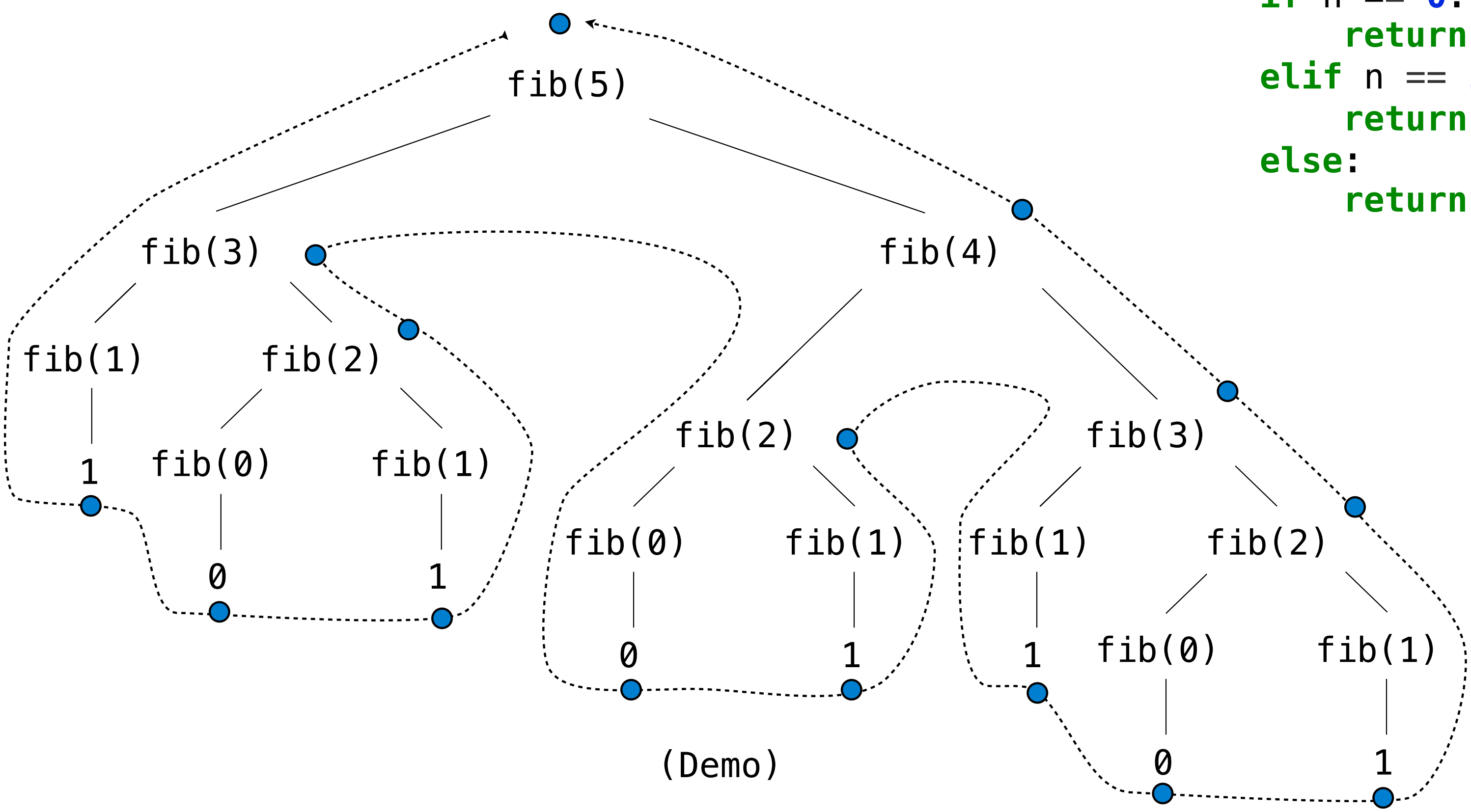



```python
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-2) + fib(n-1)
```

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

Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:




```
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

Memoization

Memoization

Idea: Remember the results that have been computed before

Memoization

Idea: Remember the results that have been computed before

```
def memo(f):
```

Memoization

Idea: Remember the results that have been computed before

```
def memo(f):
    cache = {}
```

Memoization

Idea: Remember the results that have been computed before

```
def memo(f):
    cache = {}
    def memoized(n):
```

Memoization

Idea: Remember the results that have been computed before

```
def memo(f):
    cache = {}
    def memoized(n):
        if n not in cache:
```

Memoization

Idea: Remember the results that have been computed before

```
def memo(f):
    cache = {}
    def memoized(n):
        if n not in cache:
            cache[n] = f(n)
```

Memoization

Idea: Remember the results that have been computed before

```
def memo(f):
    cache = {}
    def memoized(n):
        if n not in cache:
            cache[n] = f(n)
        return cache[n]
```

Memoization

Idea: Remember the results that have been computed before

```
def memo(f):
    cache = {}
    def memoized(n):
        if n not in cache:
            cache[n] = f(n)
        return cache[n]
    return memoized
```

Memoization

Idea: Remember the results that have been computed before

```
def memo(f):
    cache = {}
    def memoized(n):
        if n not in cache:
            cache[n] = f(n)
        return cache[n]
    return memoized
```

Keys are arguments that map to return values

Memoization

Idea: Remember the results that have been computed before

```
def memo(f):
    cache = {}
    def memoized(n):
        if n not in cache:
            cache[n] = f(n)
        return cache[n]
    return memoized
```

Keys are arguments that map to return values

Same behavior as f, if f is a pure function

Memoization

Idea: Remember the results that have been computed before

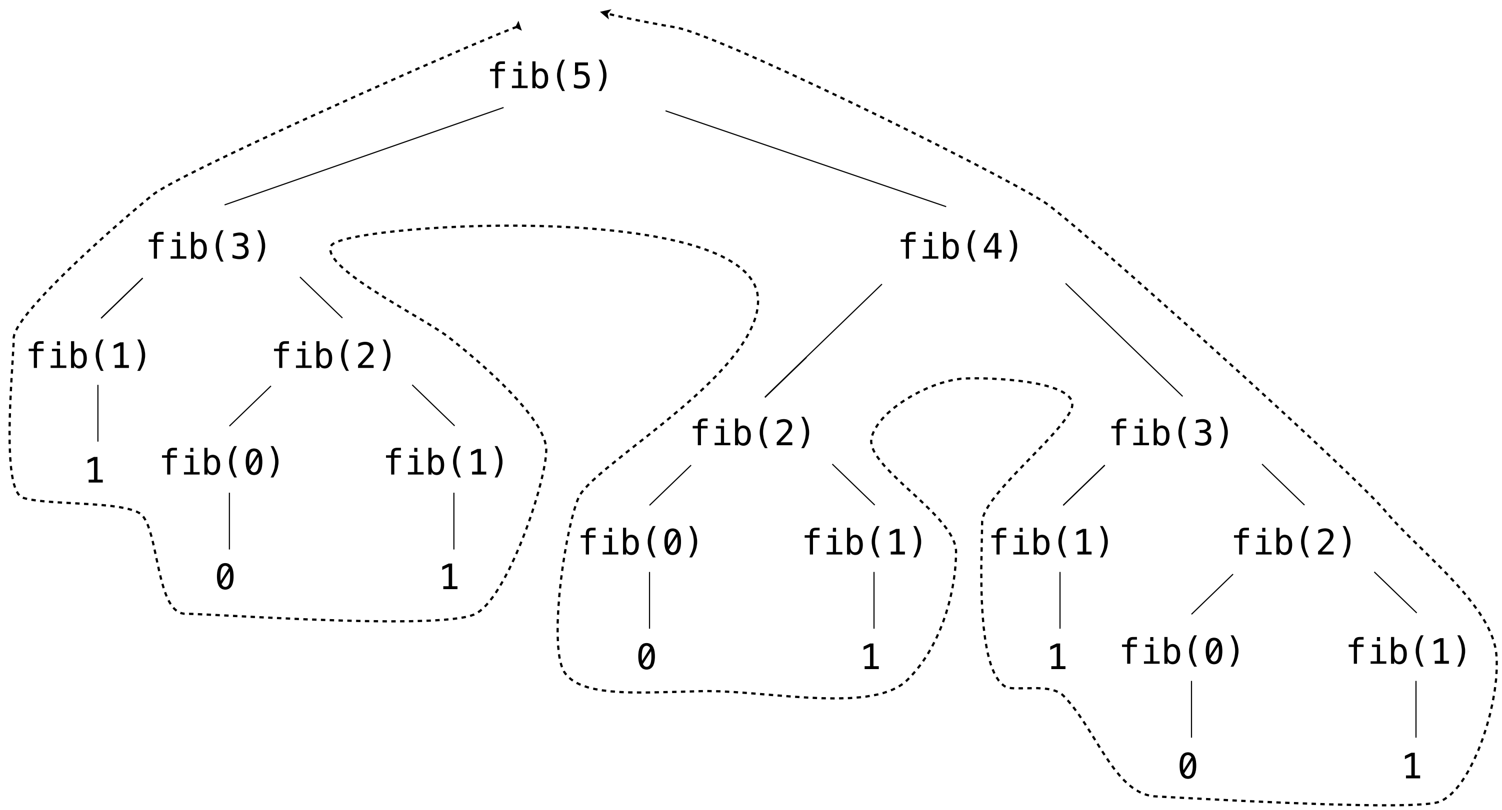
```
def memo(f):
    cache = {}
    def memoized(n):
        if n not in cache:
            cache[n] = f(n)
        return cache[n]
    return memoized
```

Keys are arguments that map to return values

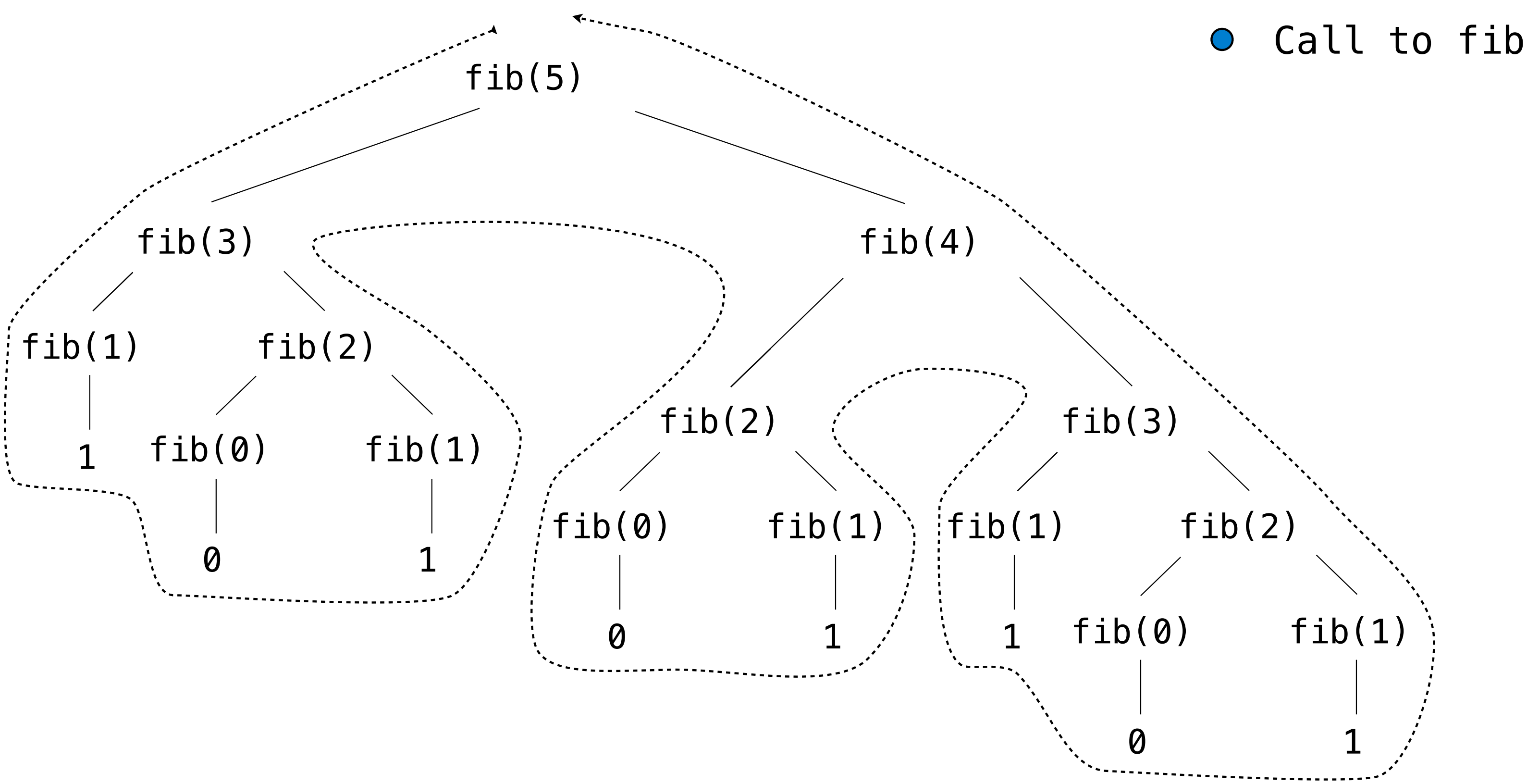
Same behavior as f, if f is a pure function

(Demo)

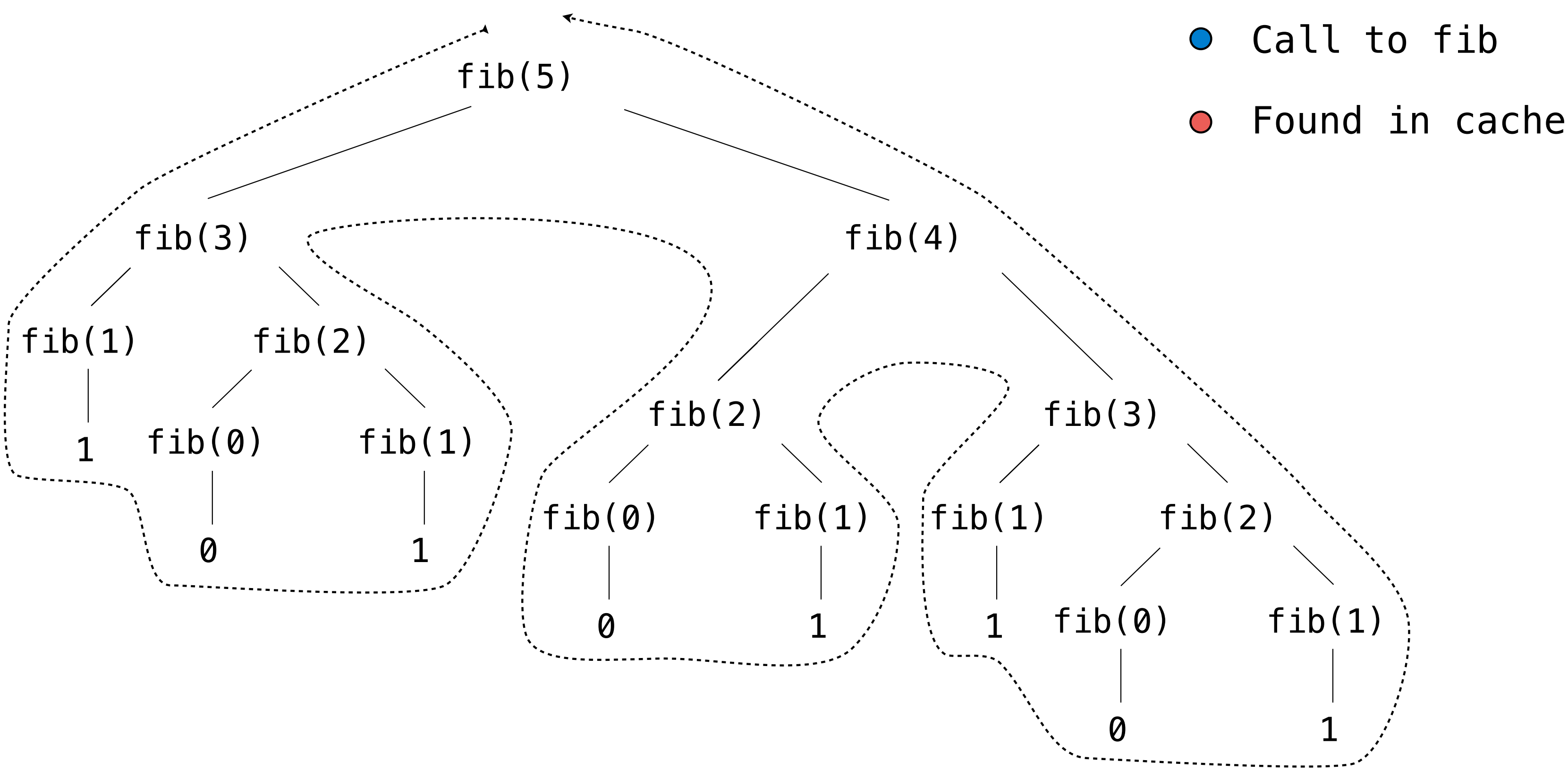
Memoized Tree Recursion



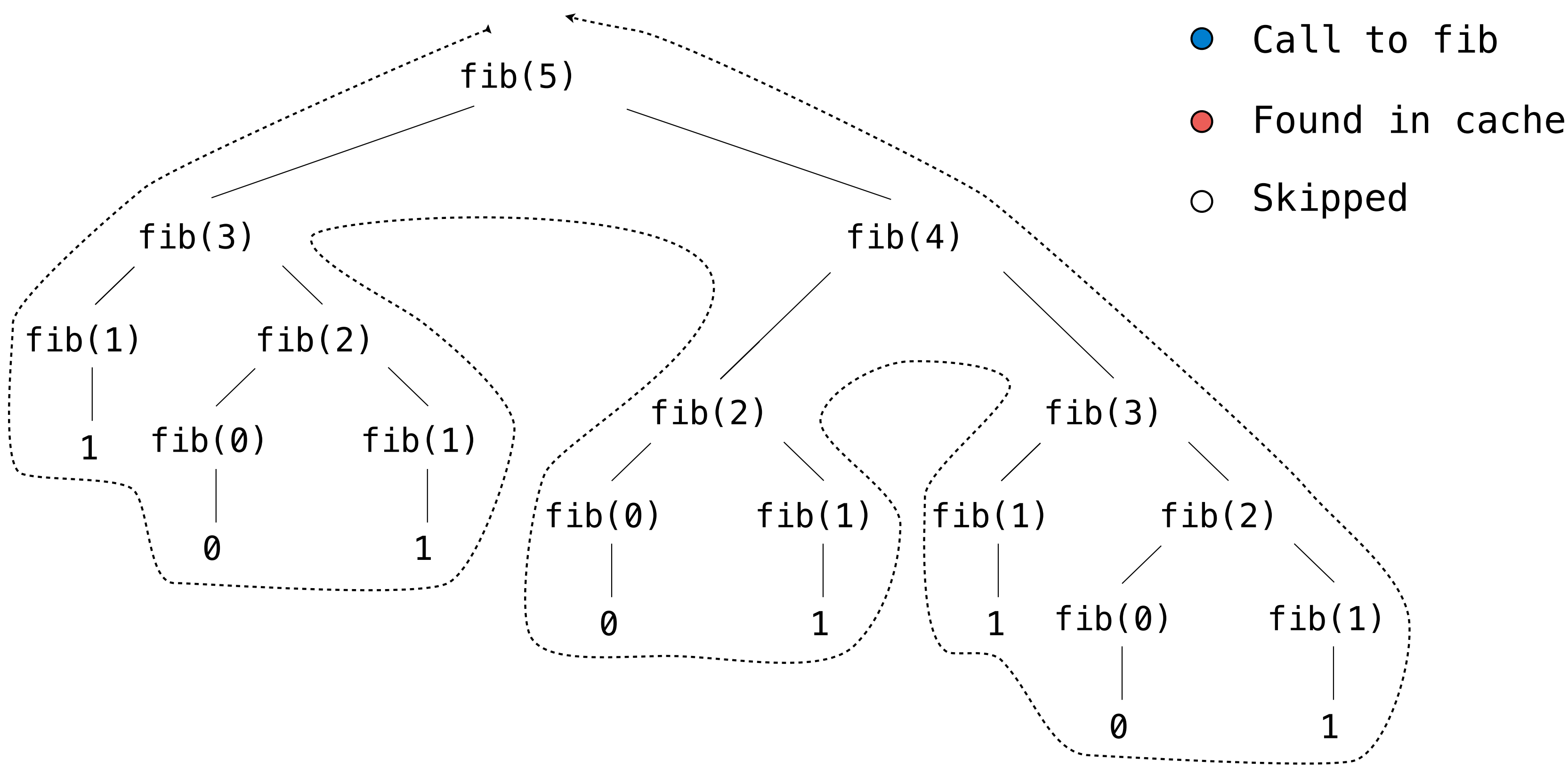
Memoized Tree Recursion



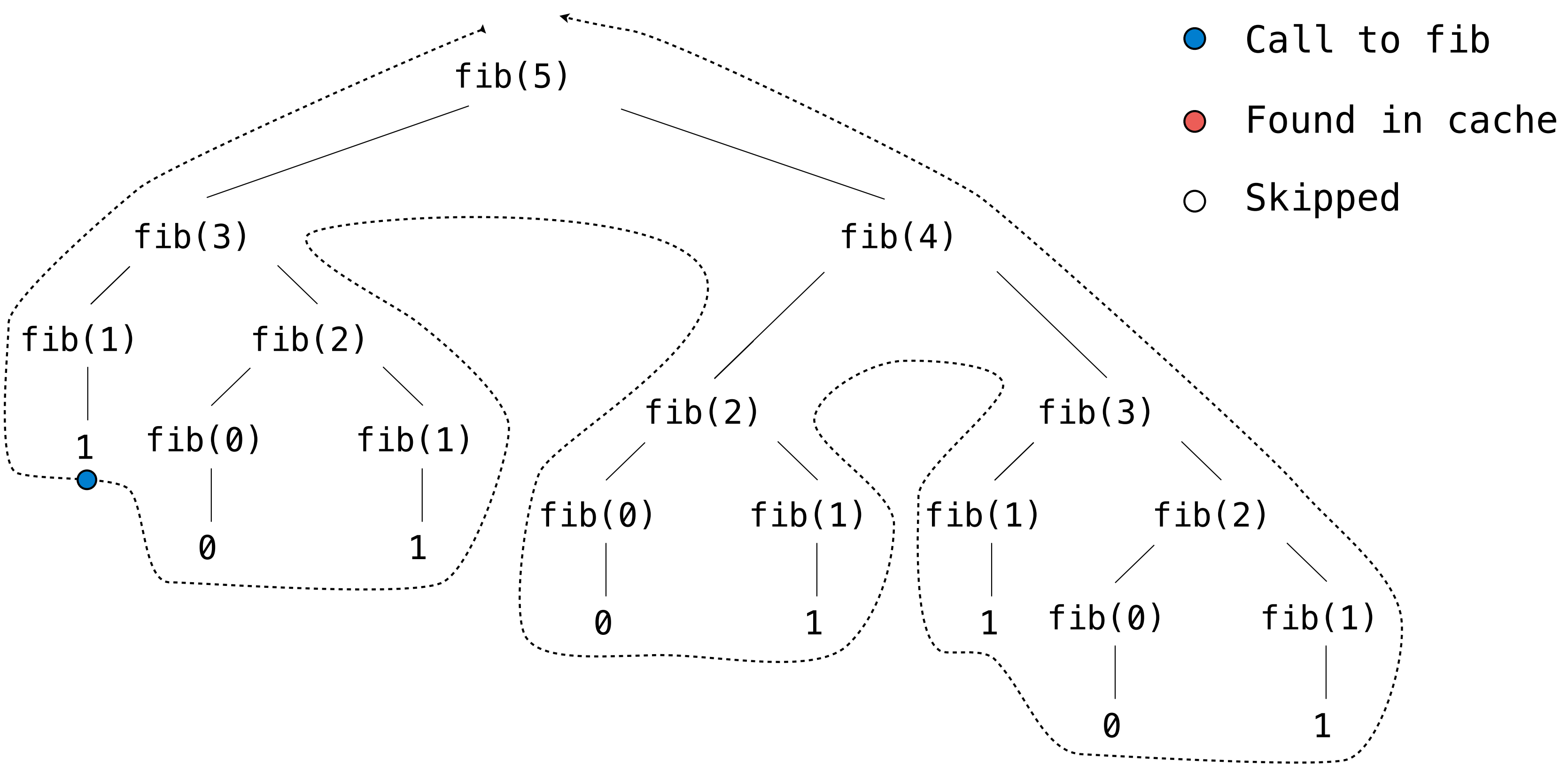
Memoized Tree Recursion



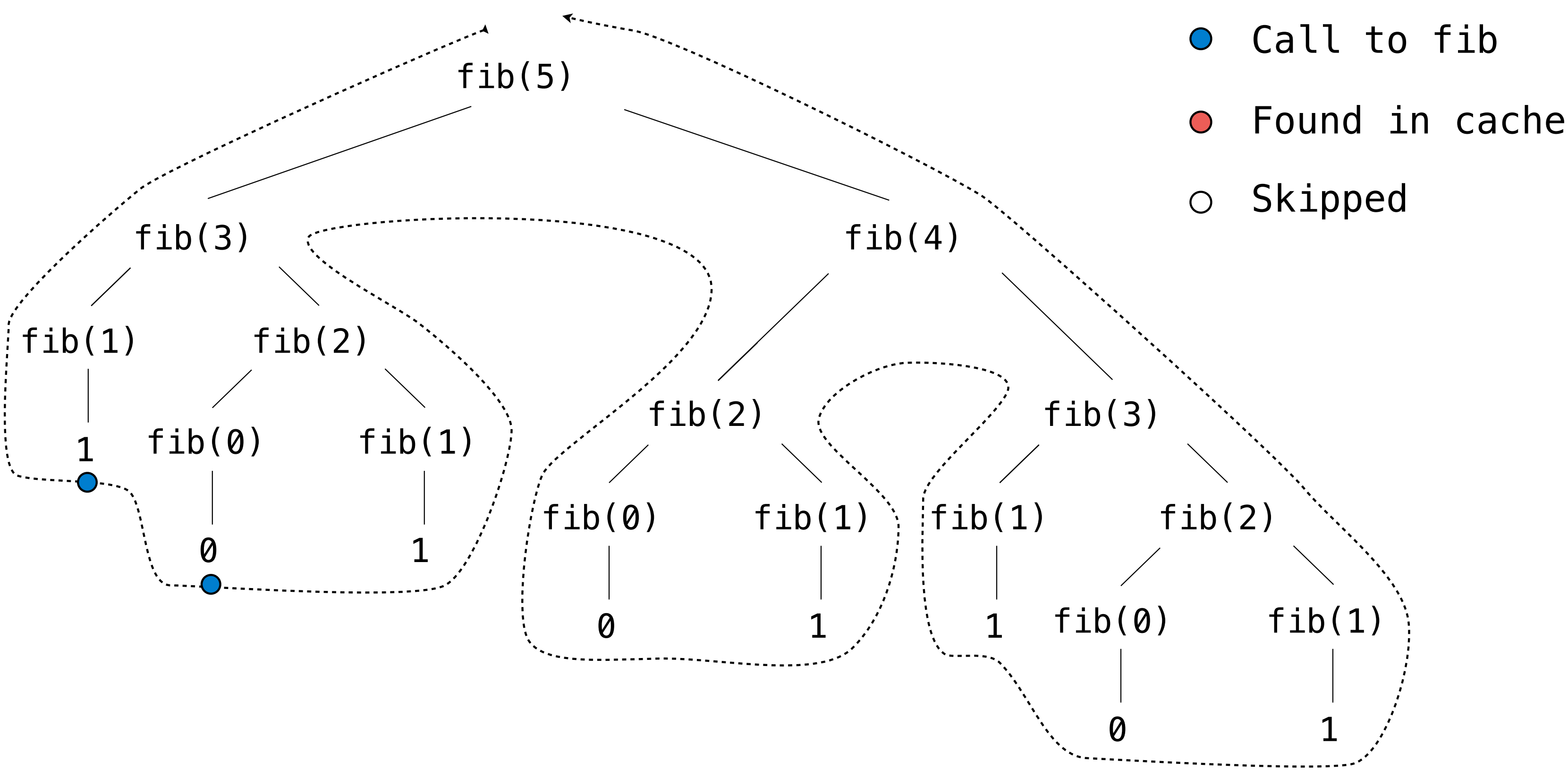
Memoized Tree Recursion



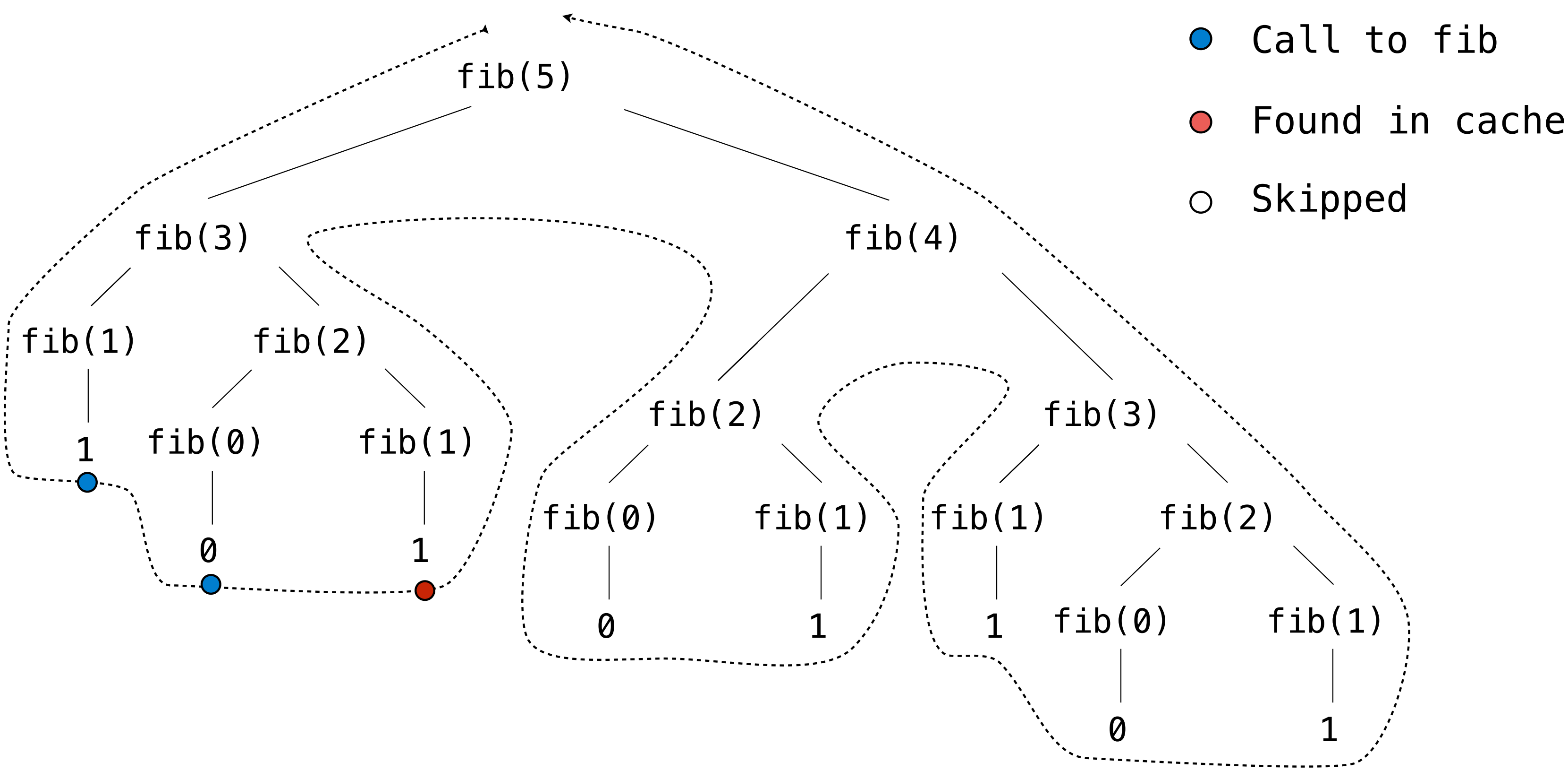
Memoized Tree Recursion



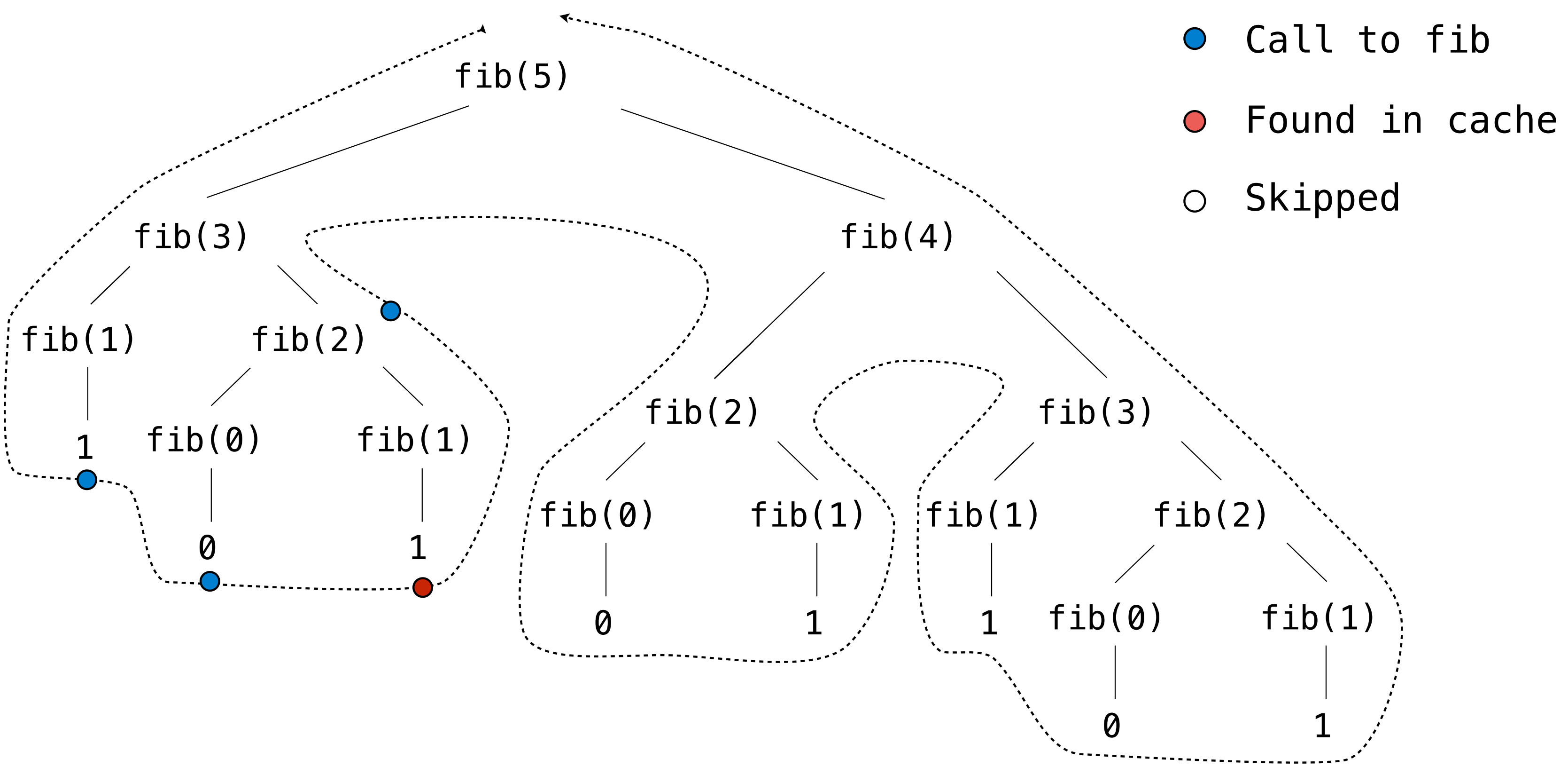
Memoized Tree Recursion



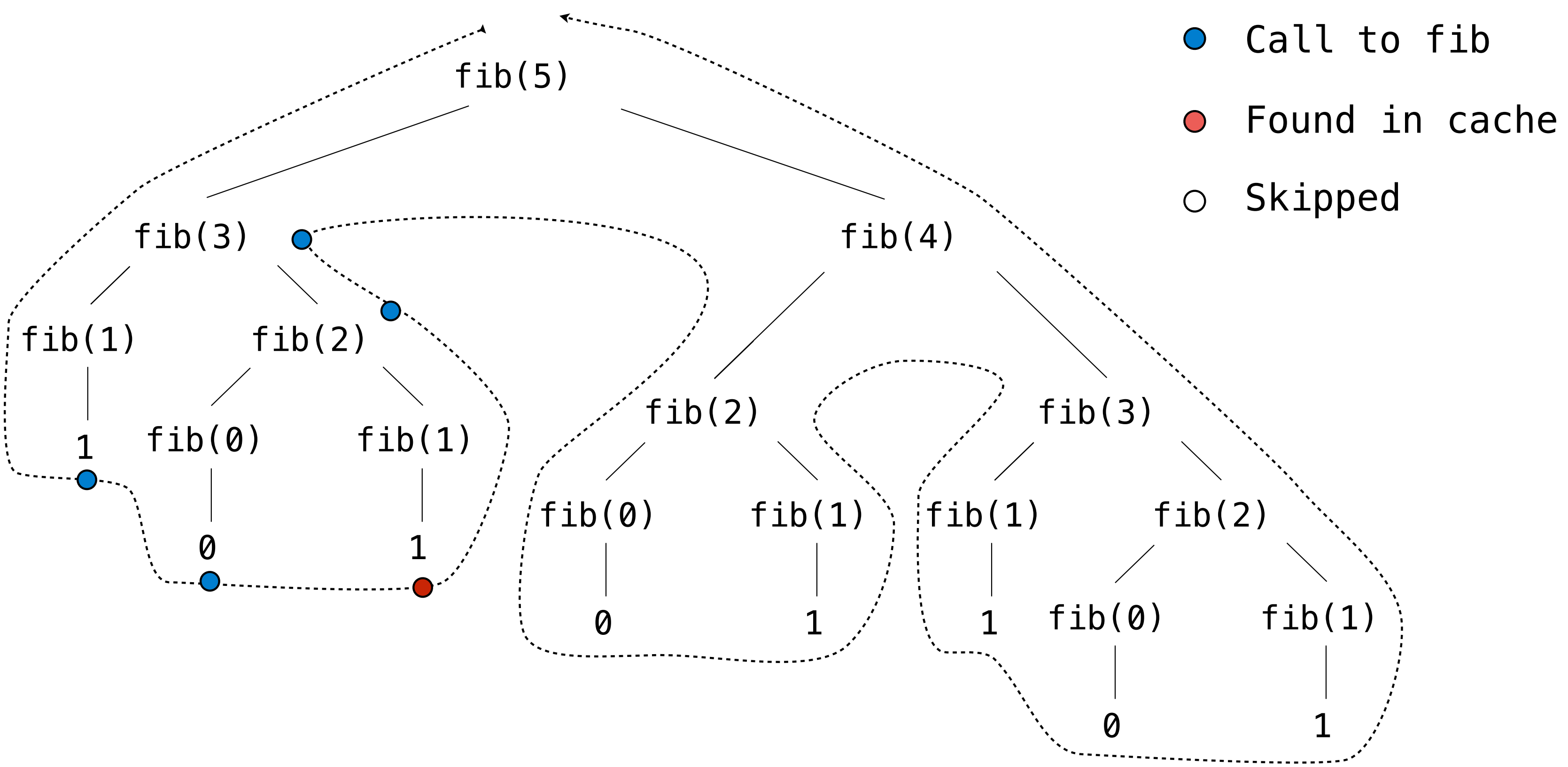
Memoized Tree Recursion



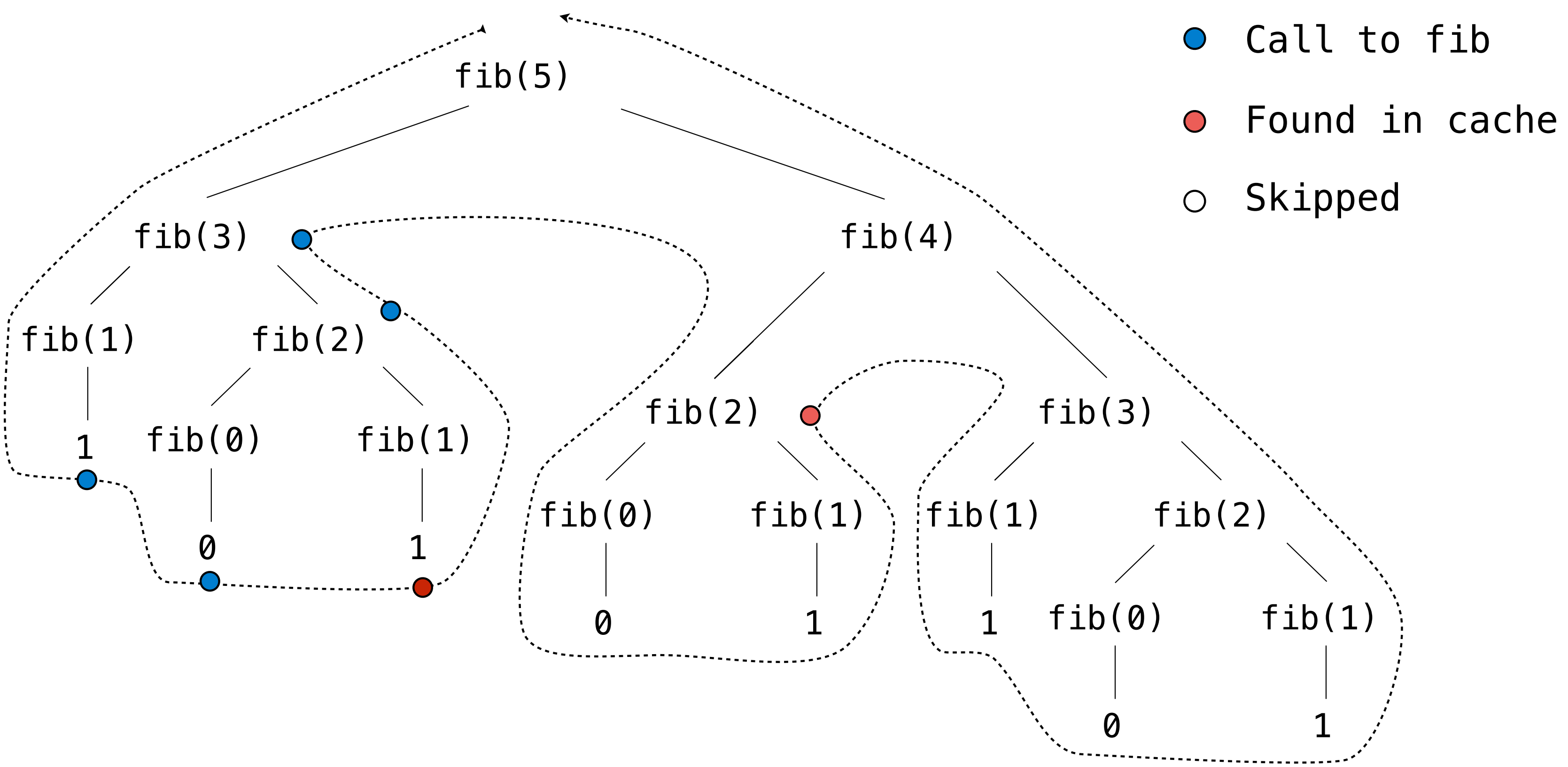
Memoized Tree Recursion



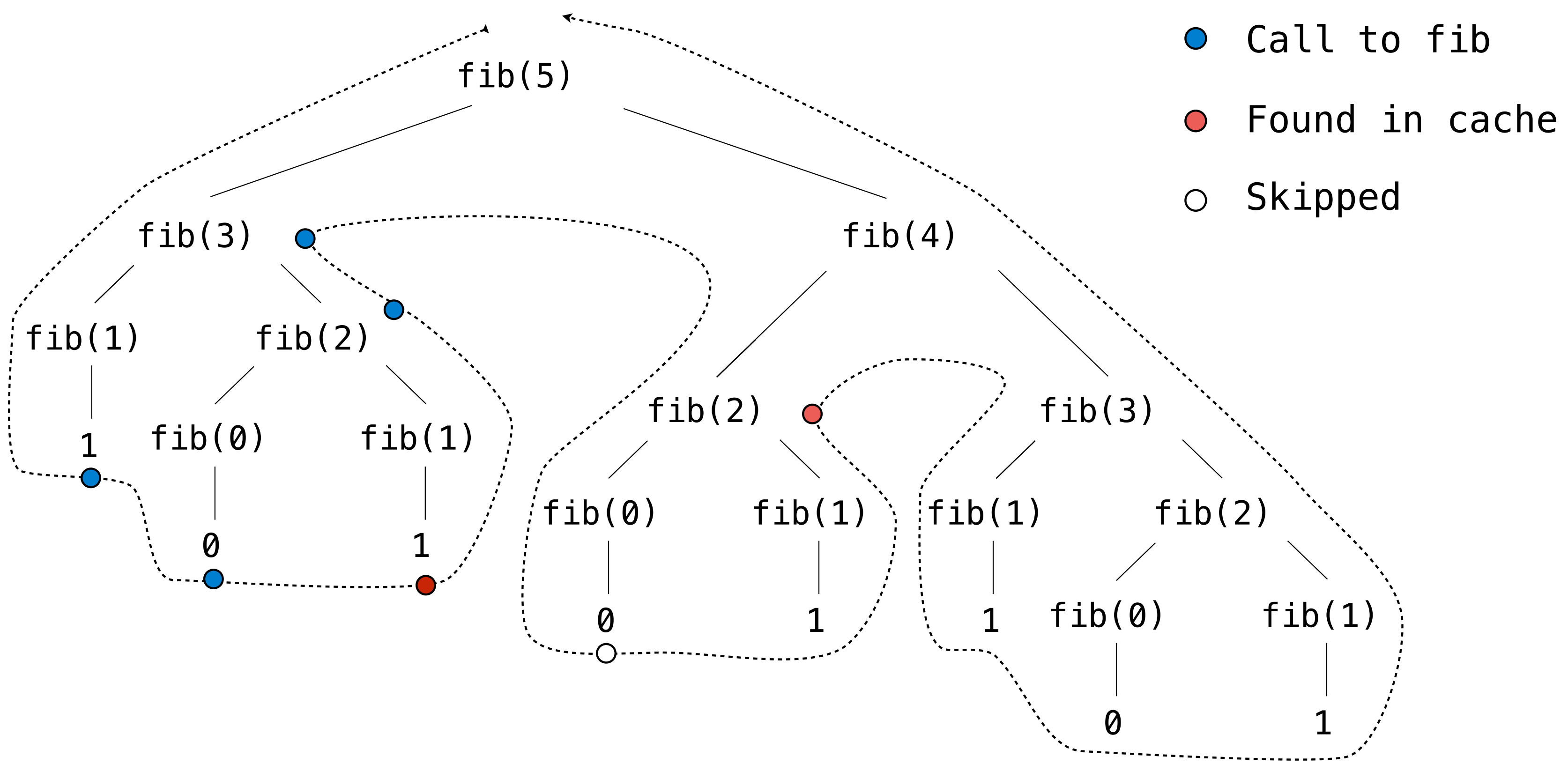
Memoized Tree Recursion



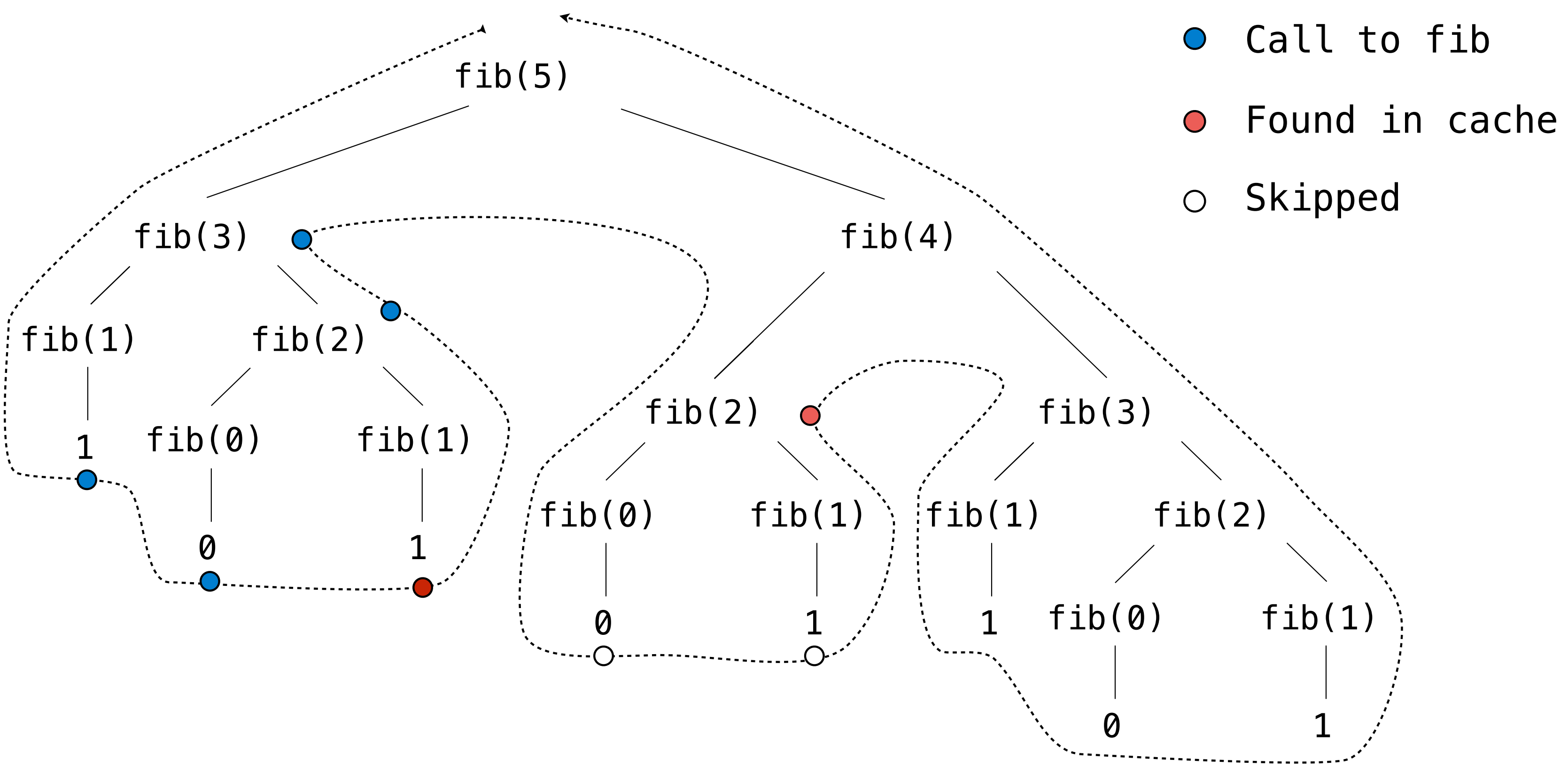
Memoized Tree Recursion



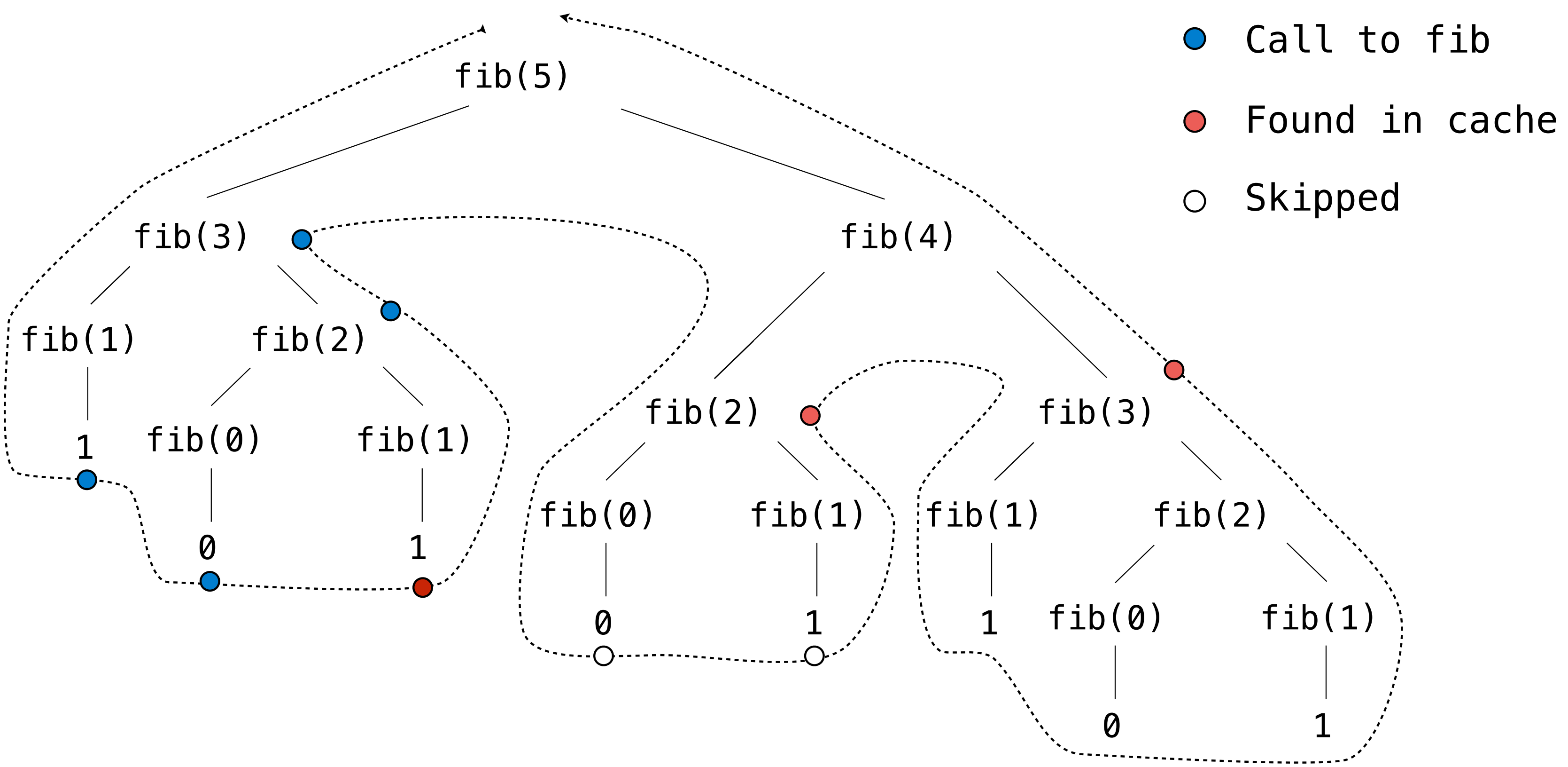
Memoized Tree Recursion



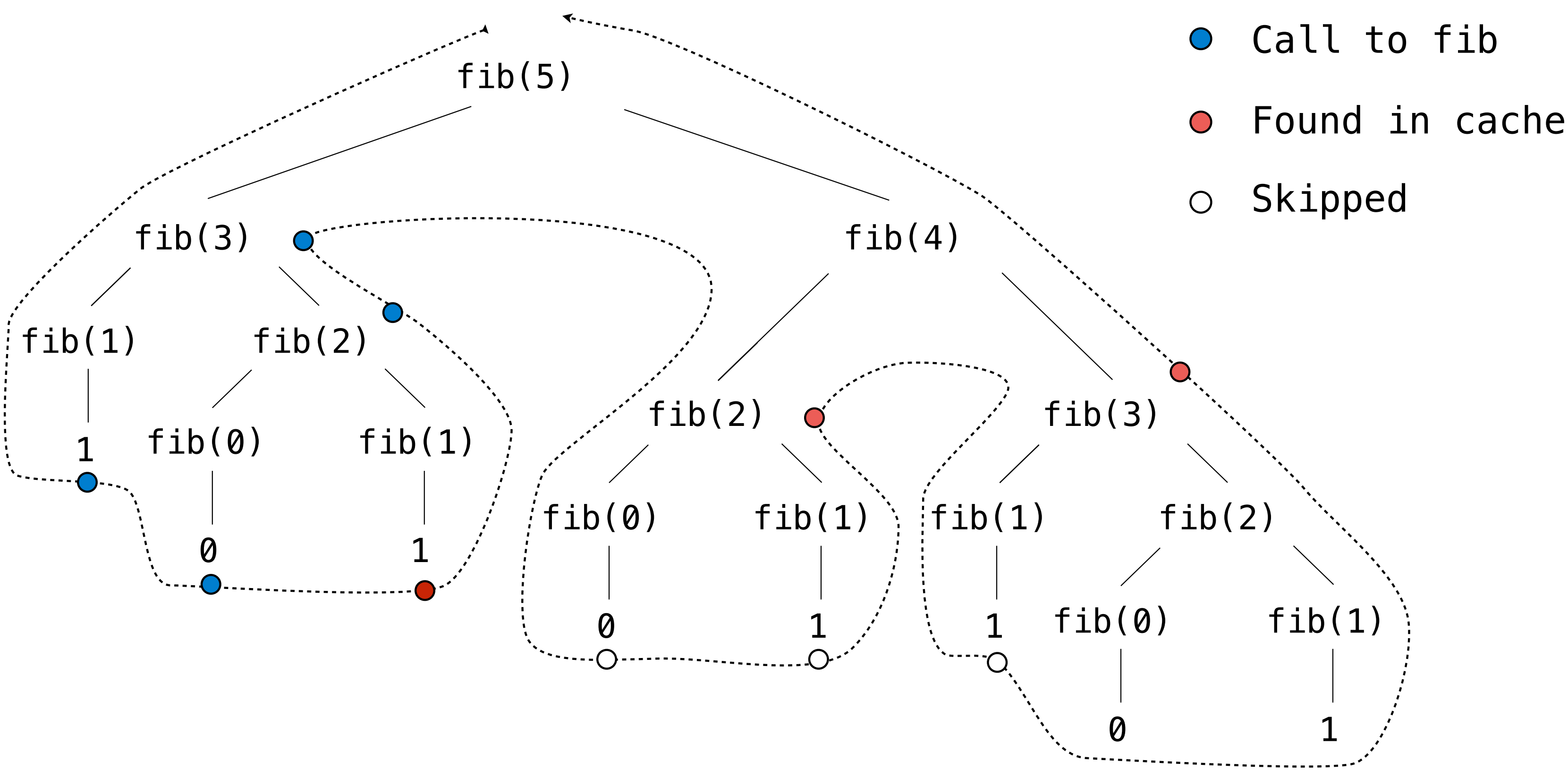
Memoized Tree Recursion



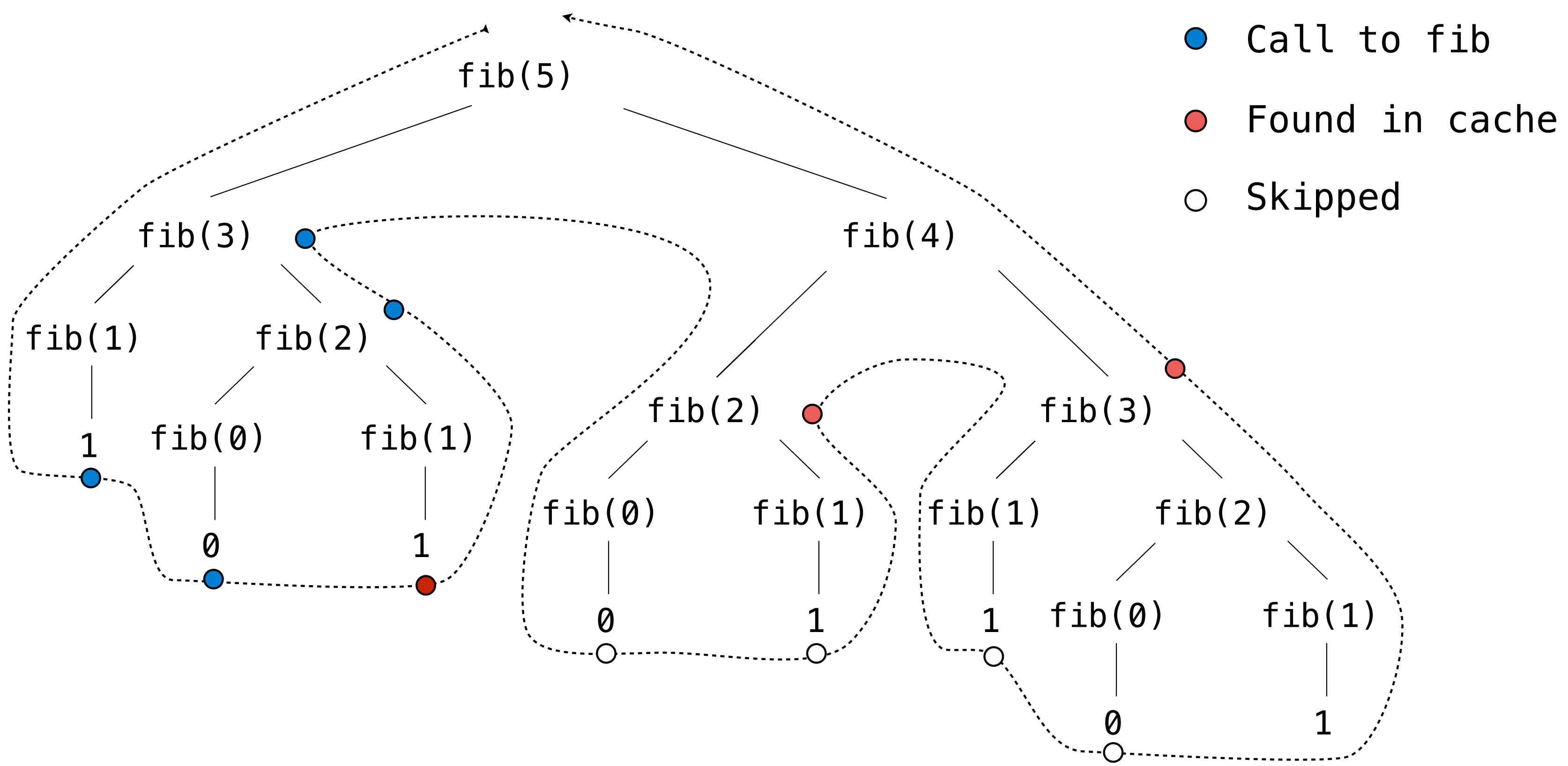
Memoized Tree Recursion



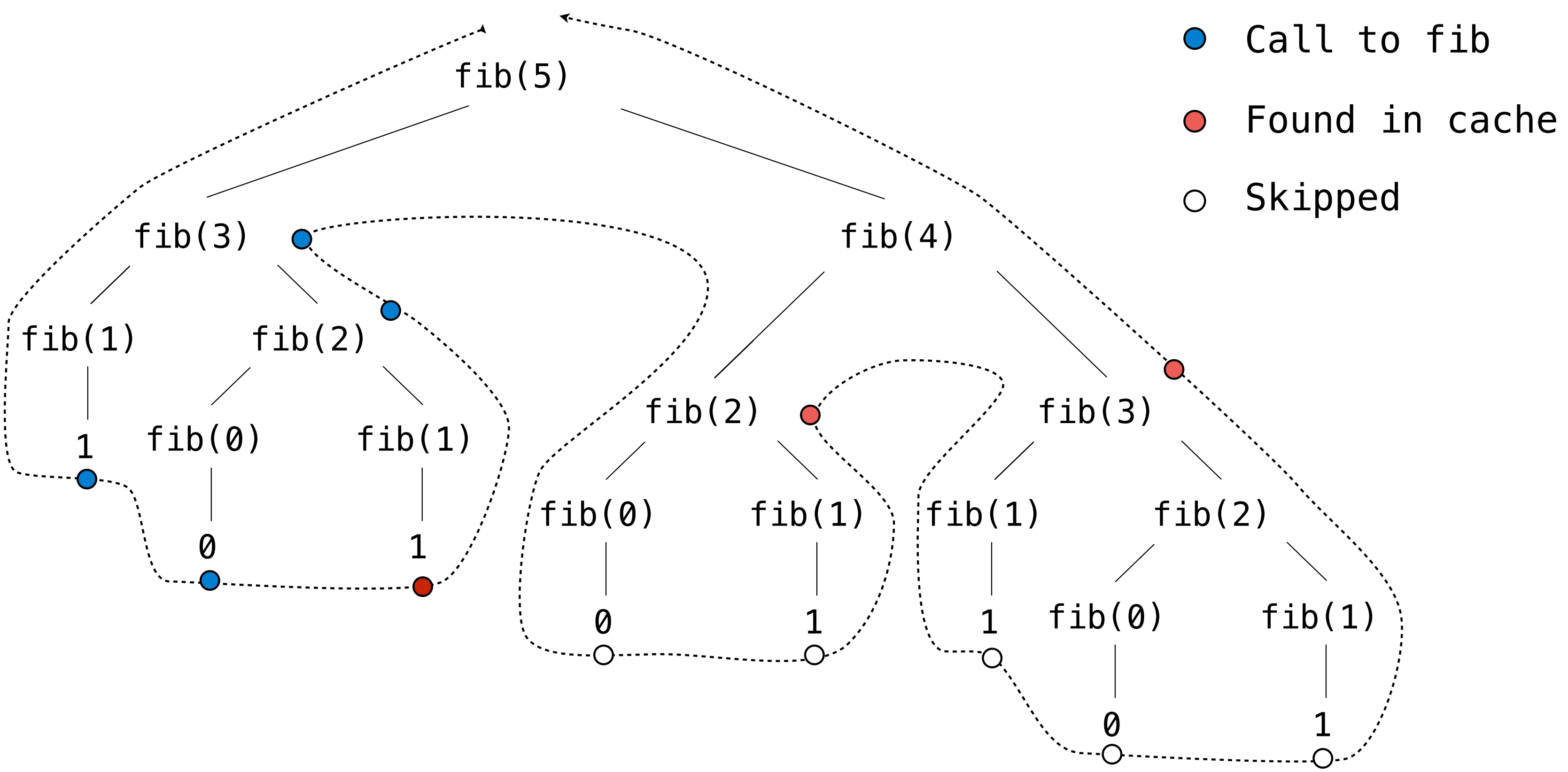
Memoized Tree Recursion



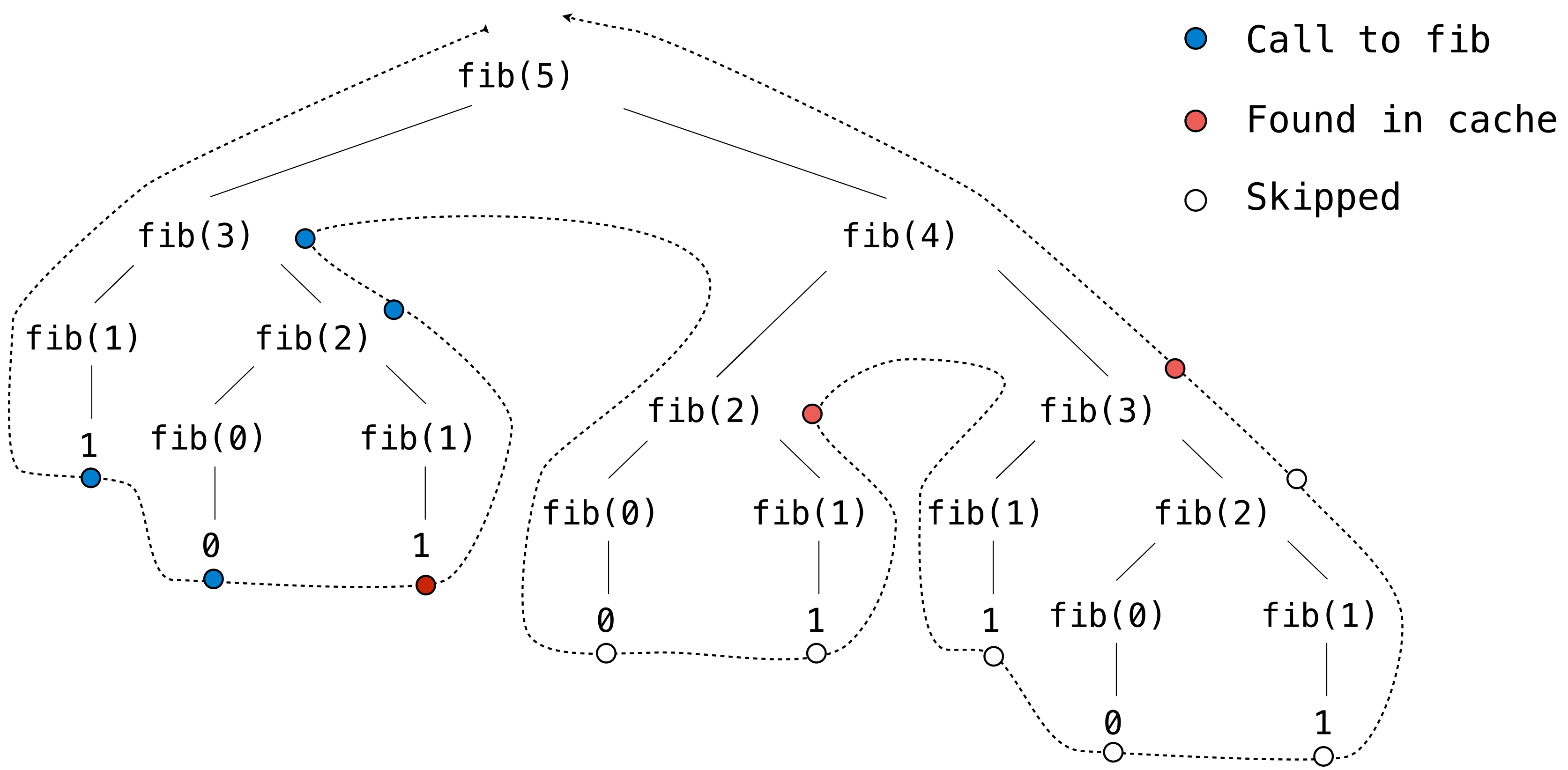
Memoized Tree Recursion



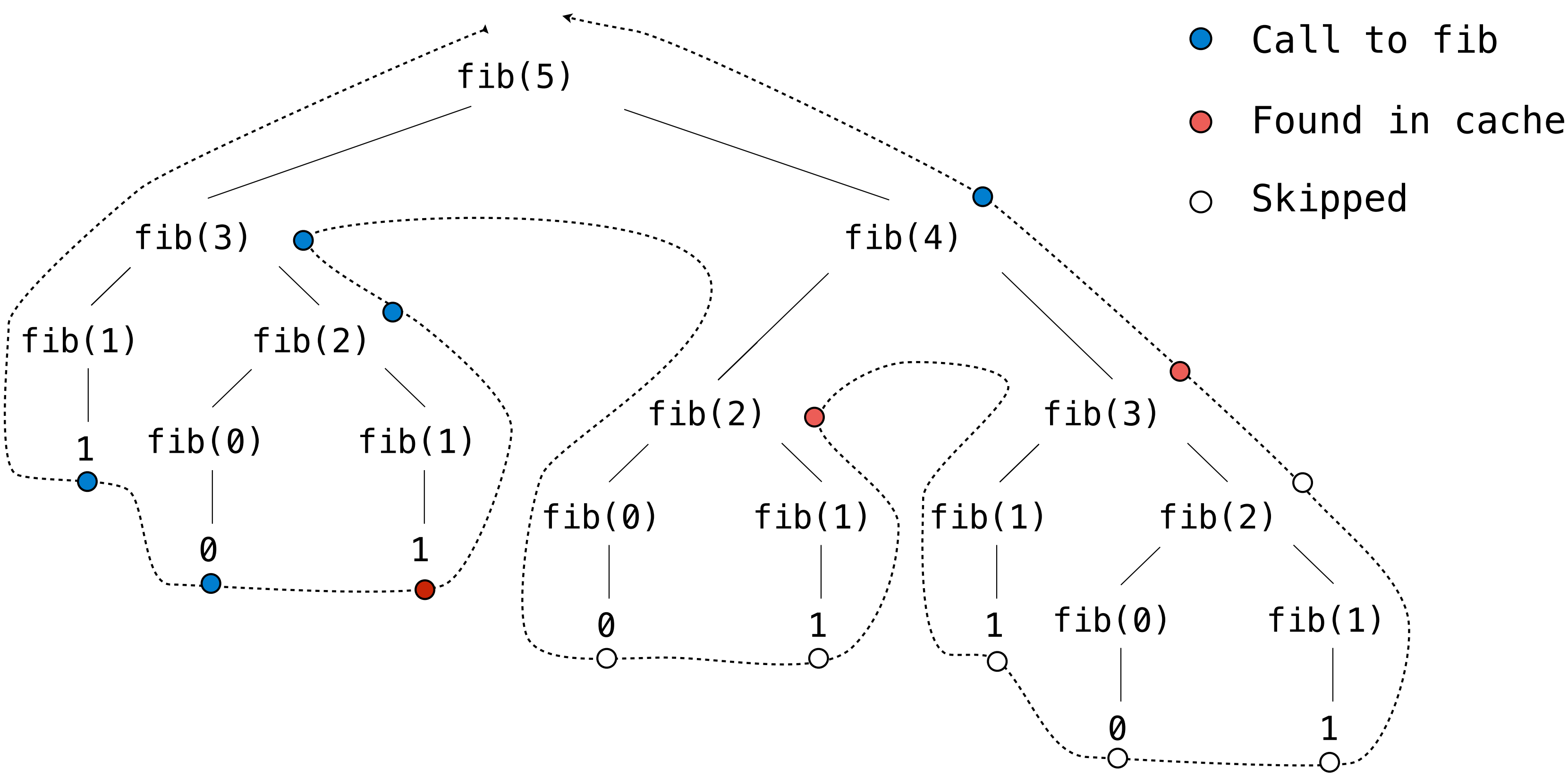
Memoized Tree Recursion



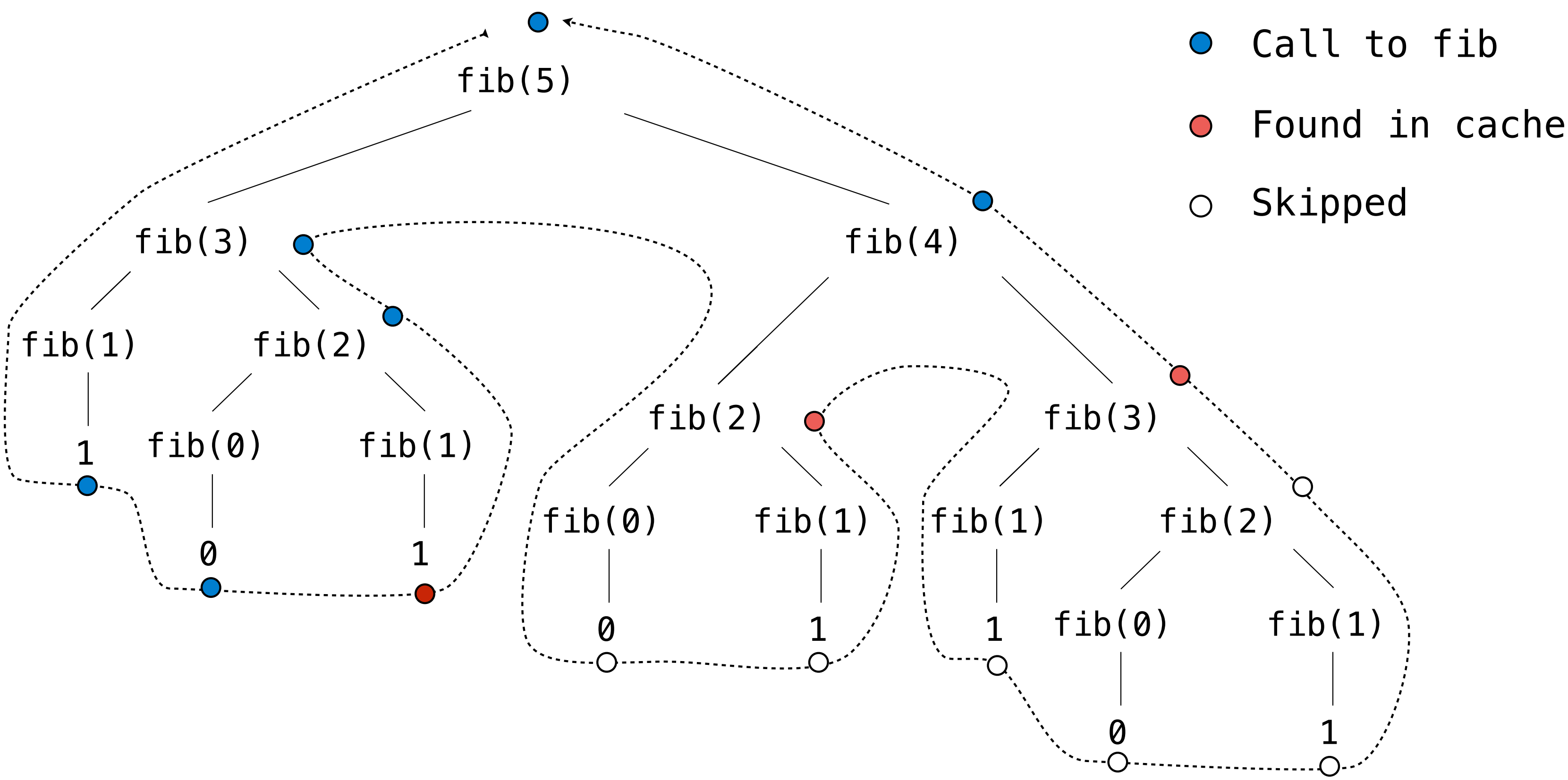
Memoized Tree Recursion



Memoized Tree Recursion



Memoized Tree Recursion



Space

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?

At any moment there is a set of active environments

The Consumption of Space

Which environment frames do we need to keep during evaluation?

At any moment there is 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?

At any moment there is a set of active environments

Values and frames in active environments consume memory

Memory that is used for other values and frames can be recycled

The Consumption of Space

Which environment frames do we need to keep during evaluation?

At any moment there is a set of active environments

Values and frames in active environments consume memory

Memory that is used for other values and frames can be recycled

Active environments:

The Consumption of Space

Which environment frames do we need to keep during evaluation?

At any moment there is a set of active environments

Values and frames in active environments consume memory

Memory that is used for other values and frames can be recycled

Active environments:

- Environments for any function calls currently being evaluated

The Consumption of Space

Which environment frames do we need to keep during evaluation?

At any moment there is a set of active environments

Values and frames in active environments consume memory

Memory that is used for other values and frames can be recycled

Active environments:

- Environments for any function calls currently being evaluated
- Parent environments of functions named in active environments

The Consumption of Space

Which environment frames do we need to keep during evaluation?

At any moment there is a set of active environments

Values and frames in active environments consume memory

Memory that is used for other values and frames can be recycled

Active environments:

- Environments for any function calls currently being evaluated
- Parent environments of functions named in active environments

(Demo)

Interactive Diagram

Fibonacci Space Consumption

Fibonacci Space Consumption

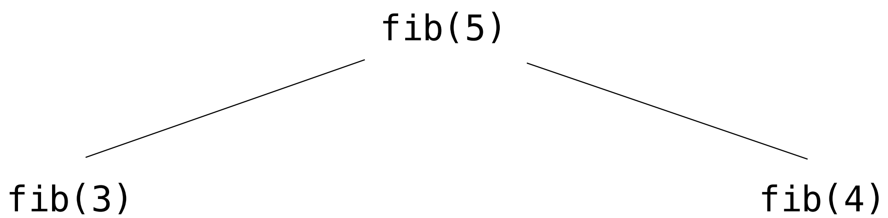
```
fib(5)
```

Fibonacci Space Consumption

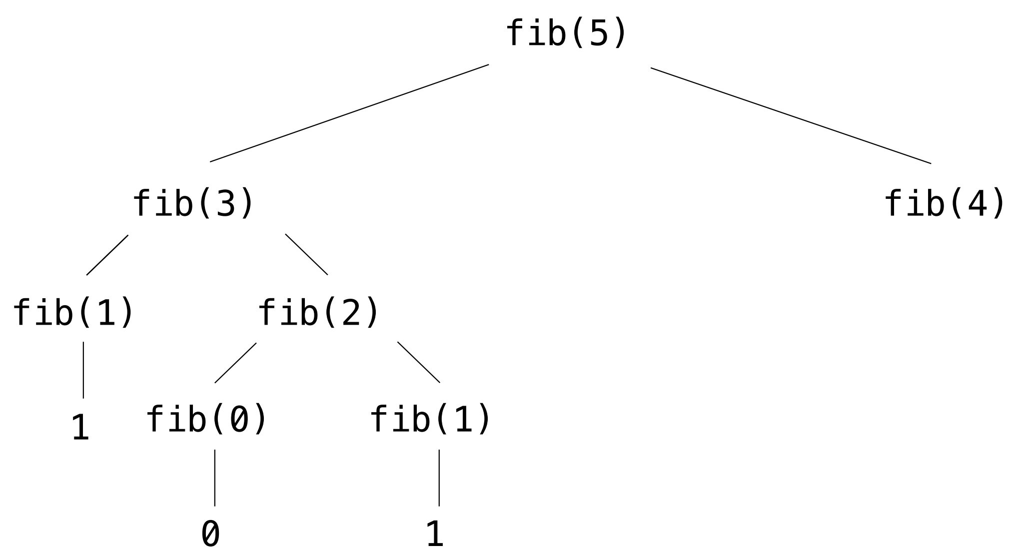
fib(5)

fib(3)

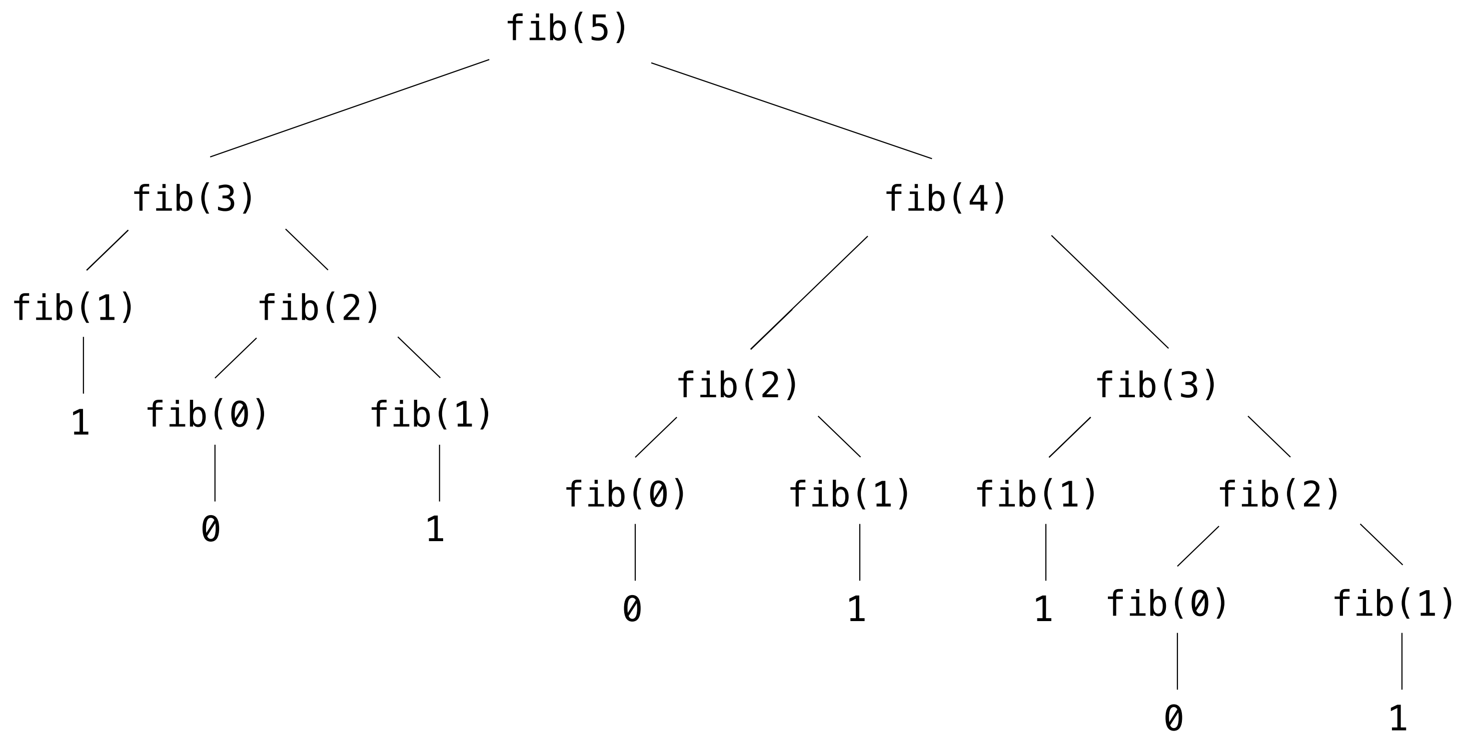
Fibonacci Space Consumption



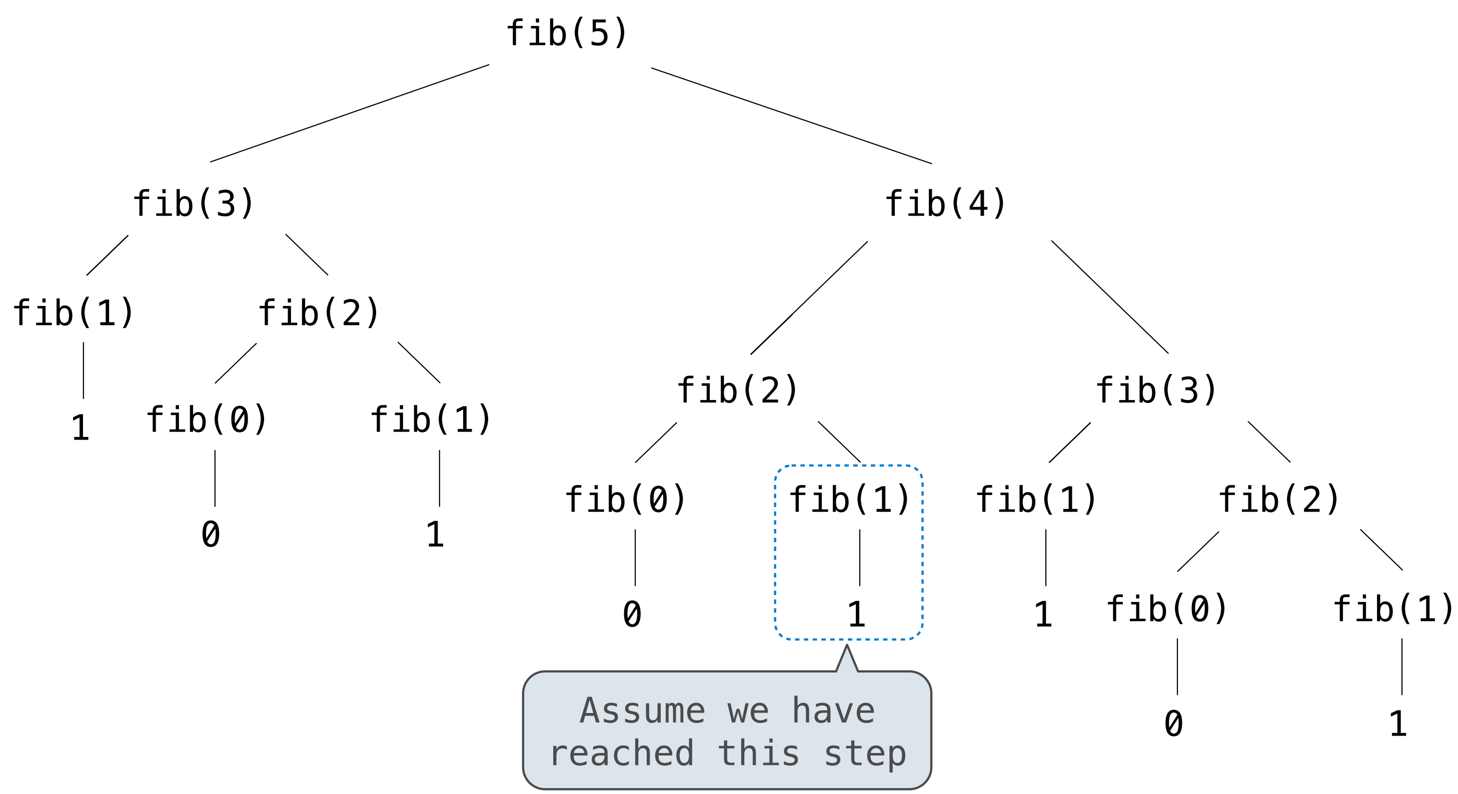
Fibonacci Space Consumption



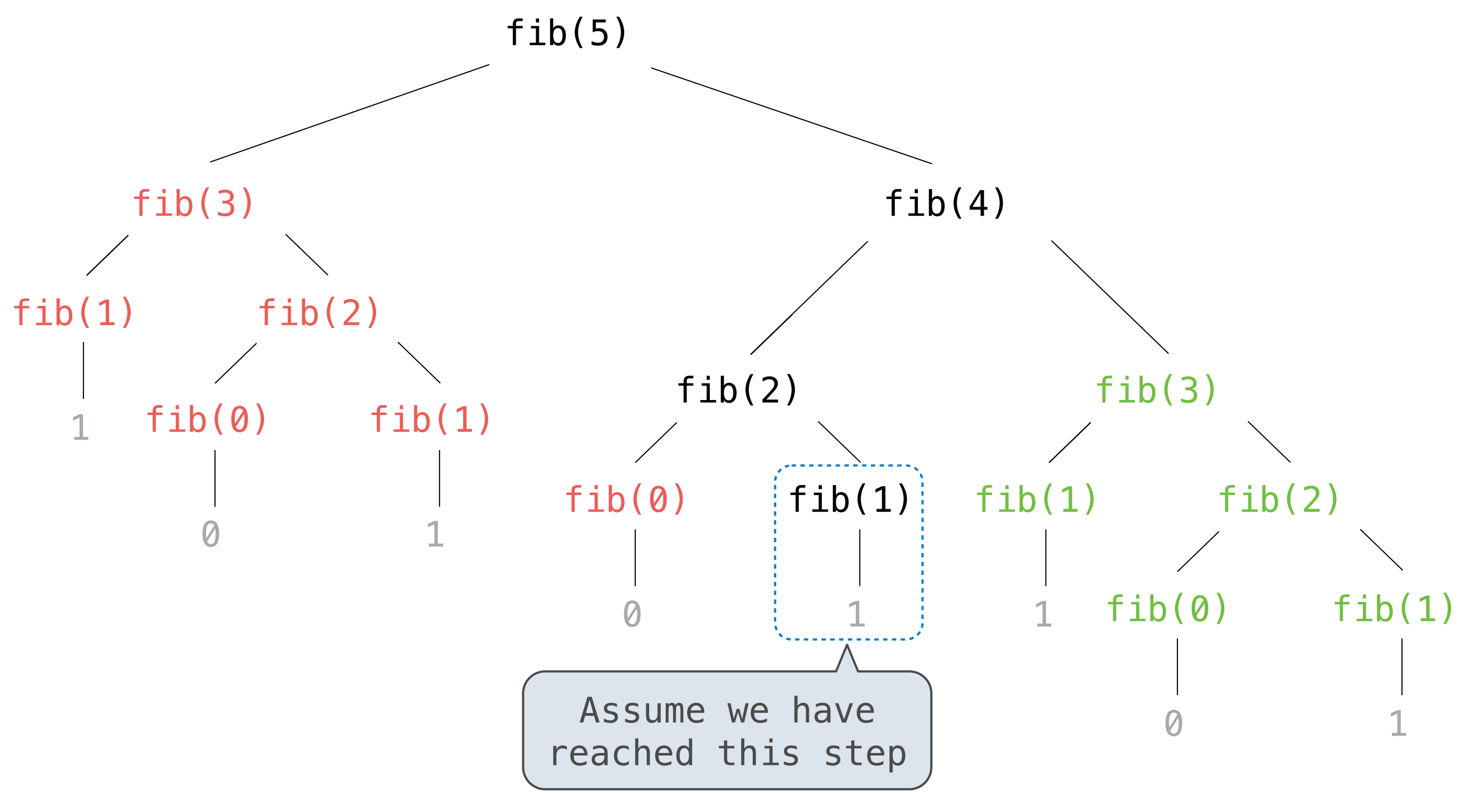
Fibonacci Space Consumption



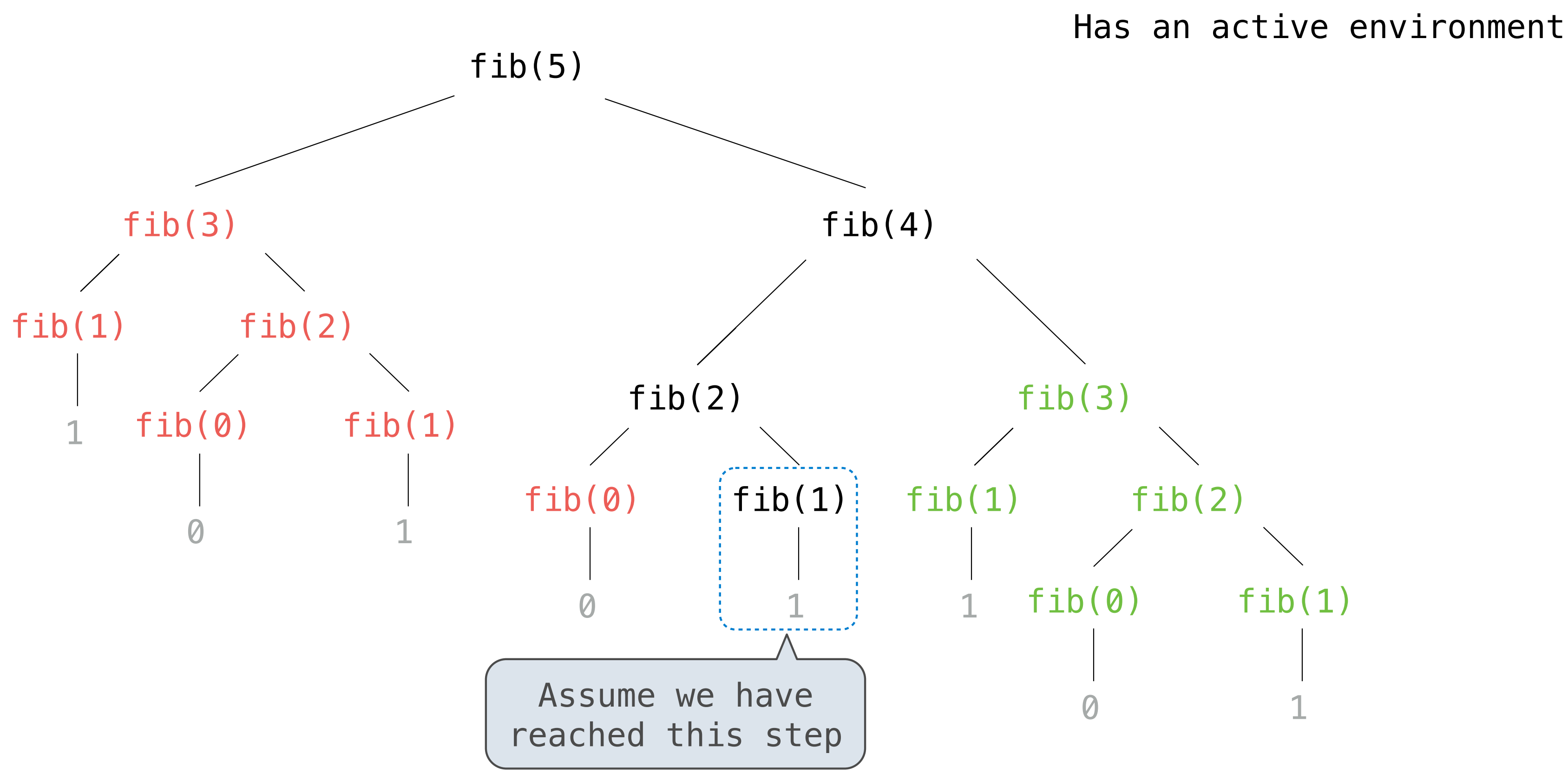
Fibonacci Space Consumption



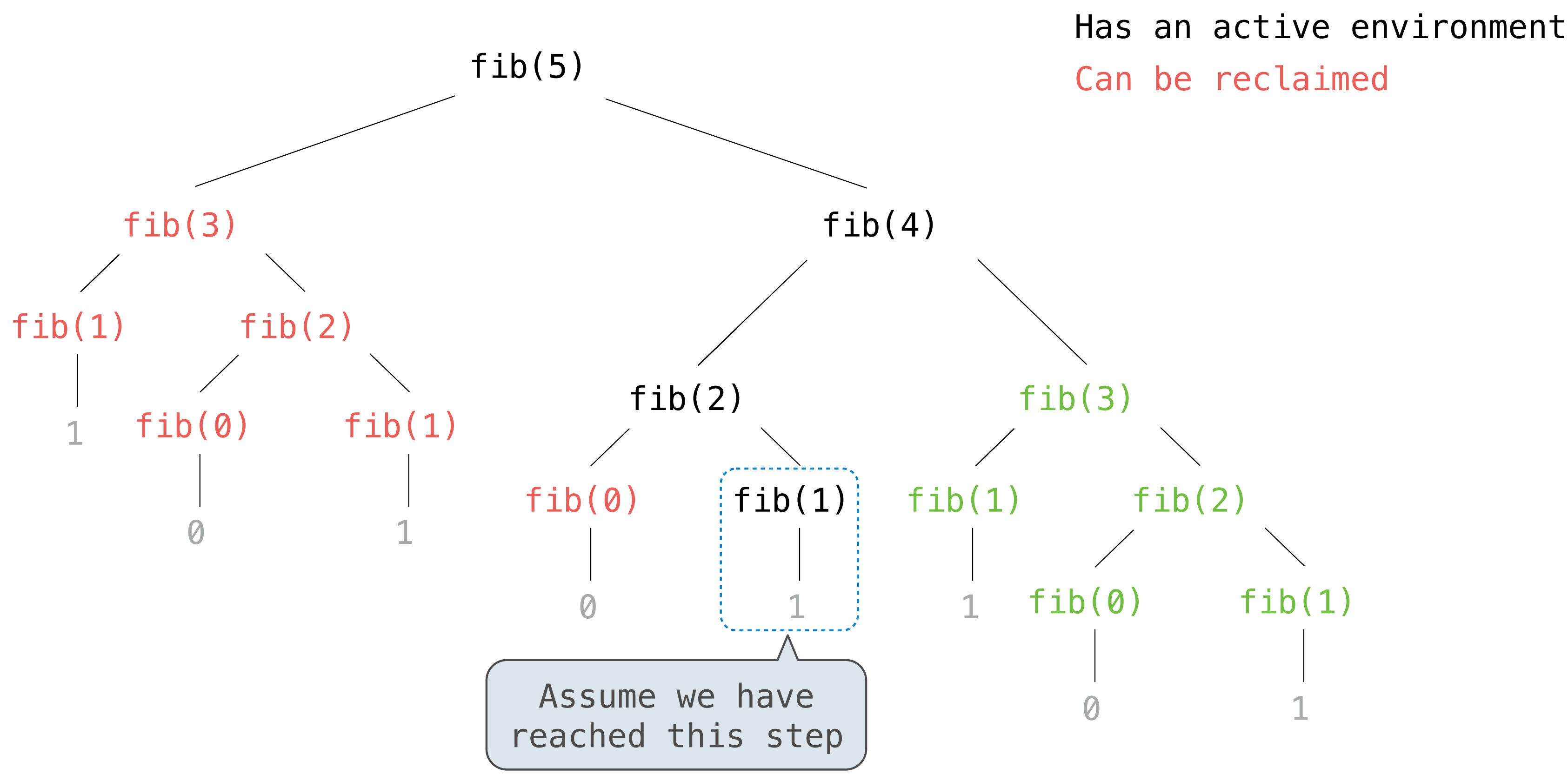
Fibonacci Space Consumption



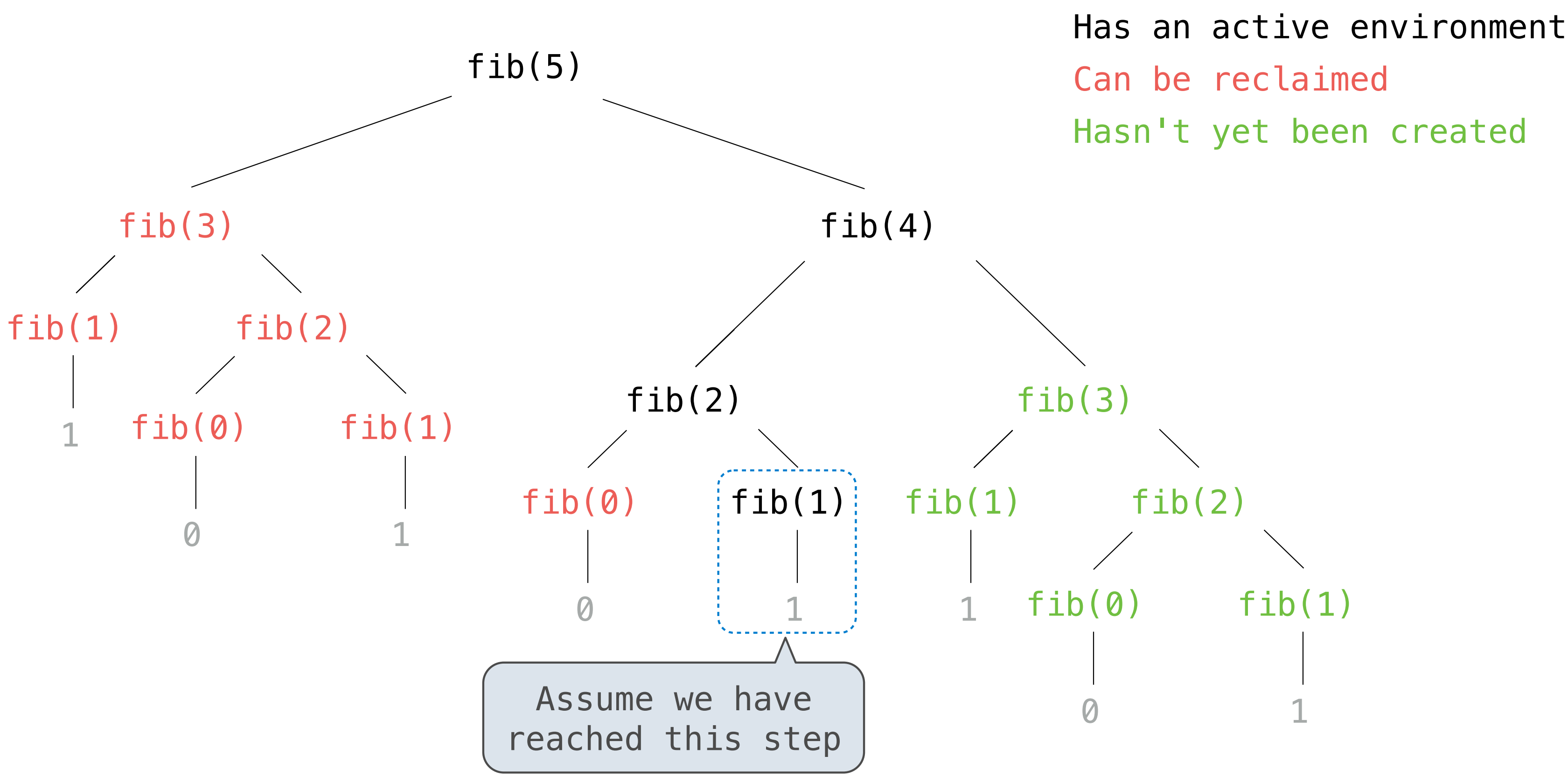
Fibonacci Space Consumption



Fibonacci Space Consumption



Fibonacci Space Consumption



Time

Comparing Implementations

Comparing Implementations

Implementations of the same functional abstraction can require different resources

Comparing Implementations

Implementations of the same functional abstraction can require different resources

Problem: How many factors does a positive integer n have?

Comparing Implementations

Implementations of the same functional abstraction can require different resources

Problem: How many factors does a positive integer n have?

A factor k of n is a positive integer that evenly divides n

Comparing Implementations

Implementations of the same functional abstraction can require different resources

Problem: How many factors does a positive integer n have?

A factor k of n is a positive integer that evenly divides n

```
def factors(n):
```

Comparing Implementations

Implementations of the same functional abstraction can require different resources

Problem: How many factors does a positive integer n have?

A factor k of n is a positive integer that evenly divides n

```
def factors(n):
```

Slow: Test each k from 1 through n

Comparing Implementations

Implementations of the same functional abstraction can require different resources

Problem: How many factors does a positive integer n have?

A factor k of n is a positive integer that evenly divides n

```
def factors(n):
```

Slow: Test each k from 1 through n

Fast: Test each k from 1 to square root n
For every k, n/k is also a factor!

Comparing Implementations

Implementations of the same functional abstraction can require different resources

Problem: How many factors does a positive integer n have?

A factor k of n is a positive integer that evenly divides n

```
def factors(n):
```

Slow: Test each k from 1 through n

Fast: Test each k from 1 to square root n
For every k, n/k is also a factor!

Question: How many time does each implementation use division? (Demo)

Comparing Implementations

Implementations of the same functional abstraction can require different resources

Problem: How many factors does a positive integer n have?

A factor k of n is a positive integer that evenly divides n

```
def factors(n):
```

Time (number of divisions)

Slow: Test each k from 1 through n

Fast: Test each k from 1 to square root n
For every k, n/k is also a factor!

Question: How many time does each implementation use division? (Demo)

Comparing Implementations

Implementations of the same functional abstraction can require different resources

Problem: How many factors does a positive integer n have?

A factor k of n is a positive integer that evenly divides n

```
def factors(n):
```

	Time (number of divisions)
Slow: Test each k from 1 through n	n
Fast: Test each k from 1 to square root n For every k, n/k is also a factor!	

Question: How many time does each implementation use division? (Demo)

Comparing Implementations

Implementations of the same functional abstraction can require different resources

Problem: How many factors does a positive integer n have?

A factor k of n is a positive integer that evenly divides n

```
def factors(n):
```

	Time (number of divisions)
Slow: Test each k from 1 through n	n
Fast: Test each k from 1 to square root n For every k, n/k is also a factor!	Greatest integer less than $\sqrt{n}$

Question: How many time does each implementation use division? (Demo)

Orders of Growth

Order of Growth

Order of Growth

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

Order of Growth

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

n: size of the problem

Order of Growth

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

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 by the "size" of a problem

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 by the "size" of a problem

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 $\mathbf{k_1}$ and $\mathbf{k_2}$ such that

Order of Growth

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

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 by the "size" of a problem

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 $\mathbf{k_1}$ and $\mathbf{k_2}$ such that

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

for all **n** larger than some minimum **m**

Order of Growth

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

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 $\mathbf{k_1}$ and $\mathbf{k_2}$ such that

$$\boxed{k_1 \cdot f(n) \leq R(n)} \leq k_2 \cdot f(n)$$

for all **n** larger than some minimum **m**

Order of Growth

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

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 $\mathbf{k_1}$ and $\mathbf{k_2}$ such that

$$k_1 \cdot f(n) \leq \boxed{R(n) \leq k_2 \cdot f(n)}$$

for all **n** larger than some minimum **m**

Order of Growth of Counting Factors

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

Problem: How many factors does a positive integer n have?

A factor k of n is a positive integer that evenly divides n

```
def factors(n):
```

Slow: Test each k from 1 through n

Fast: Test each k from 1 to square root n
For every k, n/k is also a factor!

Order of Growth of Counting Factors

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

Problem: How many factors does a positive integer n have?

A factor k of n is a positive integer that evenly divides n

```
def factors(n):
```

Time	Space

Slow: Test each k from 1 through n

Fast: Test each k from 1 to square root n
For every k, n/k is also a factor!

Order of Growth of Counting Factors

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

Problem: How many factors does a positive integer n have?

A factor k of n is a positive integer that evenly divides n

```
def factors(n):
```

	Time	Space
Slow: Test each k from 1 through n	$\Theta(n)$	$\Theta(1)$
Fast: Test each k from 1 to square root n For every k, n/k is also a factor!		

Order of Growth of Counting Factors

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

Problem: How many factors does a positive integer n have?

A factor k of n is a positive integer that evenly divides n

```
def factors(n):
```

	Time	Space
Slow: Test each k from 1 through n	$\Theta(n)$	$\Theta(1)$
Fast: Test each k from 1 to square root n For every k, n/k is also a factor!	$\Theta(\sqrt{n})$	$\Theta(1)$

Order of Growth of Counting Factors

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

Problem: How many factors does a positive integer n have?

A factor k of n is a positive integer that evenly divides n

```
def factors(n):
```

	Time	Space
Slow: Test each k from 1 through n	$\Theta(n)$	$\Theta(1)$
Fast: Test each k from 1 to square root n For every k, n/k is also a factor!	$\Theta(\sqrt{n})$	$\Theta(1)$

Assumption: integers occupy a fixed amount of space

Order of Growth of Counting Factors

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

Problem: How many factors does a positive integer n have?

A factor k of n is a positive integer that evenly divides n

```
def factors(n):
```

	Time	Space
Slow: Test each k from 1 through n	$\Theta(n)$	$\Theta(1)$
Fast: Test each k from 1 to square root n For every k, n/k is also a factor!	$\Theta(\sqrt{n})$	$\Theta(1)$

Assumption: integers occupy a fixed amount of space

(Demo)

Exponentiation

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

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

(Demo)

Exponentiation

Goal: one more multiplication lets us double the problem size

Time	Space

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


def square(x):
    return x*x

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

Exponentiation

Goal: one more multiplication lets us double the problem size

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

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

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

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

Exponentiation

Goal: one more multiplication lets us double the problem size

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


def square(x):
    return x*x

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

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

Comparing Orders of Growth

Properties of Orders of Growth

Properties of Orders of Growth

Constants: Constant terms do not affect the order of growth of a process

Properties of Orders of Growth

Constants: Constant terms do not affect the order of growth of a process

$$\Theta(n)$$

Properties of Orders of Growth

Constants: Constant terms do not affect the order of growth of a process

$$\Theta(n) \qquad \Theta(500 \cdot n)$$

Properties of Orders of Growth

Constants: Constant terms do not affect the order of growth of a process

$\Theta(n)$ $\Theta(500 \cdot n)$ $\Theta(\frac{1}{500} \cdot n)$

Properties of Orders of Growth

Constants: Constant terms do not affect the order of growth of a process

$$\Theta(n) \qquad \Theta(500 \cdot n) \qquad \Theta(\frac{1}{500} \cdot n)$$

Logarithms: The base of a logarithm does not affect the order of growth of a process

Properties of Orders of Growth

Constants: Constant terms do not affect the order of growth of a process

$$\Theta(n) \qquad \Theta(500 \cdot n) \qquad \Theta(\frac{1}{500} \cdot n)$$

Logarithms: The base of a logarithm does not affect the order of growth of a process

$$\Theta(\log_2 n)$$

Properties of Orders of Growth

Constants: Constant terms do not affect the order of growth of a process

$$\Theta(n) \qquad \Theta(500 \cdot n) \qquad \Theta(\frac{1}{500} \cdot n)$$

Logarithms: The base of a logarithm does not affect the order of growth of a process

$$\Theta(\log_2 n) \qquad \Theta(\log_{10} n)$$

Properties of Orders of Growth

Constants: Constant terms do not affect the order of growth of a process

$$\Theta(n) \qquad \Theta(500 \cdot n) \qquad \Theta(\frac{1}{500} \cdot n)$$

Logarithms: The base of a logarithm does not affect the order of growth of a process

$$\Theta(\log_2 n) \qquad \Theta(\log_{10} n) \qquad \Theta(\ln n)$$

Properties of Orders of Growth

Constants: Constant terms do not affect the order of growth of a process

$$\Theta(n) \qquad \Theta(500 \cdot n) \qquad \Theta(\frac{1}{500} \cdot n)$$

Logarithms: The base of a logarithm does not affect the order of growth of a process

$$\Theta(\log_2 n) \qquad \Theta(\log_{10} n) \qquad \Theta(\ln n)$$

Nesting: When an inner process is repeated for each step in an outer process, multiply the steps in the outer and inner processes to find the total number of steps

Properties of Orders of Growth

Constants: Constant terms do not affect the order of growth of a process

$$\Theta(n) \qquad \Theta(500 \cdot n) \qquad \Theta(\frac{1}{500} \cdot n)$$

Logarithms: The base of a logarithm does not affect the order of growth of a process

$$\Theta(\log_2 n) \qquad \Theta(\log_{10} n) \qquad \Theta(\ln n)$$

Nesting: When an inner process is repeated for each step in an outer process, multiply the steps in the outer and inner processes to find the total number of steps

```
def overlap(a, b):
    count = 0
    for item in a:
        if item in b:
            count += 1
    return count
```

Properties of Orders of Growth

Constants: Constant terms do not affect the order of growth of a process

$$\Theta(n) \qquad \Theta(500 \cdot n) \qquad \Theta(\frac{1}{500} \cdot n)$$

Logarithms: The base of a logarithm does not affect the order of growth of a process

$$\Theta(\log_2 n) \qquad \Theta(\log_{10} n) \qquad \Theta(\ln n)$$

Nesting: When an inner process is repeated for each step in an outer process, multiply the steps in the outer and inner processes to find the total number of steps

```
def overlap(a, b):
    count = 0
    for item in a:
        if item in b:
            count += 1
    return count
```

Outer: length of a

Properties of Orders of Growth

Constants: Constant terms do not affect the order of growth of a process

$$\Theta(n) \qquad \Theta(500 \cdot n) \qquad \Theta(\frac{1}{500} \cdot n)$$

Logarithms: The base of a logarithm does not affect the order of growth of a process

$$\Theta(\log_2 n) \qquad \Theta(\log_{10} n) \qquad \Theta(\ln n)$$

Nesting: When an inner process is repeated for each step in an outer process, multiply the steps in the outer and inner processes to find the total number of steps

```
def overlap(a, b):
    count = 0
    for item in a:          Outer: length of a
        if item in b:       Inner: length of b
            count += 1
    return count
```

Properties of Orders of Growth

Constants: Constant terms do not affect the order of growth of a process

$$\Theta(n) \qquad \Theta(500 \cdot n) \qquad \Theta(\frac{1}{500} \cdot n)$$

Logarithms: The base of a logarithm does not affect the order of growth of a process

$$\Theta(\log_2 n) \qquad \Theta(\log_{10} n) \qquad \Theta(\ln n)$$

Nesting: When an inner process is repeated for each step in an outer process, multiply the steps in the outer and inner processes to find the total number of steps

```
def overlap(a, b):
    count = 0
    for item in a:
        if item in b:
            count += 1
    return count
```

Outer: length of a

Inner: length of b

If a and b are both length **n**, then overlap takes $\Theta(n^2)$ steps

Properties of Orders of Growth

Constants: Constant terms do not affect the order of growth of a process

$$\Theta(n) \qquad \Theta(500 \cdot n) \qquad \Theta(\frac{1}{500} \cdot n)$$

Logarithms: The base of a logarithm does not affect the order of growth of a process

$$\Theta(\log_2 n) \qquad \Theta(\log_{10} n) \qquad \Theta(\ln n)$$

Nesting: When an inner process is repeated for each step in an outer process, multiply the steps in the outer and inner processes to find the total number of steps

```
def overlap(a, b):
    count = 0
    for item in a:
        if item in b:
            count += 1
    return count
```

Outer: length of a

Inner: length of b

If a and b are both length **n**, then overlap takes $\Theta(n^2)$ steps

Lower-order terms: The fastest-growing part of the computation dominates the total

Properties of Orders of Growth

Constants: Constant terms do not affect the order of growth of a process

$$\Theta(n) \qquad \Theta(500 \cdot n) \qquad \Theta(\frac{1}{500} \cdot n)$$

Logarithms: The base of a logarithm does not affect the order of growth of a process

$$\Theta(\log_2 n) \qquad \Theta(\log_{10} n) \qquad \Theta(\ln n)$$

Nesting: When an inner process is repeated for each step in an outer process, multiply the steps in the outer and inner processes to find the total number of steps

```
def overlap(a, b):
    count = 0
    for item in a:           Outer: length of a
        if item in b:        Inner: length of b
            count += 1
    return count
```

If a and b are both length **n**, then overlap takes $\Theta(n^2)$ steps

Lower-order terms: The fastest-growing part of the computation dominates the total

$$\Theta(n^2)$$

Properties of Orders of Growth

Constants: Constant terms do not affect the order of growth of a process

$$\Theta(n) \qquad \Theta(500 \cdot n) \qquad \Theta(\frac{1}{500} \cdot n)$$

Logarithms: The base of a logarithm does not affect the order of growth of a process

$$\Theta(\log_2 n) \qquad \Theta(\log_{10} n) \qquad \Theta(\ln n)$$

Nesting: When an inner process is repeated for each step in an outer process, multiply the steps in the outer and inner processes to find the total number of steps

```
def overlap(a, b):
    count = 0
    for item in a:          Outer: length of a
        if item in b:       Inner: length of b
            count += 1
    return count
```

If a and b are both length **n**, then overlap takes $\Theta(n^2)$ steps

Lower-order terms: The fastest-growing part of the computation dominates the total

$$\Theta(n^2) \qquad \Theta(n^2 + n)$$

Properties of Orders of Growth

Constants: Constant terms do not affect the order of growth of a process

$$\Theta(n) \qquad \Theta(500 \cdot n) \qquad \Theta(\frac{1}{500} \cdot n)$$

Logarithms: The base of a logarithm does not affect the order of growth of a process

$$\Theta(\log_2 n) \qquad \Theta(\log_{10} n) \qquad \Theta(\ln n)$$

Nesting: When an inner process is repeated for each step in an outer process, multiply the steps in the outer and inner processes to find the total number of steps

```
def overlap(a, b):
    count = 0
    for item in a:          Outer: length of a
        if item in b:       Inner: length of b
            count += 1
    return count
```

If a and b are both length **n**, then overlap takes $\Theta(n^2)$ steps

Lower-order terms: The fastest-growing part of the computation dominates the total

$$\Theta(n^2) \qquad \Theta(n^2 + n) \qquad \Theta(n^2 + 500 \cdot n + \log_2 n + 1000)$$

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` $\phi = \dfrac{1+\sqrt{5}}{2} \approx 1.61828$

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` $\phi = \frac{1+\sqrt{5}}{2} \approx 1.61828$

`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., `overlap`

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., `overlap`

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., `overlap`

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 $\phi = \frac{1+\sqrt{5}}{2} \approx 1.61828$

Incrementing the problem scales R(n) by a factor

$\Theta(n^2)$ Quadratic growth. E.g., `overlap`

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

$\Theta(n)$ Linear growth. E.g., slow `factors` or `exp`

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., overlap

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

$\Theta(n)$ Linear growth. E.g., slow factors or exp

$\Theta(\sqrt{n})$

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., overlap

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

$\Theta(n)$ Linear growth. E.g., slow factors or exp

$\Theta(\sqrt{n})$ Square root growth. E.g., factors_fast

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., `overlap`

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

$\Theta(n)$ Linear growth. E.g., slow `factors` or `exp`

$\Theta(\sqrt{n})$ Square root growth. E.g., `factors_fast`

$\Theta(\log n)$

Comparing orders of growth (n is the problem size)

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

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

Incrementing the problem scales R(n) by a factor

$\Theta(n^2)$ Quadratic growth. E.g., `overlap`

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

$\Theta(n)$ Linear growth. E.g., slow `factors` or `exp`

$\Theta(\sqrt{n})$ Square root growth. E.g., `factors_fast`

$\Theta(\log n)$ Logarithmic growth. E.g., `exp_fast`

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., overlap

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

$\Theta(n)$ Linear growth. E.g., slow factors or exp

$\Theta(\sqrt{n})$ Square root growth. E.g., factors_fast

$\Theta(\log n)$ Logarithmic growth. E.g., exp_fast

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., overlap

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

$\Theta(n)$ Linear growth. E.g., slow factors or exp

$\Theta(\sqrt{n})$ Square root growth. E.g., factors_fast

$\Theta(\log n)$ Logarithmic growth. E.g., exp_fast

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., overlap

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

$\Theta(n)$ Linear growth. E.g., slow factors or exp

$\Theta(\sqrt{n})$ Square root growth. E.g., factors_fast

$\Theta(\log n)$ Logarithmic growth. E.g., exp_fast

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., overlap Incrementing n increases R(n) by the problem size n
$\Theta(n)$	Linear growth. E.g., slow factors or exp
$\Theta(\sqrt{n})$	Square root growth. E.g., factors_fast
$\Theta(\log n)$	Logarithmic growth. E.g., exp_fast 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., overlap Incrementing n increases R(n) by the problem size n
$\Theta(n)$	Linear growth. E.g., slow factors or exp
$\Theta(\sqrt{n})$	Square root growth. E.g., factors_fast
$\Theta(\log n)$	Logarithmic growth. E.g., exp_fast Doubling the problem only increments R(n).
$\Theta(1)$	Constant. The problem size doesn't matter