Efficiency

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)

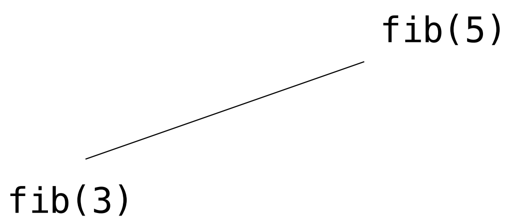
```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-2) + fib(n-1)
```

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

Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:

```
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)
```

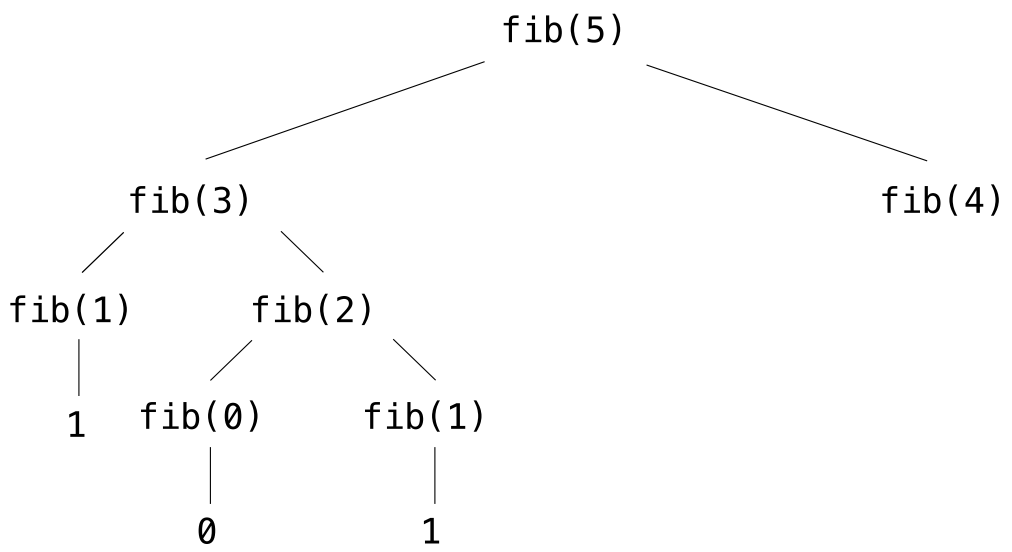
fib(5)

fib(3)   fib(4)

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

Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:



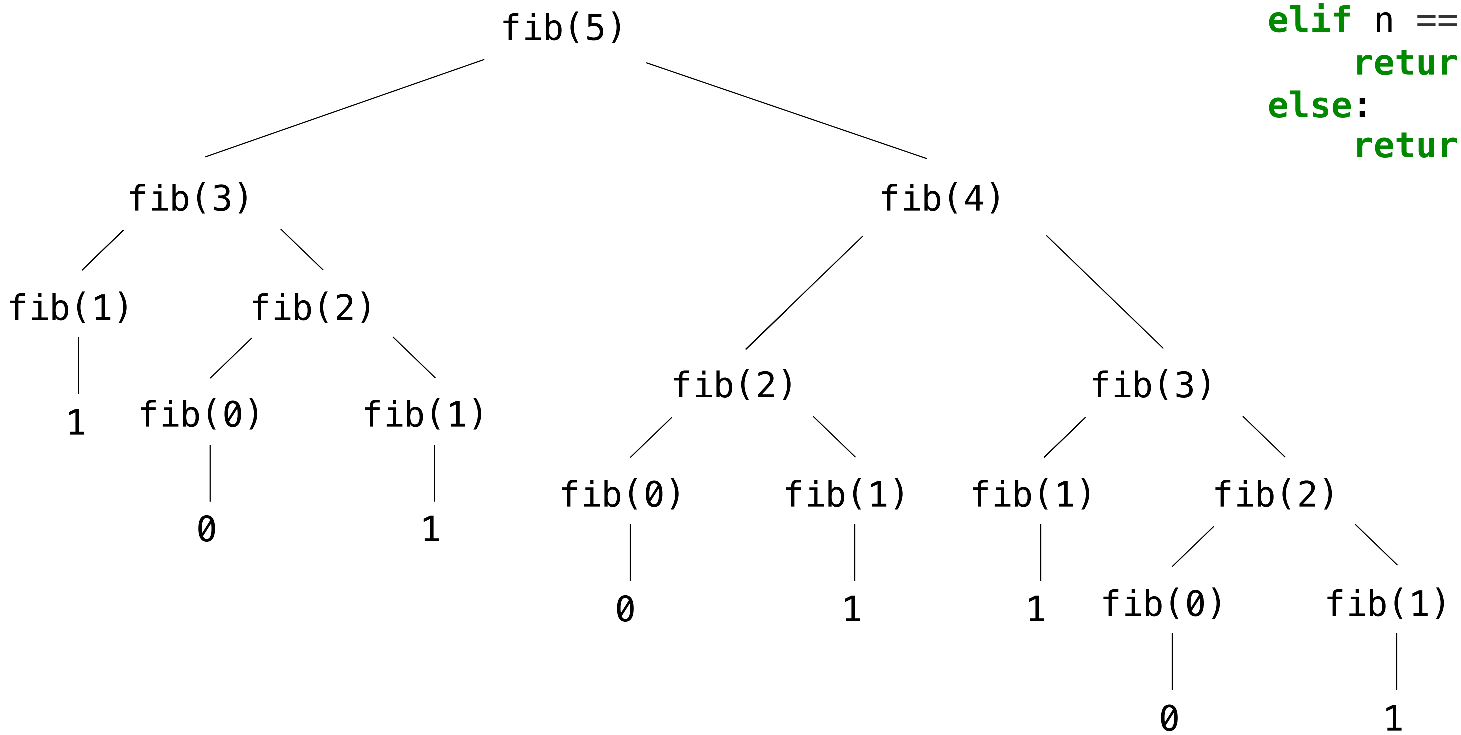

```python
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-2) + fib(n-1)
```

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

Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:

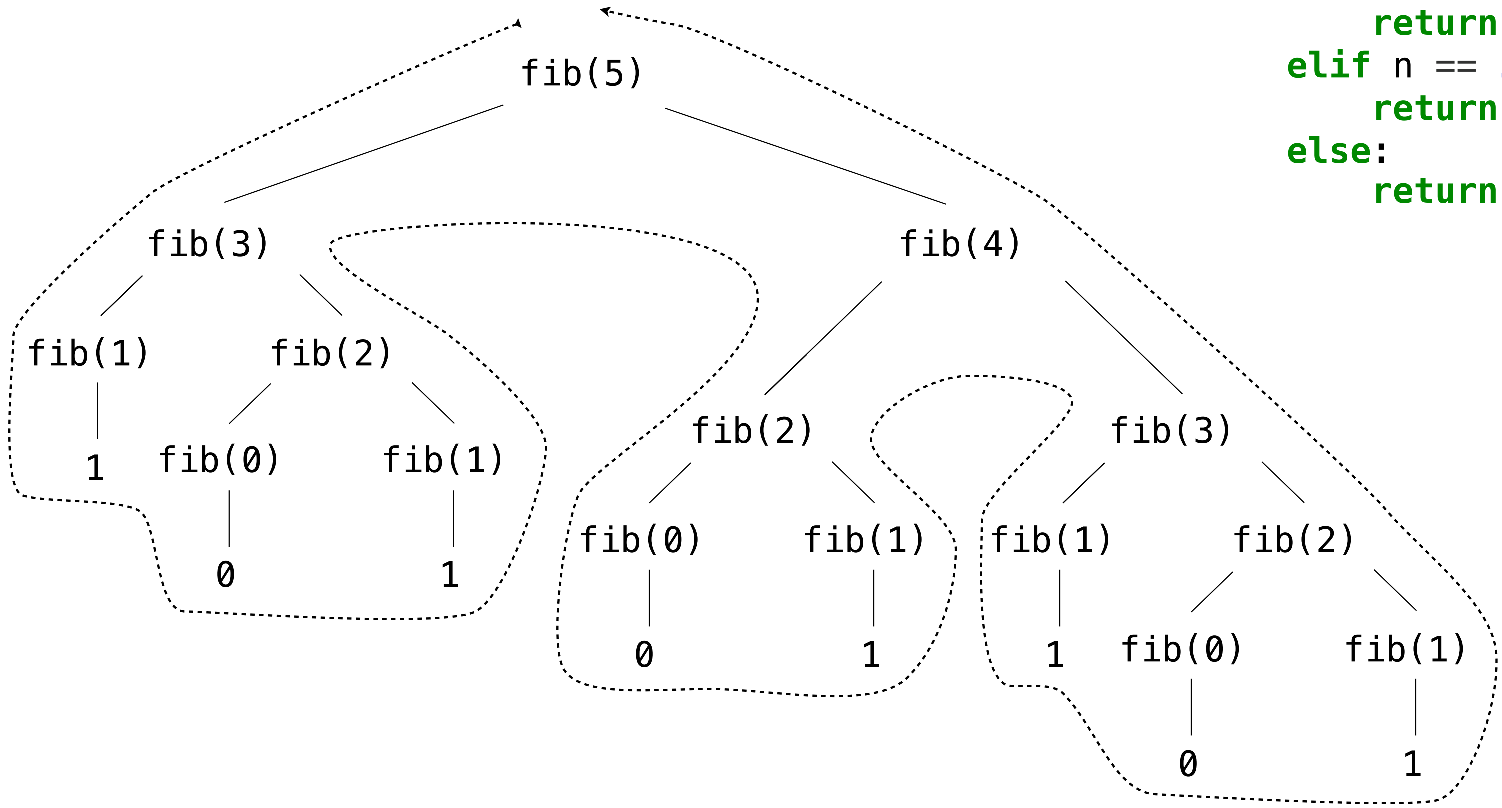
```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-2) + fib(n-1)
```



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

Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:



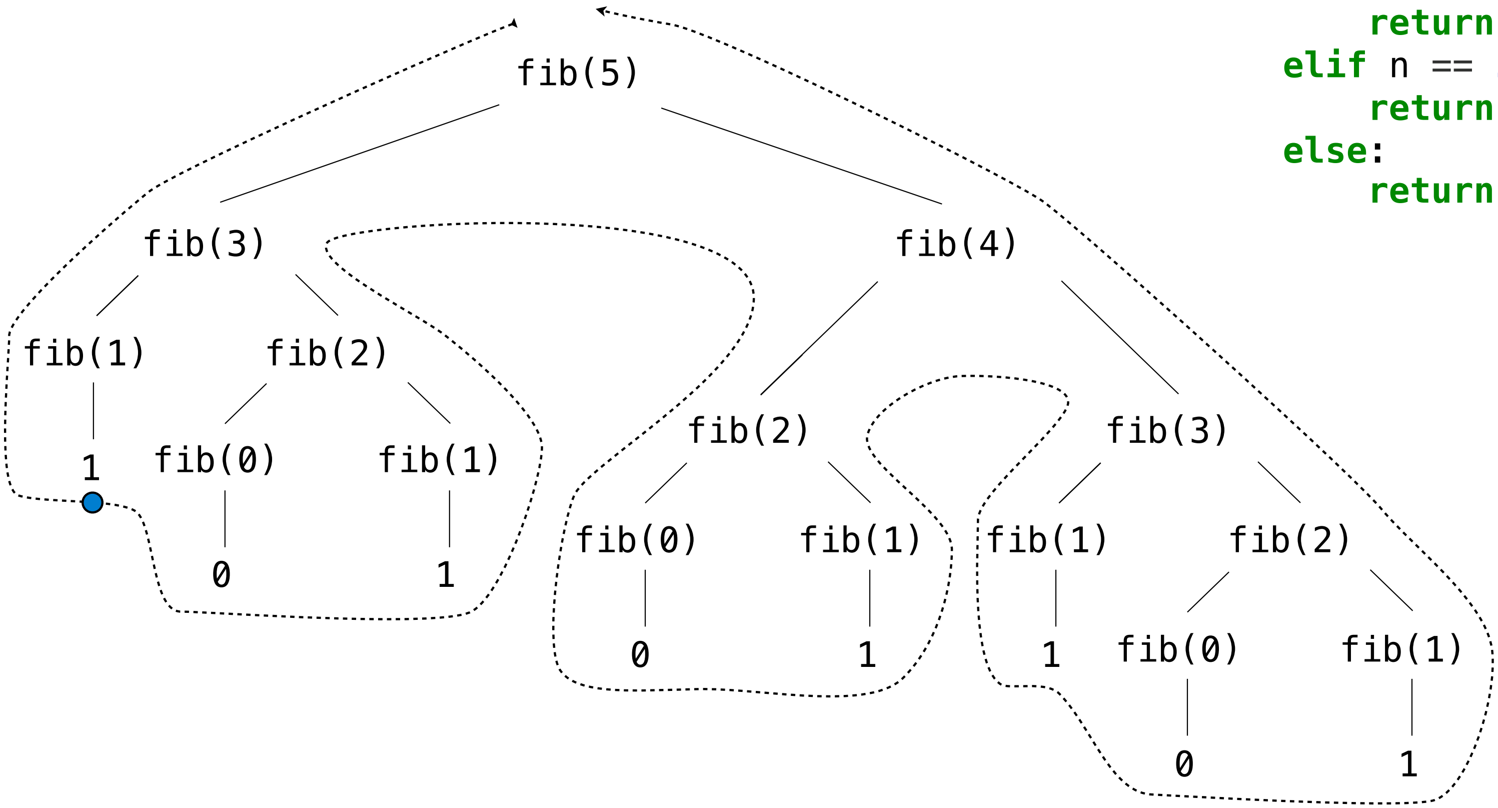

```python
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-2) + fib(n-1)
```

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)
```

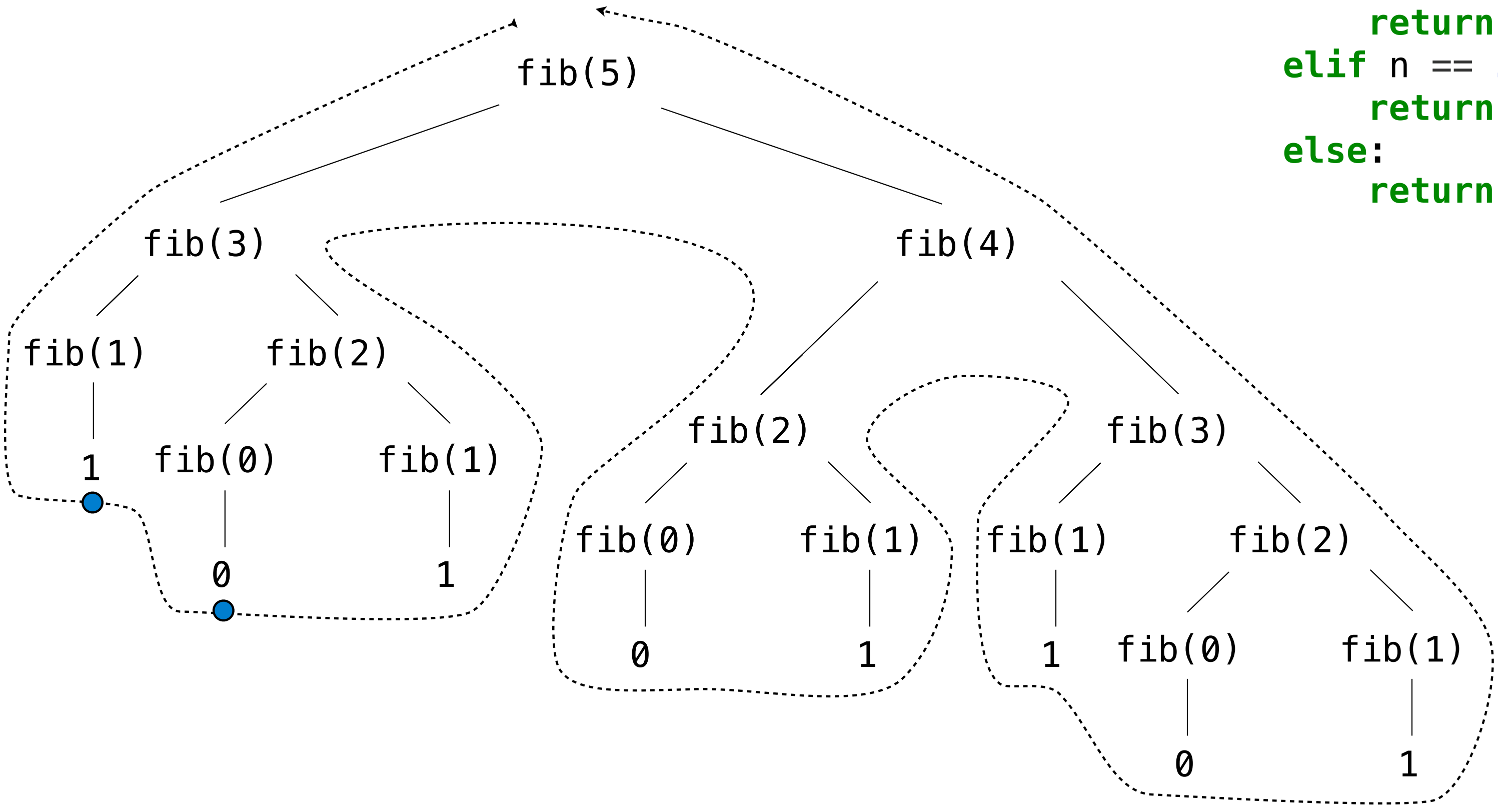


http://en.wikipedia.org/wiki/File:Fibonacci.jpg

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)
```

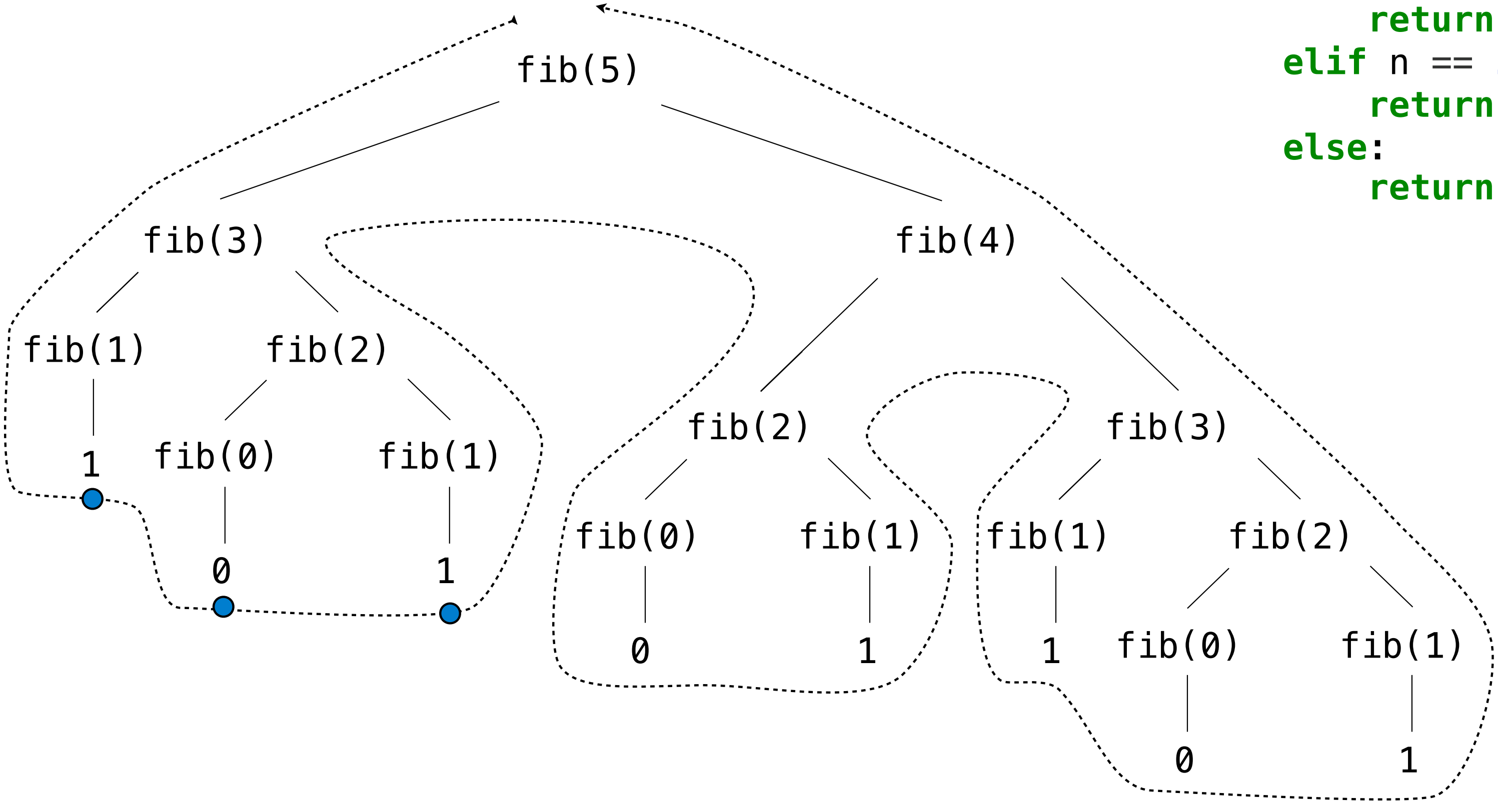


http://en.wikipedia.org/wiki/File:Fibonacci.jpg

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)
```

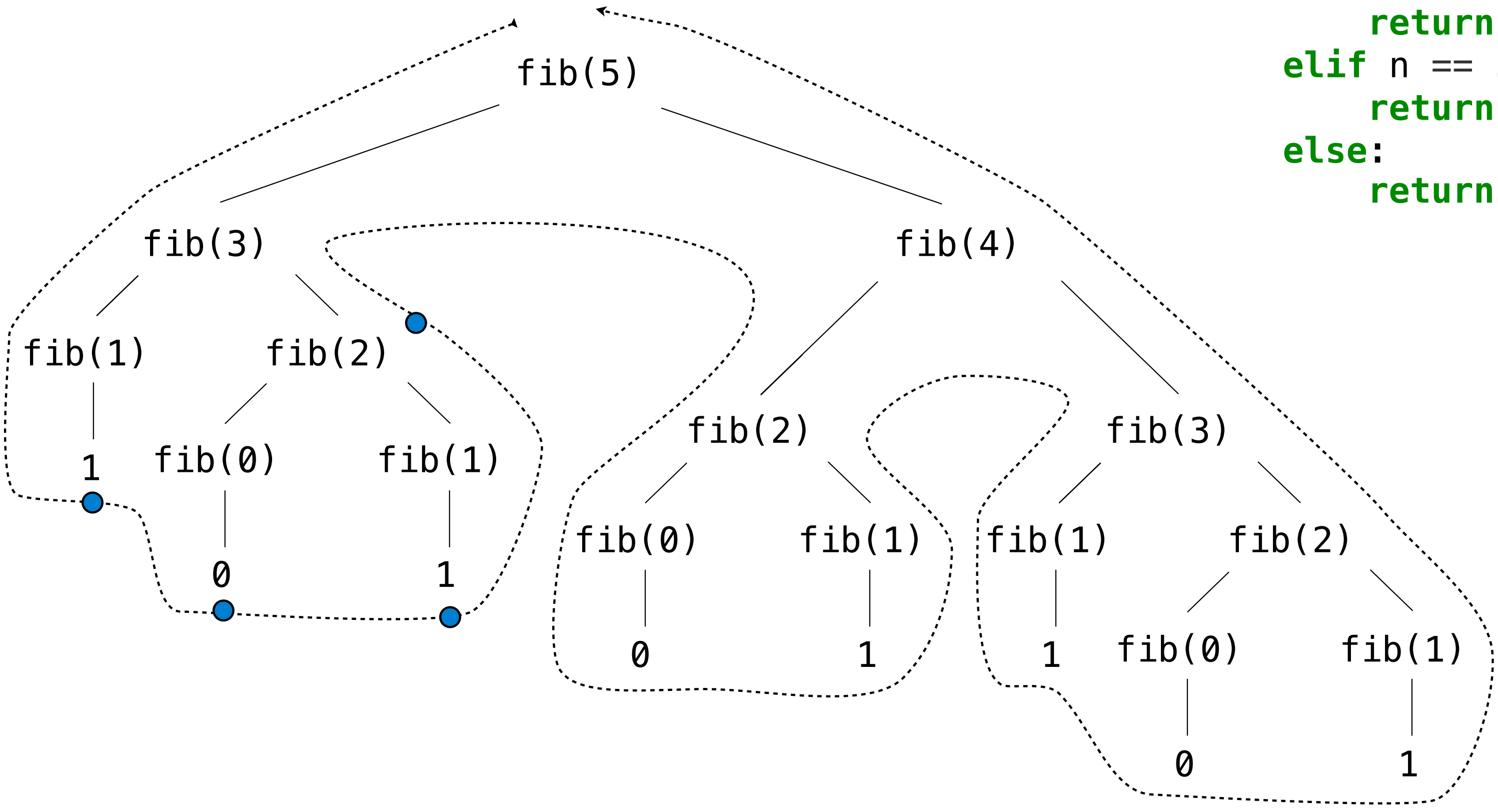


http://en.wikipedia.org/wiki/File:Fibonacci.jpg

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)
```

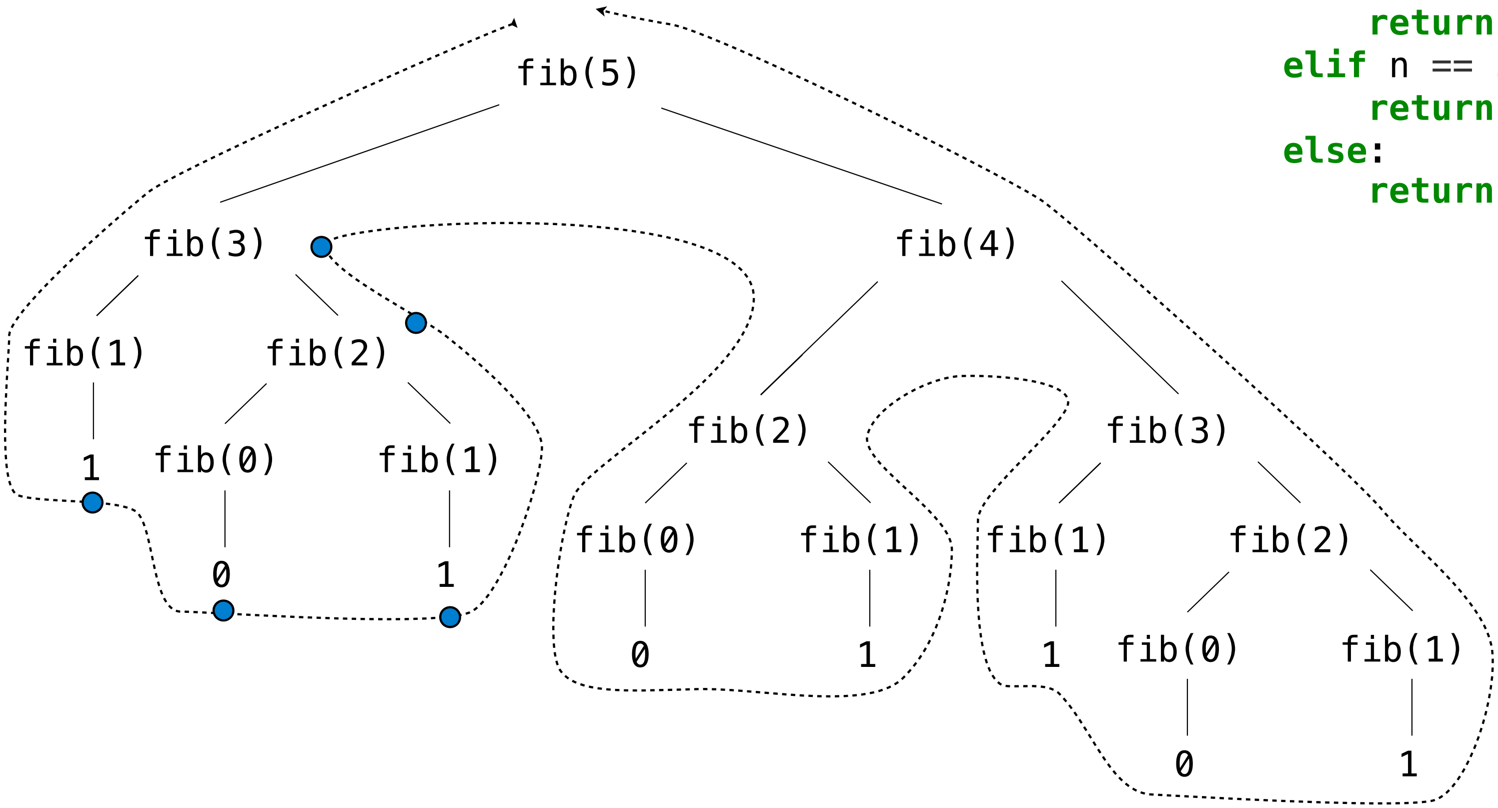


http://en.wikipedia.org/wiki/File:Fibonacci.jpg

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)
```

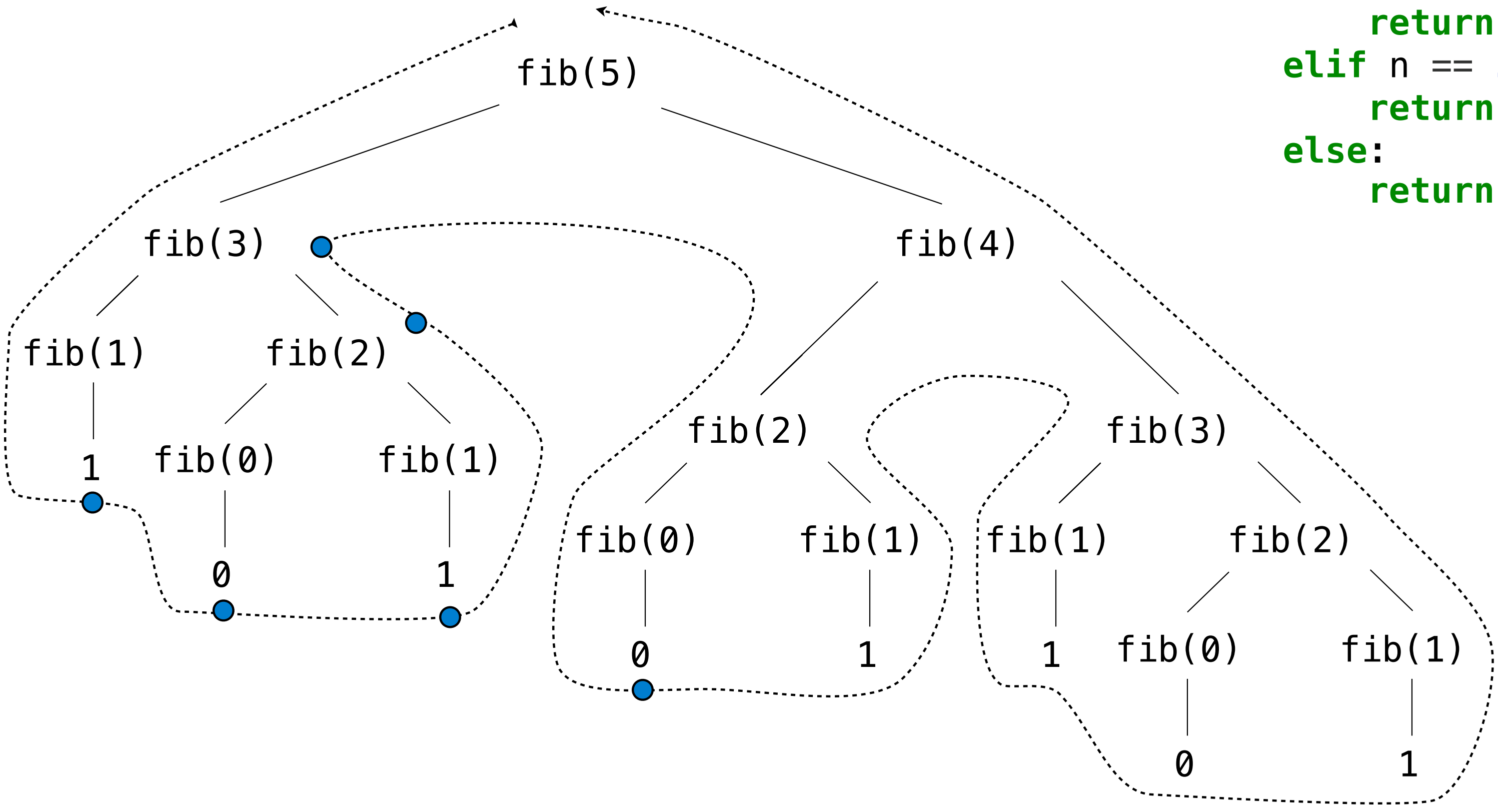


http://en.wikipedia.org/wiki/File:Fibonacci.jpg

Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:

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

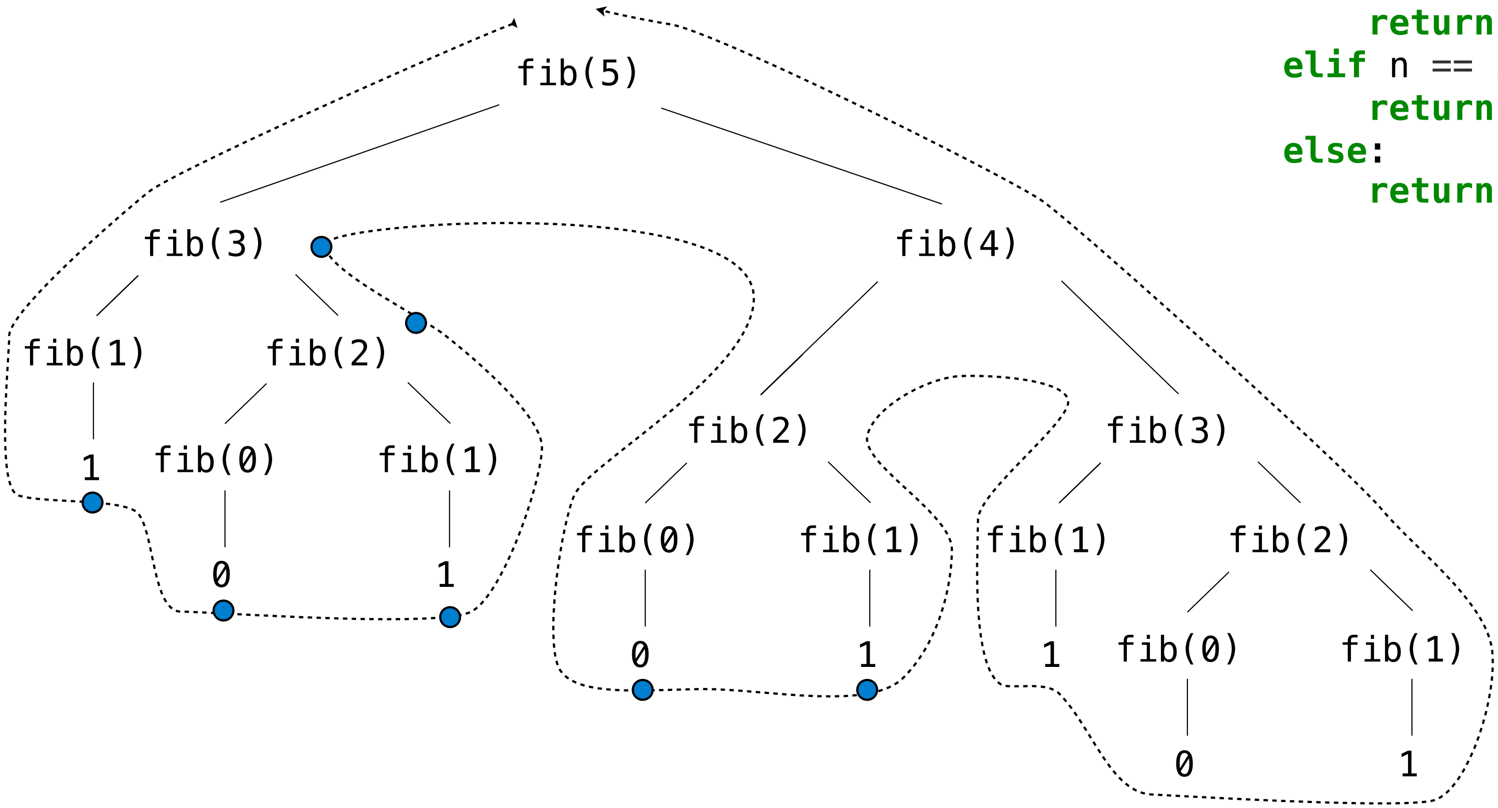


http://en.wikipedia.org/wiki/File:Fibonacci.jpg

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)
```

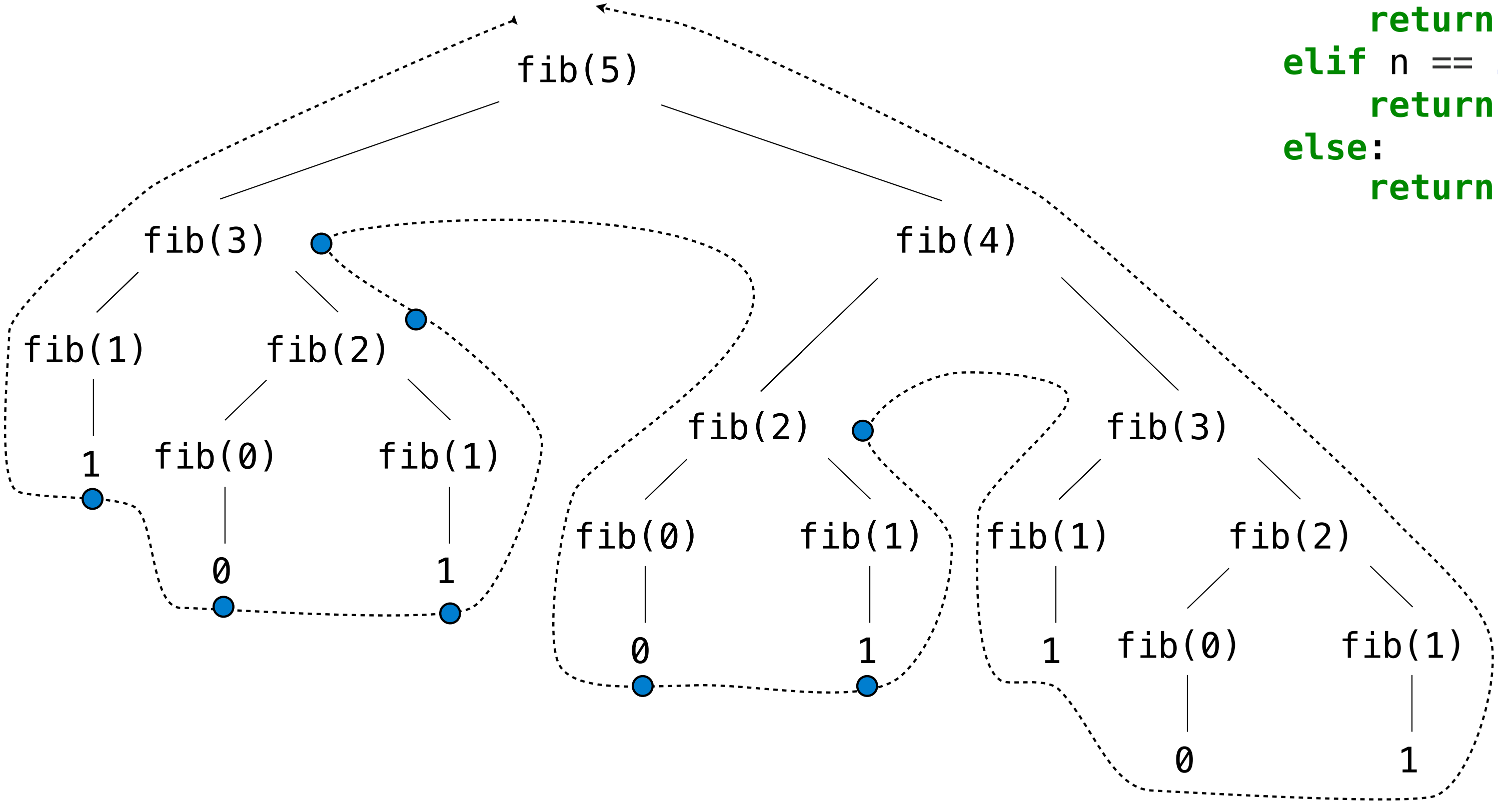


http://en.wikipedia.org/wiki/File:Fibonacci.jpg

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)
```

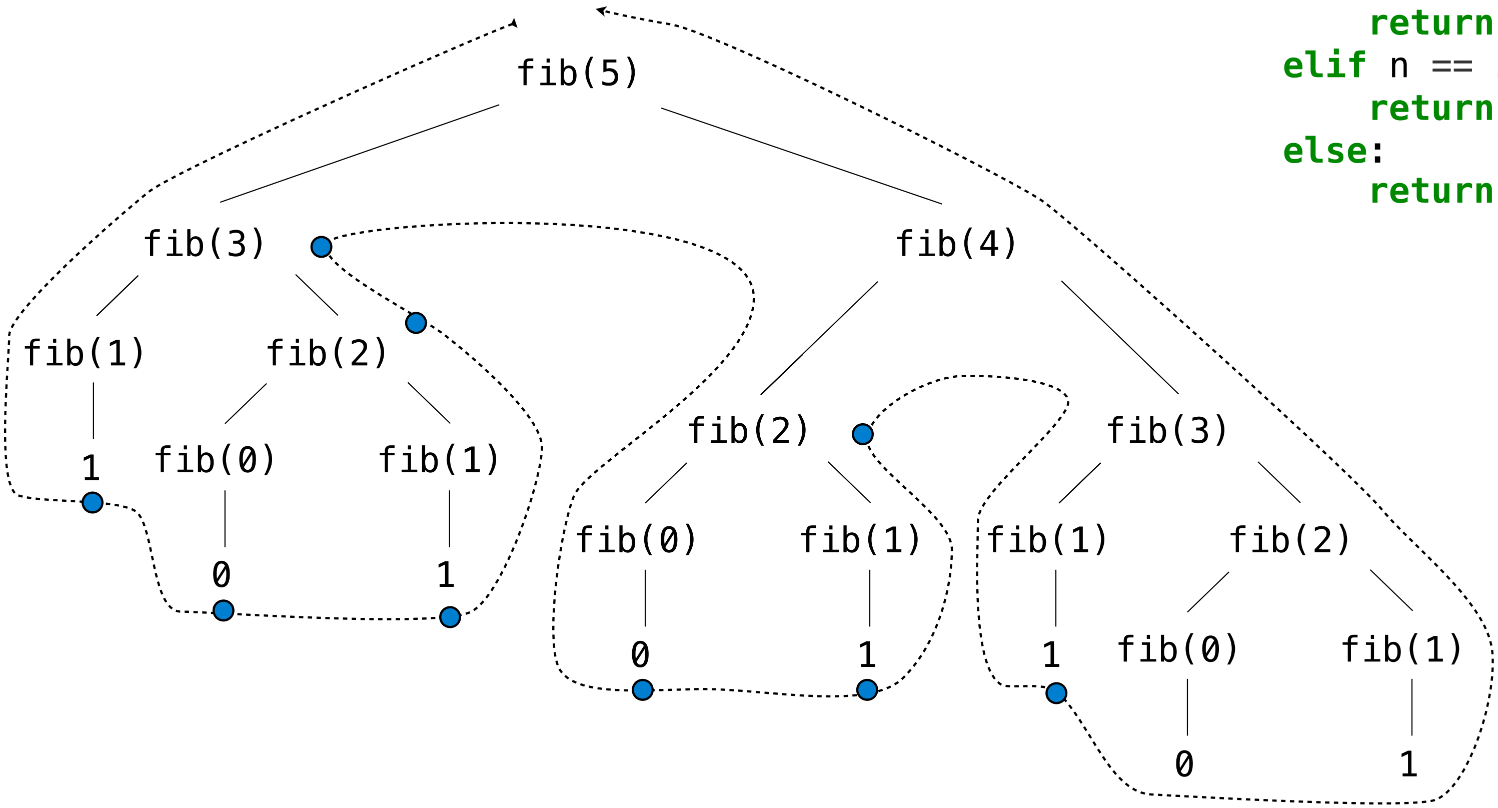


http://en.wikipedia.org/wiki/File:Fibonacci.jpg

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)
```

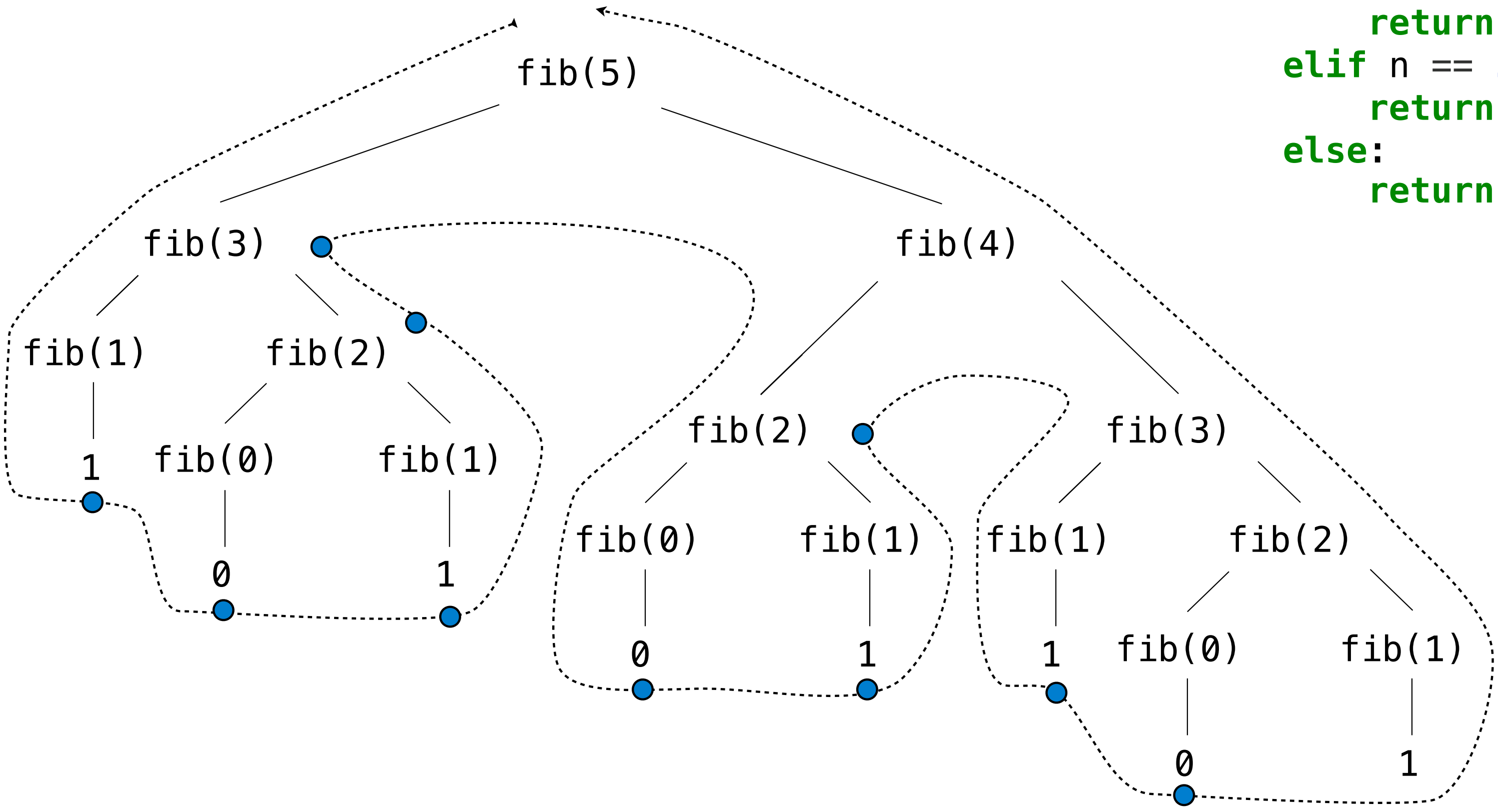


http://en.wikipedia.org/wiki/File:Fibonacci.jpg

Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:

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

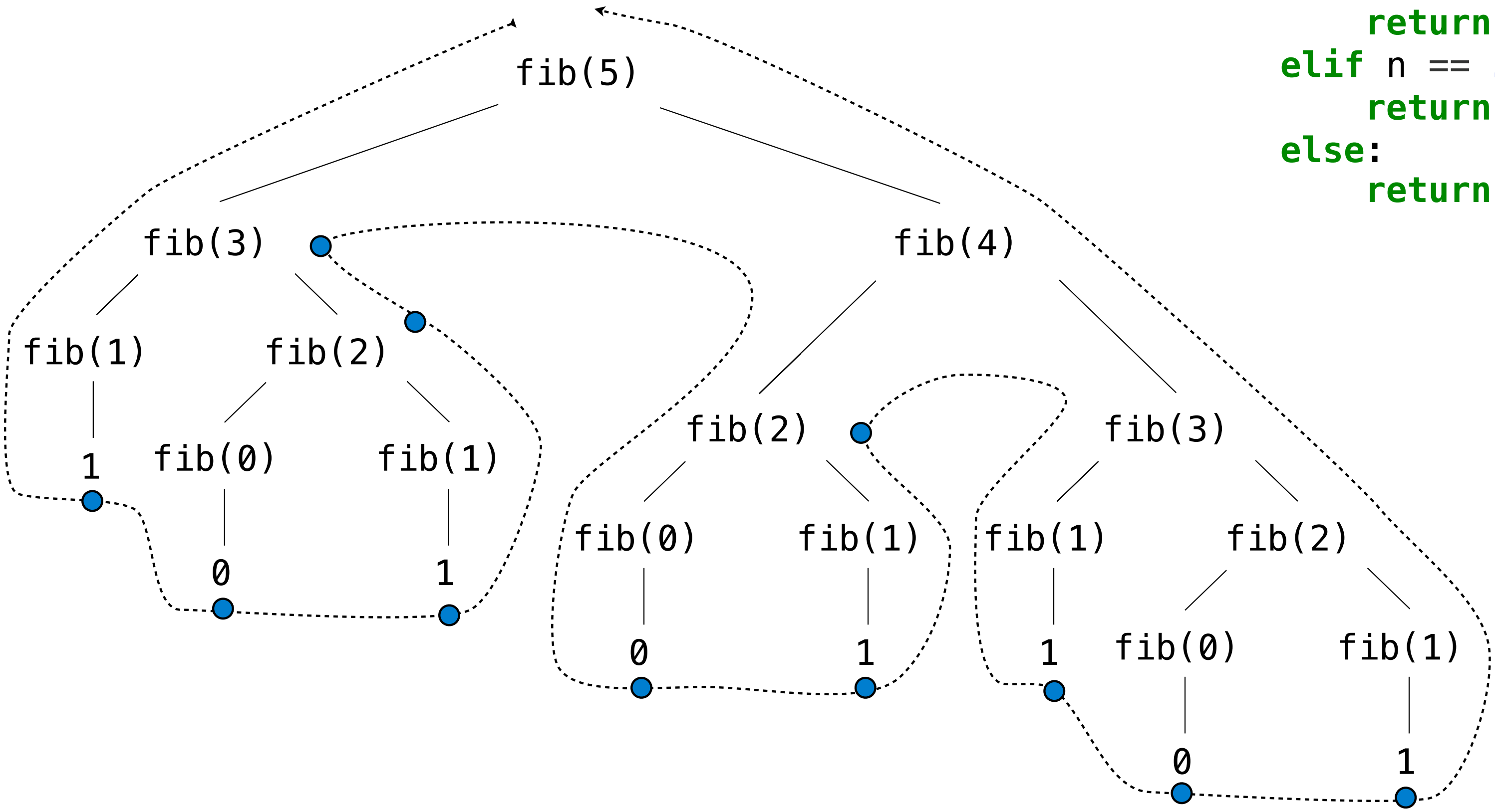


http://en.wikipedia.org/wiki/File:Fibonacci.jpg

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)
```

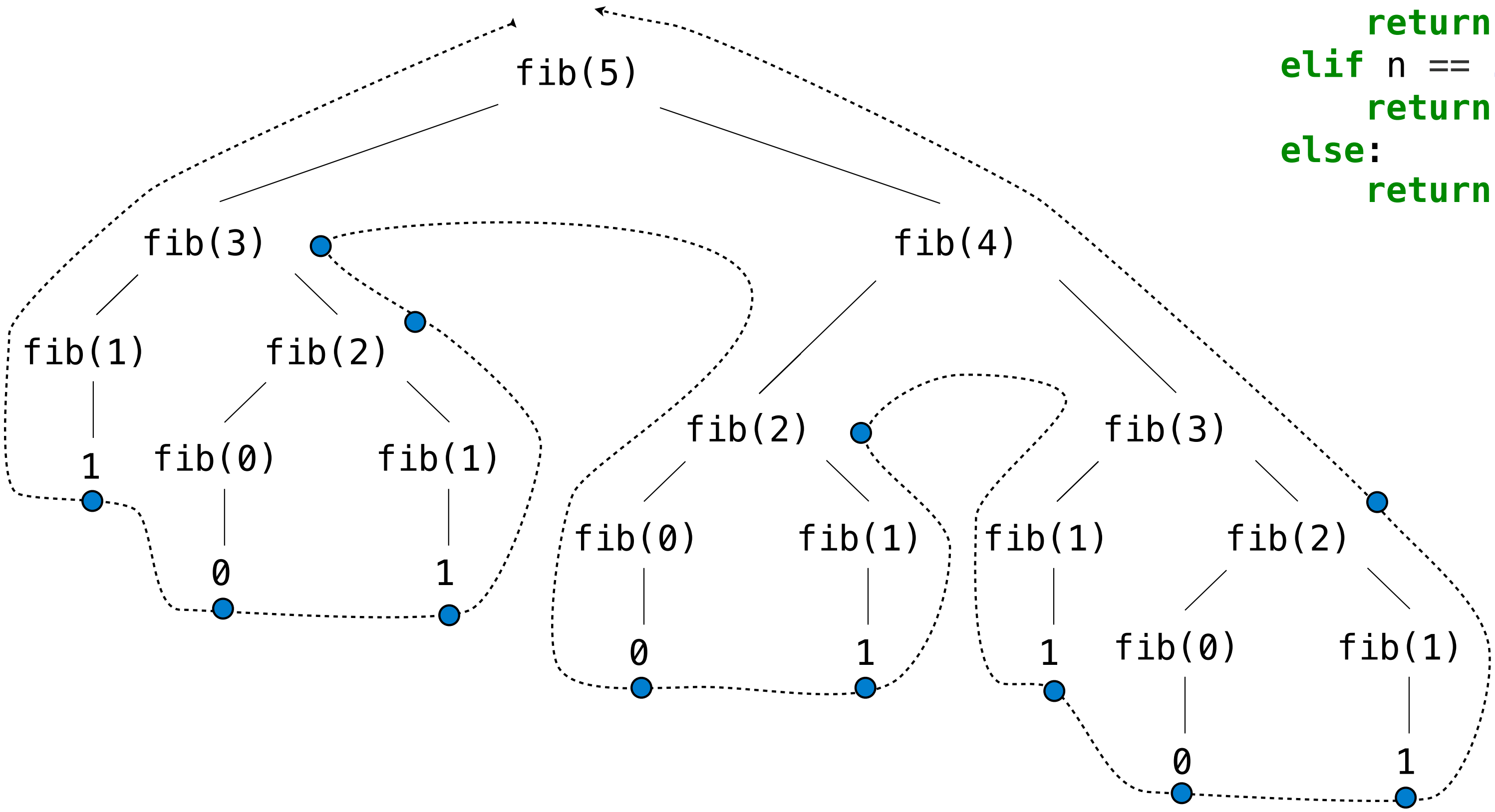


http://en.wikipedia.org/wiki/File:Fibonacci.jpg

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)
```

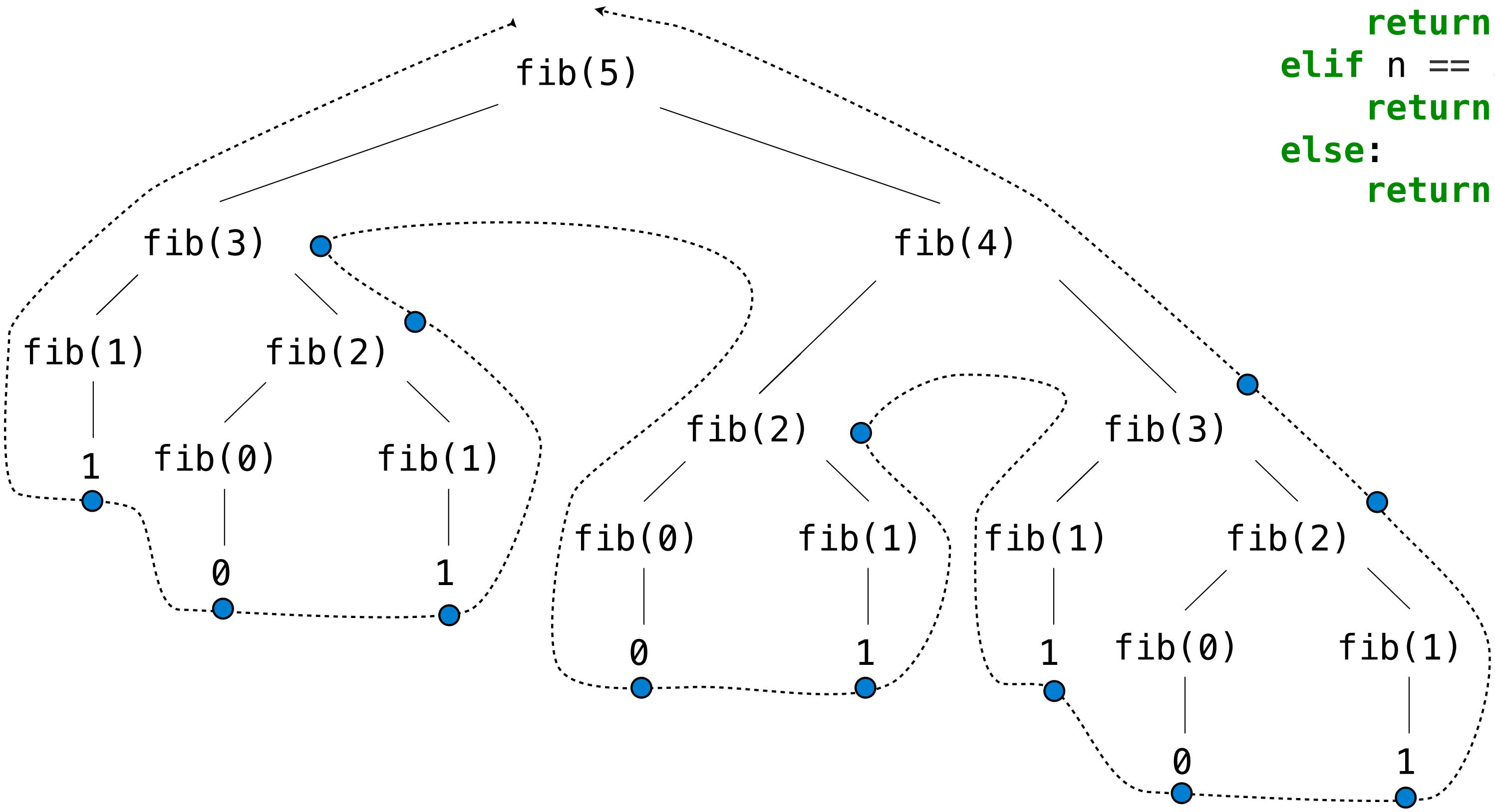


http://en.wikipedia.org/wiki/File:Fibonacci.jpg

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)
```

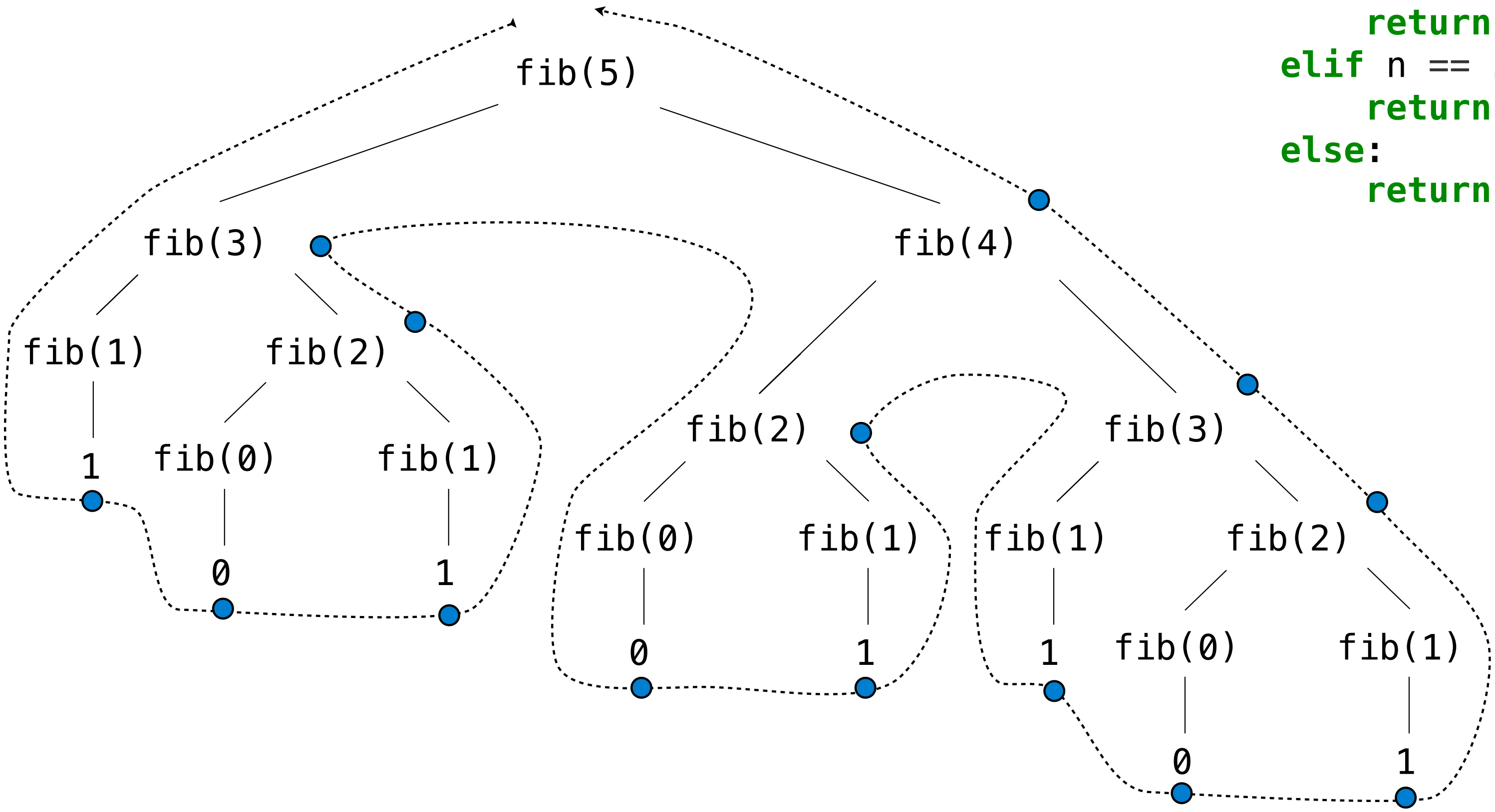


http://en.wikipedia.org/wiki/File:Fibonacci.jpg

Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:

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

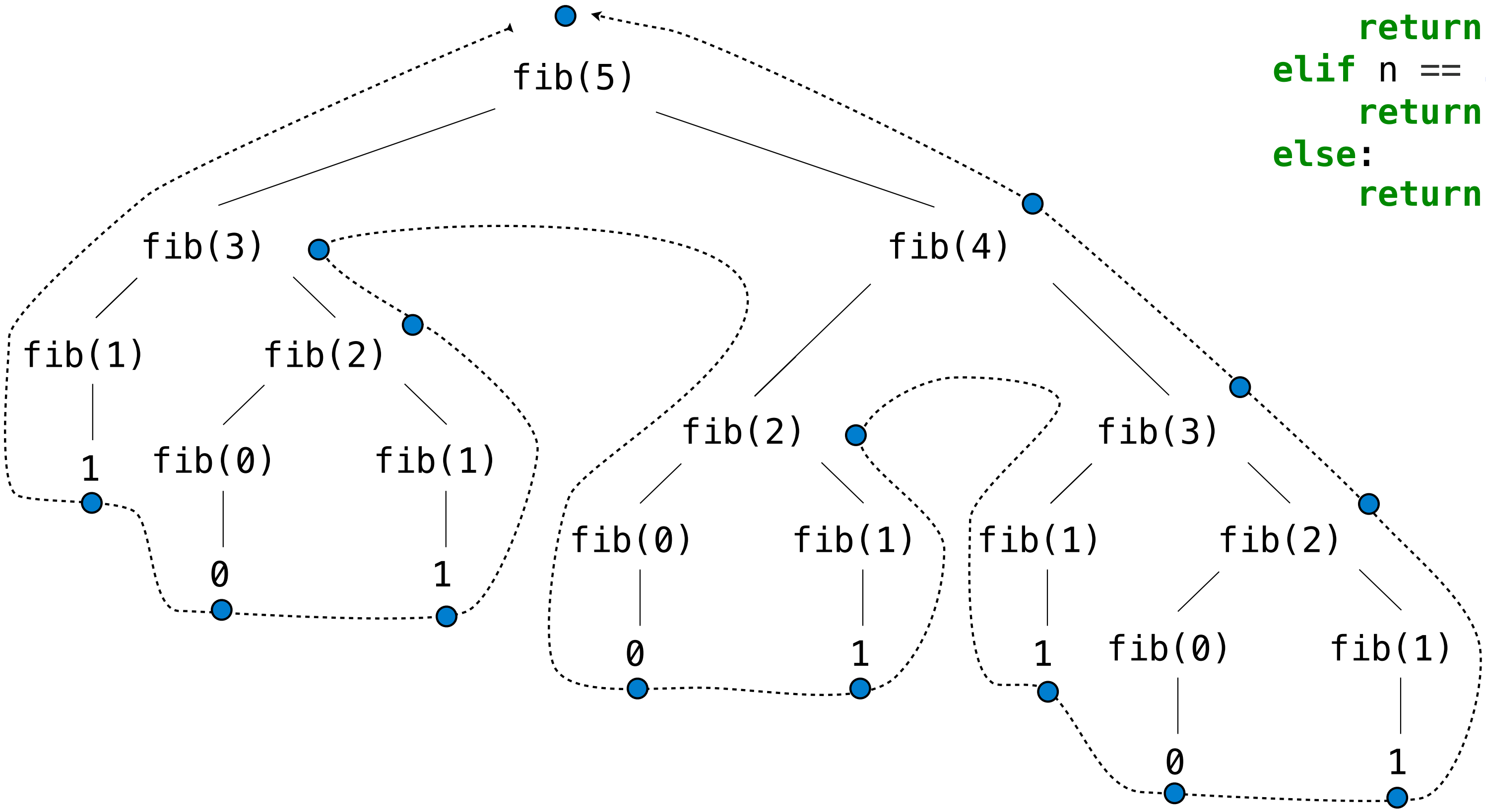


http://en.wikipedia.org/wiki/File:Fibonacci.jpg

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)
```

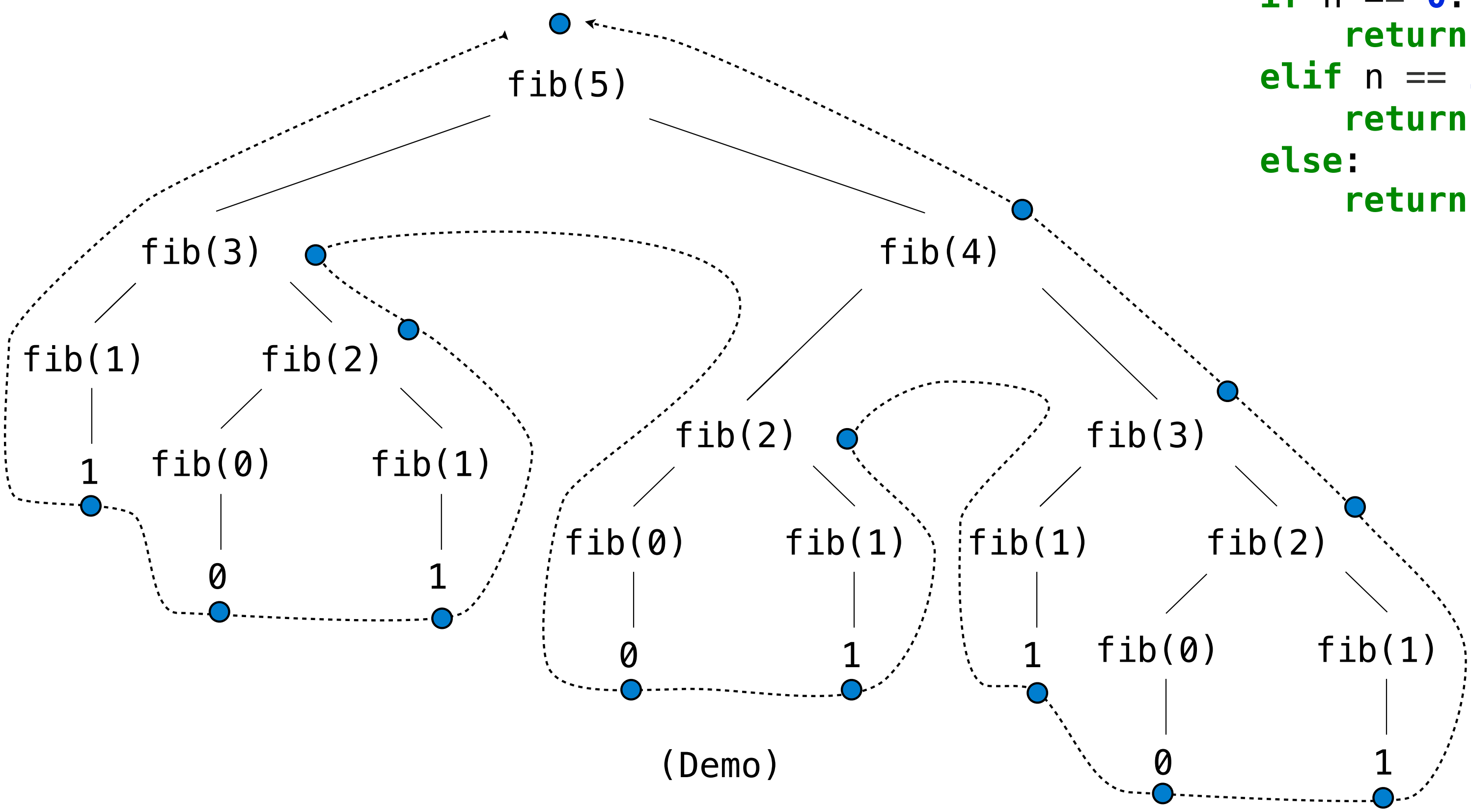


http://en.wikipedia.org/wiki/File:Fibonacci.jpg

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)
```



(Demo)

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

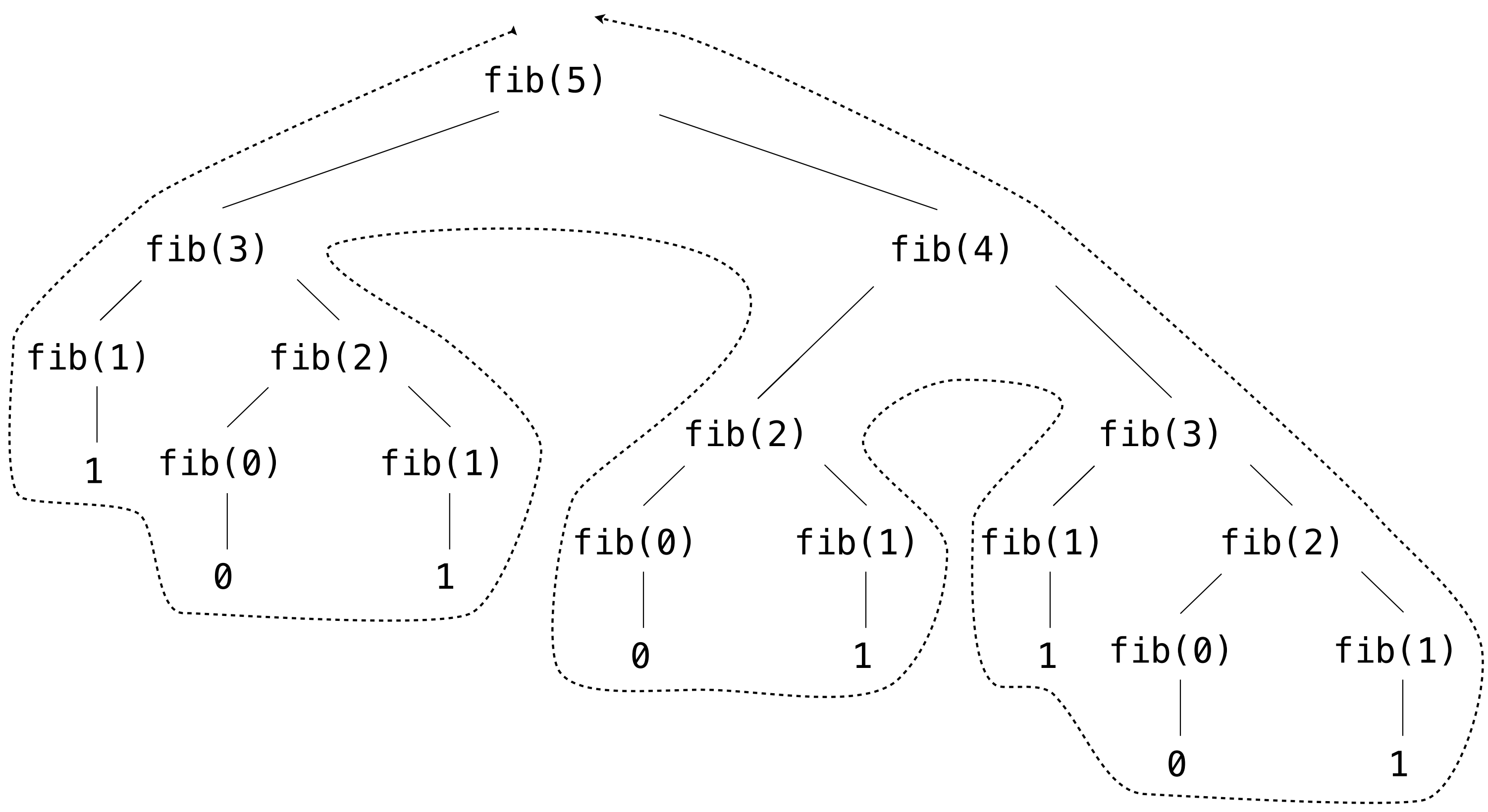
```
def memo(f):
    cache = {}
    def memoized(n):
        if n not in cache:
            cache[n] = f(n)
        return cache[n]
    return memoized
```

Keys are arguments that map to return values

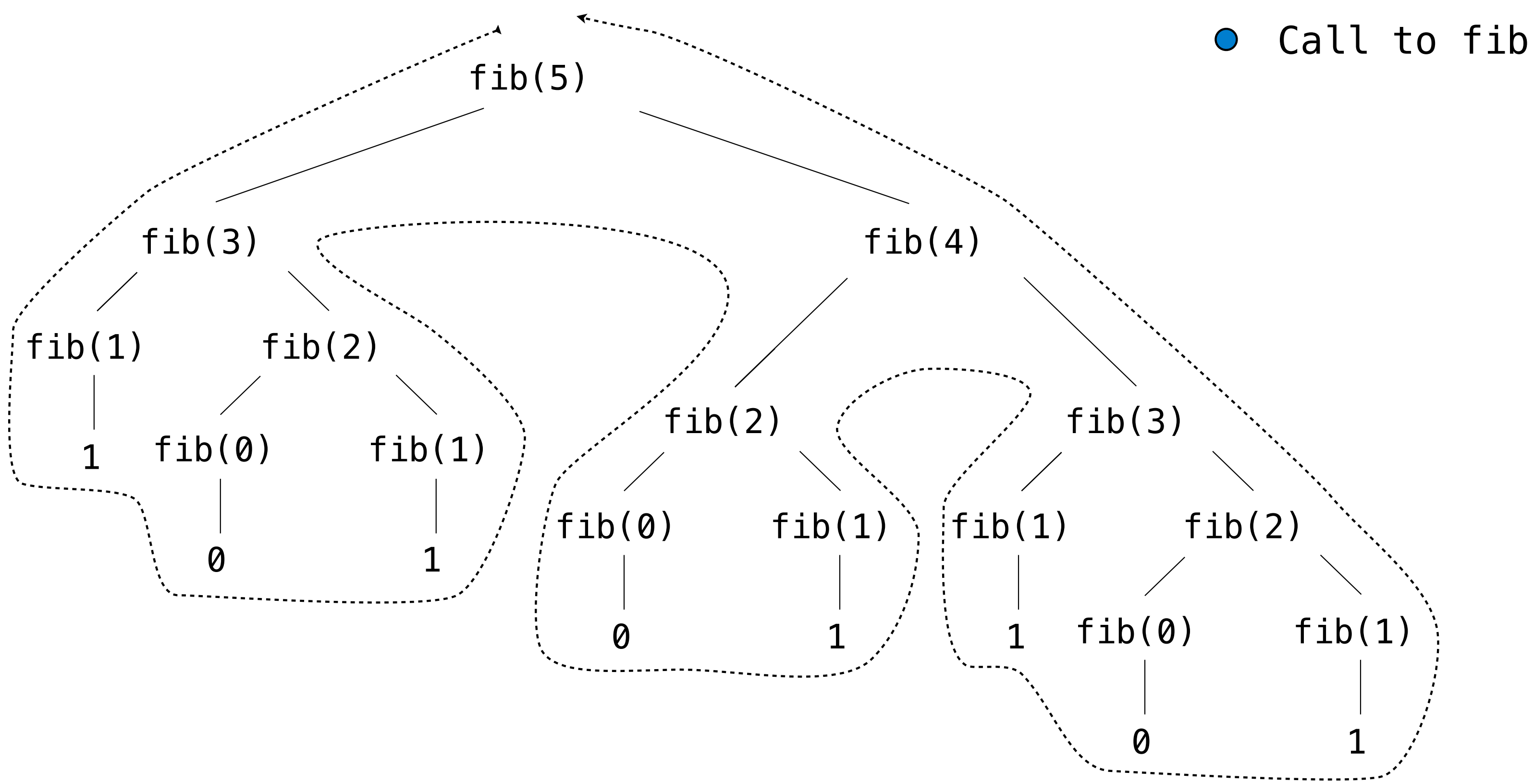
Same behavior as f, if f is a pure function

(Demo)

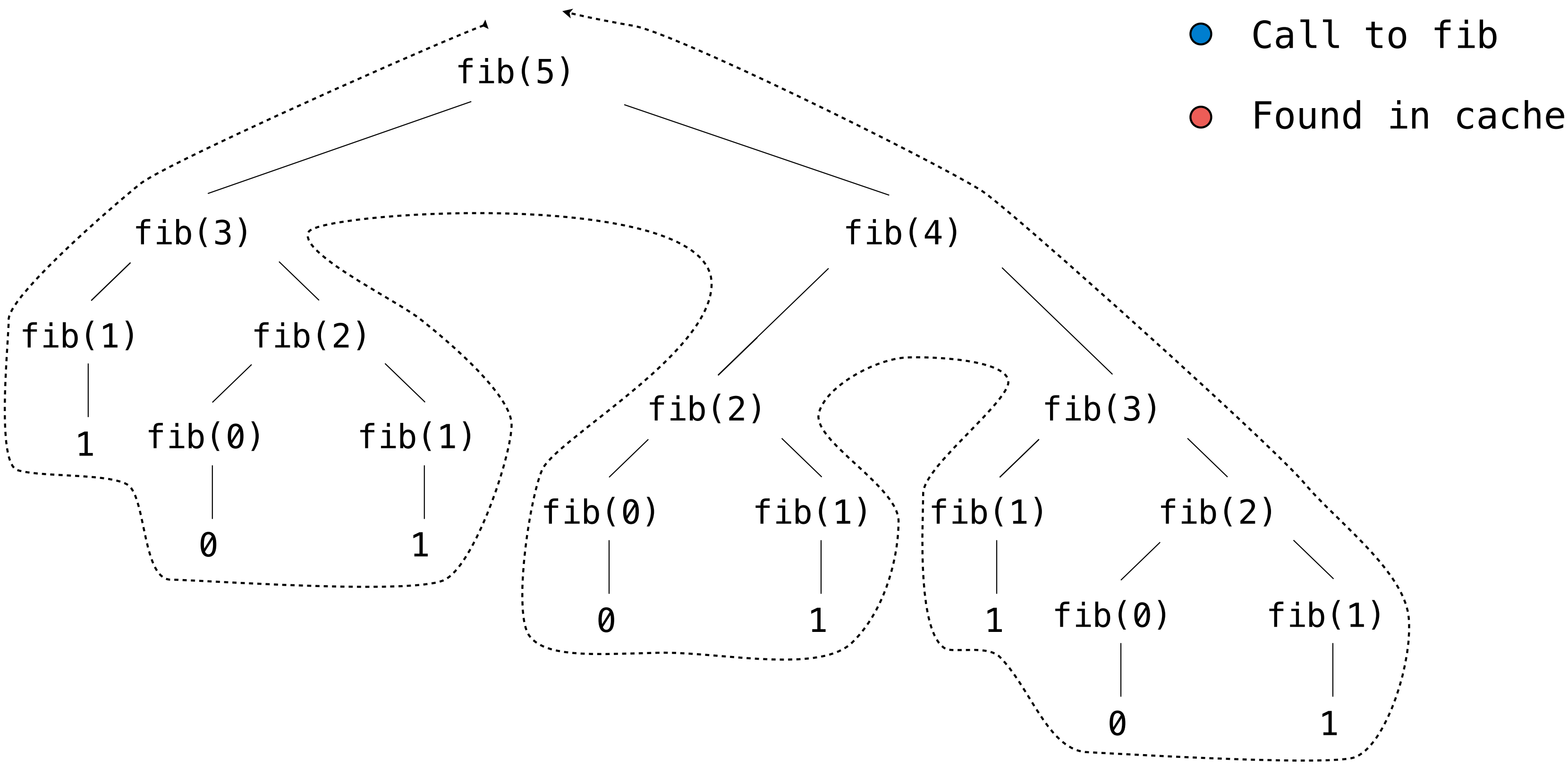
Memoized Tree Recursion



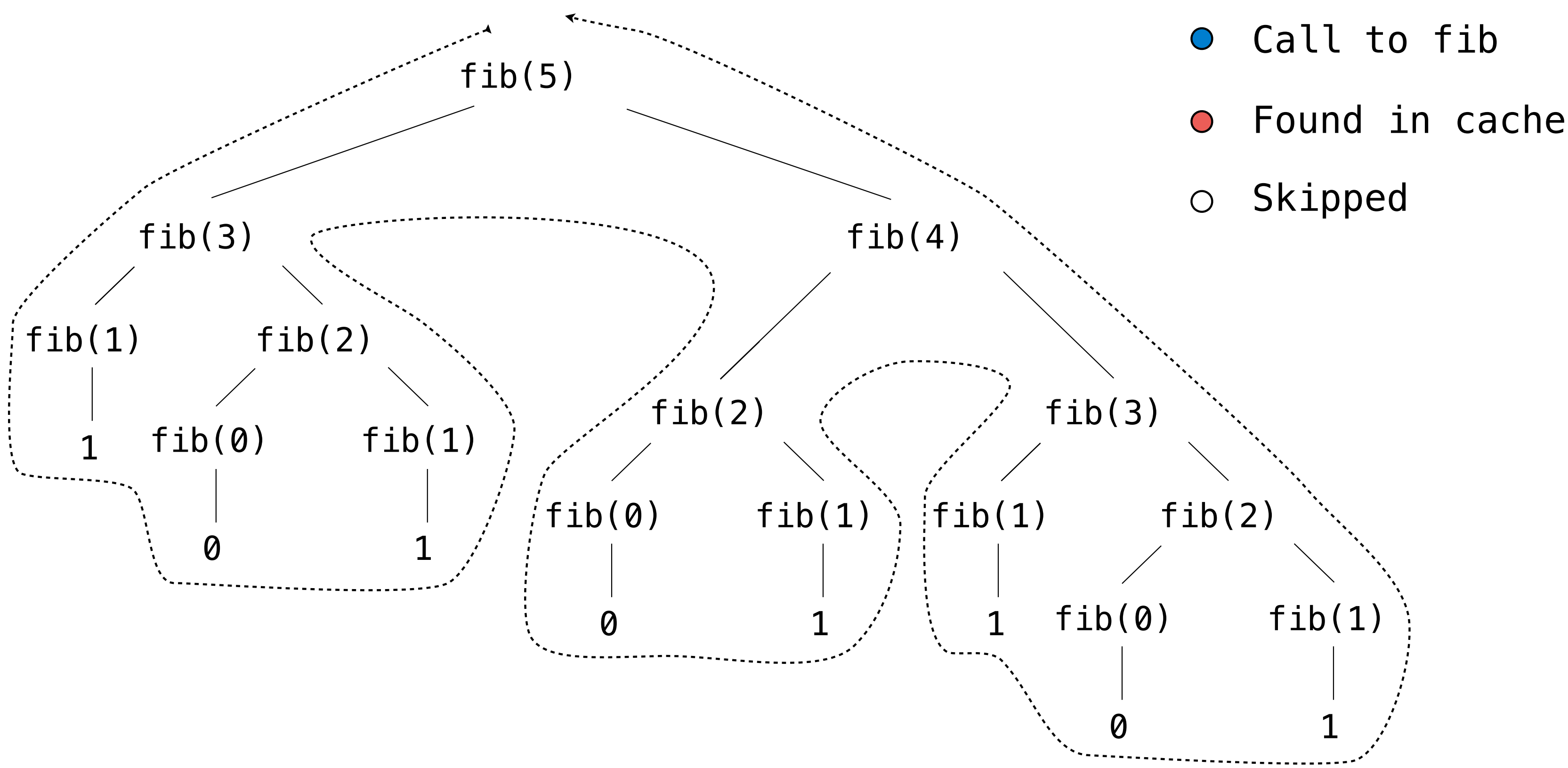
Memoized Tree Recursion



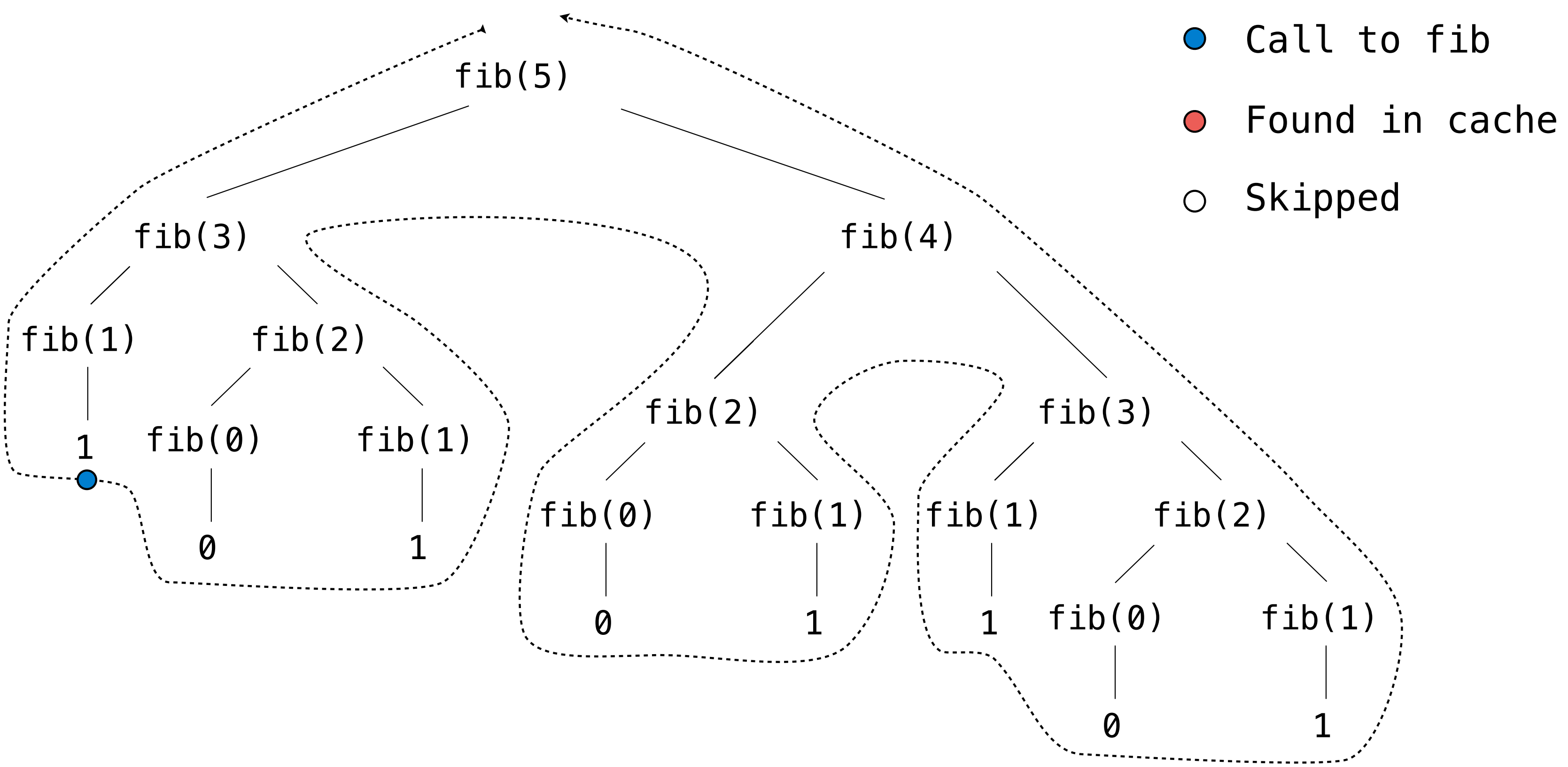
Memoized Tree Recursion



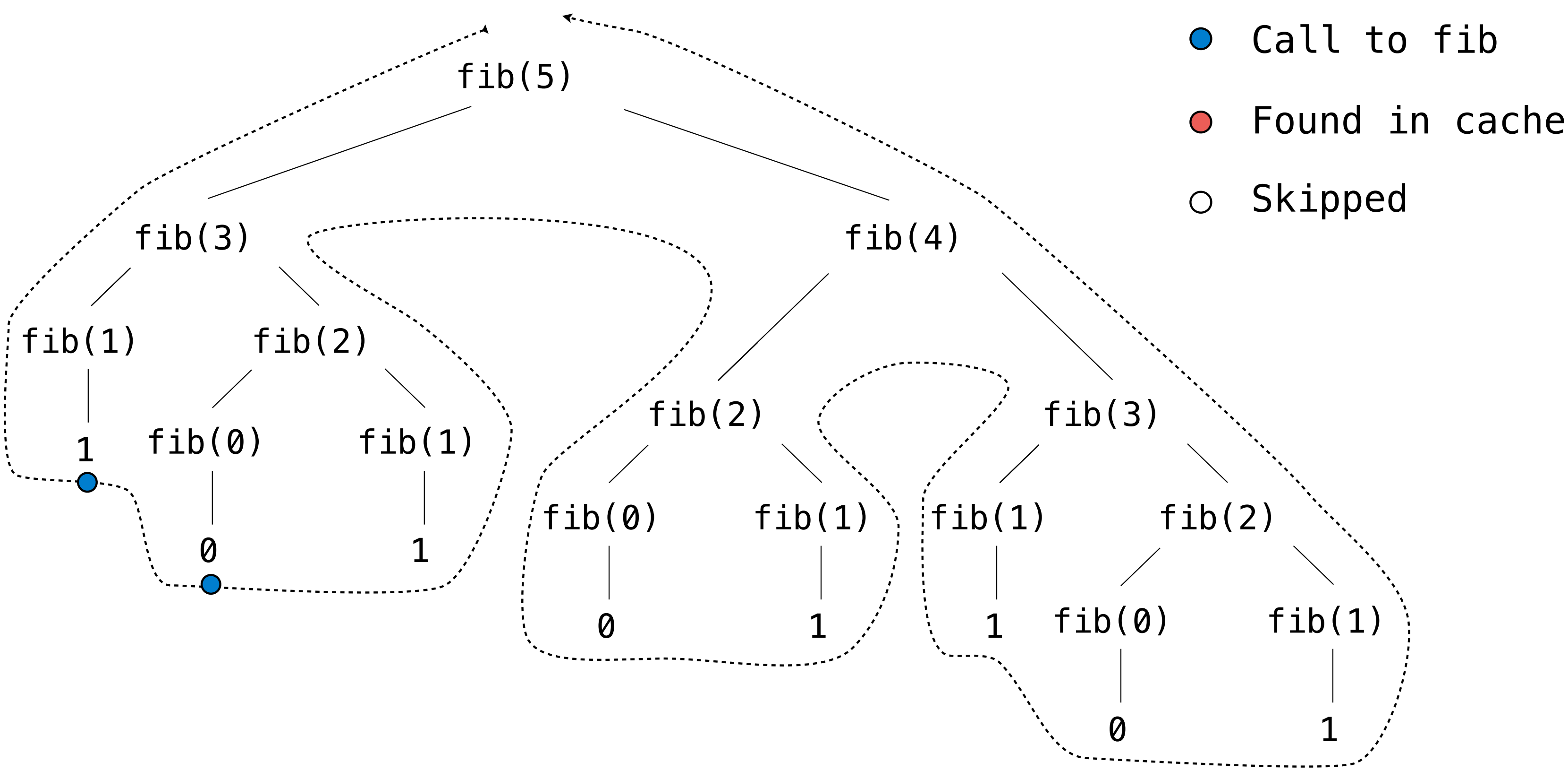
Memoized Tree Recursion



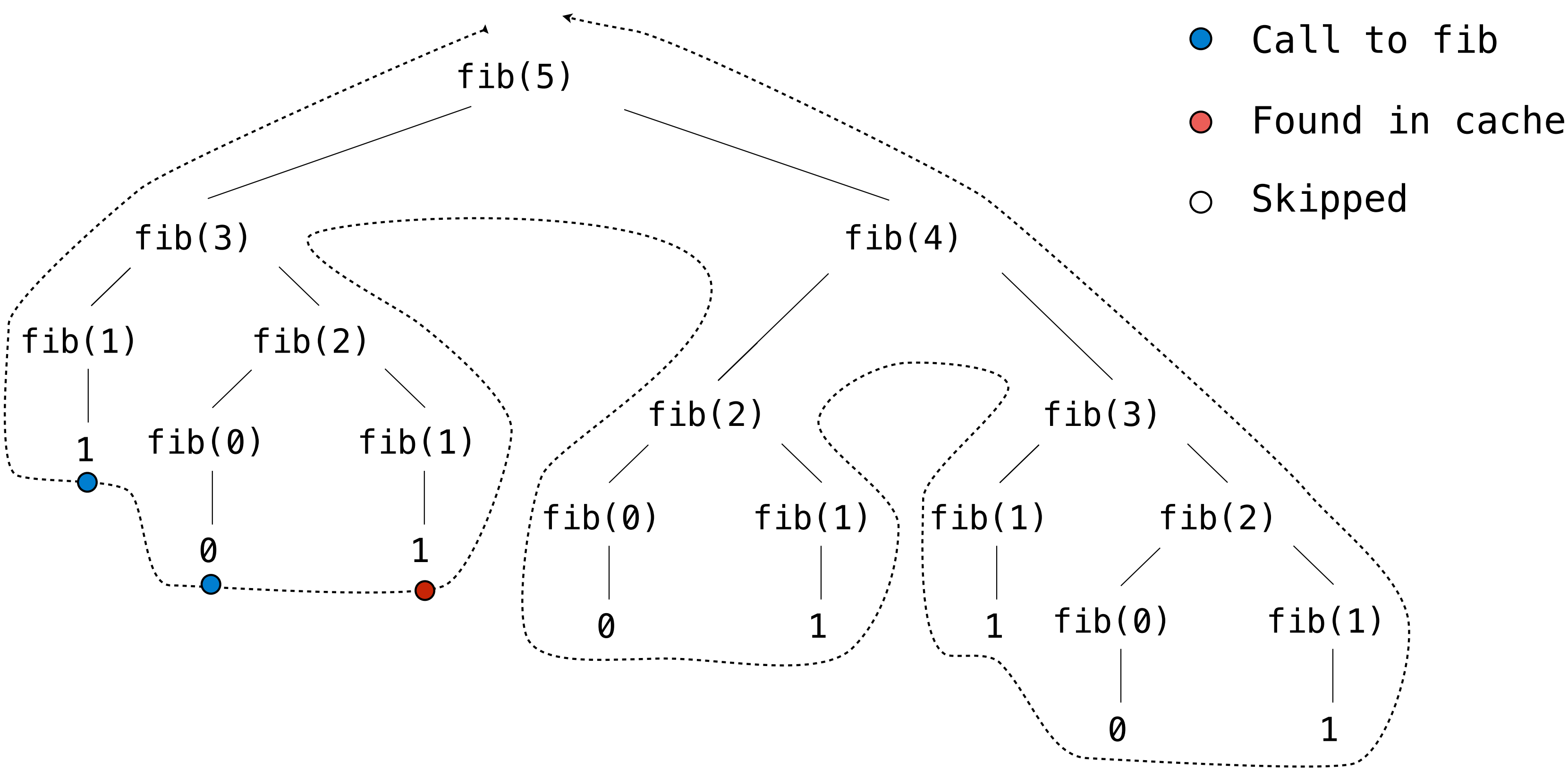
Memoized Tree Recursion



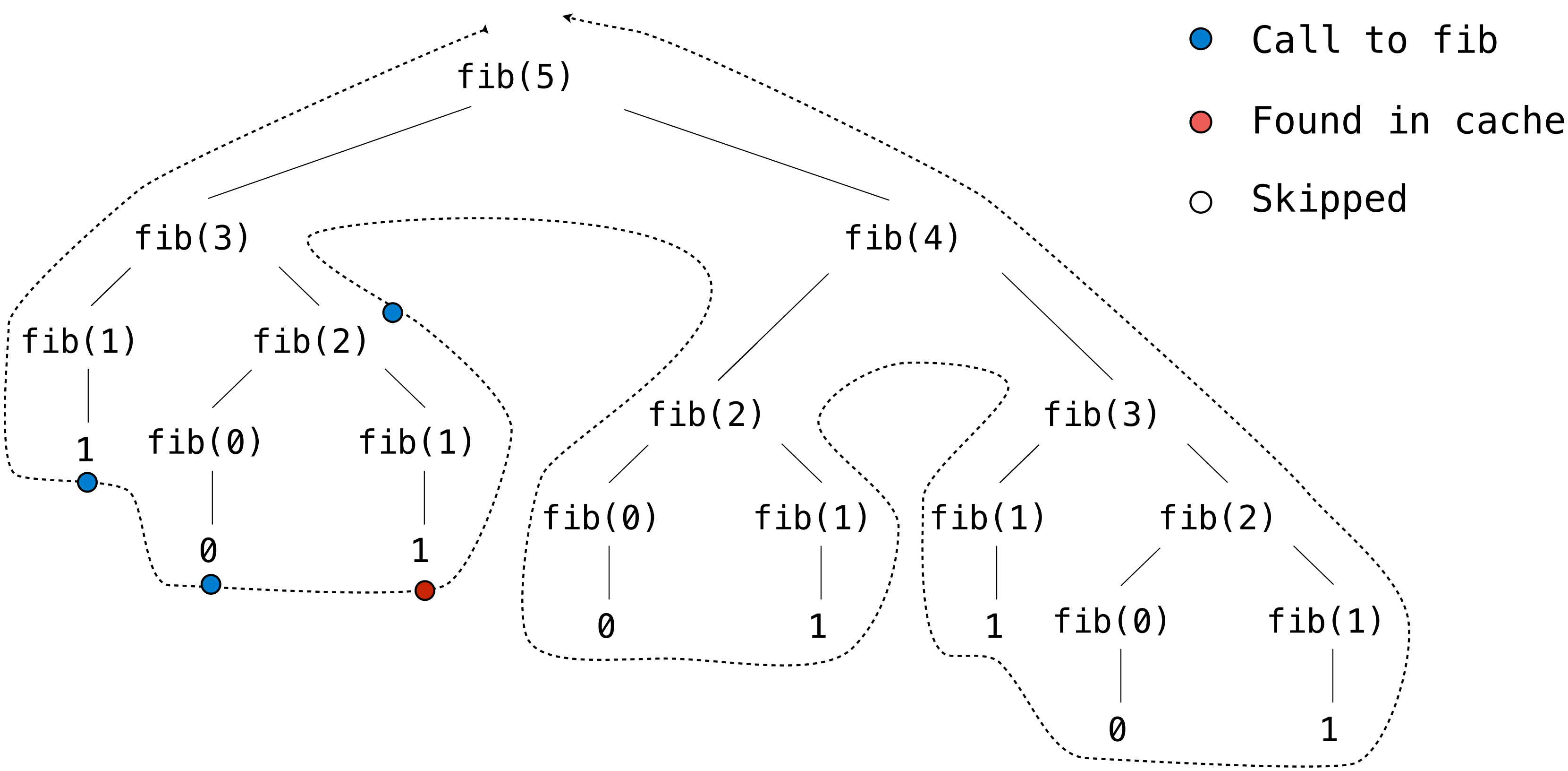
Memoized Tree Recursion



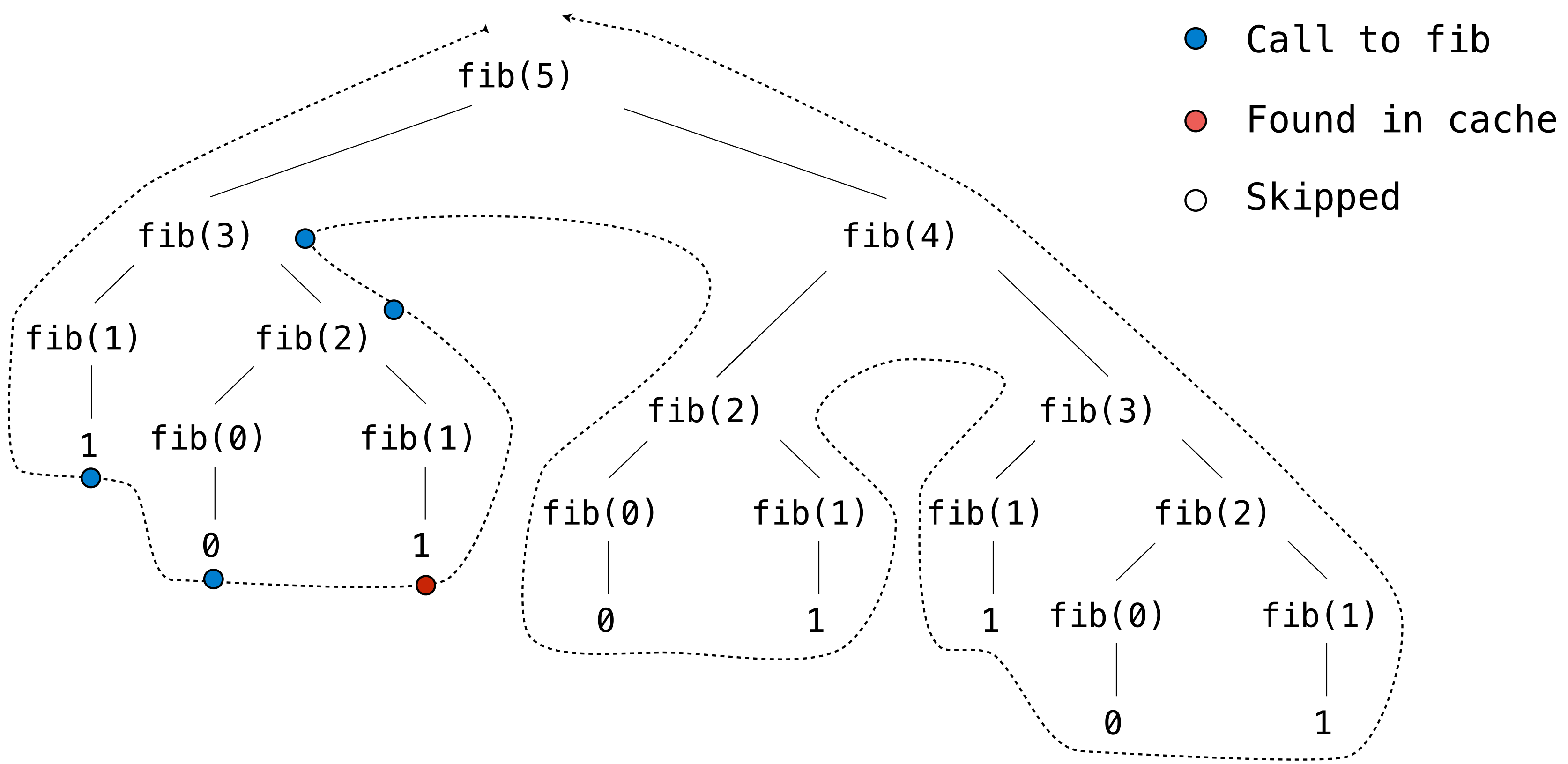
Memoized Tree Recursion



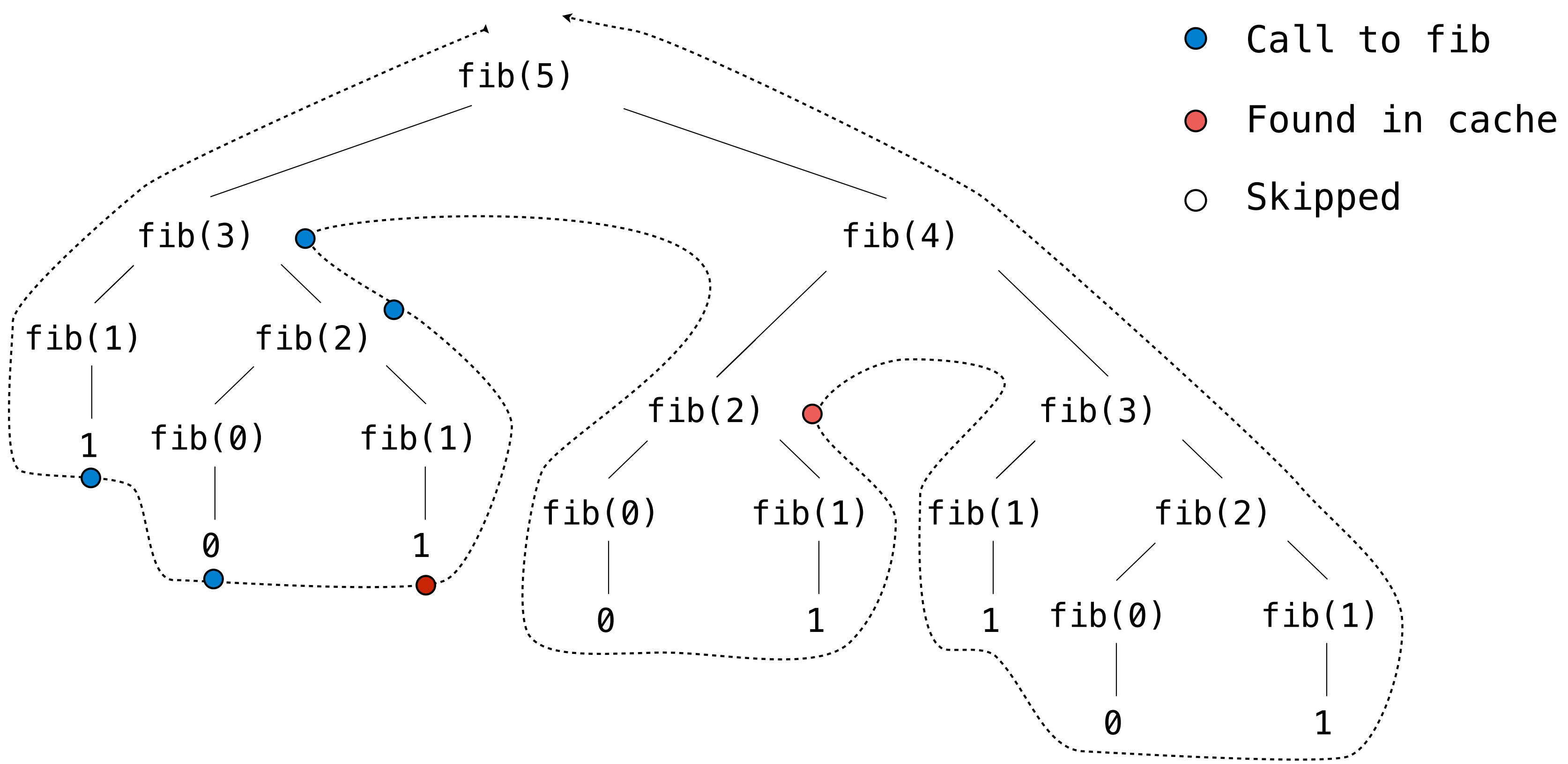
Memoized Tree Recursion



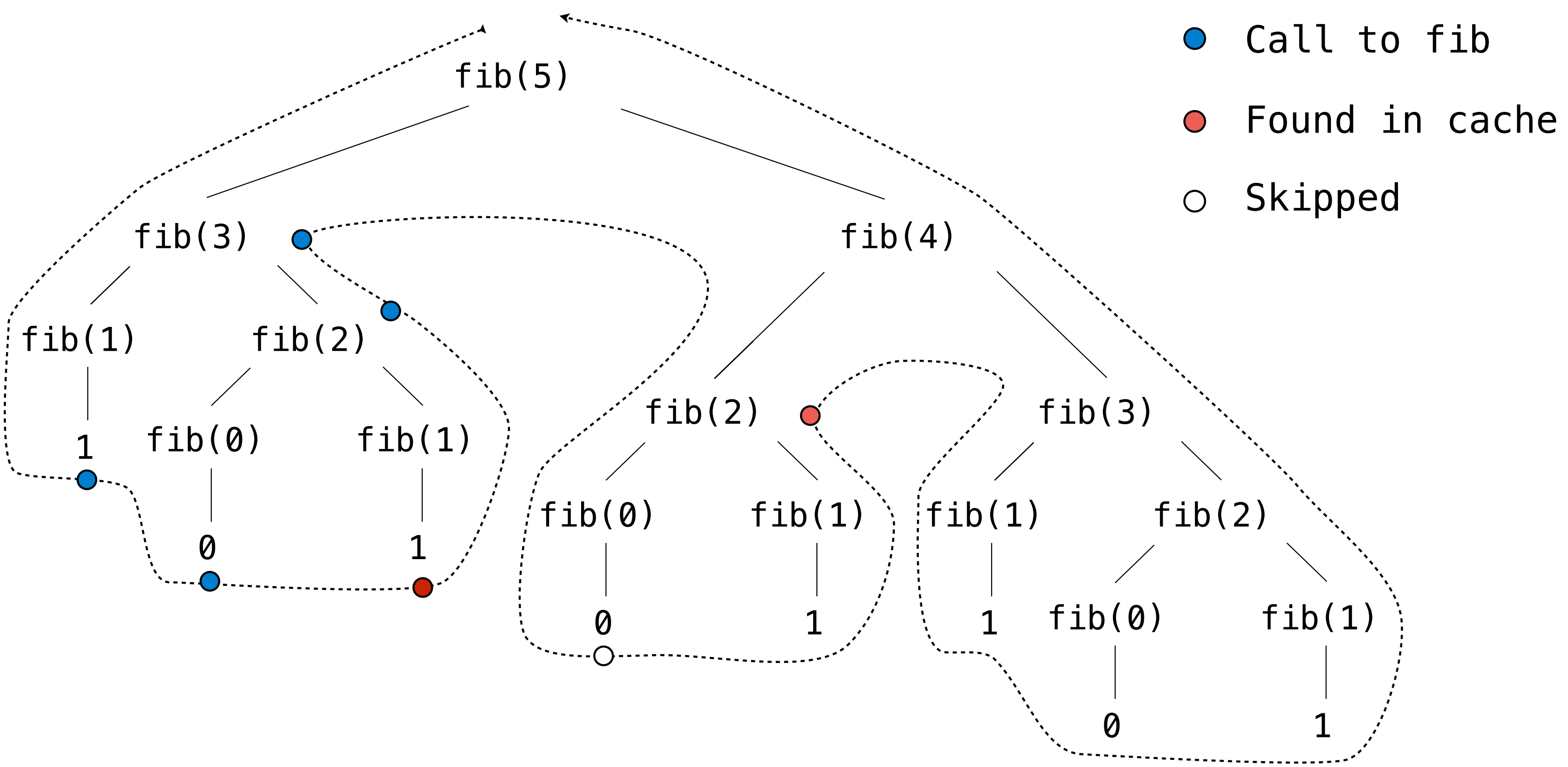
Memoized Tree Recursion



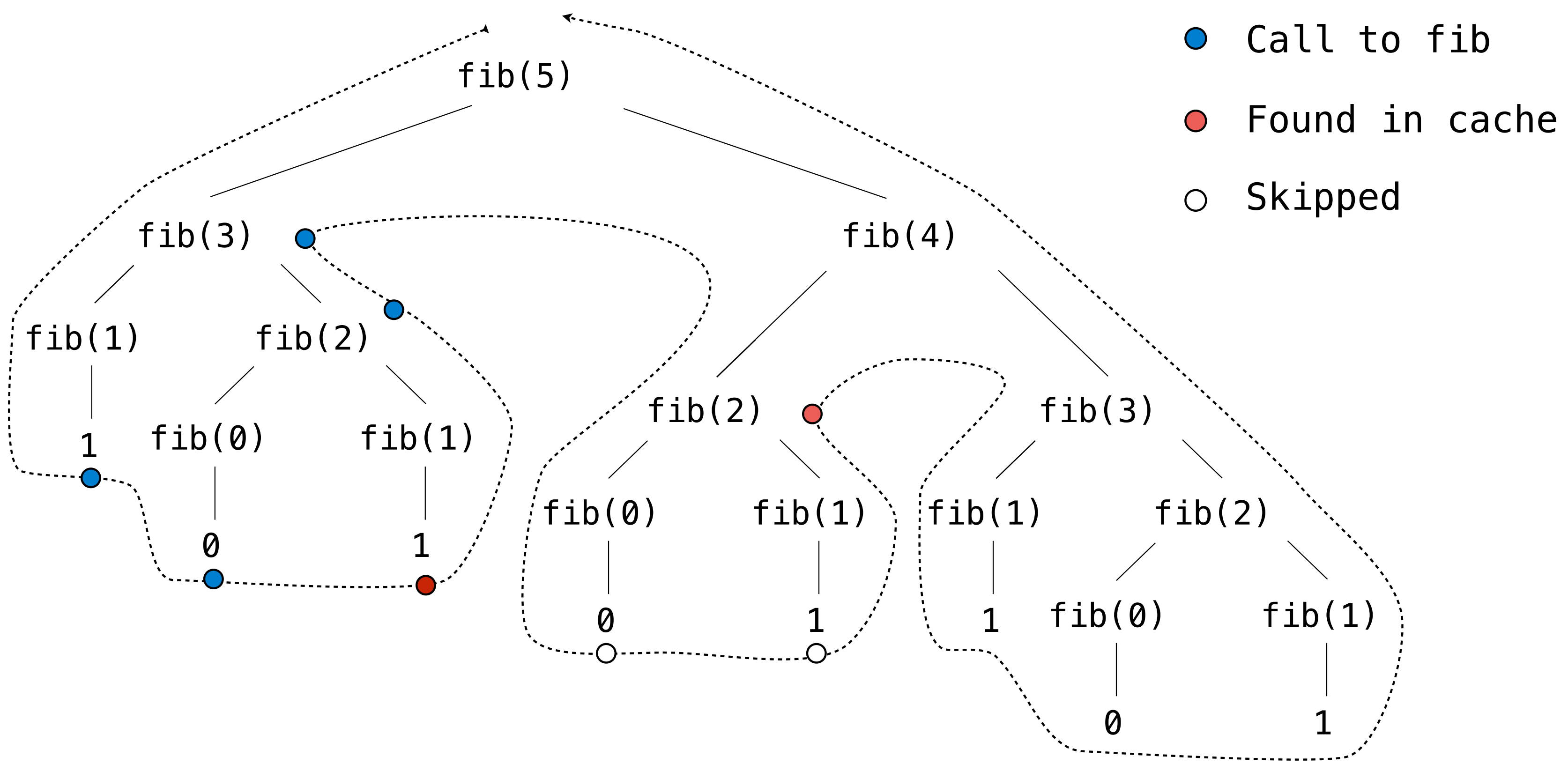
Memoized Tree Recursion



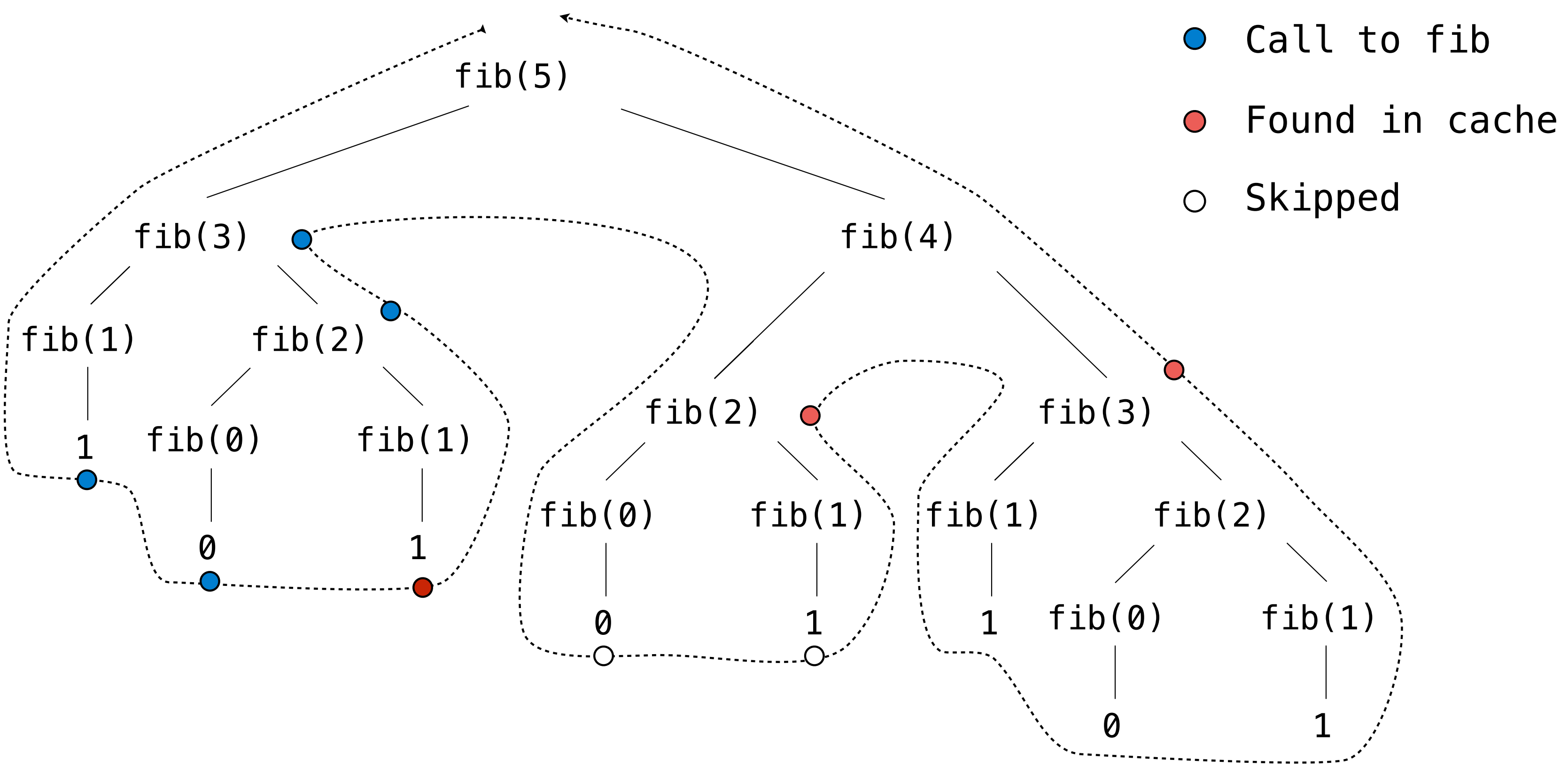
Memoized Tree Recursion



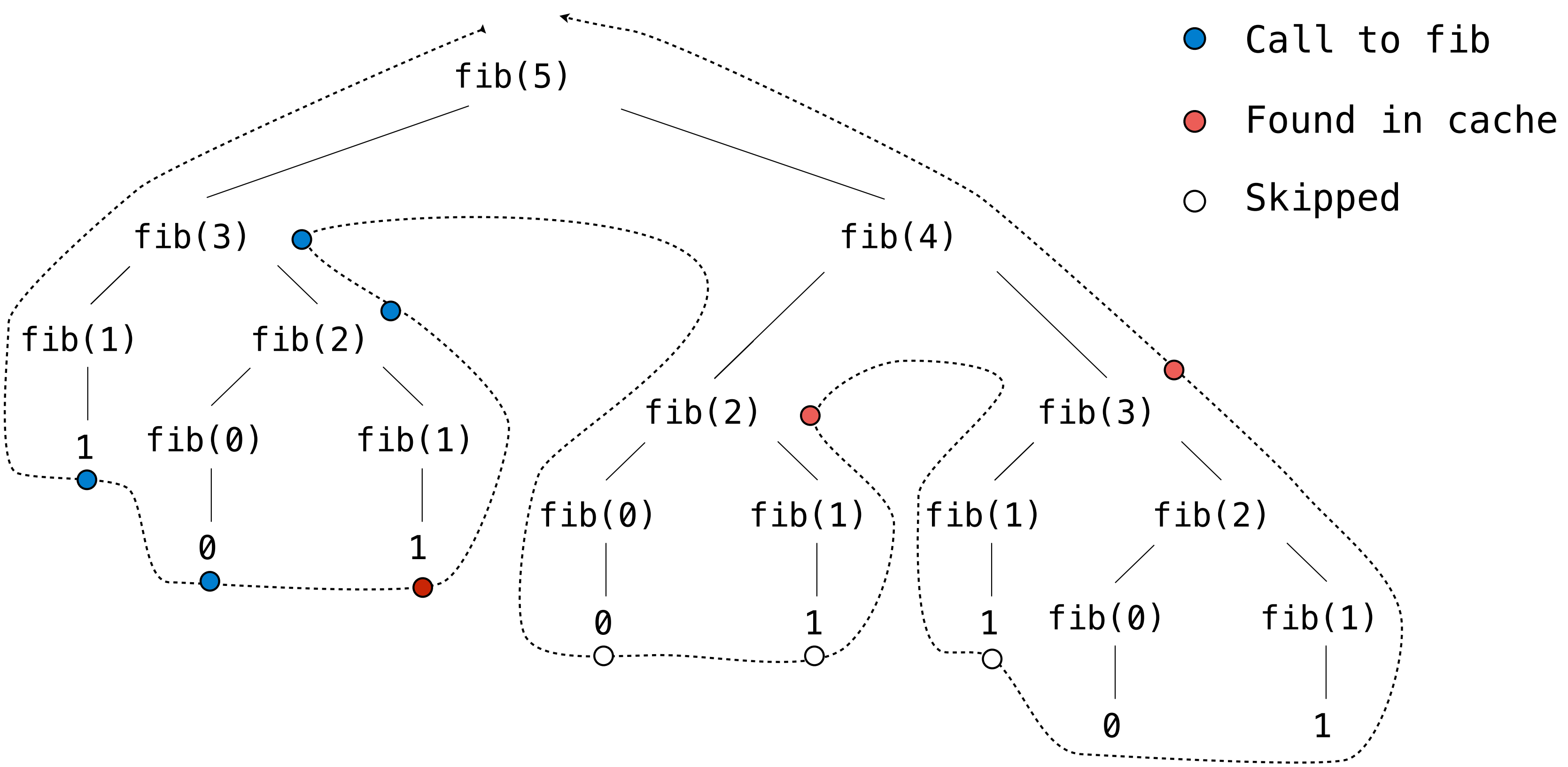
Memoized Tree Recursion



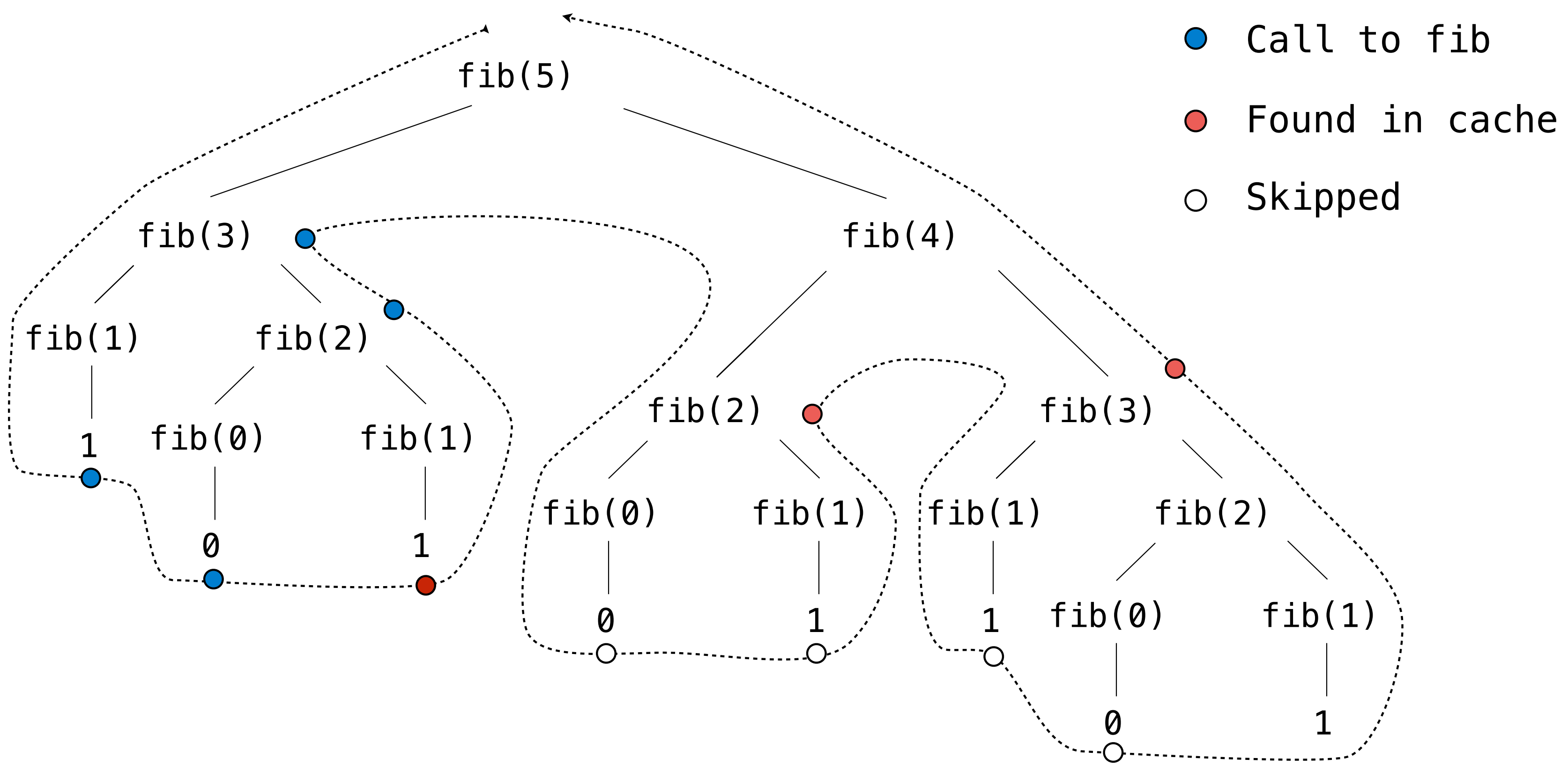
Memoized Tree Recursion



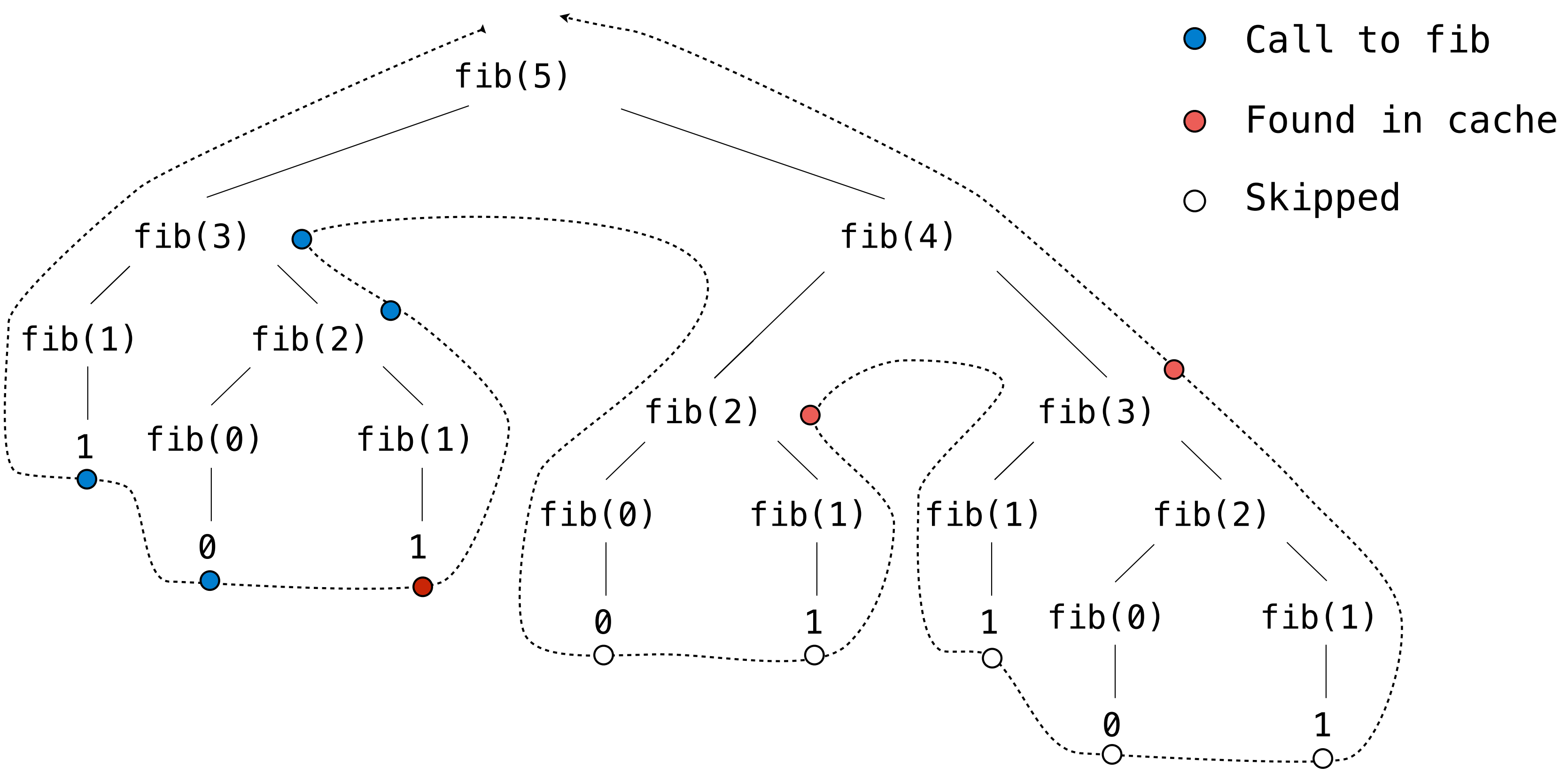
Memoized Tree Recursion



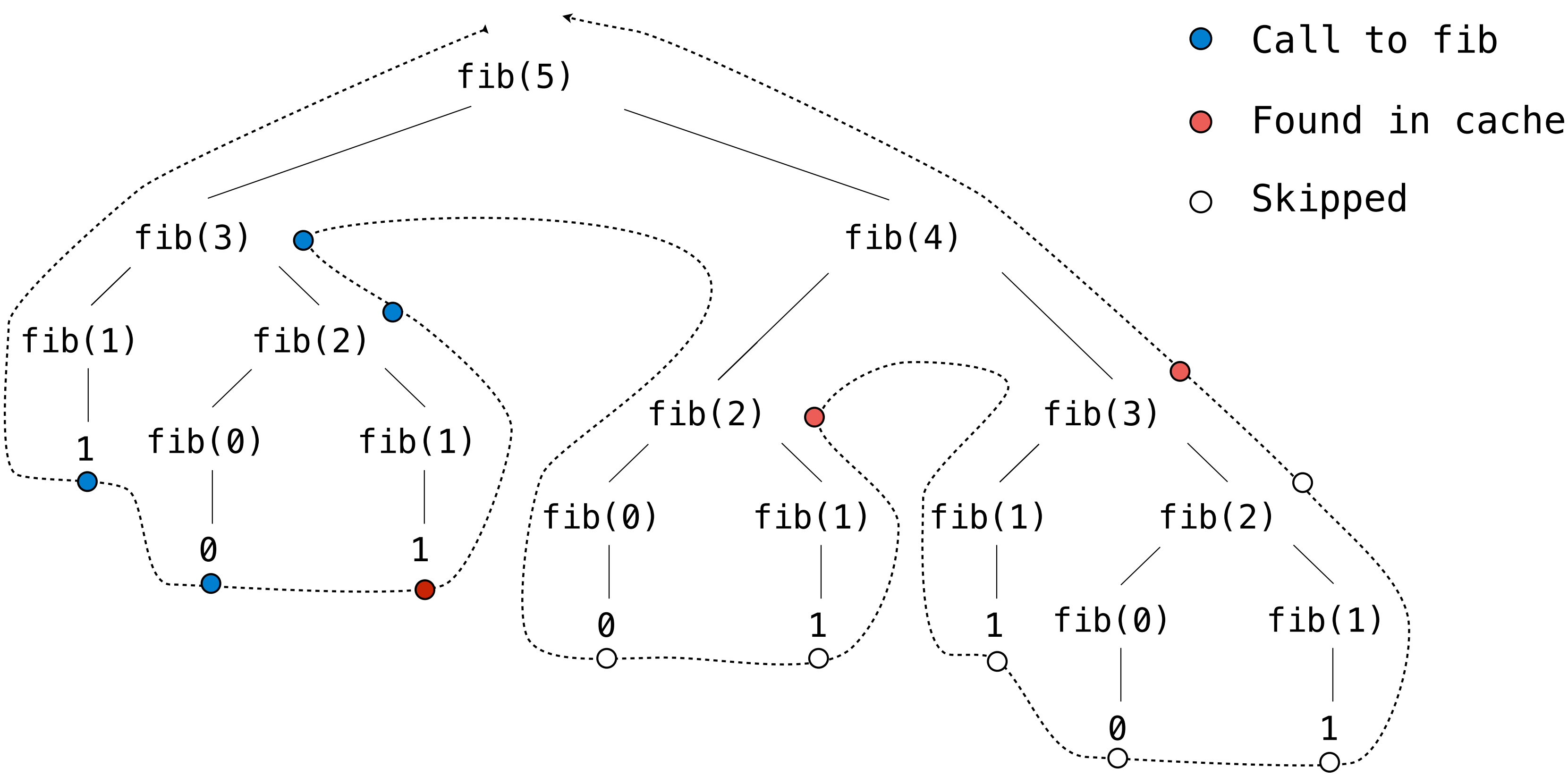
Memoized Tree Recursion



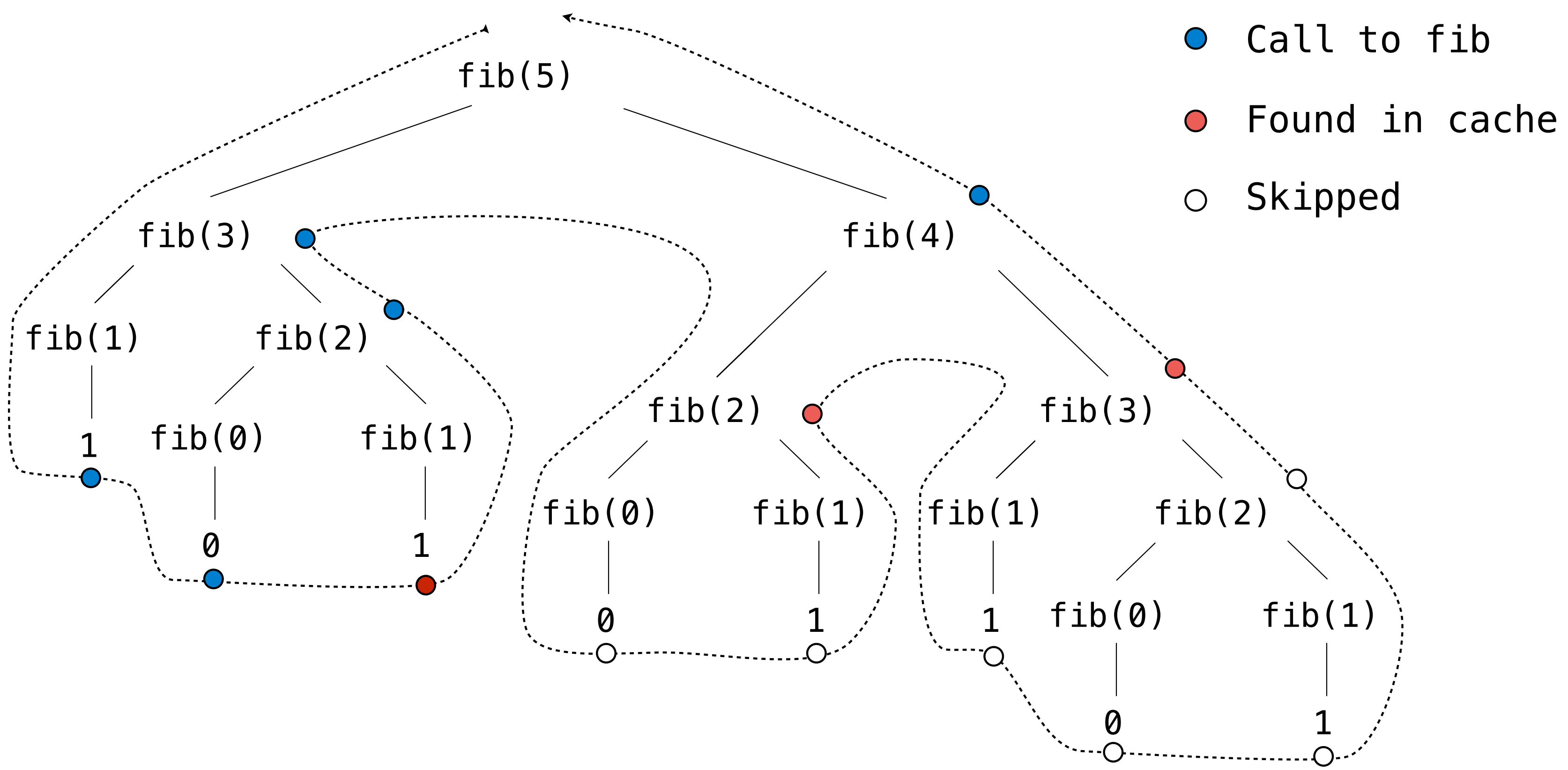
Memoized Tree Recursion



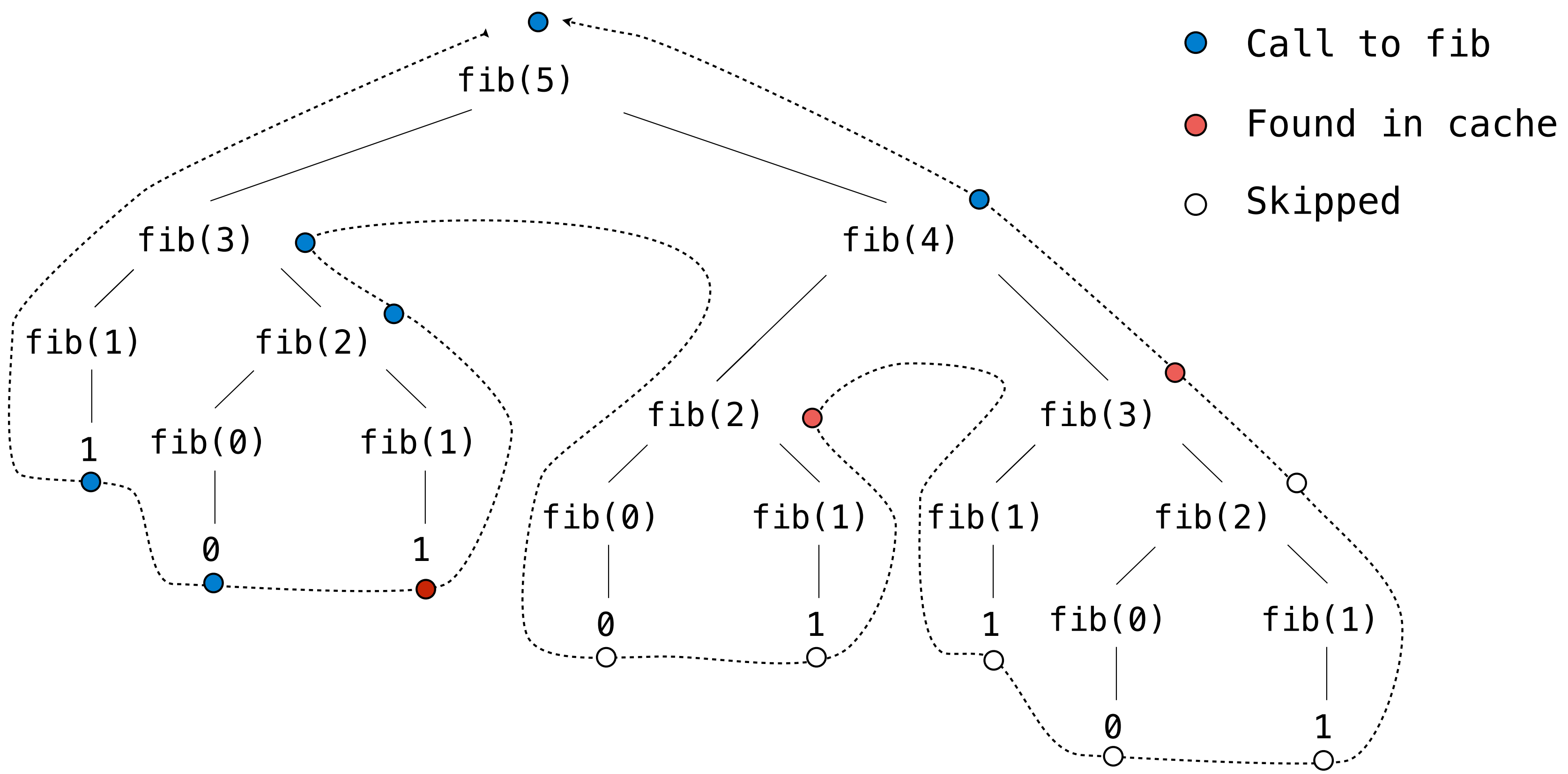
Memoized Tree Recursion



Memoized Tree Recursion



Memoized Tree Recursion



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 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)

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
    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 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)

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

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

(Demo)

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)
```

Linear time:

- Doubling the input **doubles** the time
- 1024x the input takes 1024x as much time

```
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)

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

Logarithmic time:

- Doubling the input **increases** the time by a constant C
- 1024x the input increases the time by only 10 times C

Orders of Growth

Quadratic Time

Functions that process all pairs of values in a sequence of length n take quadratic time

Quadratic Time

Functions that process all pairs of values in a sequence of length n take quadratic time

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

overlap([3, 5, 7, 6], [4, 5, 6, 5])
```

Quadratic Time

Functions that process all pairs of values in a sequence of length n take quadratic time

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

overlap([3, 5, 7, 6], [4, 5, 6, 5])
```

	3	5	7	6
4	0	0	0	0
5	0	1	0	0
6	0	0	0	1
5	0	1	0	0

Quadratic Time

Functions that process all pairs of values in a sequence of length n take quadratic time

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

overlap([3, 5, 7, 6], [4, 5, 6, 5])
```

	3	5	7	6
4	0	0	0	0
5	0	1	0	0
6	0	0	0	1
5	0	1	0	0

Quadratic Time

Functions that process all pairs of values in a sequence of length n take quadratic time

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

overlap([3, 5, 7, 6], [4, 5, 6, 5])
```

	3	5	7	6
4	0	0	0	0
5	0	1	0	0
6	0	0	0	1
5	0	1	0	0

(Demo)

Exponential Time

Tree-recursive functions can take exponential time

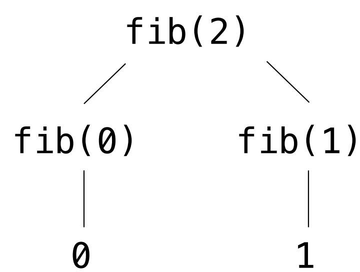
```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-2) + fib(n-1)
```

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

Exponential Time

Tree-recursive functions can take exponential time

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

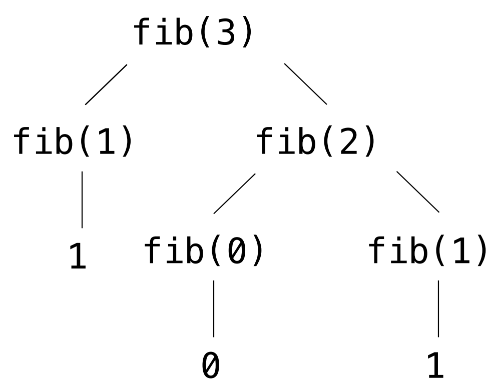


http://en.wikipedia.org/wiki/File:Fibonacci.jpg

Exponential Time

Tree-recursive functions can take exponential time

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

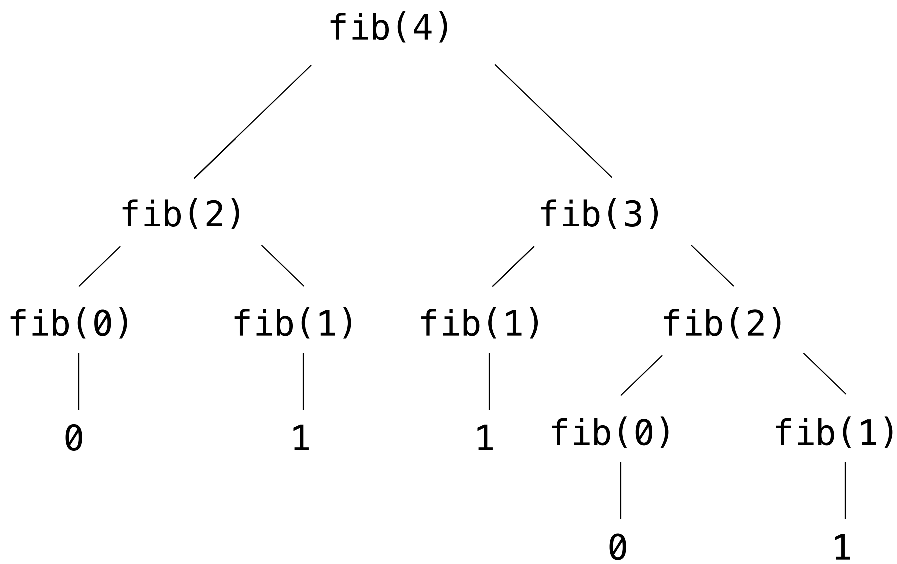


http://en.wikipedia.org/wiki/File:Fibonacci.jpg

Exponential Time

Tree-recursive functions can take exponential time

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

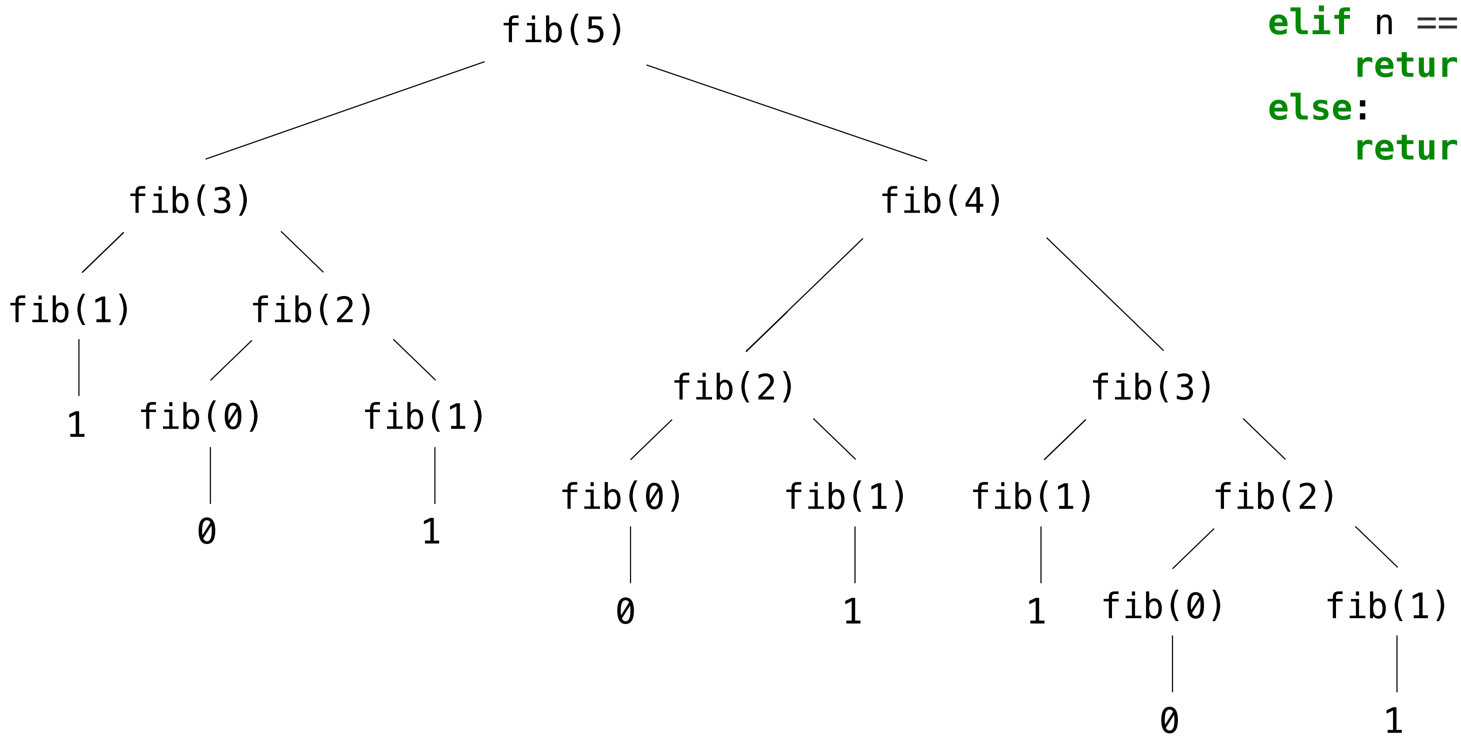


http://en.wikipedia.org/wiki/File:Fibonacci.jpg

Exponential Time

Tree-recursive functions can take exponential time

```
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

Common Orders of Growth

Exponential growth. E.g., recursive fib

Quadratic growth. E.g., overlap

Linear growth. E.g., slow exp

Logarithmic growth. E.g., exp_fast

Constant growth. Increasing *n* doesn't affect time

Common Orders of Growth

Exponential growth. E.g., recursive `fib`

$$a \cdot b^{n+1} = (a \cdot b^n) \cdot b$$

Quadratic growth. E.g., `overlap`

$$a \cdot (n+1)^2 = (a \cdot n^2) + a \cdot (2n+1)$$

Linear growth. E.g., slow `exp`

$$a \cdot (n+1) = (a \cdot n) + a$$

Logarithmic growth. E.g., `exp_fast`

$$a \cdot \ln(2 \cdot n) = (a \cdot \ln n) + a \cdot \ln 2$$

Constant growth. Increasing *n* doesn't affect time

Common Orders of Growth

Time for input n+1

Exponential growth. E.g., recursive `fib`

$$a \cdot b^{n+1} = (a \cdot b^n) \cdot b$$

Quadratic growth. E.g., `overlap`

$$a \cdot (n+1)^2 = (a \cdot n^2) + a \cdot (2n+1)$$

Linear growth. E.g., slow `exp`

$$a \cdot (n+1) = (a \cdot n) + a$$

Logarithmic growth. E.g., `exp_fast`

$$a \cdot \ln(2 \cdot n) = (a \cdot \ln n) + a \cdot \ln 2$$

Constant growth. Increasing *n* doesn't affect time

Time for input n+1	Time for input n

Common Orders of Growth

Exponential growth. E.g., recursive `fib`

$$a \cdot b^{n+1} = (a \cdot b^n) \cdot b$$

Quadratic growth. E.g., `overlap`

$$a \cdot (n+1)^2 = (a \cdot n^2) + a \cdot (2n+1)$$

Linear growth. E.g., slow `exp`

$$a \cdot (n+1) = (a \cdot n) + a$$

Logarithmic growth. E.g., `exp_fast`

$$a \cdot \ln(2 \cdot n) = (a \cdot \ln n) + a \cdot \ln 2$$

Constant growth. Increasing *n* doesn't affect time

Common Orders of Growth

Time for input n+1 | Time for input n

Exponential growth. E.g., recursive `fib`

$$a \cdot b^{n+1} = (a \cdot b^n) \cdot b$$

Quadratic growth. E.g., `overlap`

$$a \cdot (n+1)^2 = (a \cdot n^2) + a \cdot (2n+1)$$

Linear growth. E.g., slow `exp`

$$a \cdot (n+1) = (a \cdot n) + a$$

Logarithmic growth. E.g., `exp_fast`

$$a \cdot \ln(2 \cdot n) = (a \cdot \ln n) + a \cdot \ln 2$$

Constant growth. Increasing *n* doesn't affect time

Common Orders of Growth

Time for input n+1

Time for input n

Exponential growth. E.g., recursive `fib`

$$\boxed{a \cdot b^{n+1}} = \boxed{(a \cdot b^n)} \cdot b$$

Quadratic growth. E.g., `overlap`

$$a \cdot (n+1)^2 = (a \cdot n^2) + a \cdot (2n+1)$$

Linear growth. E.g., slow `exp`

$$a \cdot (n+1) = (a \cdot n) + a$$

Logarithmic growth. E.g., `exp_fast`

$$a \cdot \ln(2 \cdot n) = (a \cdot \ln n) + a \cdot \ln 2$$

Constant growth. Increasing *n* doesn't affect time

Time for input n+1

Time for input n

Common Orders of Growth

Exponential growth. E.g., recursive `fib`

Incrementing *n* multiplies *time* by a constant

$$a \cdot b^{n+1} = (a \cdot b^n) \cdot b$$

Quadratic growth. E.g., `overlap`

$$a \cdot (n+1)^2 = (a \cdot n^2) + a \cdot (2n+1)$$

Linear growth. E.g., slow `exp`

$$a \cdot (n+1) = (a \cdot n) + a$$

Logarithmic growth. E.g., `exp_fast`

$$a \cdot \ln(2 \cdot n) = (a \cdot \ln n) + a \cdot \ln 2$$

Constant growth. Increasing *n* doesn't affect time

Common Orders of Growth

Time for input n+1

Time for input n

Exponential growth. E.g., recursive `fib`

Incrementing n multiplies *time* by a constant

$$\boxed{a \cdot b^{n+1}} = \boxed{(a \cdot b^n)} \cdot b$$

Quadratic growth. E.g., `overlap`

$$\boxed{a \cdot (n+1)^2} = (a \cdot n^2) + a \cdot (2n+1)$$

Linear growth. E.g., slow `exp`

$$a \cdot (n+1) = (a \cdot n) + a$$

Logarithmic growth. E.g., `exp_fast`

$$a \cdot \ln(2 \cdot n) = (a \cdot \ln n) + a \cdot \ln 2$$

Constant growth. Increasing n doesn't affect time

Common Orders of Growth

Time for input n+1		Time for input n

Exponential growth. E.g., recursive `fib`

Incrementing *n* multiplies *time* by a constant

$$\boxed{a \cdot b^{n+1}} = \boxed{(a \cdot b^n)} \cdot b$$

Quadratic growth. E.g., `overlap`

$$\boxed{a \cdot (n+1)^2} = \boxed{(a \cdot n^2)} + a \cdot (2n+1)$$

Linear growth. E.g., slow `exp`

$$a \cdot (n+1) = (a \cdot n) + a$$

Logarithmic growth. E.g., `exp_fast`

$$a \cdot \ln(2 \cdot n) = (a \cdot \ln n) + a \cdot \ln 2$$

Constant growth. Increasing *n* doesn't affect time

Common Orders of Growth

Time for input n+1

Time for input n

Exponential growth. E.g., recursive fib

Incrementing *n* multiplies *time* by a constant

$$\boxed{a \cdot b^{n+1}} = \boxed{(a \cdot b^n)} \cdot b$$

Quadratic growth. E.g., overlap

Incrementing *n* increases *time* by *n* times a constant

$$\boxed{a \cdot (n+1)^2} = \boxed{(a \cdot n^2)} + a \cdot (2n+1)$$

Linear growth. E.g., slow exp

$$a \cdot (n+1) = (a \cdot n) + a$$

Logarithmic growth. E.g., exp_fast

$$a \cdot \ln(2 \cdot n) = (a \cdot \ln n) + a \cdot \ln 2$$

Constant growth. Increasing *n* doesn't affect time

Time for input n+1

Time for input n

Common Orders of Growth

Exponential growth. E.g., recursive fib

Incrementing n multiplies *time* by a constant

$$a \cdot b^{n+1} = (a \cdot b^n) \cdot b$$

Quadratic growth. E.g., overlap

Incrementing n increases *time* by n times a constant

$$a \cdot (n+1)^2 = (a \cdot n^2) + a \cdot (2n+1)$$

Linear growth. E.g., slow exp

$$a \cdot (n+1) = (a \cdot n) + a$$

Logarithmic growth. E.g., exp_fast

$$a \cdot \ln(2 \cdot n) = (a \cdot \ln n) + a \cdot \ln 2$$

Constant growth. Increasing n doesn't affect time

Common Orders of Growth

Time for input n+1 | Time for input n

Exponential growth. E.g., recursive `fib`

Incrementing *n* multiplies *time* by a constant

$$\boxed{a \cdot b^{n+1}} = \boxed{(a \cdot b^n)} \cdot b$$

Quadratic growth. E.g., `overlap`

Incrementing *n* increases *time* by *n* times a constant

$$\boxed{a \cdot (n+1)^2} = \boxed{(a \cdot n^2)} + a \cdot (2n+1)$$

Linear growth. E.g., slow `exp`

$$\boxed{a \cdot (n+1)} = \boxed{(a \cdot n)} + a$$

Logarithmic growth. E.g., `exp_fast`

$$a \cdot \ln(2 \cdot n) = (a \cdot \ln n) + a \cdot \ln 2$$

Constant growth. Increasing *n* doesn't affect time

Common Orders of Growth

Time for input n+1	Time for input n

Exponential growth. E.g., recursive `fib`

Incrementing *n* multiplies *time* by a constant

$$\boxed{a \cdot b^{n+1}} = \boxed{(a \cdot b^n)} \cdot b$$

Quadratic growth. E.g., `overlap`

Incrementing *n* increases *time* by *n* times a constant

$$\boxed{a \cdot (n+1)^2} = \boxed{(a \cdot n^2)} + a \cdot (2n+1)$$

Linear growth. E.g., slow `exp`

Incrementing *n* increases *time* by a constant

$$\boxed{a \cdot (n+1)} = \boxed{(a \cdot n)} + a$$

Logarithmic growth. E.g., `exp_fast`

$$a \cdot \ln(2 \cdot n) = (a \cdot \ln n) + a \cdot \ln 2$$

Constant growth. Increasing *n* doesn't affect time

Common Orders of Growth

Time for n+n

Time for input n+1

Time for input n

Exponential growth. E.g., recursive fib

Incrementing *n* multiplies *time* by a constant

$$a \cdot b^{n+1} = (a \cdot b^n) \cdot b$$

Quadratic growth. E.g., overlap

Incrementing *n* increases *time* by *n* times a constant

$$a \cdot (n+1)^2 = (a \cdot n^2) + a \cdot (2n+1)$$

Linear growth. E.g., slow exp

Incrementing *n* increases *time* by a constant

$$a \cdot (n+1) = (a \cdot n) + a$$

Logarithmic growth. E.g., exp_fast

$$a \cdot \ln(2 \cdot n) = (a \cdot \ln n) + a \cdot \ln 2$$

Constant growth. Increasing *n* doesn't affect time

Common Orders of Growth

Time for n+n

Time for input n+1

Time for input n

Exponential growth. E.g., recursive `fib`

Incrementing *n* multiplies *time* by a constant

$$a \cdot b^{n+1} = (a \cdot b^n) \cdot b$$

Quadratic growth. E.g., `overlap`

Incrementing *n* increases *time* by *n* times a constant

$$a \cdot (n+1)^2 = (a \cdot n^2) + a \cdot (2n+1)$$

Linear growth. E.g., slow `exp`

Incrementing *n* increases *time* by a constant

$$a \cdot (n+1) = (a \cdot n) + a$$

Logarithmic growth. E.g., `exp_fast`

$$a \cdot \ln(2 \cdot n) = (a \cdot \ln n) + a \cdot \ln 2$$

Constant growth. Increasing *n* doesn't affect time

Common Orders of Growth

Time for n+n | Time for input n+1 | Time for input n

Exponential growth. E.g., recursive `fib`

Incrementing *n* multiplies *time* by a constant

$$\boxed{a \cdot b^{n+1}} = \boxed{(a \cdot b^n)} \cdot b$$

Quadratic growth. E.g., `overlap`

Incrementing *n* increases *time* by *n* times a constant

$$\boxed{a \cdot (n+1)^2} = \boxed{(a \cdot n^2)} + a \cdot (2n+1)$$

Linear growth. E.g., slow `exp`

Incrementing *n* increases *time* by a constant

$$\boxed{a \cdot (n+1)} = \boxed{(a \cdot n)} + a$$

Logarithmic growth. E.g., `exp_fast`

$$\boxed{a \cdot \ln(2 \cdot n)} = \boxed{(a \cdot \ln n)} + a \cdot \ln 2$$

Constant growth. Increasing *n* doesn't affect time

Common Orders of Growth

Time for n+n | Time for input n+1 | Time for input n

Exponential growth. E.g., recursive `fib`

Incrementing *n* multiplies *time* by a constant

$$\boxed{a \cdot b^{n+1}} = \boxed{(a \cdot b^n)} \cdot b$$

Quadratic growth. E.g., `overlap`

Incrementing *n* increases *time* by *n* times a constant

$$\boxed{a \cdot (n+1)^2} = \boxed{(a \cdot n^2)} + a \cdot (2n+1)$$

Linear growth. E.g., slow `exp`

Incrementing *n* increases *time* by a constant

$$\boxed{a \cdot (n+1)} = \boxed{(a \cdot n)} + a$$

Logarithmic growth. E.g., `exp_fast`

Doubling *n* only increments *time* by a constant

$$\boxed{a \cdot \ln(2 \cdot n)} = \boxed{(a \cdot \ln n)} + a \cdot \ln 2$$

Constant growth. Increasing *n* doesn't affect time

Space

Space and Environments

Space and Environments

Which environment frames do we need to keep during evaluation?

Space and Environments

Which environment frames do we need to keep during evaluation?

At any moment there is a set of active environments

Space and Environments

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

Space and Environments

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

Space and Environments

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:

Space and Environments

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

Space and Environments

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

Space and Environments

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)

pythontutor.com/
composingprograms.html#code=def%20fib%28n%29%3A%0A%20%20%20%20if%20n%20%3D%3D%200%20or%20n%20%3D%3D%201%3A%0A%20%20%20%20%20%20%20%20return%20n%0A%20%20%20%20else%3A%0A%20%20%20%20%20%20%20%20return%20fib%28n-2%29%20%2B%20fib%28n-1%29%0A%20%20%20%20%20%20%20%20%0Afib%286%29&mode=display&
origin=composingprograms.js&cumulative=false&py=3&rawInputLstJSON=[]&curInstr=1

Fibonacci Space Consumption

Fibonacci Space Consumption

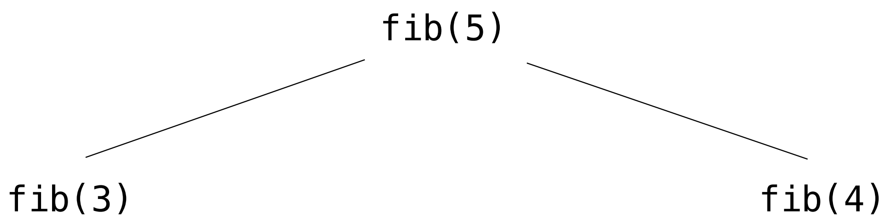
```
fib(5)
```

Fibonacci Space Consumption

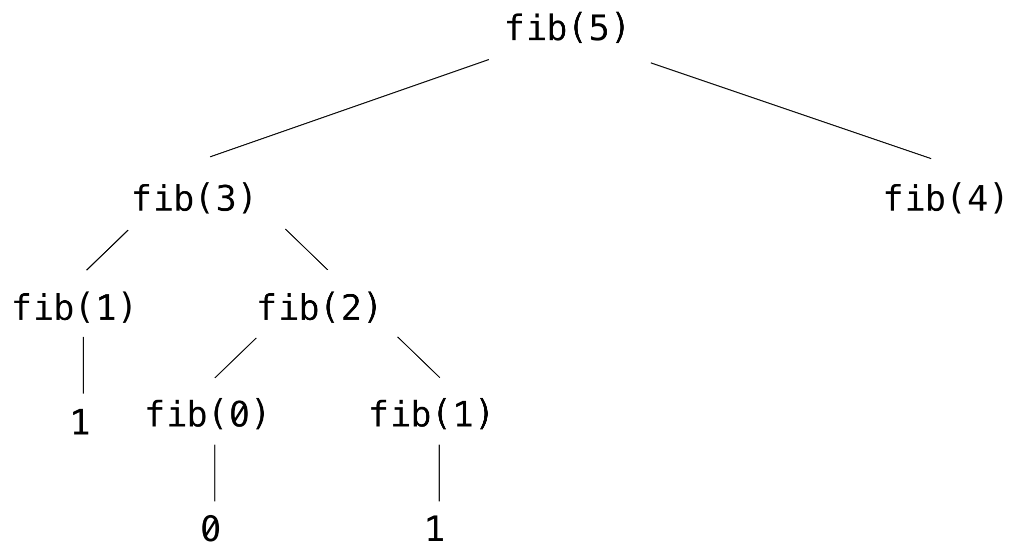
```
                fib(5)

        fib(3)
```

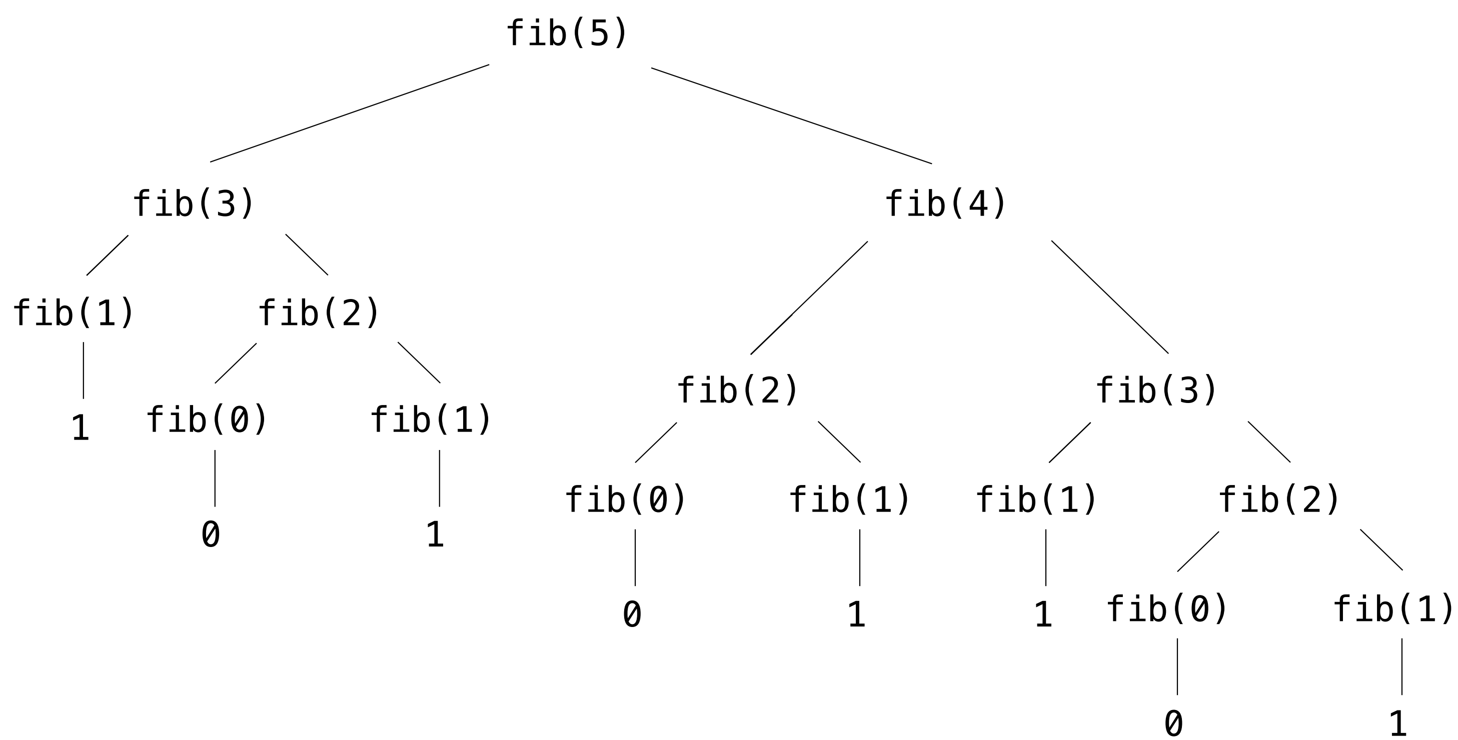
Fibonacci Space Consumption



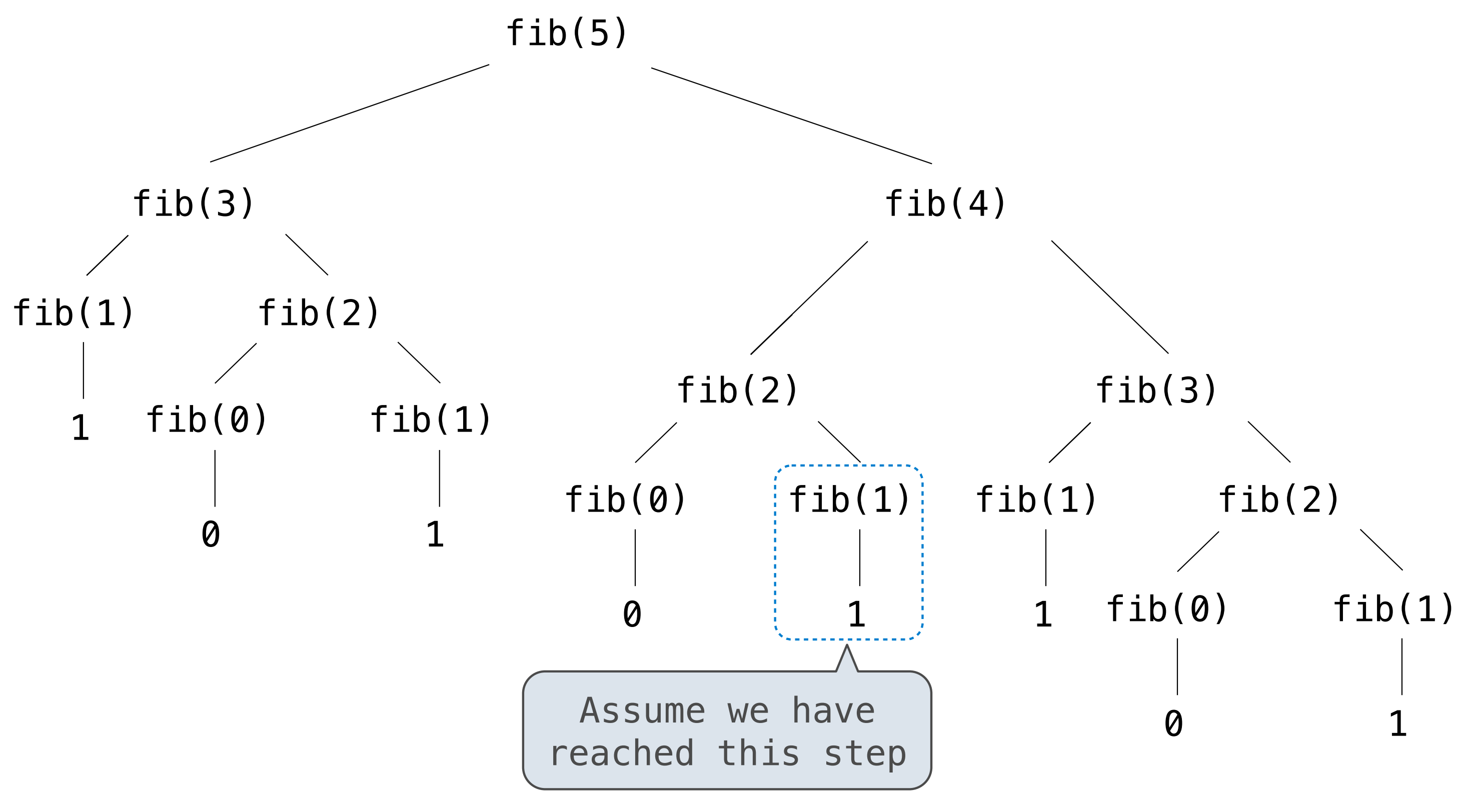
Fibonacci Space Consumption



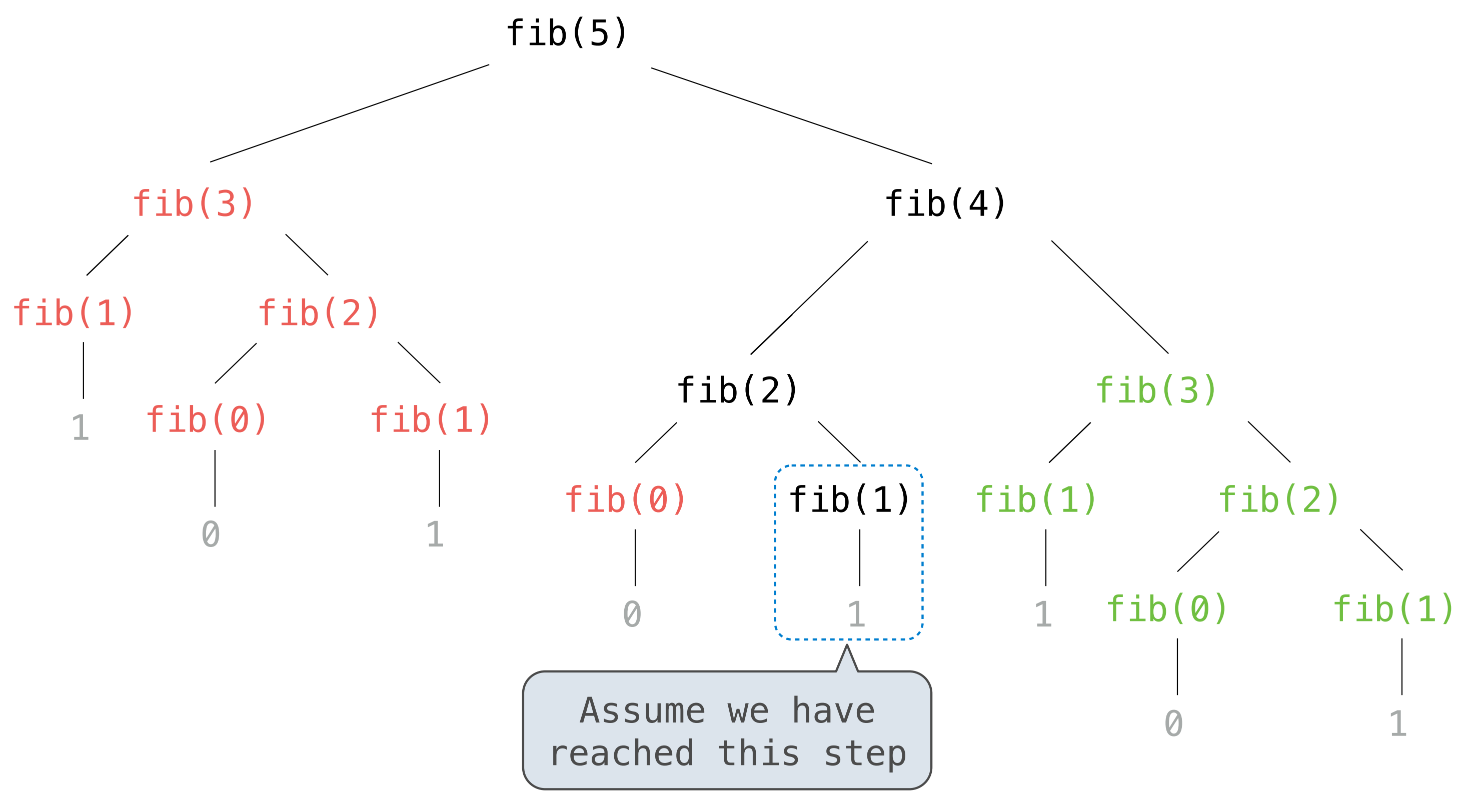
Fibonacci Space Consumption



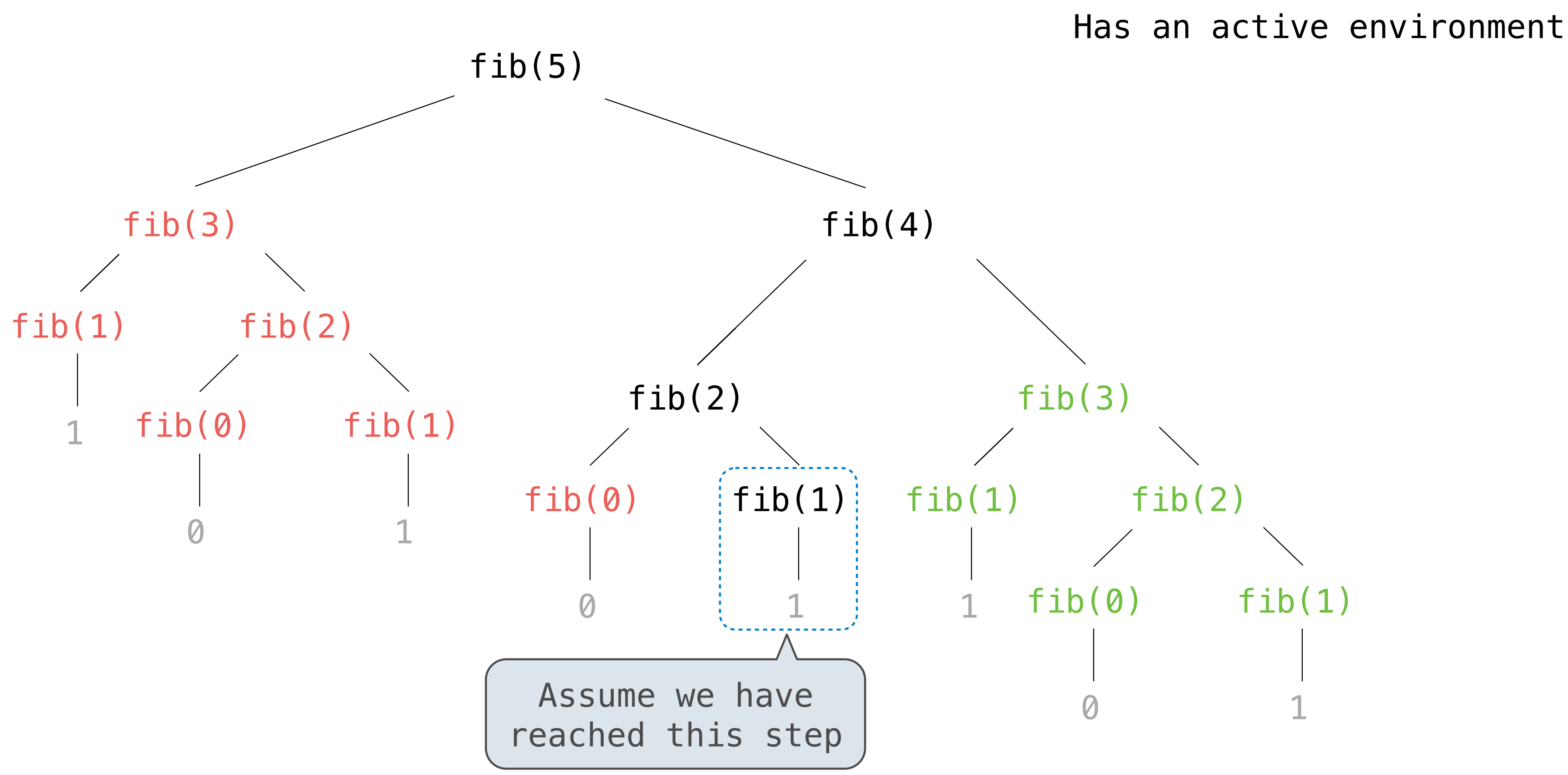
Fibonacci Space Consumption



Fibonacci Space Consumption



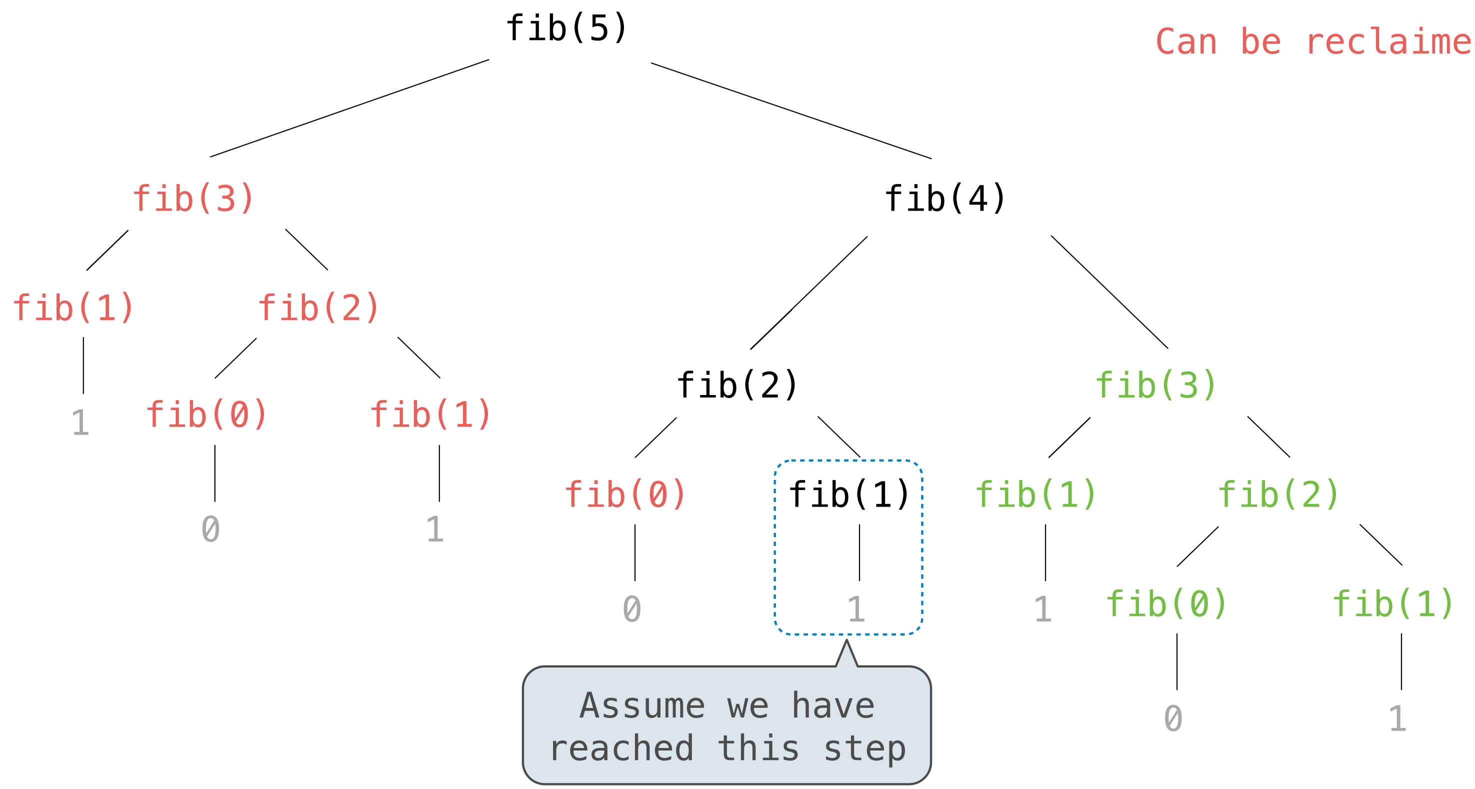
Fibonacci Space Consumption



Fibonacci Space Consumption

Has an active environment

Can be reclaimed



Fibonacci Space Consumption