61A Lecture 20

Friday, October 17

Announcements

Announcements

- Guerrilla Section 4 on Sunday 10/19: Object-oriented programming and recursive data

Announcements

- Guerrilla Section 4 on Sunday 10/19: Object-oriented programming and recursive data
  - Meet in 271 Soda: Vanguard section from 12-2pm; Main section from 2:30-4:30pm

Announcements

- Guerrilla Section 4 on Sunday 10/19: Object-oriented programming and recursive data
  - Meet in 271 Soda: Vanguard section from 12-2pm; Main section from 2:30-4:30pm
- Homework 6 is due Monday 10/20 @ 11:59pm

Announcements

- Guerrilla Section 4 on Sunday 10/19: Object-oriented programming and recursive data
  - Meet in 271 Soda: Vanguard section from 12–2pm; Main section from 2:30–4:30pm
- Homework 6 is due Monday 10/20 @ 11:59pm
  - Homework party on Monday 10/20 6pm–8pm in 2050 VLSB

Announcements

- Guerrilla Section 4 on Sunday 10/19: Object-oriented programming and recursive data
  - Meet in 271 Soda: Vanguard section from 12-2pm; Main section from 2:30-4:30pm
- Homework 6 is due Monday 10/20 @ 11:59pm
  - Homework party on Monday 10/20 6pm-8pm in 2050 VLSB
- Project 3 is due Thursday 10/23 @ 11:59pm

Announcements

- Guerrilla Section 4 on Sunday 10/19: Object-oriented programming and recursive data
  - Meet in 271 Soda: Vanguard section from 12-2pm; Main section from 2:30-4:30pm
- Homework 6 is due Monday 10/20 @ 11:59pm
  - Homework party on Monday 10/20 6pm-8pm in 2050 VLSB
- Project 3 is due Thursday 10/23 @ 11:59pm
- Midterm 2 is on Monday 10/27 7pm-9pm

Announcements

- Guerrilla Section 4 on Sunday 10/19: Object-oriented programming and recursive data
  - Meet in 271 Soda: Vanguard section from 12-2pm; Main section from 2:30-4:30pm
- Homework 6 is due Monday 10/20 @ 11:59pm
  - Homework party on Monday 10/20 6pm-8pm in 2050 VLSB
- Project 3 is due Thursday 10/23 @ 11:59pm
- Midterm 2 is on Monday 10/27 7pm-9pm
  - Class Conflict? Fill out the conflict form at the top of http://cs61a.org

Announcements

- Guerrilla Section 4 on Sunday 10/19: Object-oriented programming and recursive data
  - Meet in 271 Soda: Vanguard section from 12-2pm; Main section from 2:30-4:30pm
- Homework 6 is due Monday 10/20 @ 11:59pm
  - Homework party on Monday 10/20 6pm-8pm in 2050 VLSB
- Project 3 is due Thursday 10/23 @ 11:59pm
- Midterm 2 is on Monday 10/27 7pm-9pm
  - Class Conflict? Fill out the conflict form at the top of http://cs61a.org
  - Review session on Saturday 10/25 3pm-6pm in 2050 VLSB

Announcements

- Guerrilla Section 4 on Sunday 10/19: Object-oriented programming and recursive data
  - Meet in 271 Soda: Vanguard section from 12-2pm; Main section from 2:30-4:30pm
- Homework 6 is due Monday 10/20 @ 11:59pm
  - Homework party on Monday 10/20 6pm-8pm in 2050 VLSB
- Project 3 is due Thursday 10/23 @ 11:59pm
- Midterm 2 is on Monday 10/27 7pm-9pm
  - Class Conflict? Fill out the conflict form at the top of http://cs61a.org
  - Review session on Saturday 10/25 3pm-6pm in 2050 VLSB
- CSUA and Hackers@Berkeley are holding a hack-a-thon on Saturday for 61A students

Announcements

- Guerrilla Section 4 on Sunday 10/19: Object-oriented programming and recursive data
  - Meet in 271 Soda: Vanguard section from 12-2pm; Main section from 2:30-4:30pm
- Homework 6 is due Monday 10/20 @ 11:59pm
  - Homework party on Monday 10/20 6pm-8pm in 2050 VLSB
- Project 3 is due Thursday 10/23 @ 11:59pm
- Midterm 2 is on Monday 10/27 7pm-9pm
  - Class Conflict? Fill out the conflict form at the top of http://cs61a.org
  - Review session on Saturday 10/25 3pm-6pm in 2050 VLSB
- CSUA and Hackers@Berkeley are holding a hack-a-thon on Saturday for 61A students
  - 10am - 11pm in Wozniak Lounge

Introducing Cohorts

Introducing Cohorts

```
Each of you has been randomly placed in the cohort of a patron computer scientist
```

Introducing Cohorts

Each of you has been randomly placed in the cohort of a patron computer scientist

00: Ada Lovelace

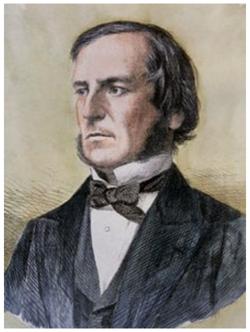
Wrote first program

Introducing Cohorts

Each of you has been randomly placed in the cohort of a patron computer scientist

00: Ada Lovelace

Wrote first program



10: George Boole

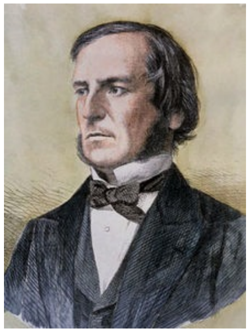
Invented boolean logic

Introducing Cohorts

Each of you has been randomly placed in the cohort of a patron computer scientist

00: Ada Lovelace

Wrote first program



10: George Boole

Invented boolean logic

01: Haskell Curry

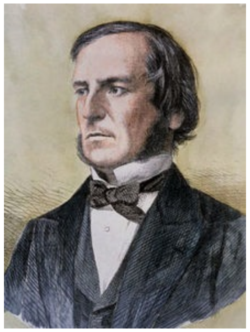
Math for functional programming

Introducing Cohorts

Each of you has been randomly placed in the cohort of a patron computer scientist

00: Ada Lovelace

Wrote first program



10: George Boole

Invented boolean logic

01: Haskell Curry

Math for functional programming

11: Grace Hopper

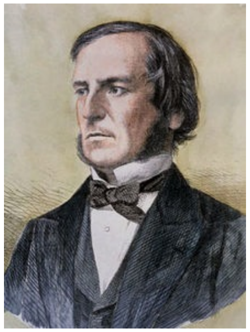
Wrote first compiler

Introducing Cohorts

Each of you has been randomly placed in the cohort of a patron computer scientist

00: Ada Lovelace

Wrote first program



10: George Boole

Invented boolean logic

01: Haskell Curry

Math for functional programming

11: Grace Hopper

Wrote first compiler

cs61a.org/cohorts.html

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:

fib(5)

fib(3)

```
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)

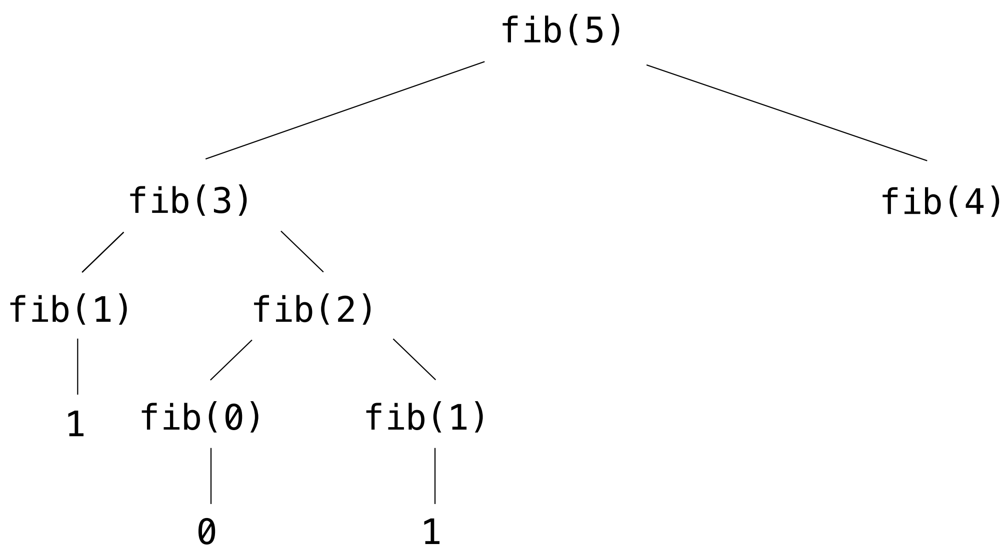
fib(3)   fib(4)

```
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:



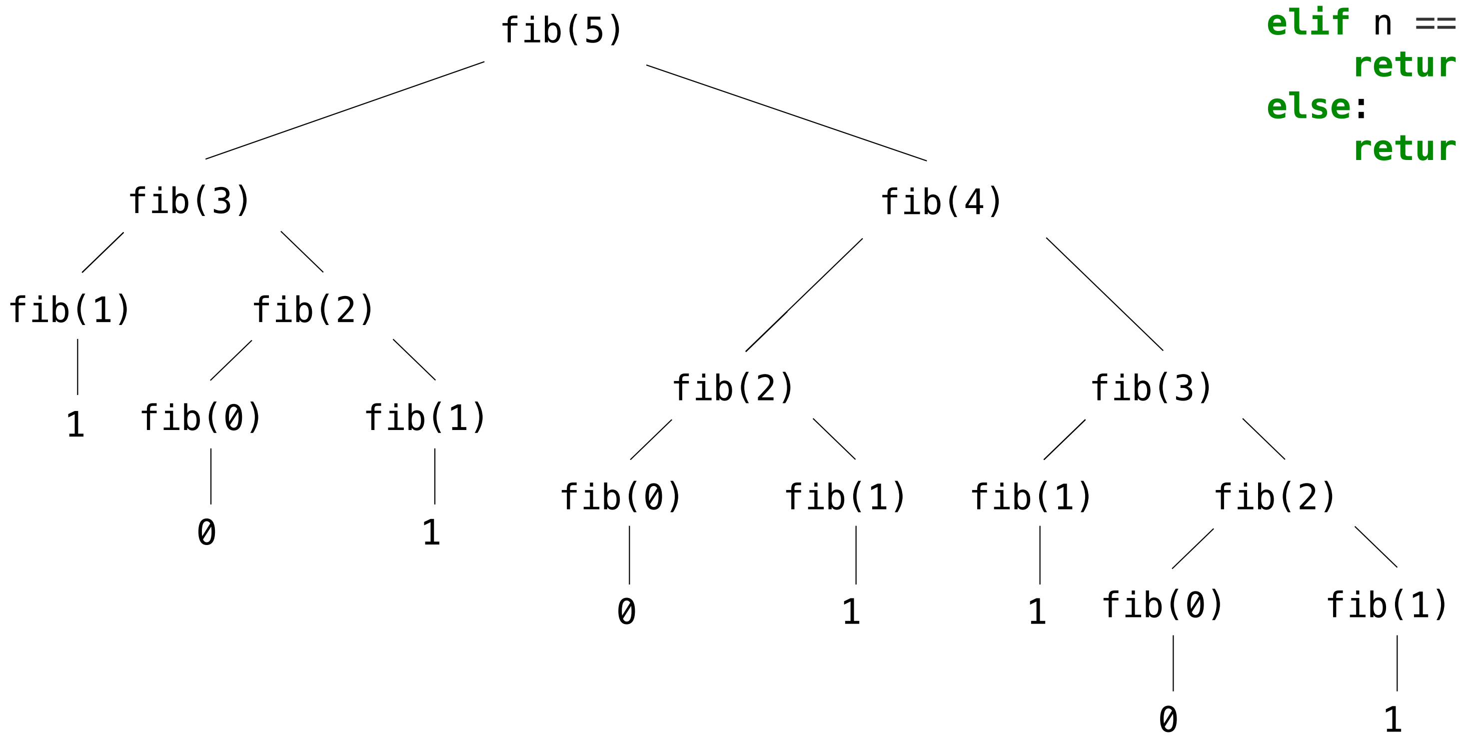

```python
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-2) + fib(n-1)
```

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)
```

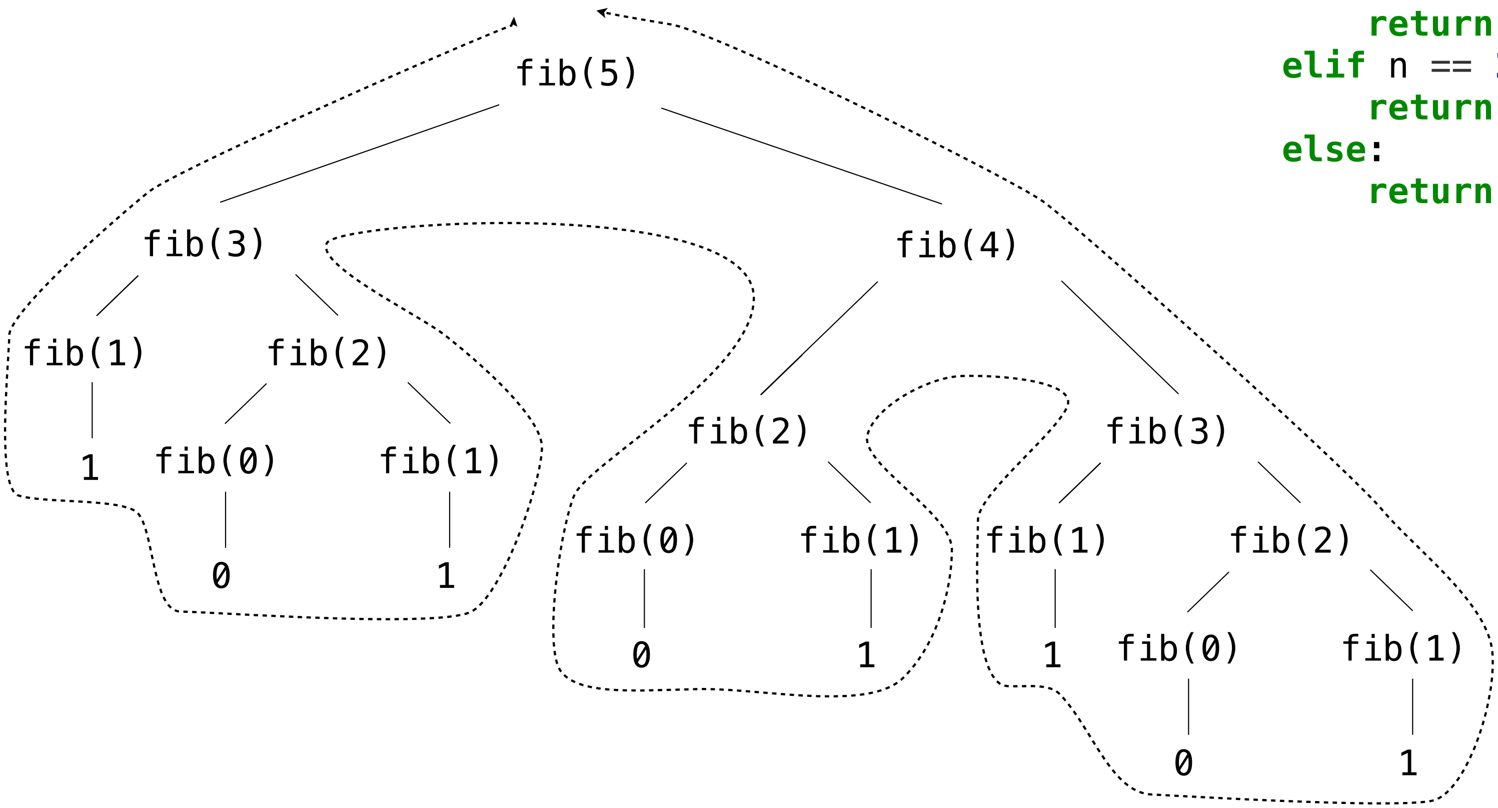


http://en.wikipedia.org/wiki/File:Fibonacci.jpg

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)
```

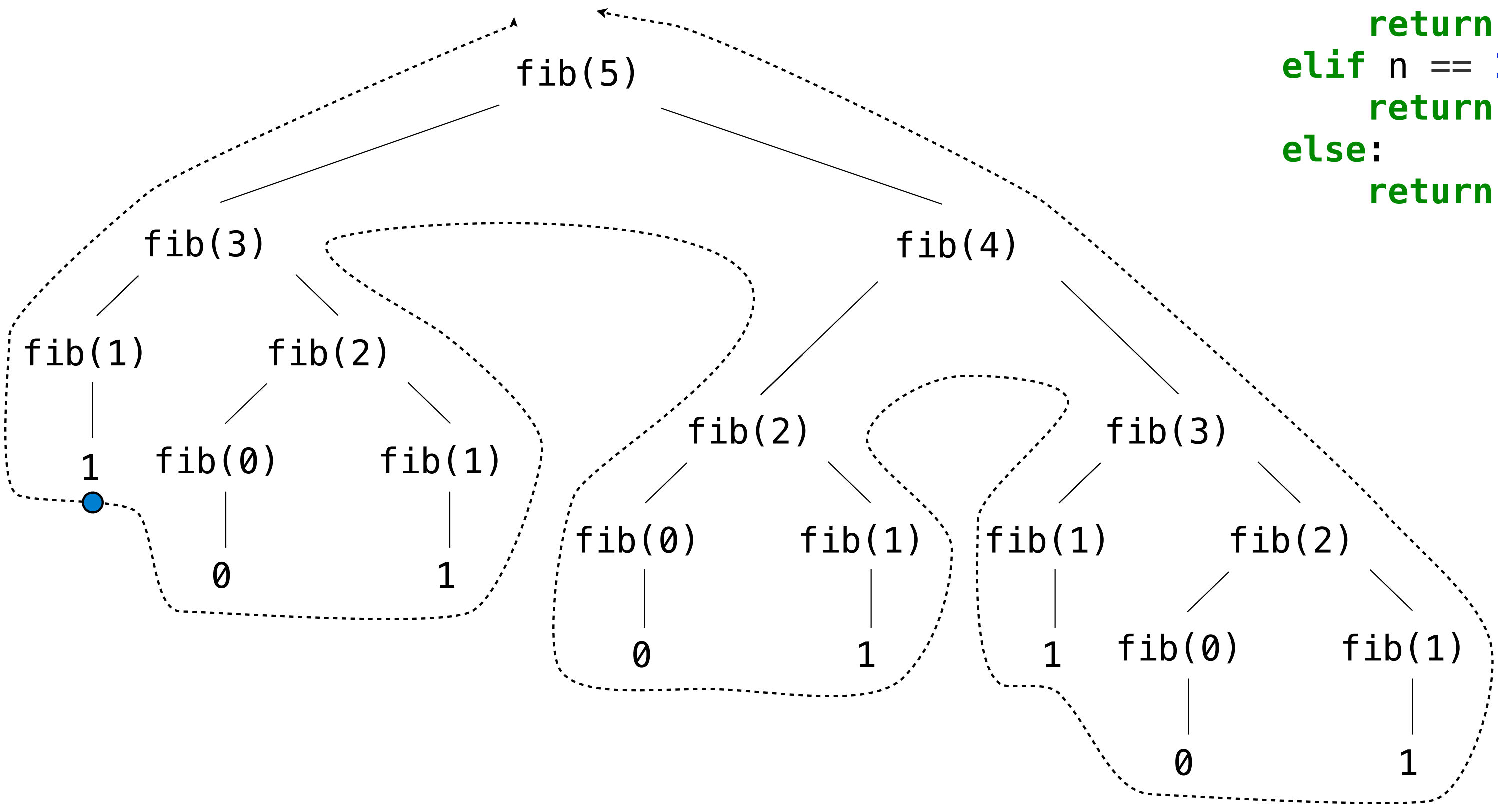


http://en.wikipedia.org/wiki/File:Fibonacci.jpg

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)
```

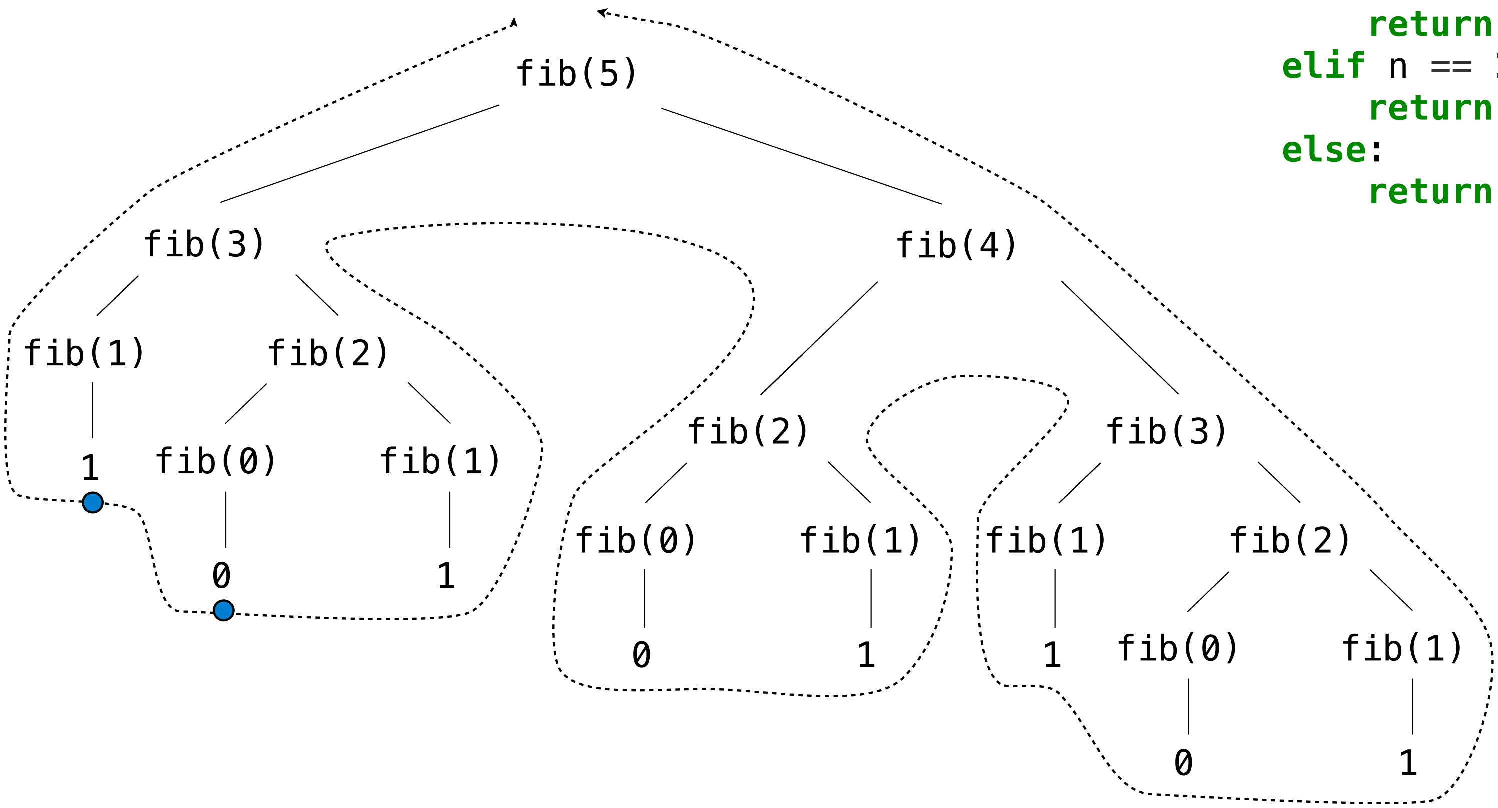


http://en.wikipedia.org/wiki/File:Fibonacci.jpg

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)
```

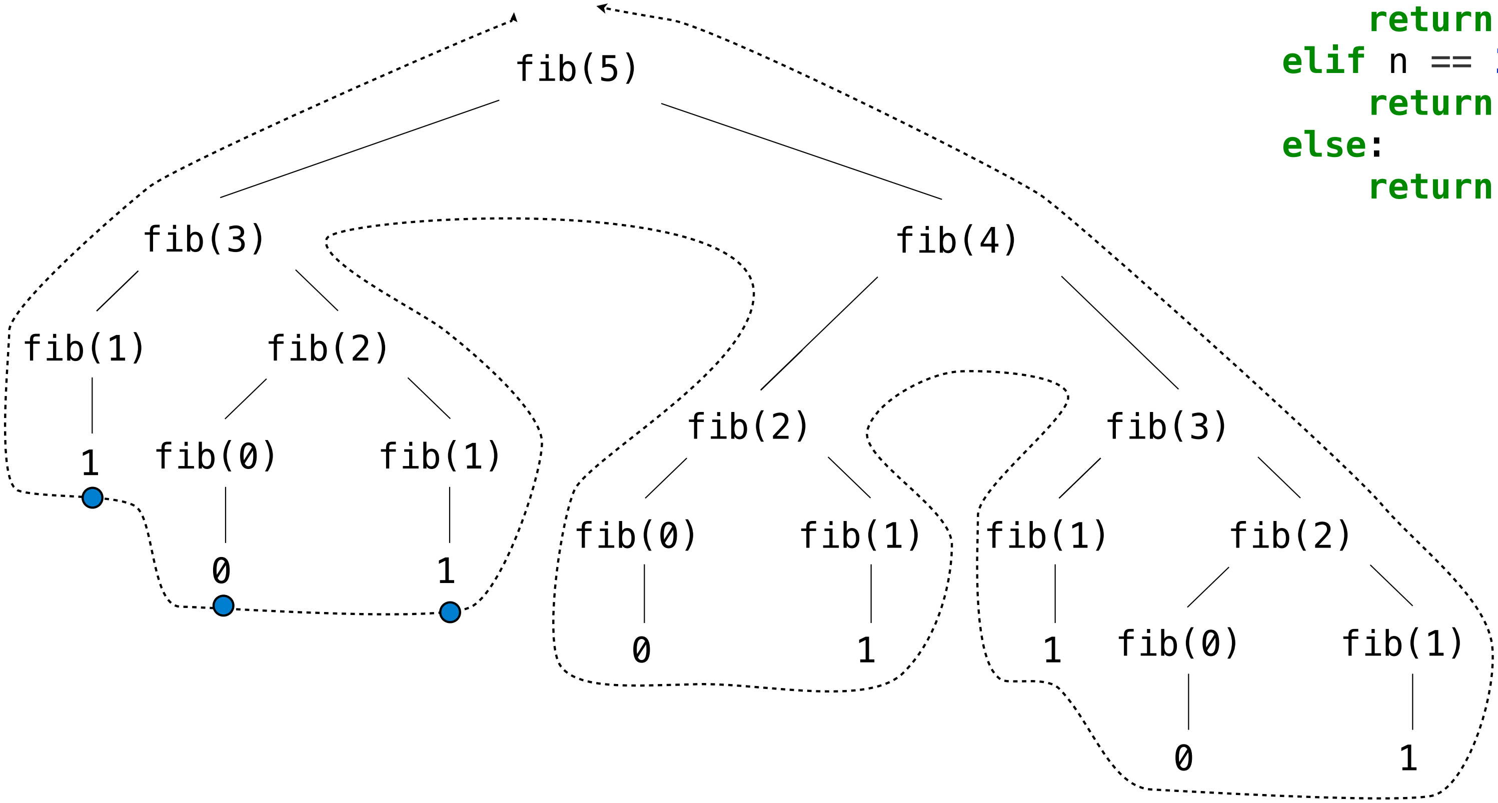


http://en.wikipedia.org/wiki/File:Fibonacci.jpg

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)
```

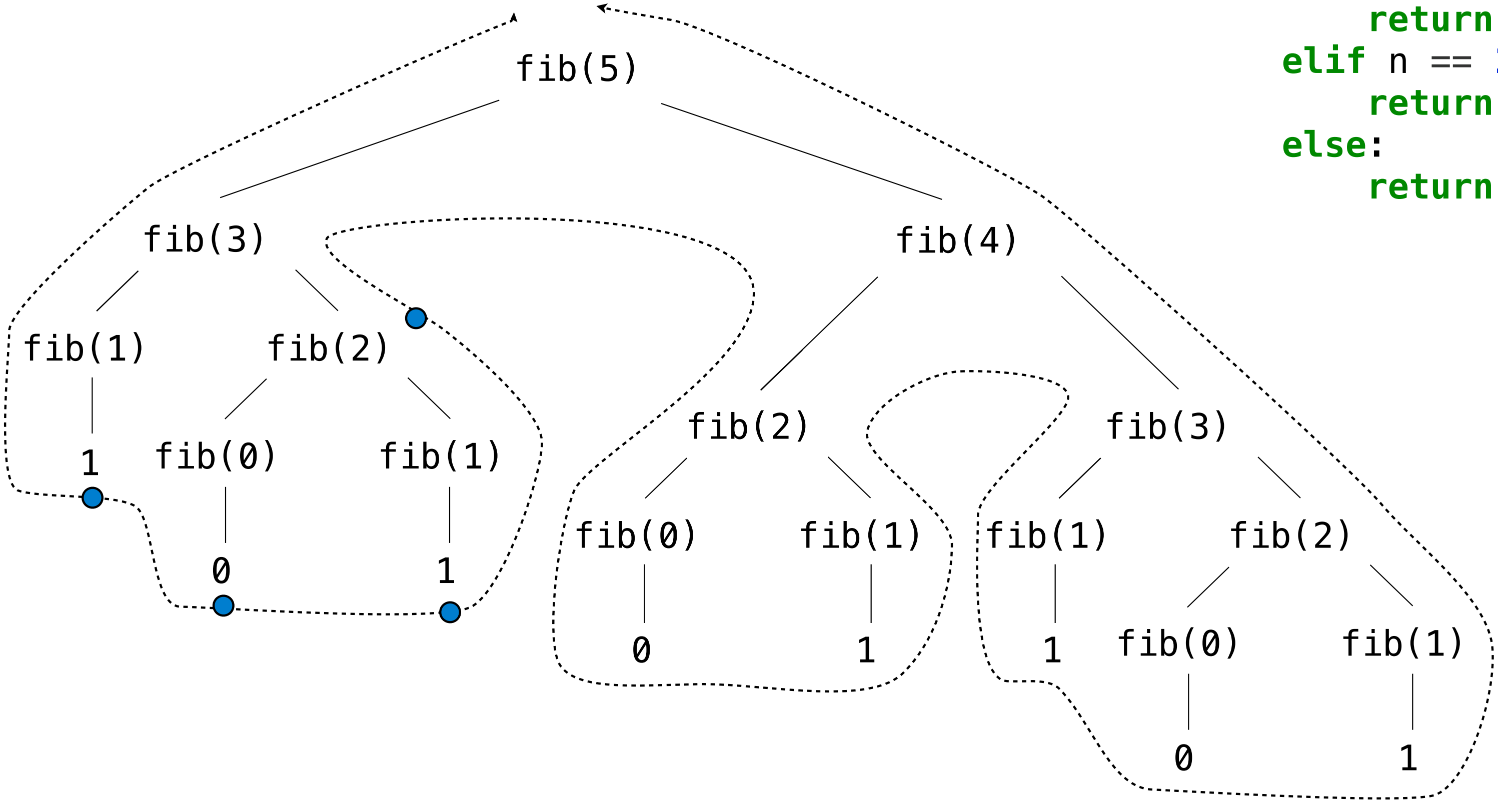


http://en.wikipedia.org/wiki/File:Fibonacci.jpg

Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:

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

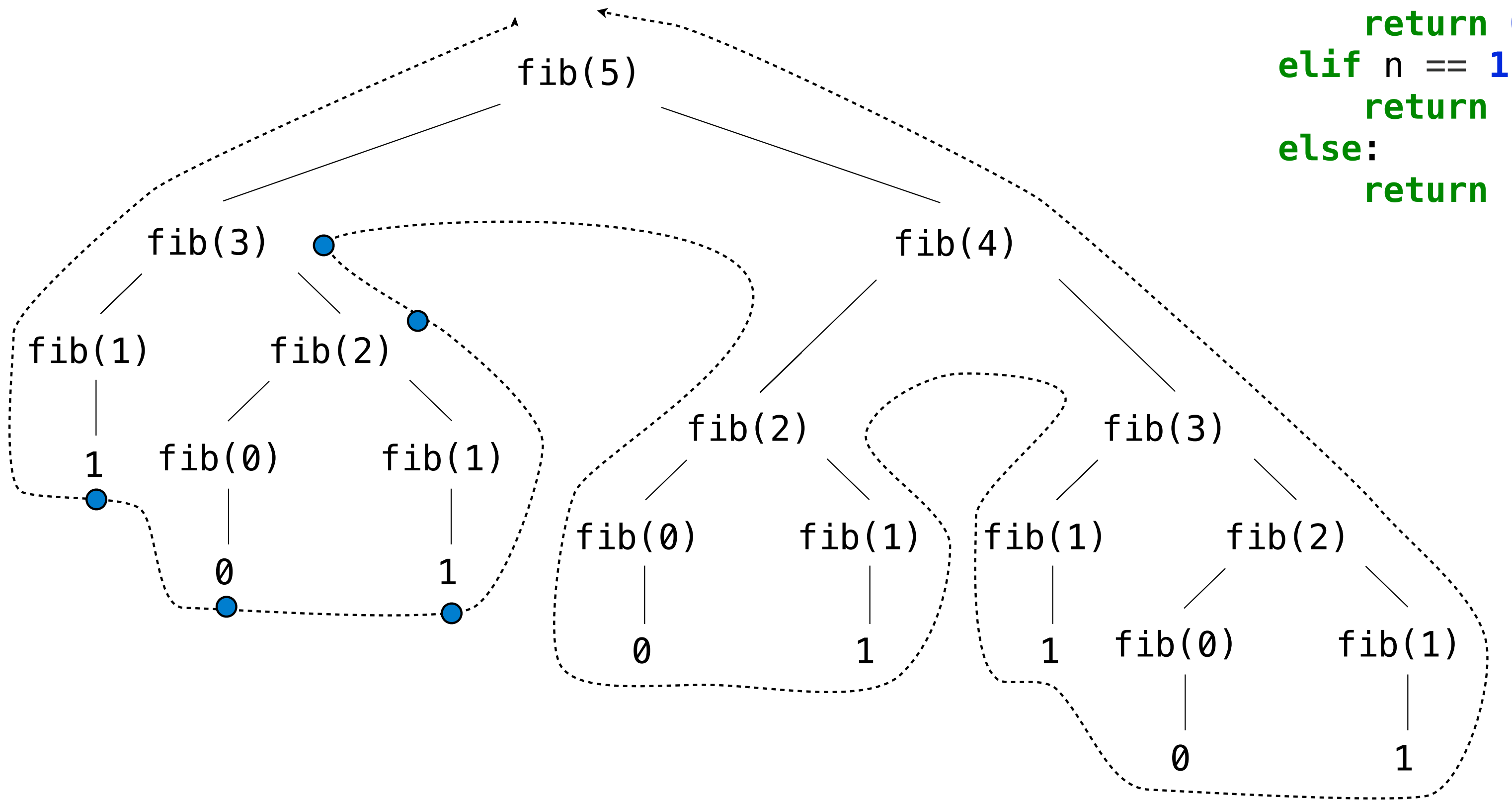


http://en.wikipedia.org/wiki/File:Fibonacci.jpg

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)
```

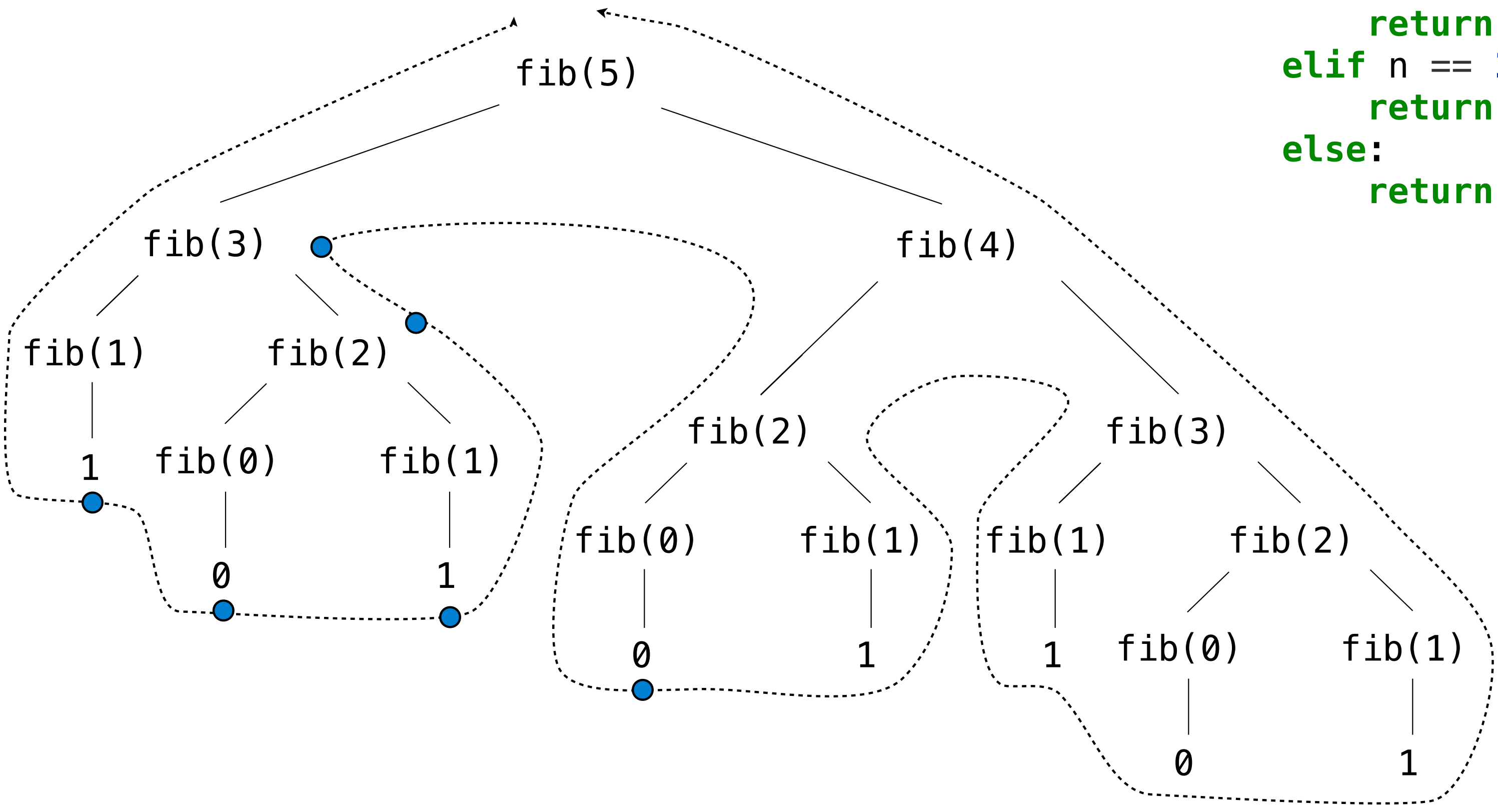


http://en.wikipedia.org/wiki/File:Fibonacci.jpg

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)
```

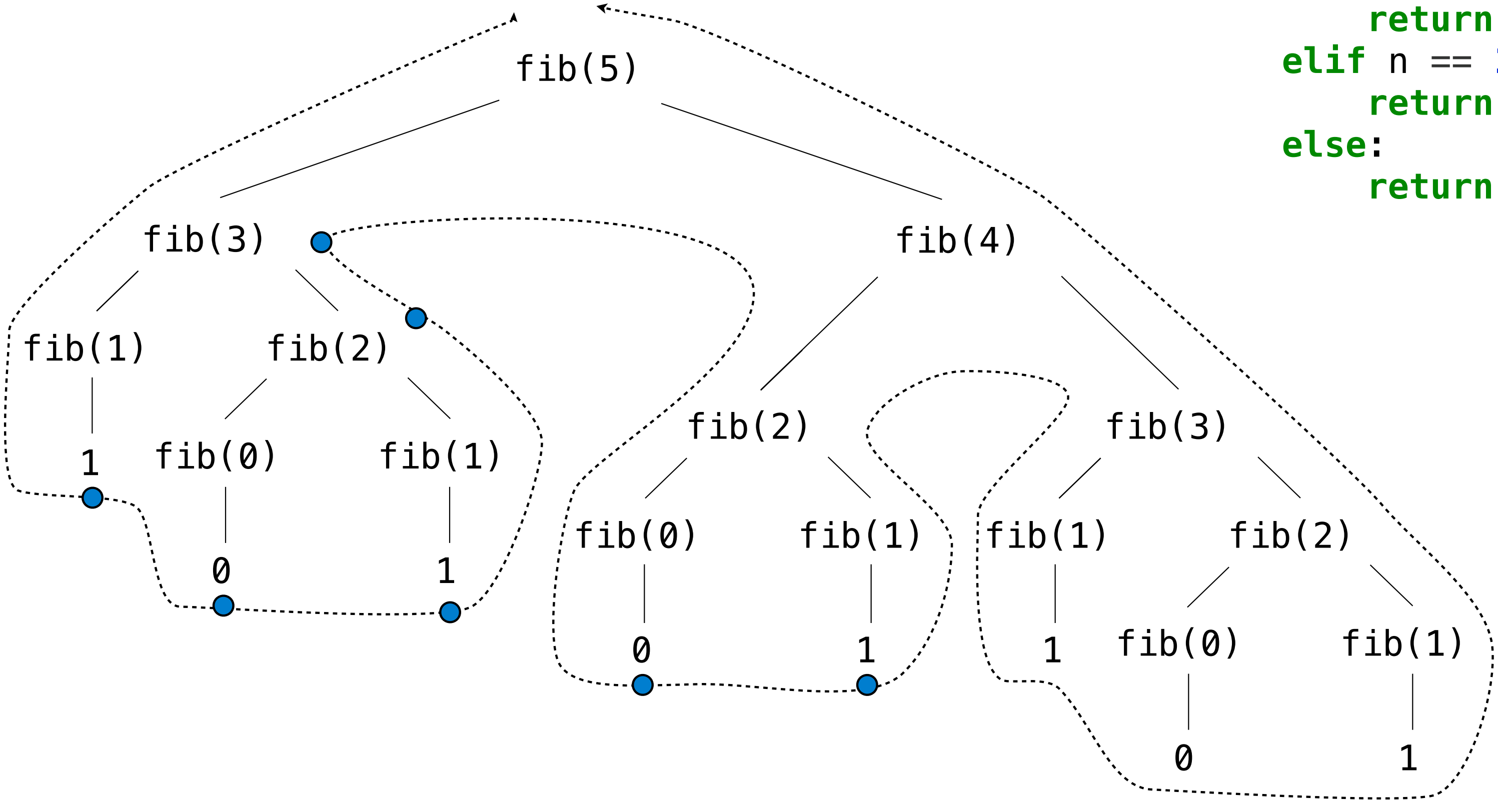


http://en.wikipedia.org/wiki/File:Fibonacci.jpg

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)
```

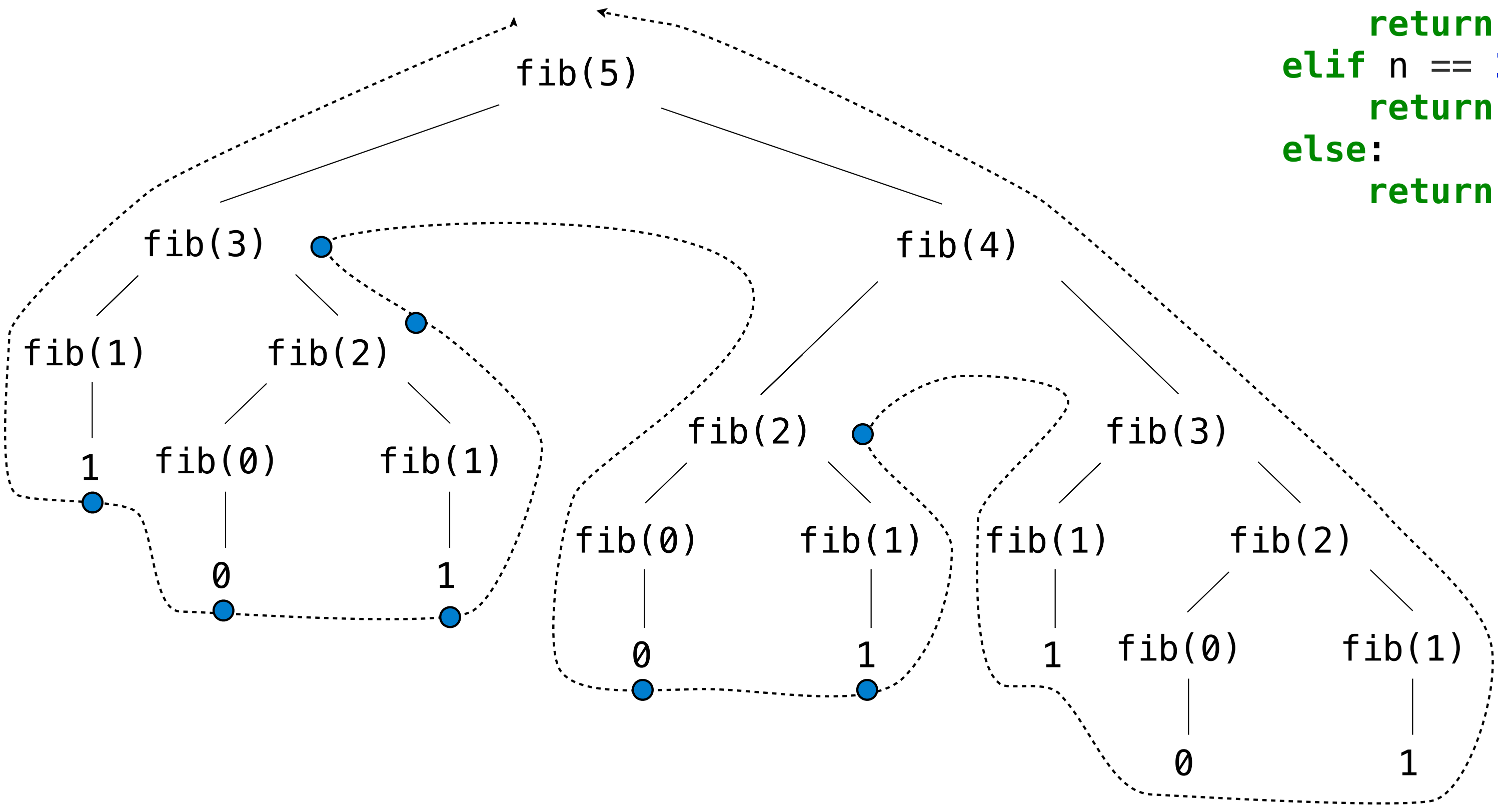


http://en.wikipedia.org/wiki/File:Fibonacci.jpg

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)
```

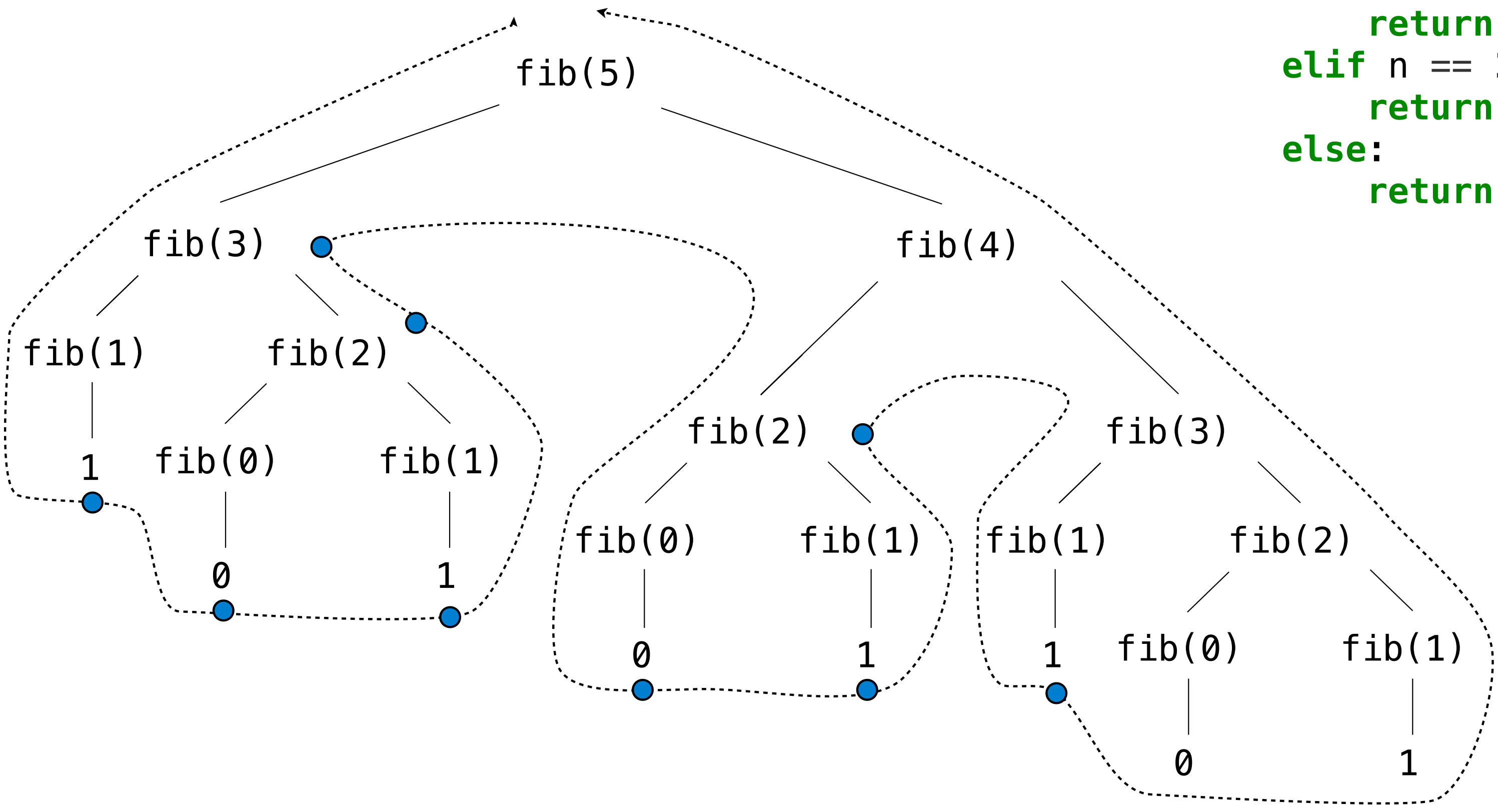


http://en.wikipedia.org/wiki/File:Fibonacci.jpg

Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:

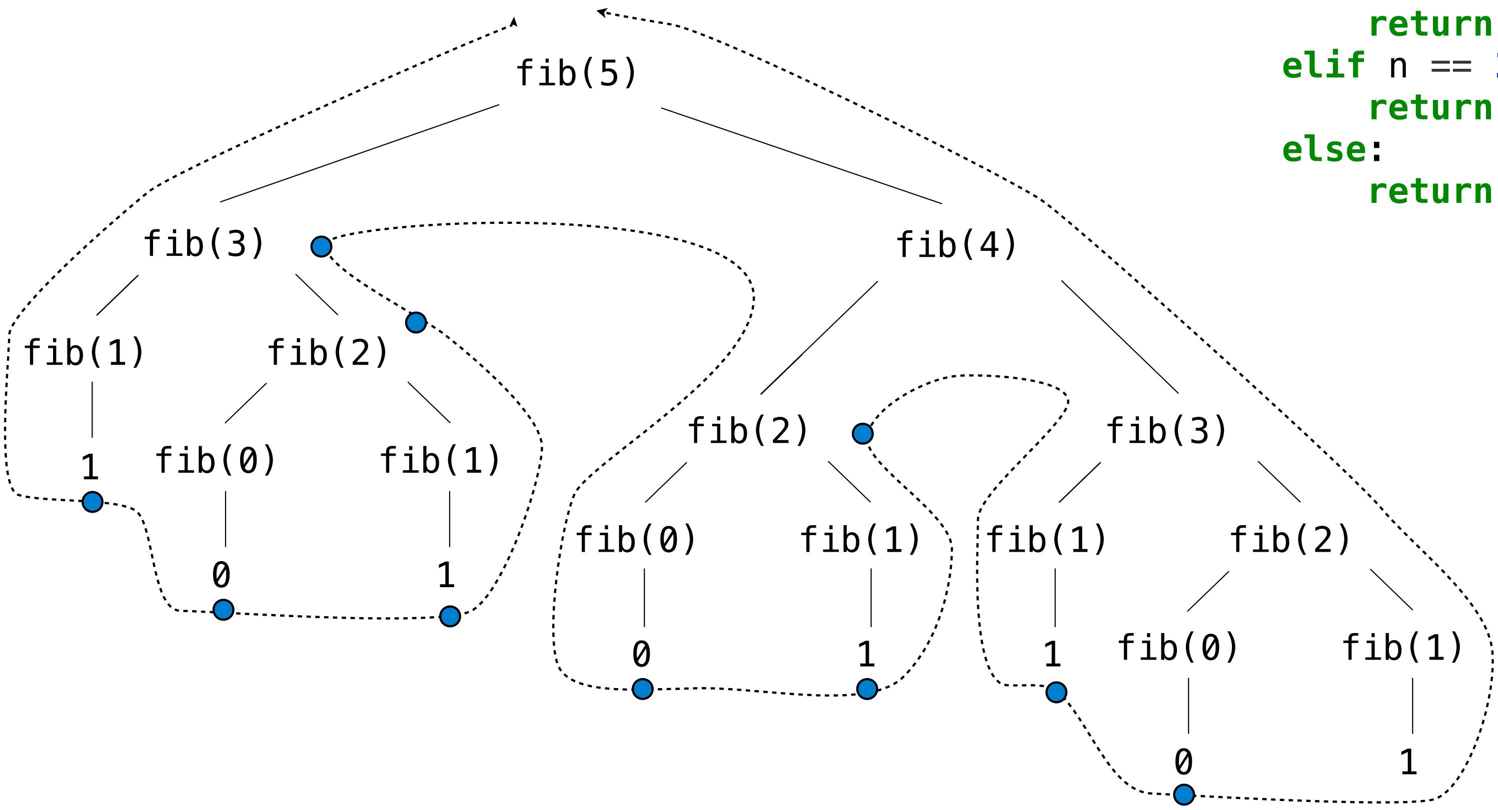
```python
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-2) + fib(n-1)
```



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

Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:



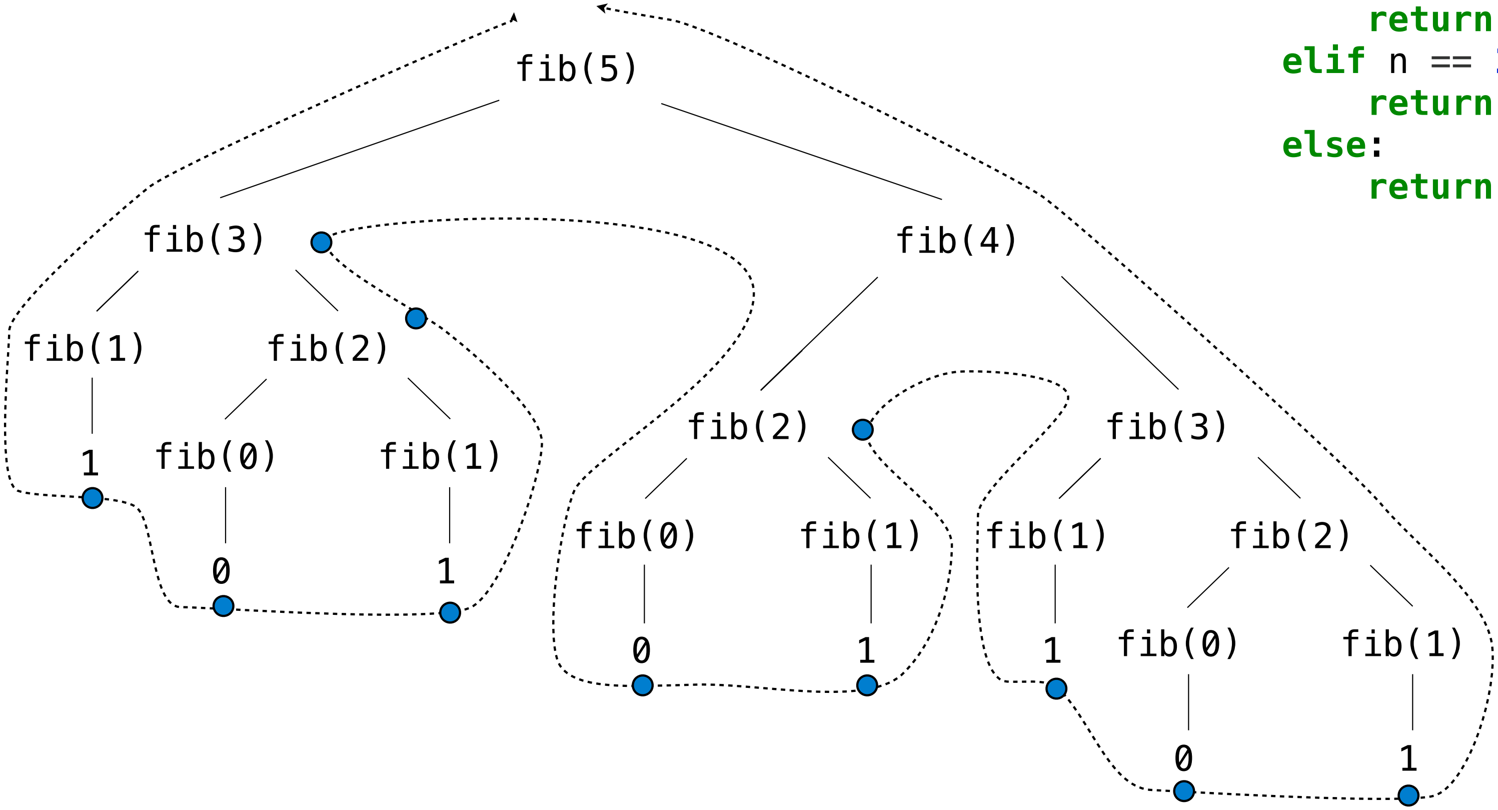

```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-2) + fib(n-1)
```

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)
```

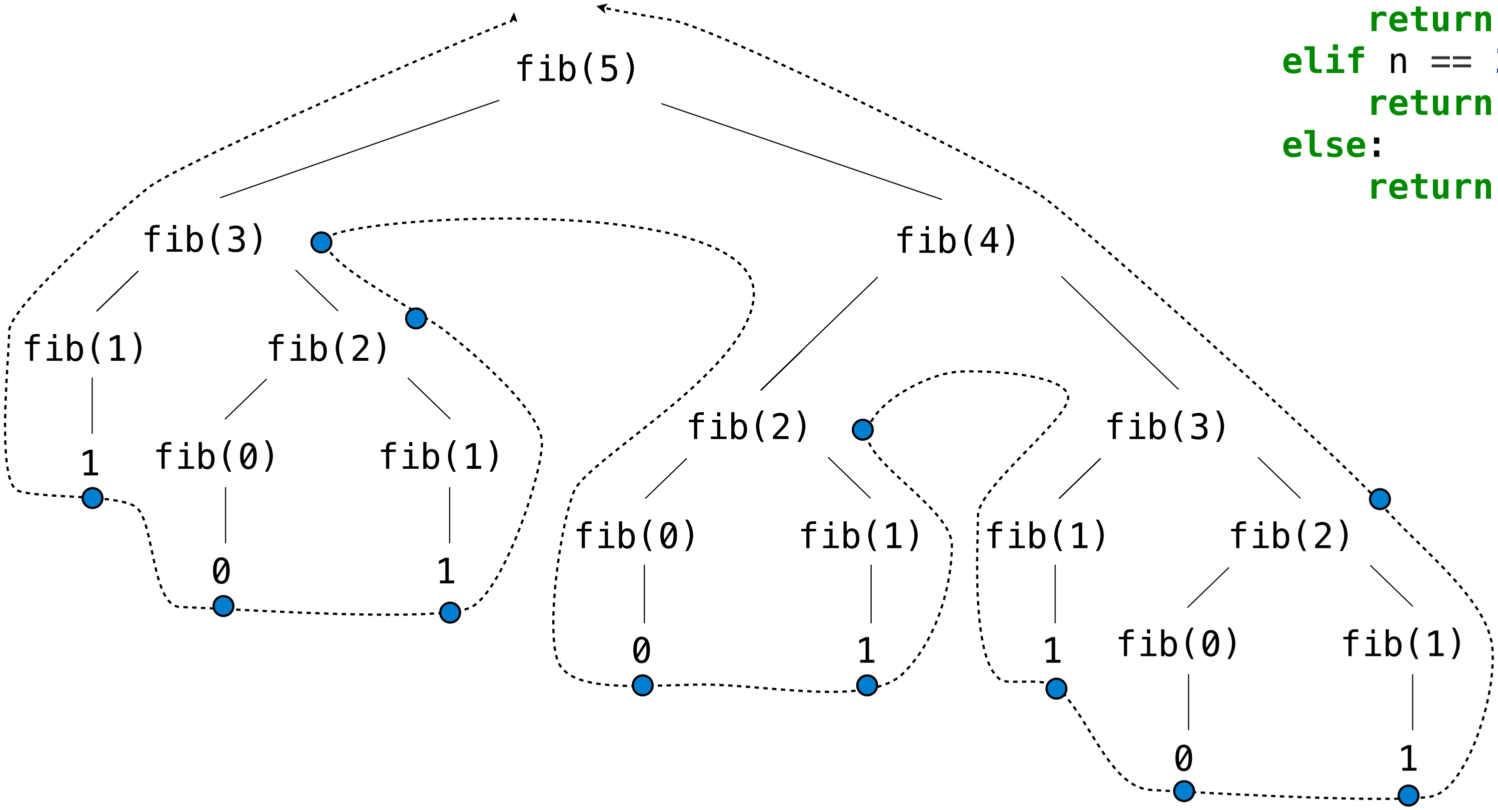


http://en.wikipedia.org/wiki/File:Fibonacci.jpg

Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:

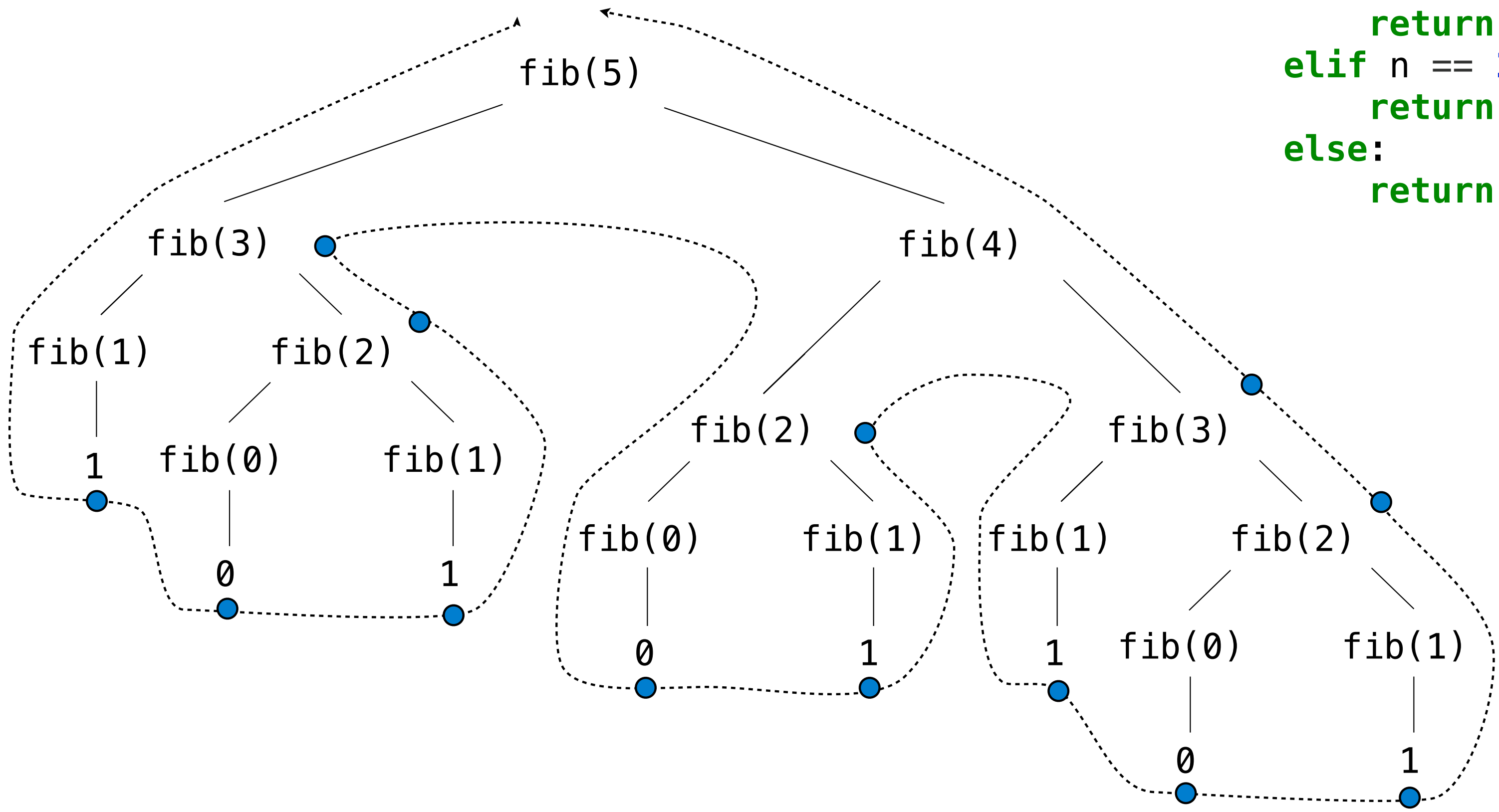
```python
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-2) + fib(n-1)
```



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

Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:



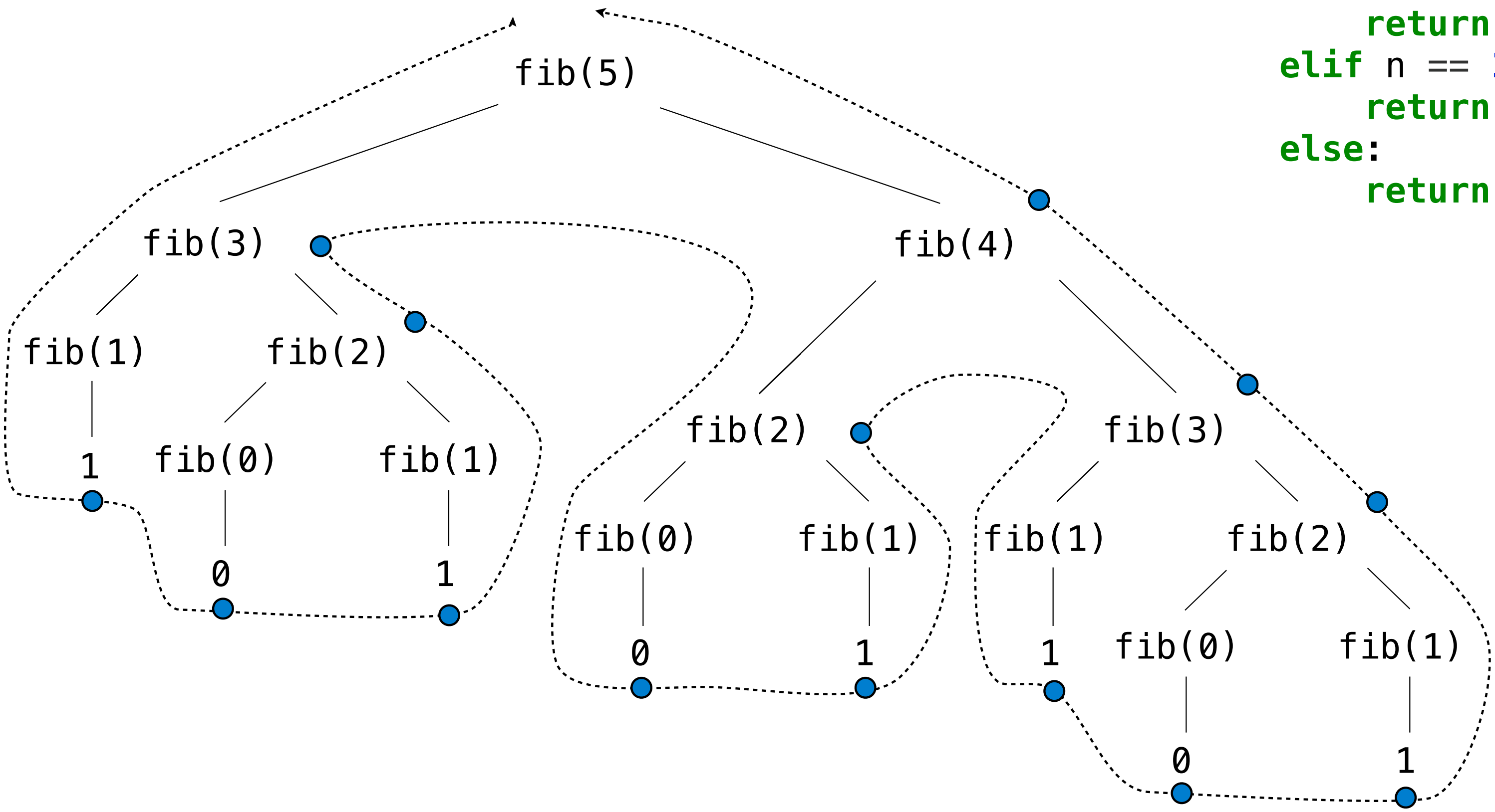

```python
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-2) + fib(n-1)
```

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)
```

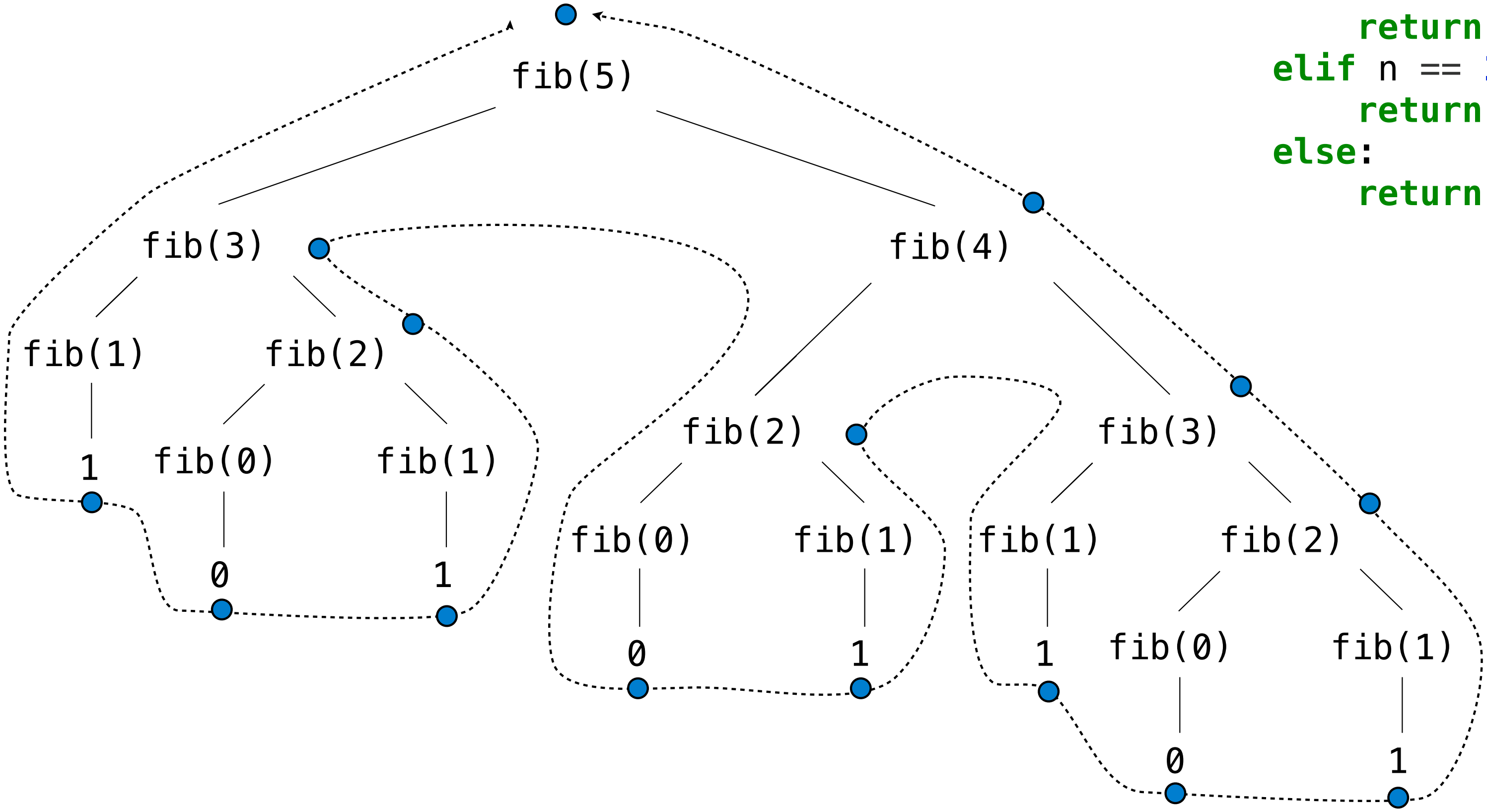


http://en.wikipedia.org/wiki/File:Fibonacci.jpg

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)
```

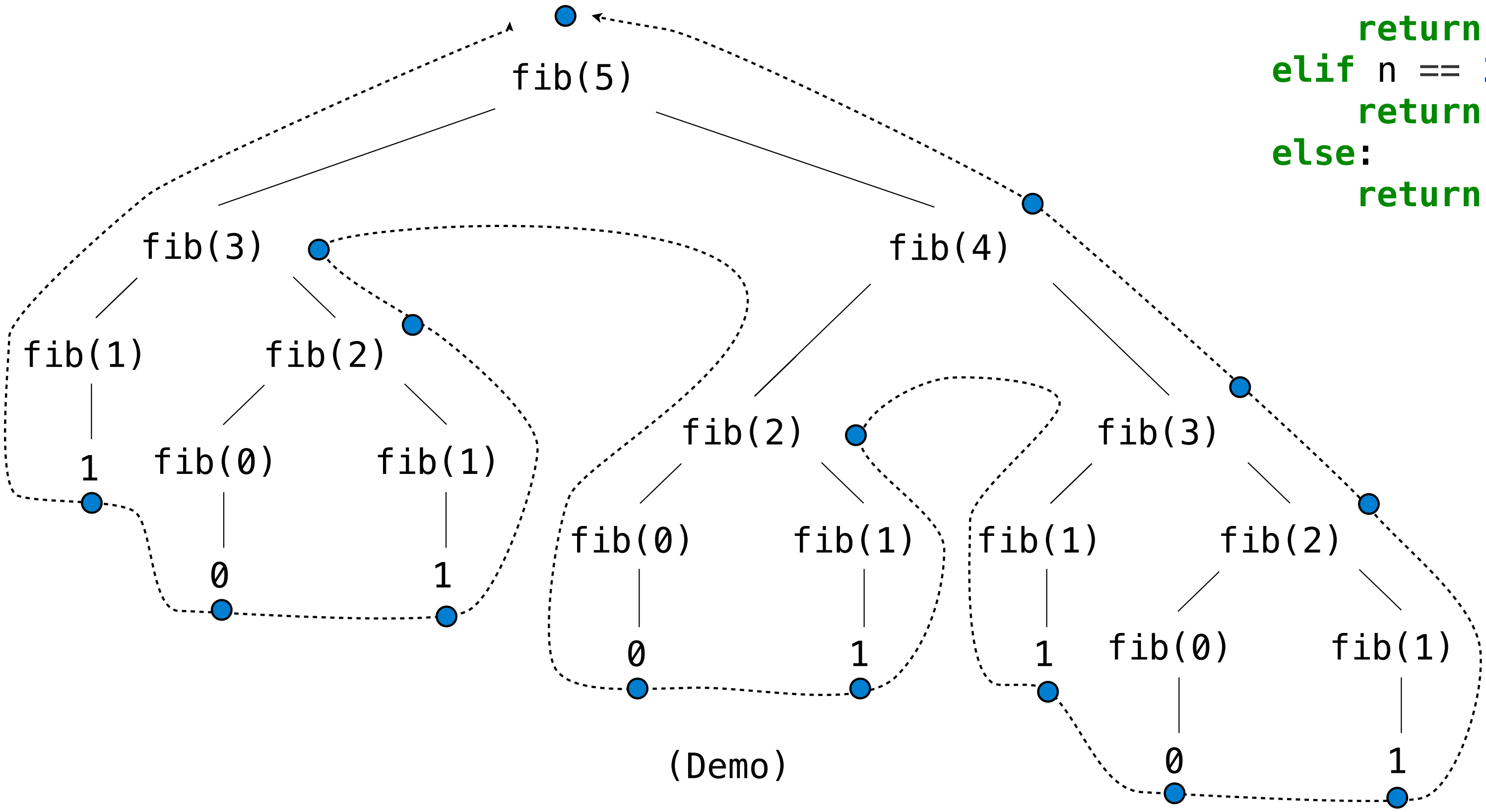


http://en.wikipedia.org/wiki/File:Fibonacci.jpg

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)
```



(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
```

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

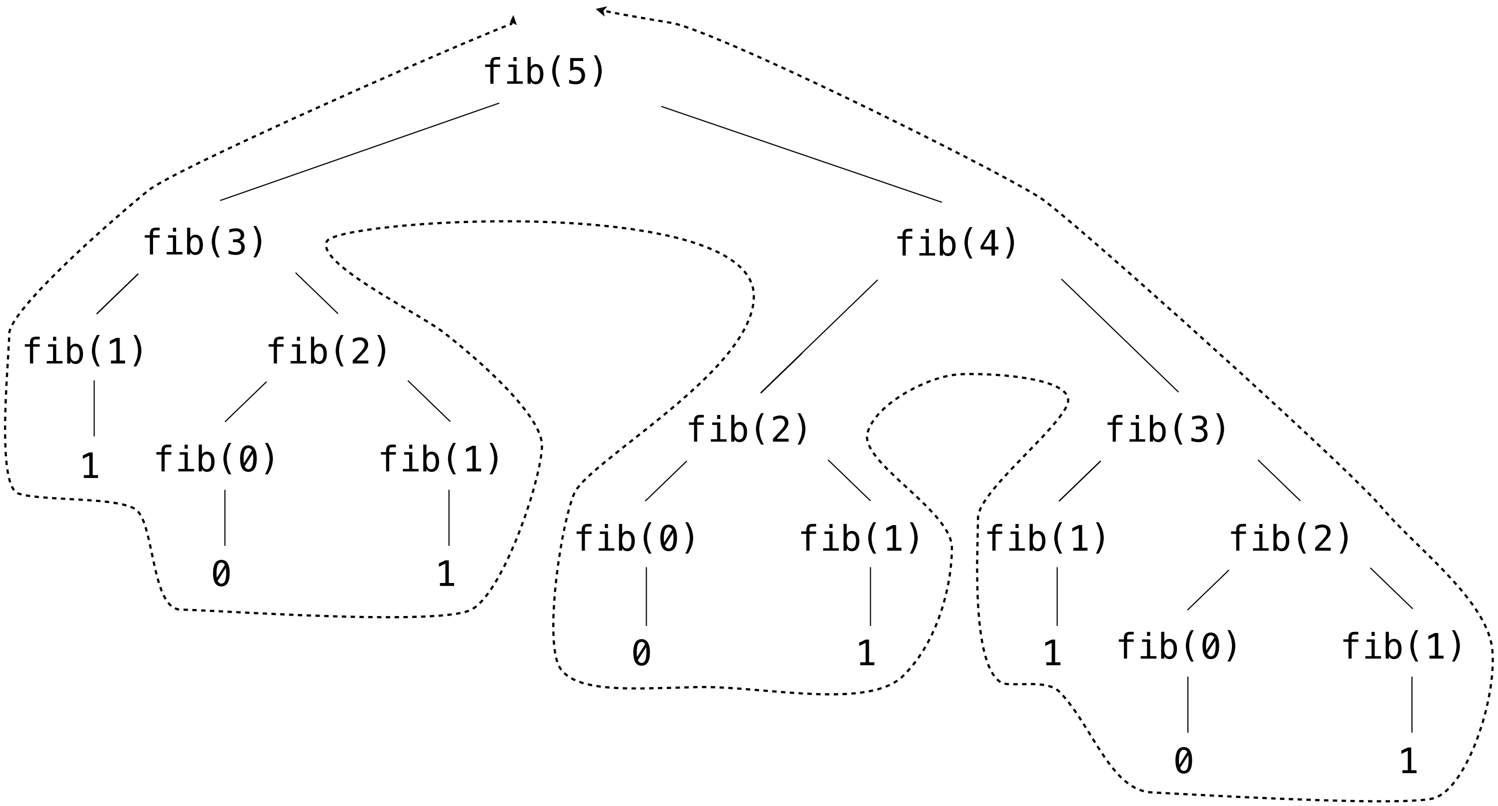
```
def memo(f):
    cache = {}
    def memoized(n):
        if n not in cache:
            cache[n] = f(n)
        return cache[n]
    return memoized
```

Keys are arguments that map to return values

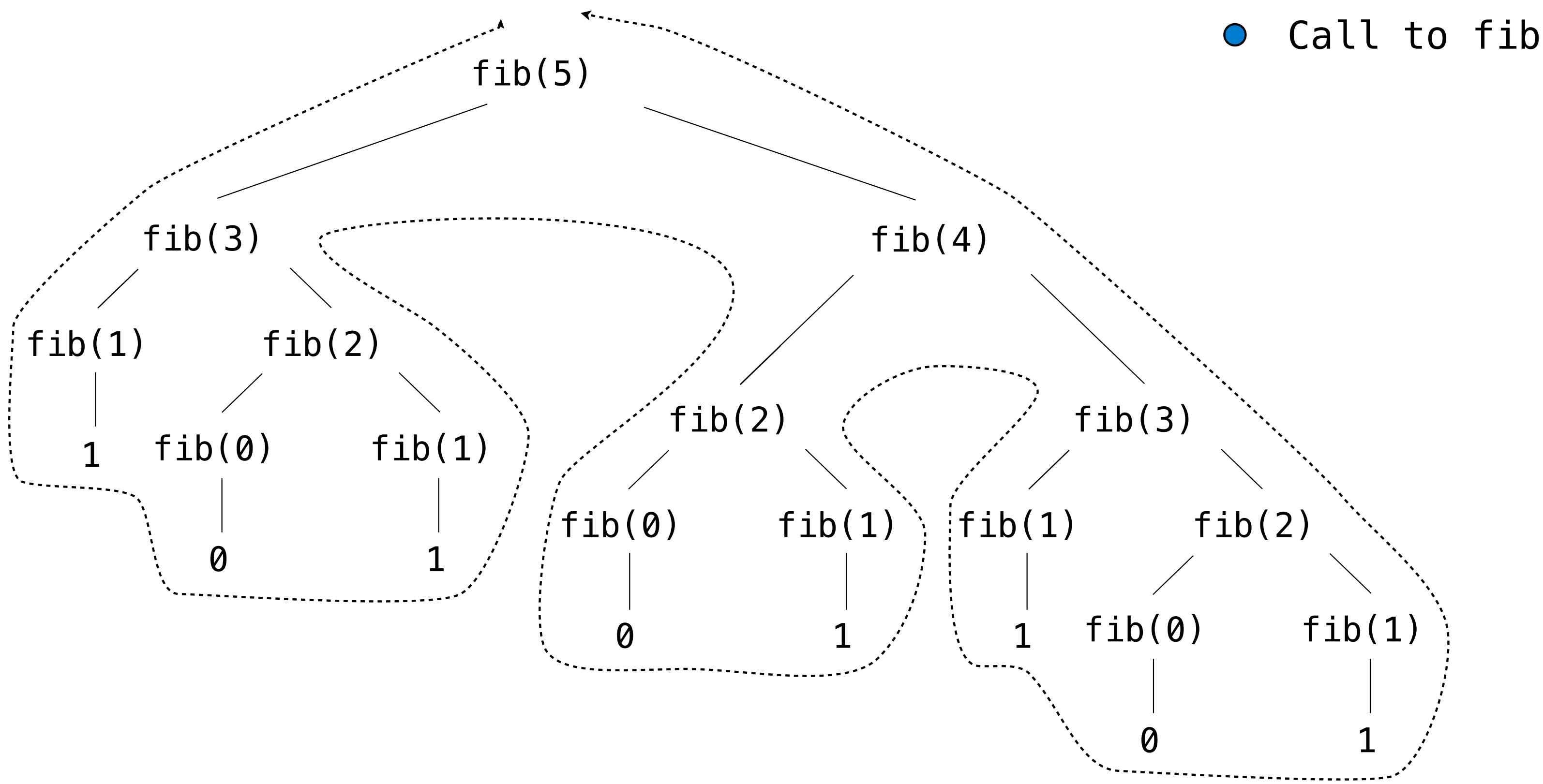
Same behavior as f, if f is a pure function

(Demo)

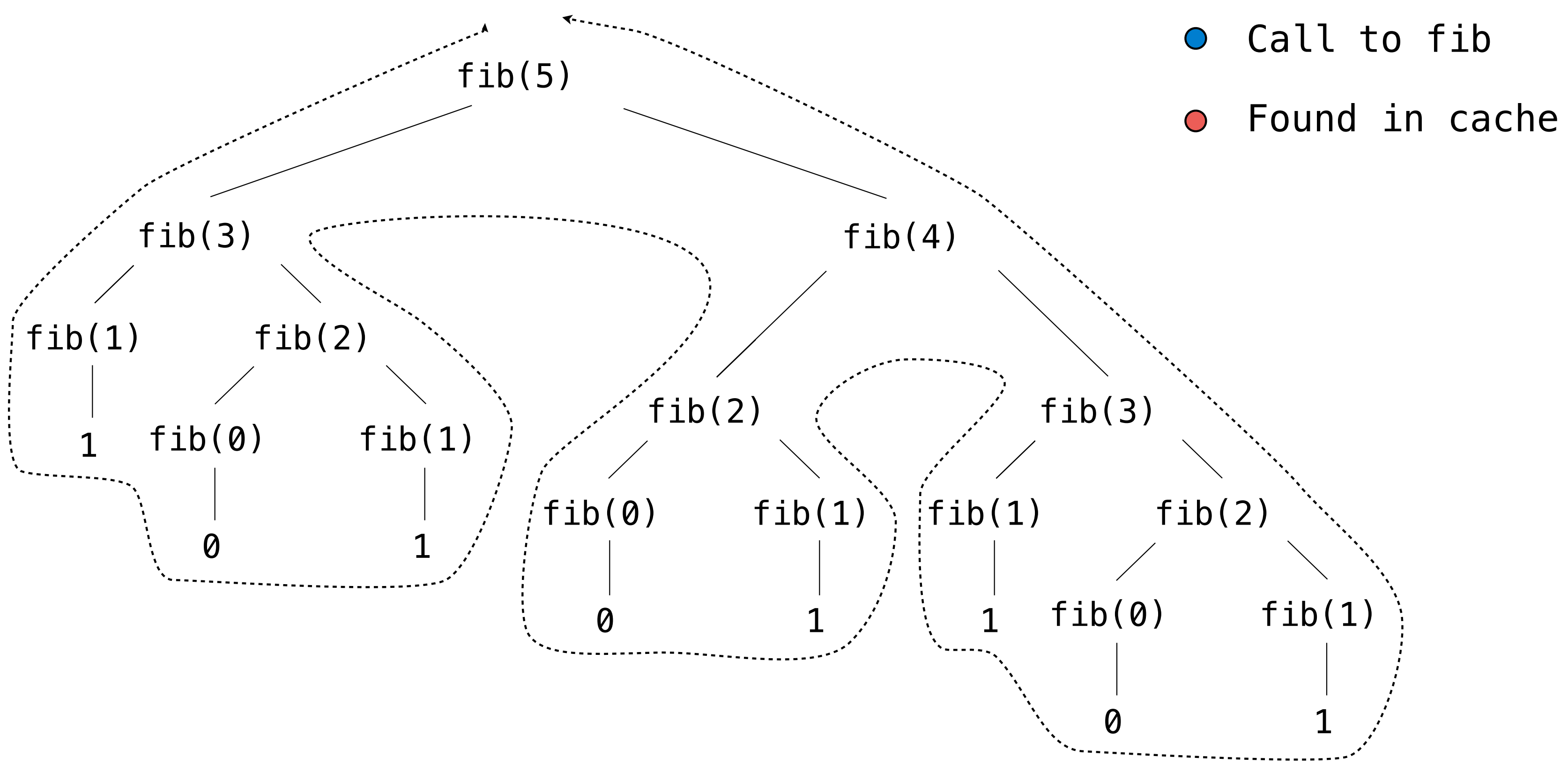
Memoized Tree Recursion



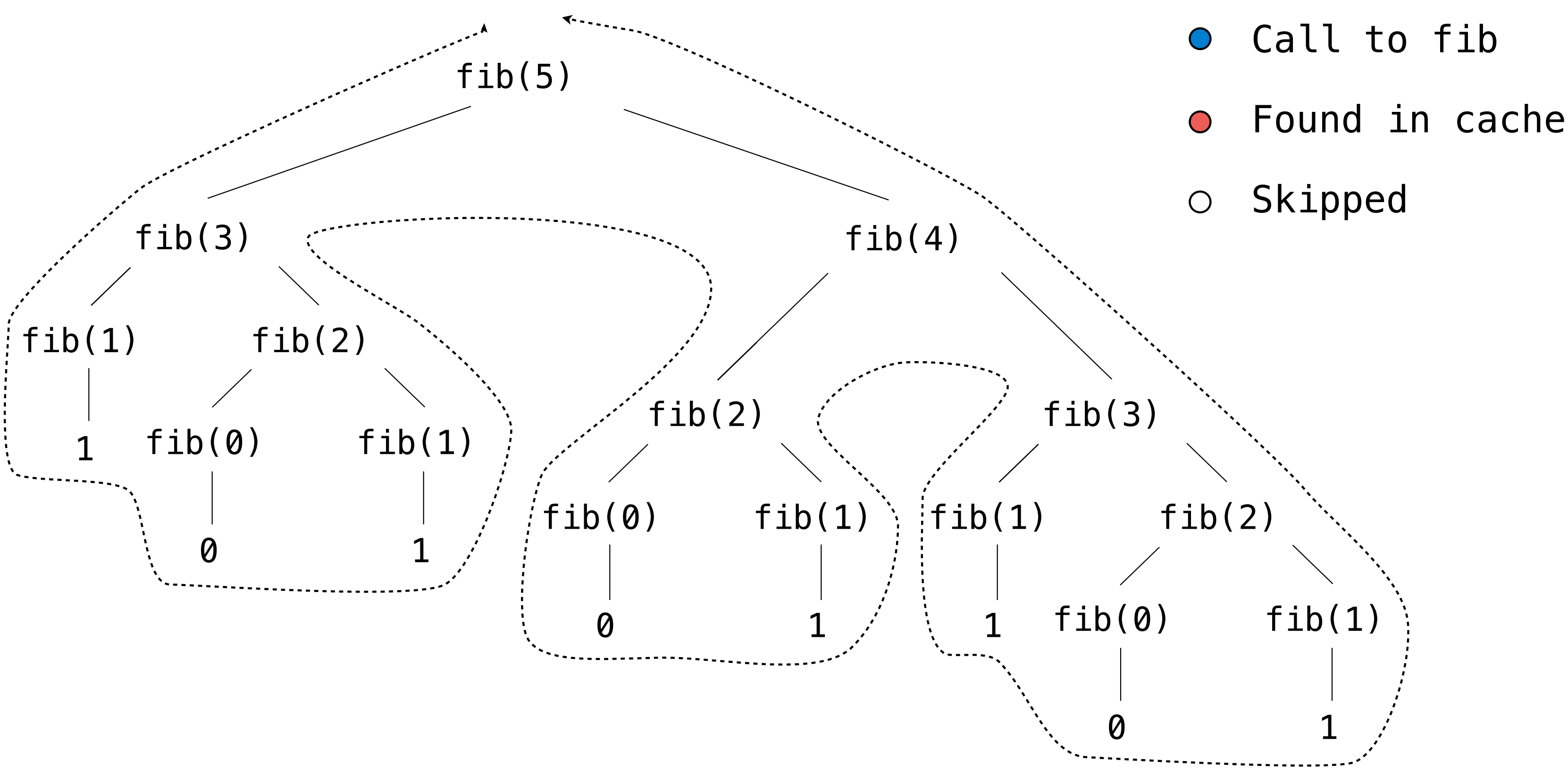
Memoized Tree Recursion



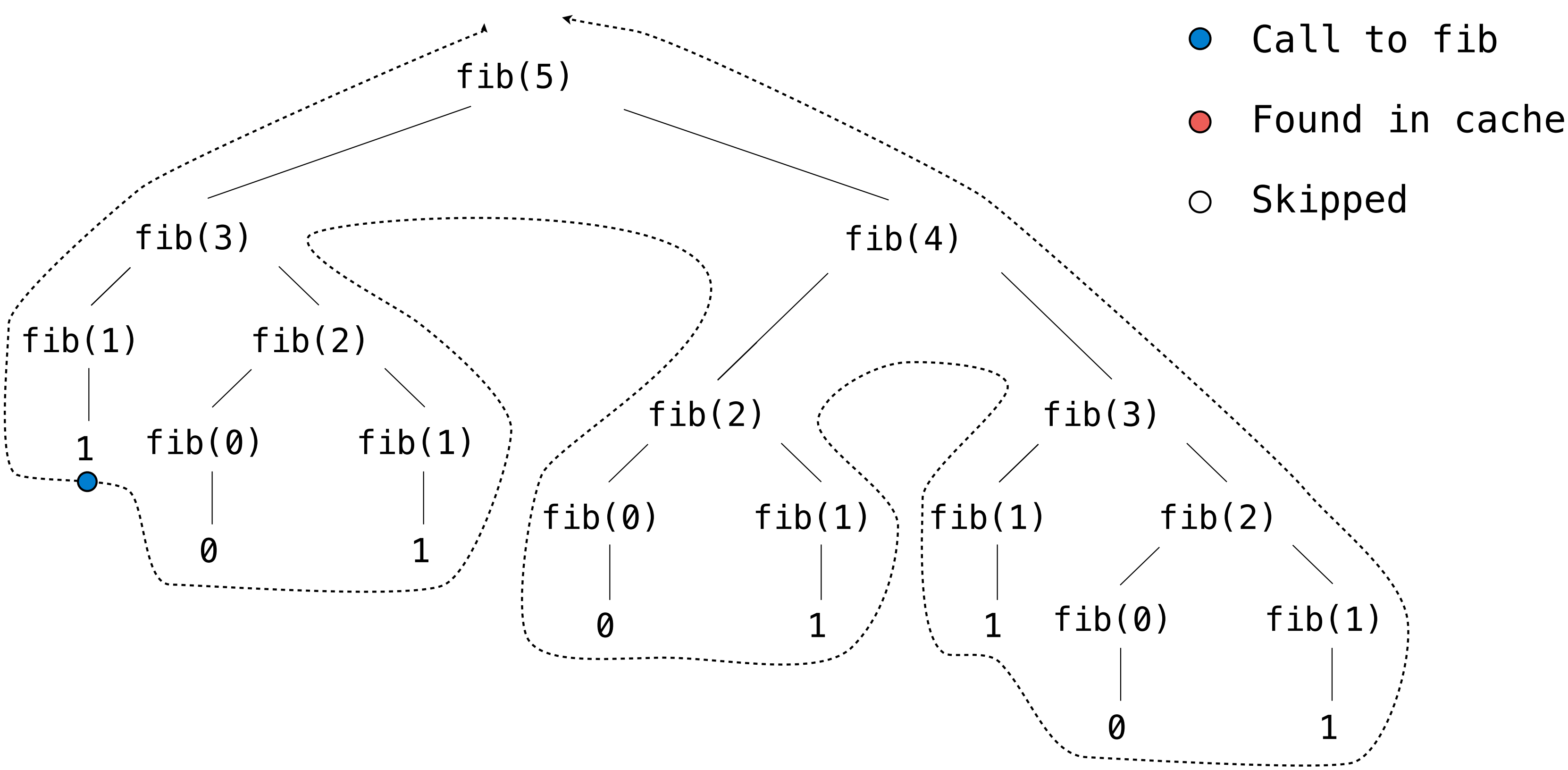
Memoized Tree Recursion



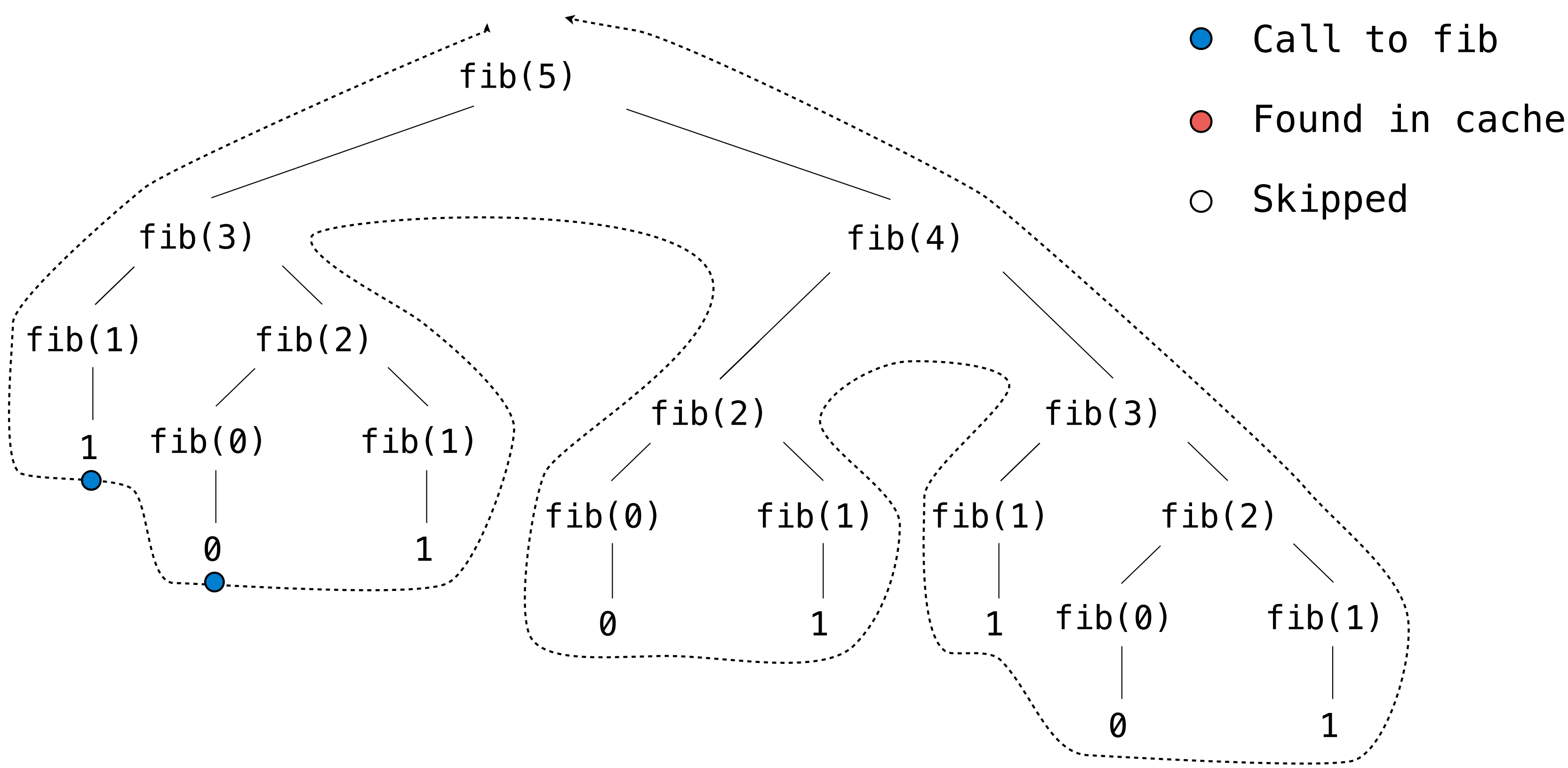
Memoized Tree Recursion



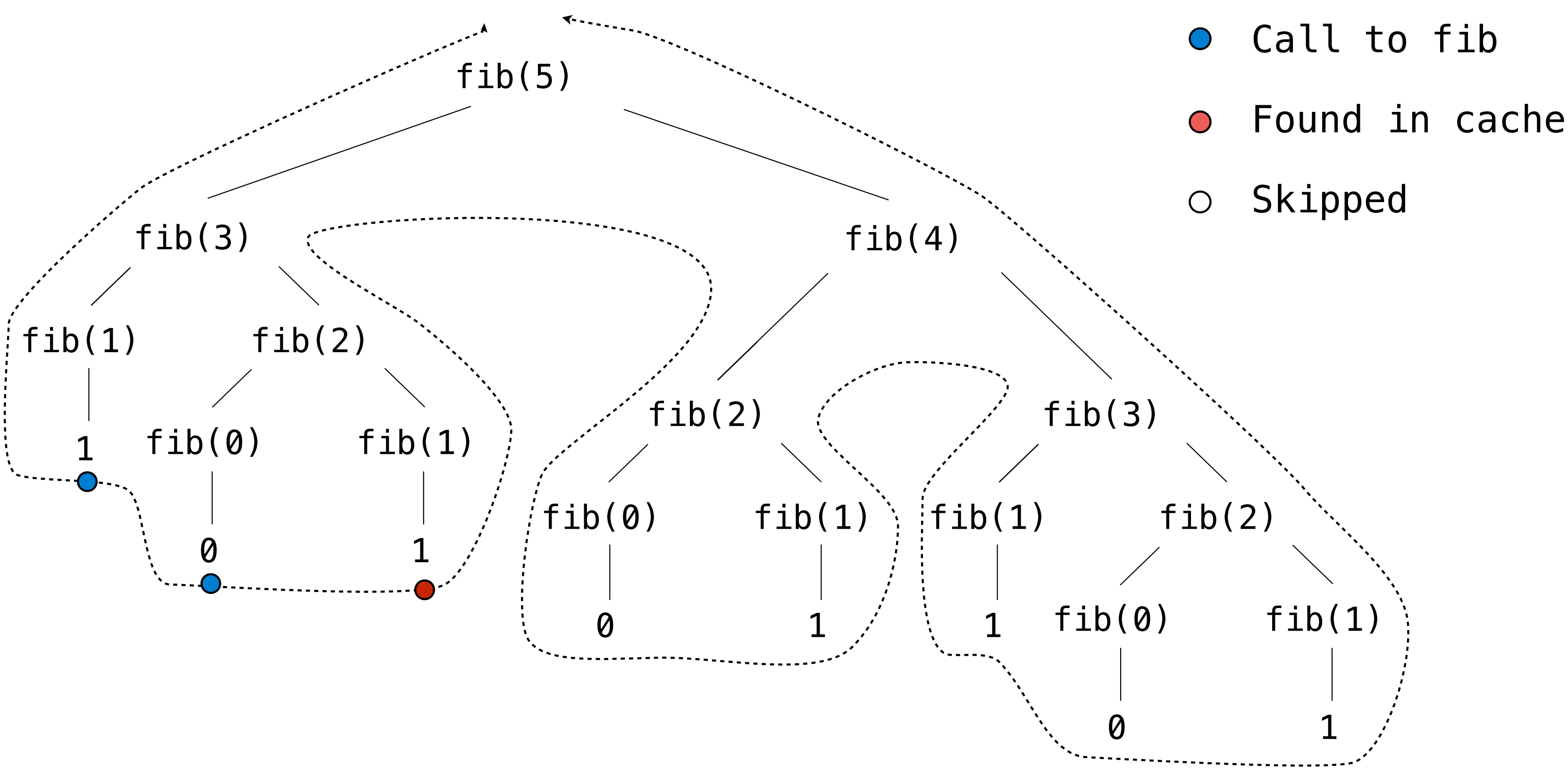
Memoized Tree Recursion



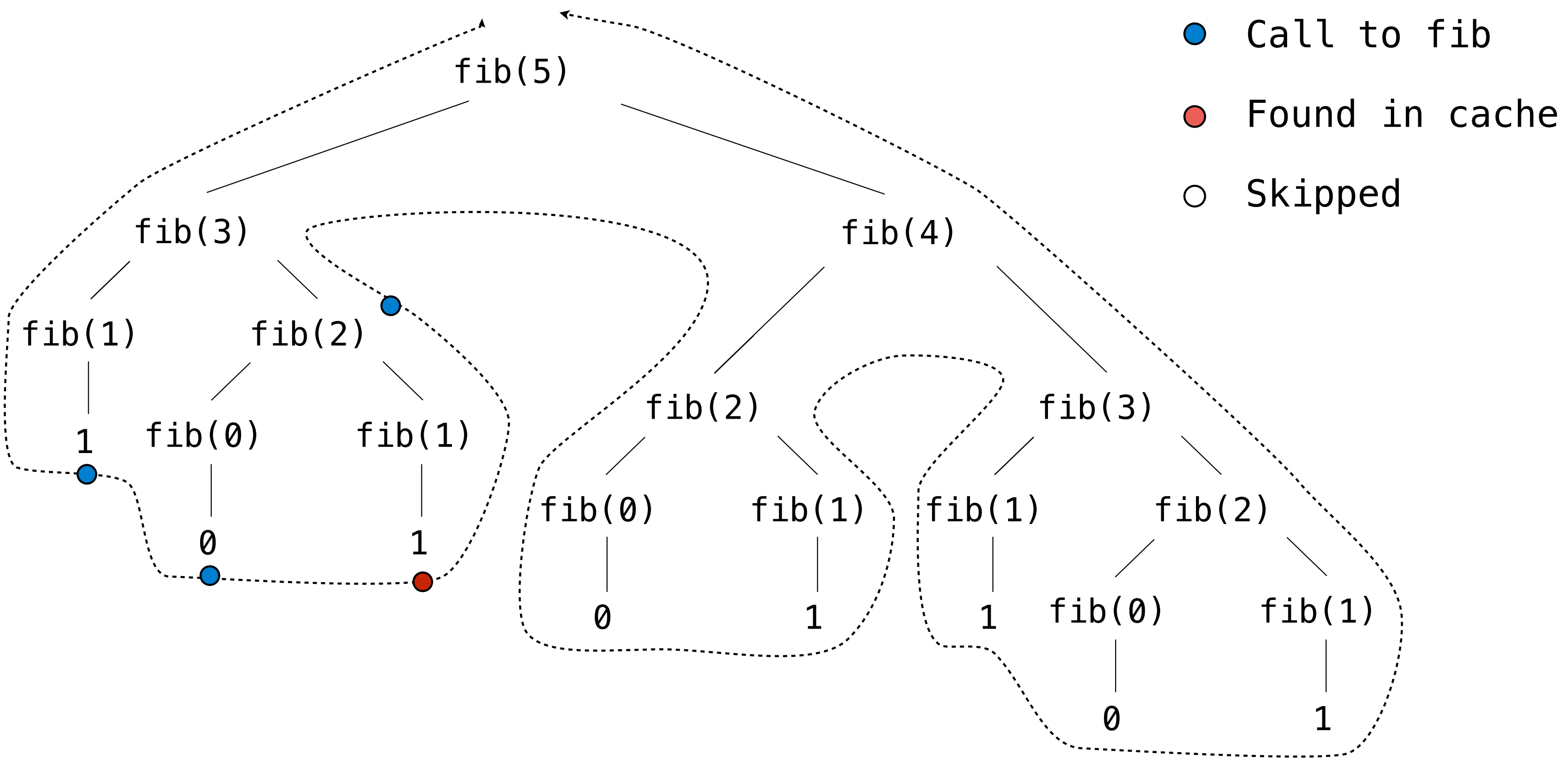
Memoized Tree Recursion



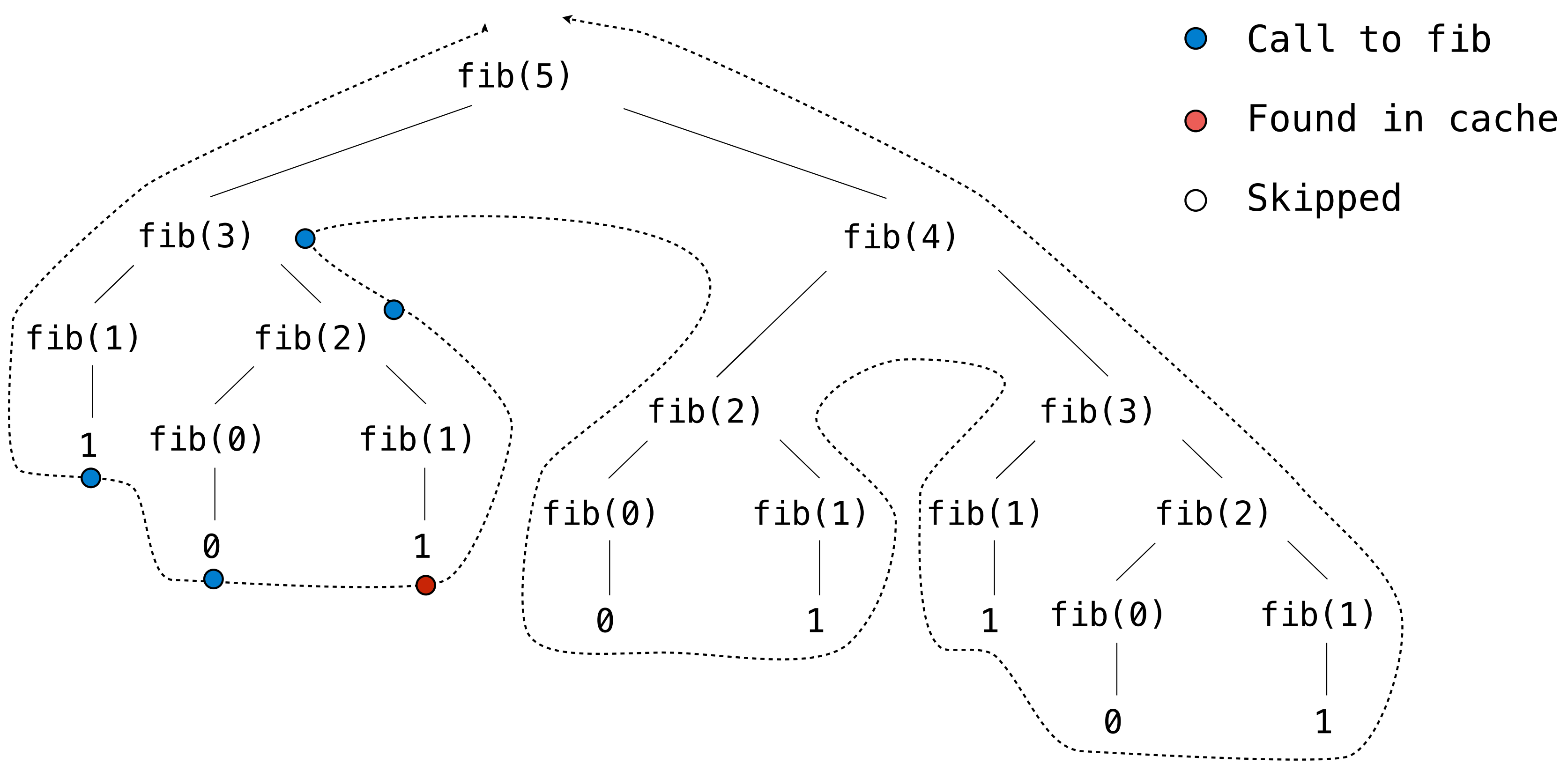
Memoized Tree Recursion



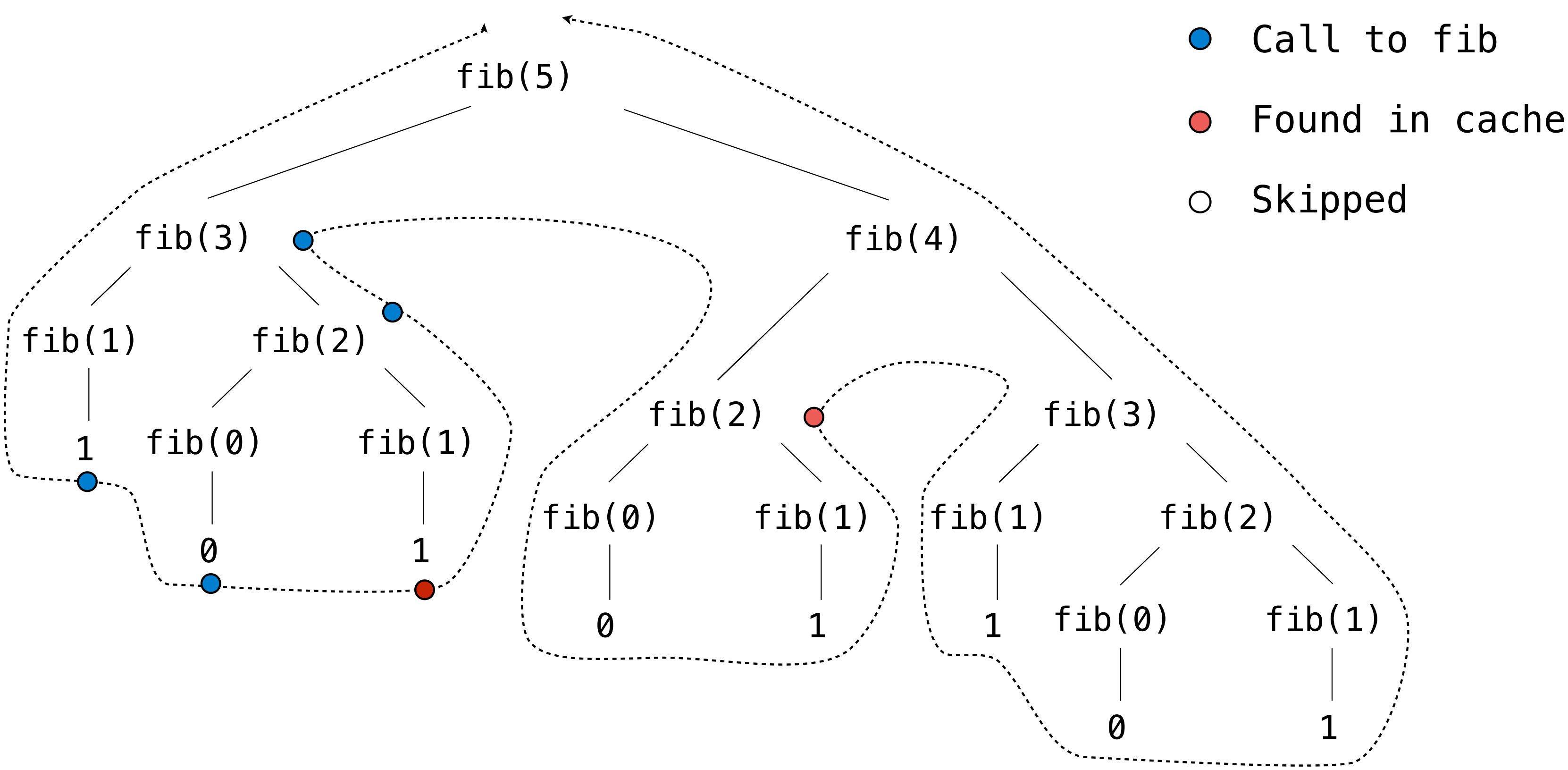
Memoized Tree Recursion



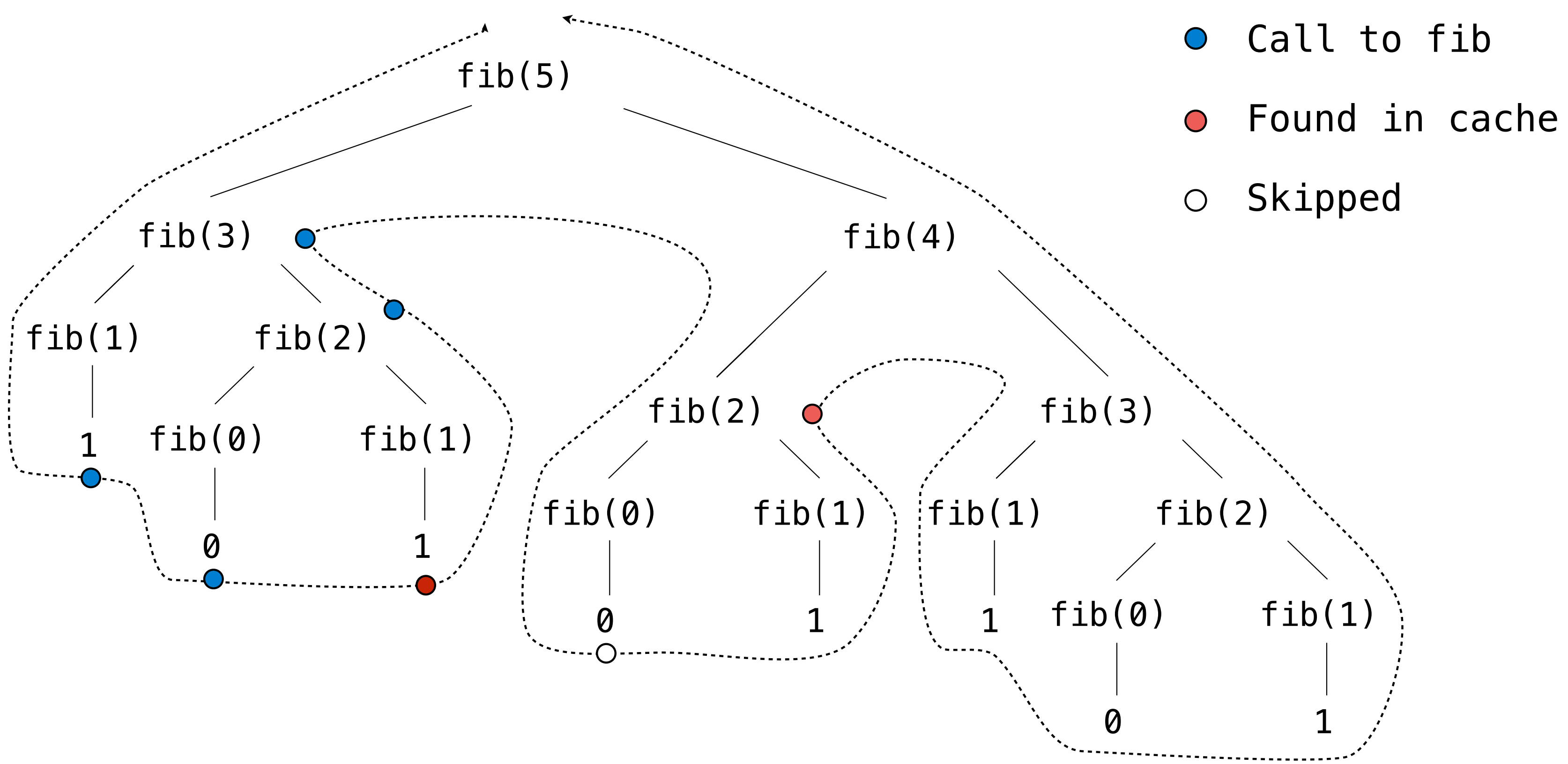
Memoized Tree Recursion



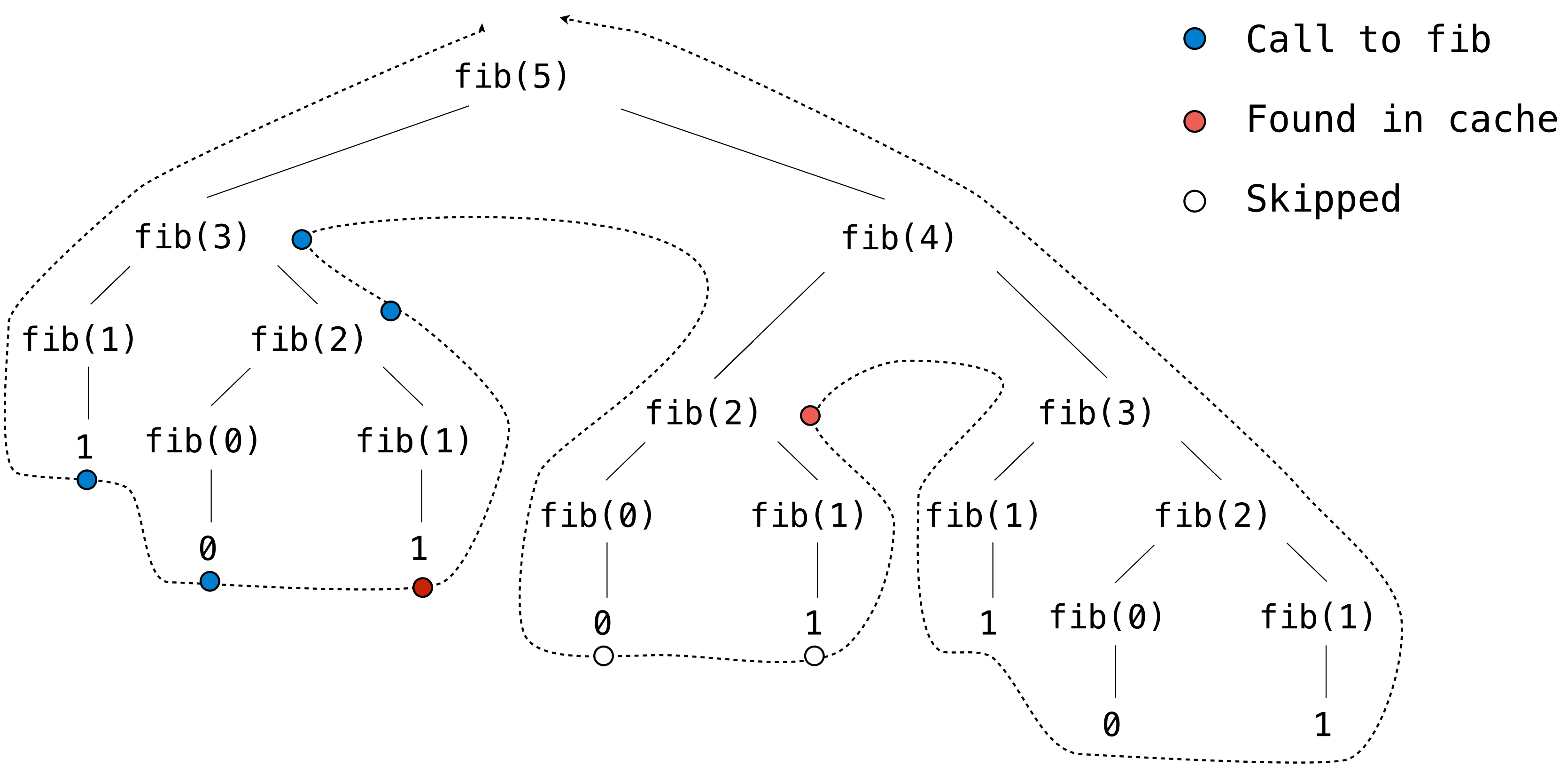
Memoized Tree Recursion



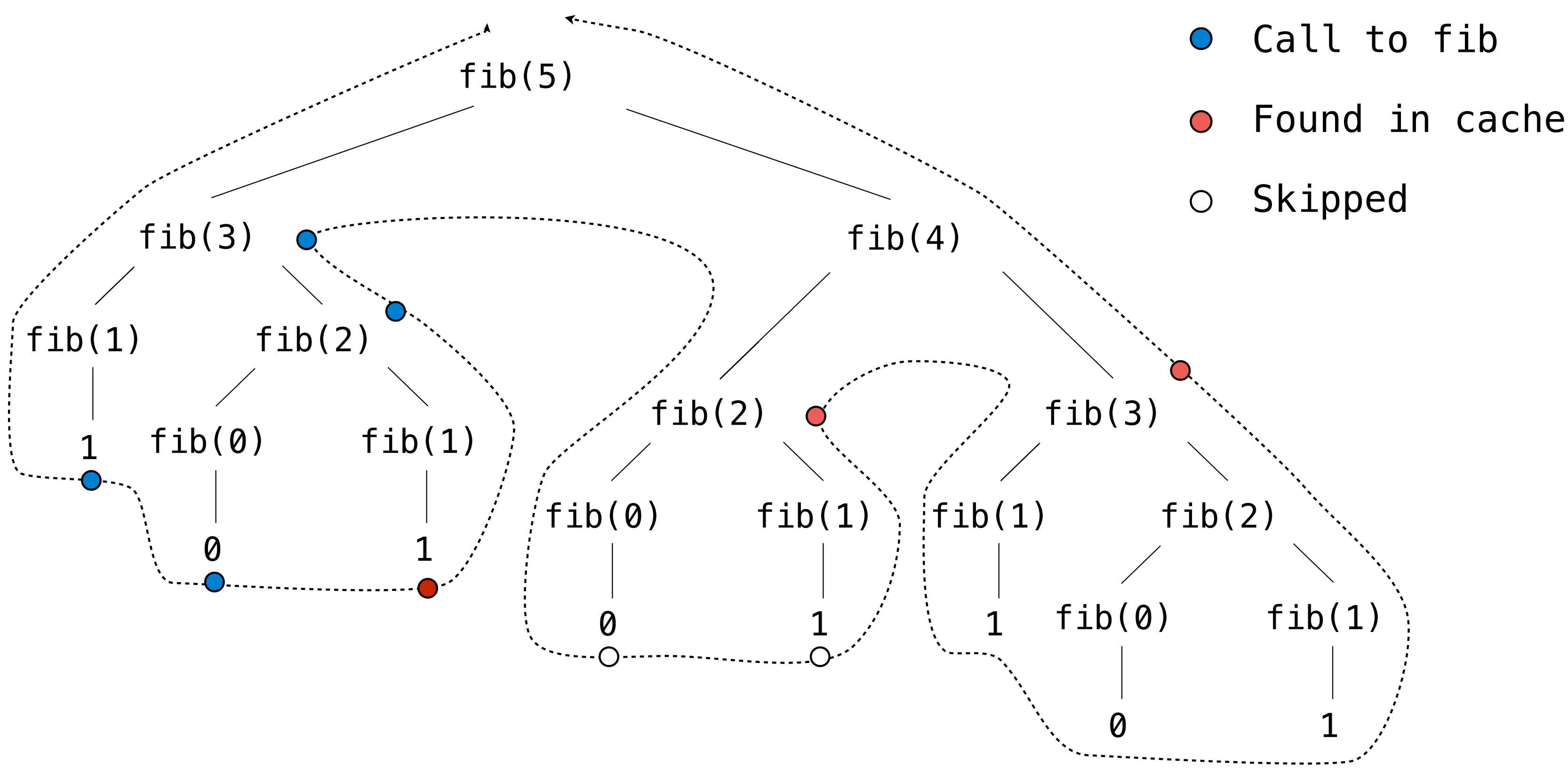
Memoized Tree Recursion



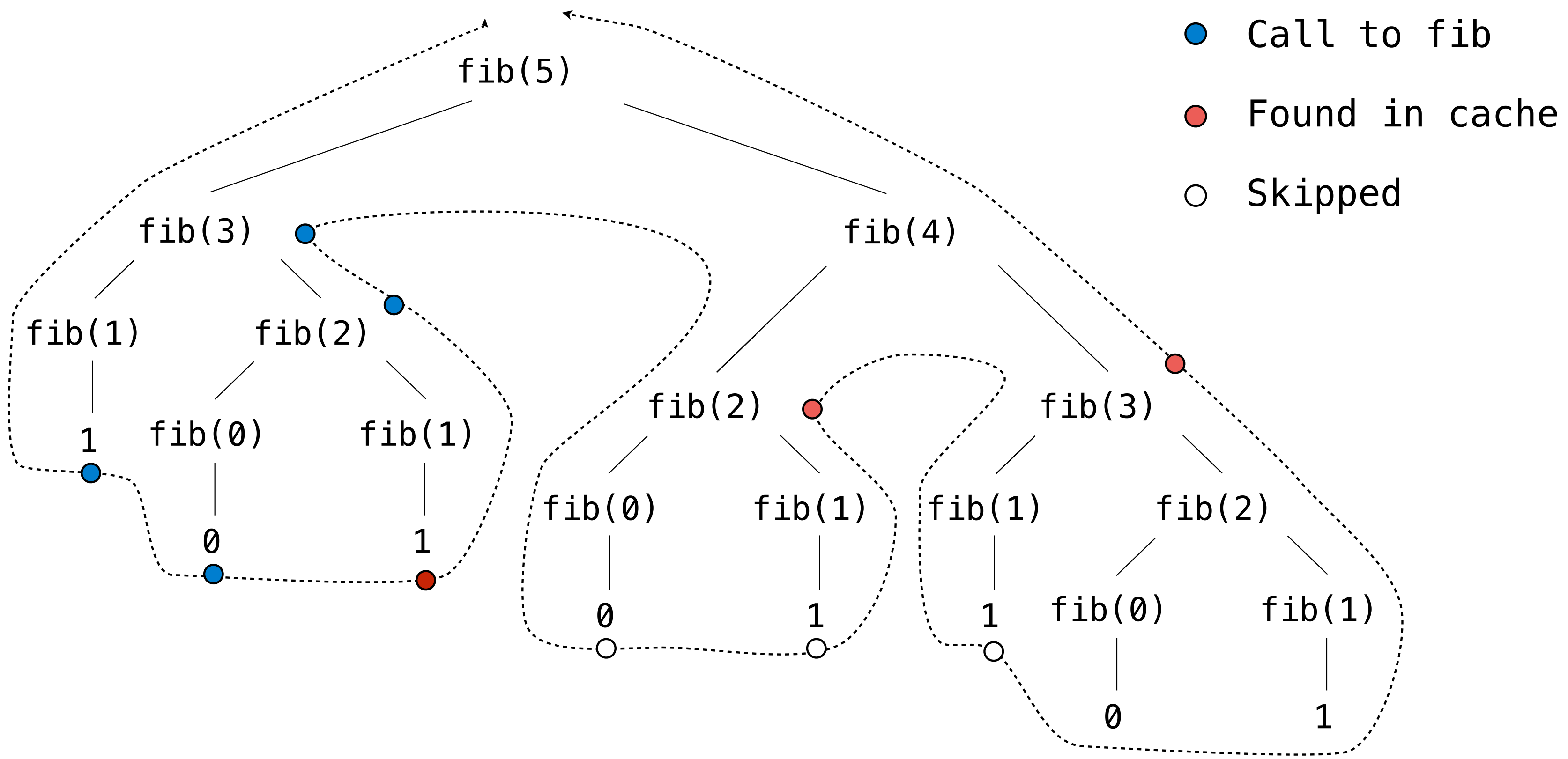
Memoized Tree Recursion



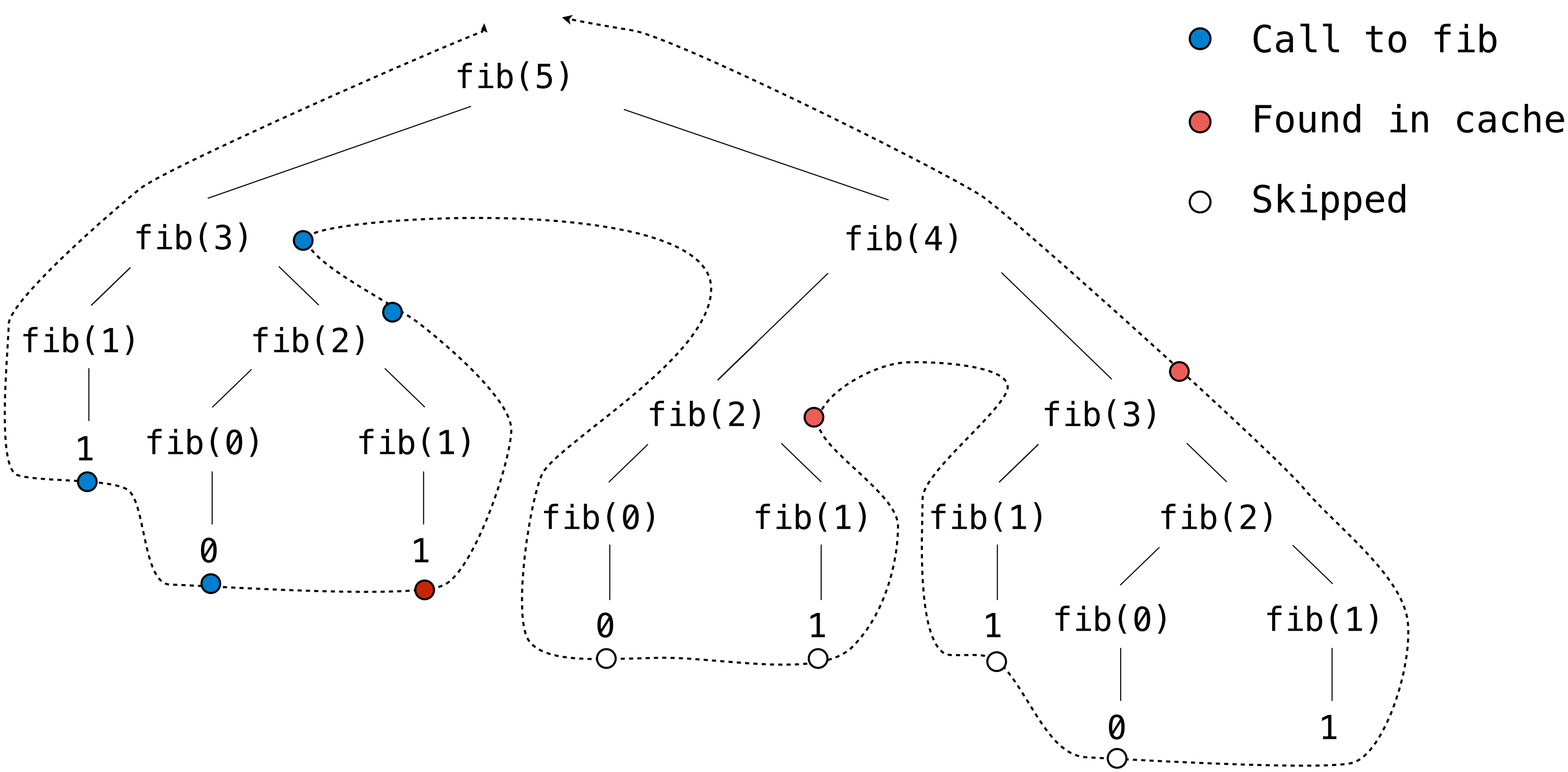
Memoized Tree Recursion



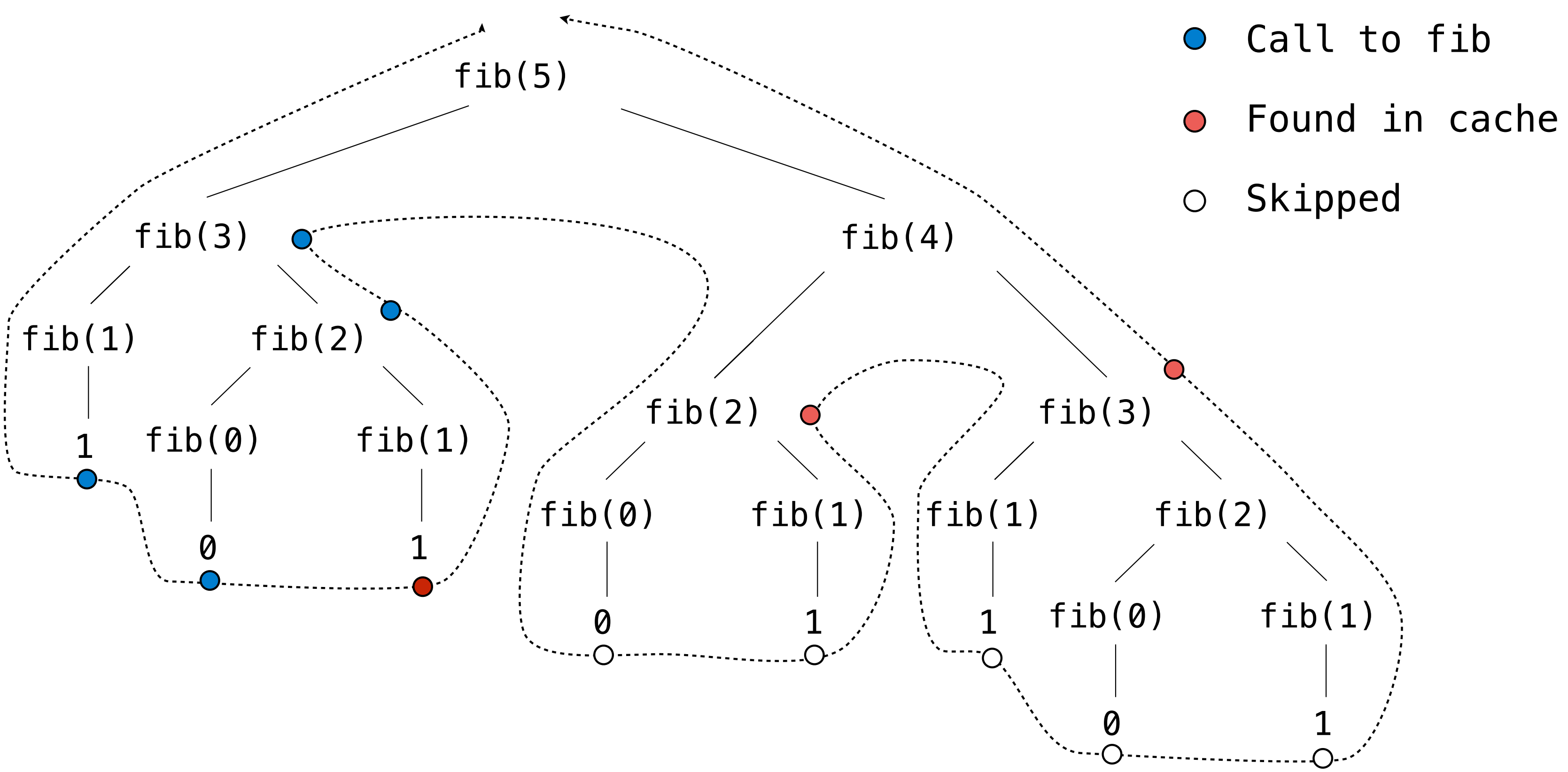
Memoized Tree Recursion



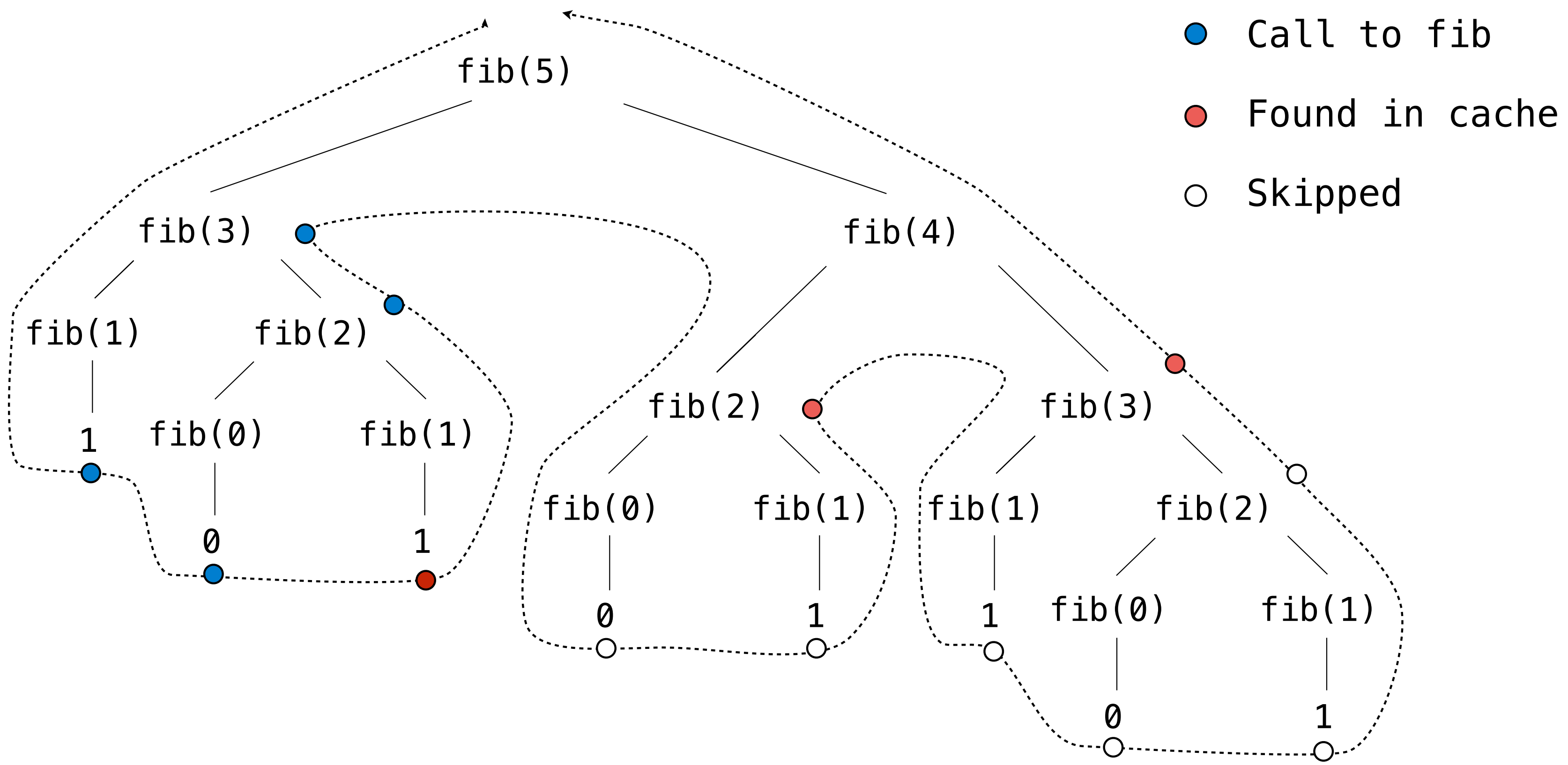
Memoized Tree Recursion



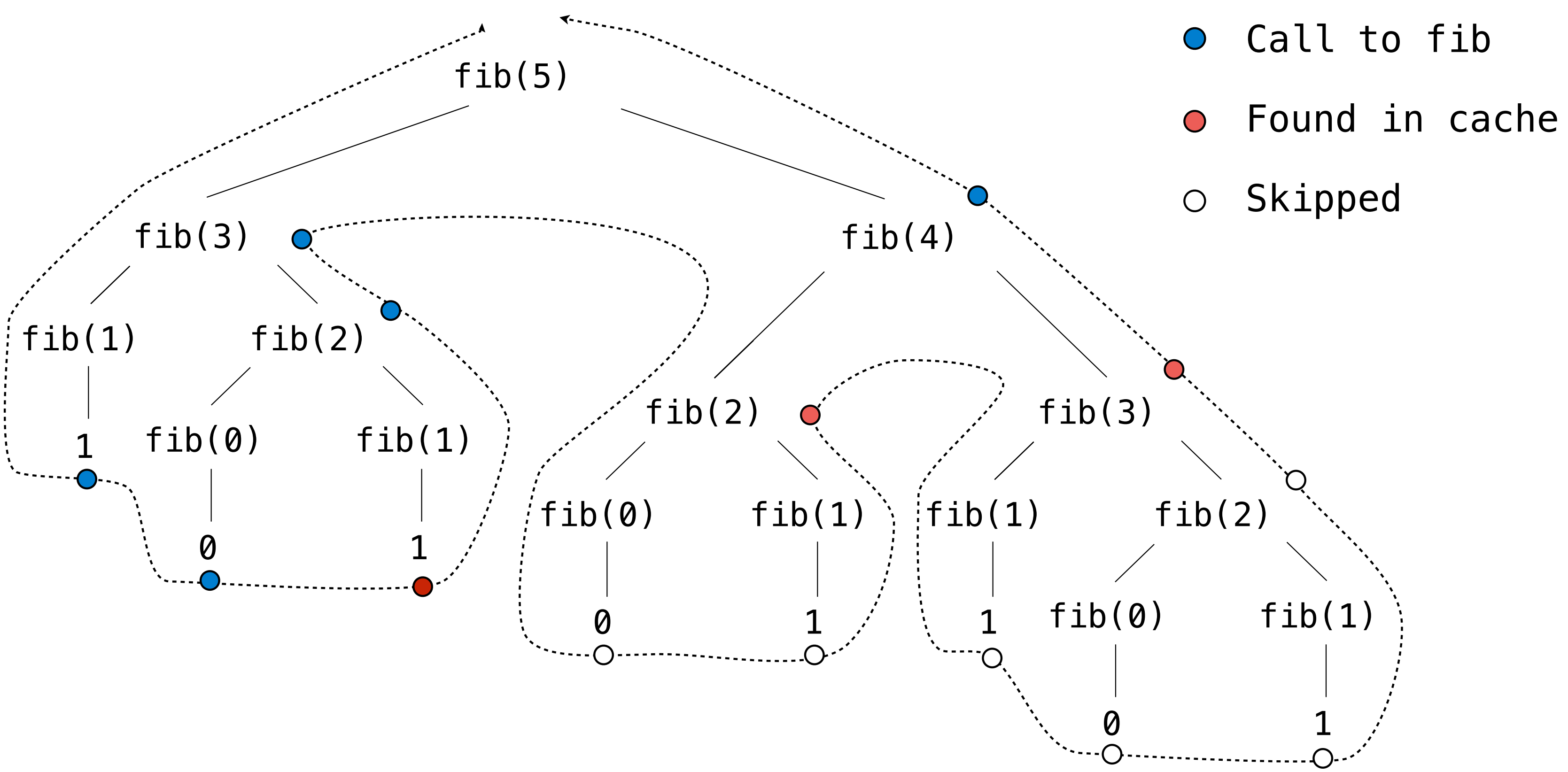
Memoized Tree Recursion



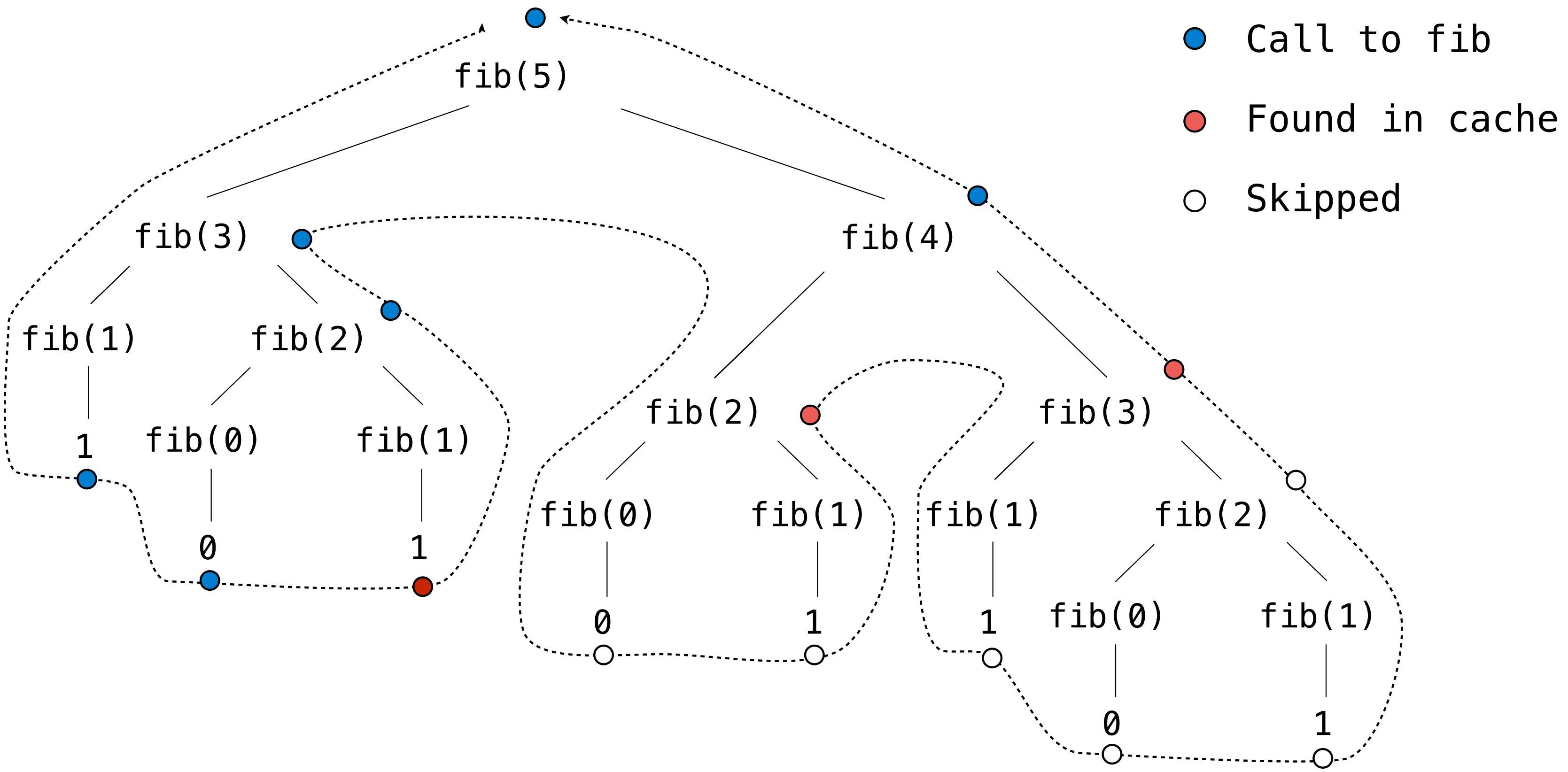
Memoized Tree Recursion



Memoized Tree Recursion



Memoized Tree Recursion



Linked List Class

Linked Lists as Objects

Linked Lists as Objects

Linked list idea: Pairs are sufficient to represent sequences of arbitrary length

Linked Lists as Objects

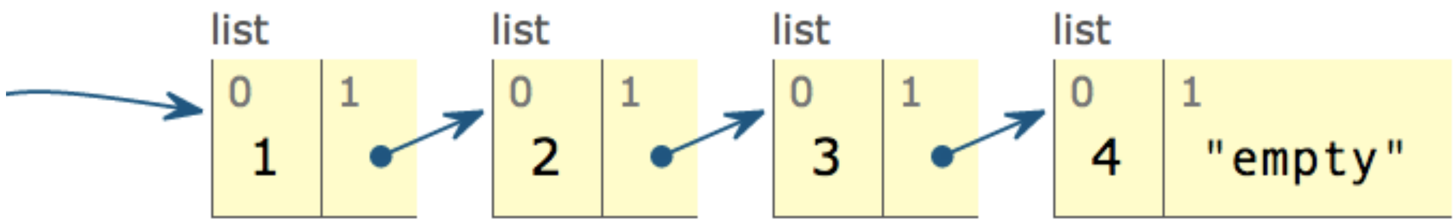
Linked list idea: Pairs are sufficient to represent sequences of arbitrary length

1 2 3 4

Linked Lists as Objects

`Linked list idea: Pairs are sufficient to represent sequences of arbitrary length`

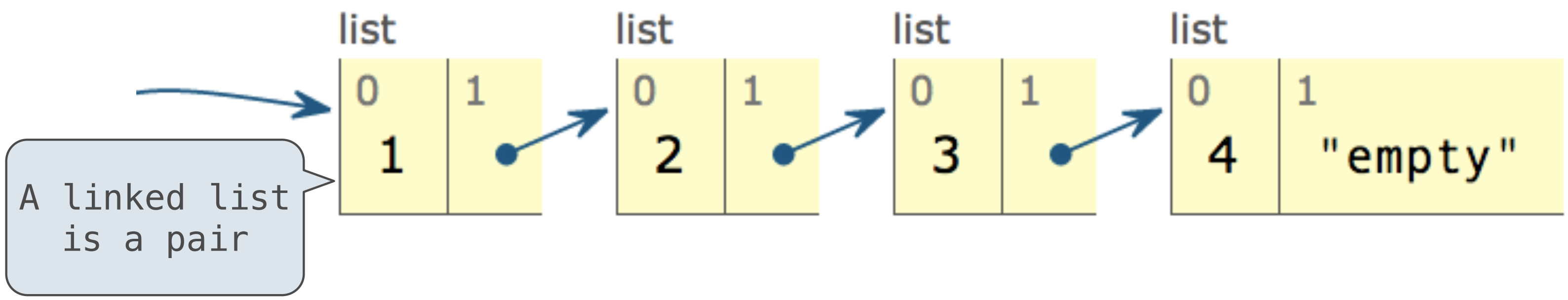
`1 2 3 4`



Linked Lists as Objects

Linked list idea: Pairs are sufficient to represent sequences of arbitrary length

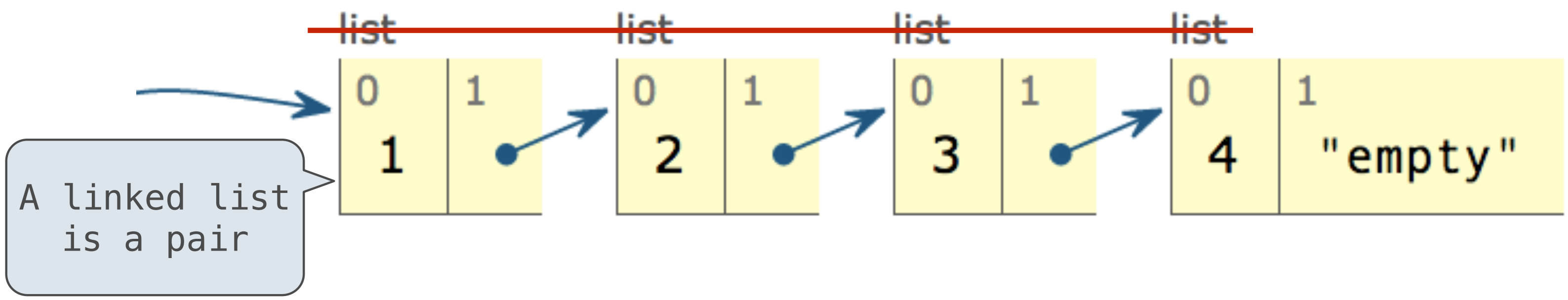
1 2 3 4



Linked Lists as Objects

Linked list idea: Pairs are sufficient to represent sequences of arbitrary length

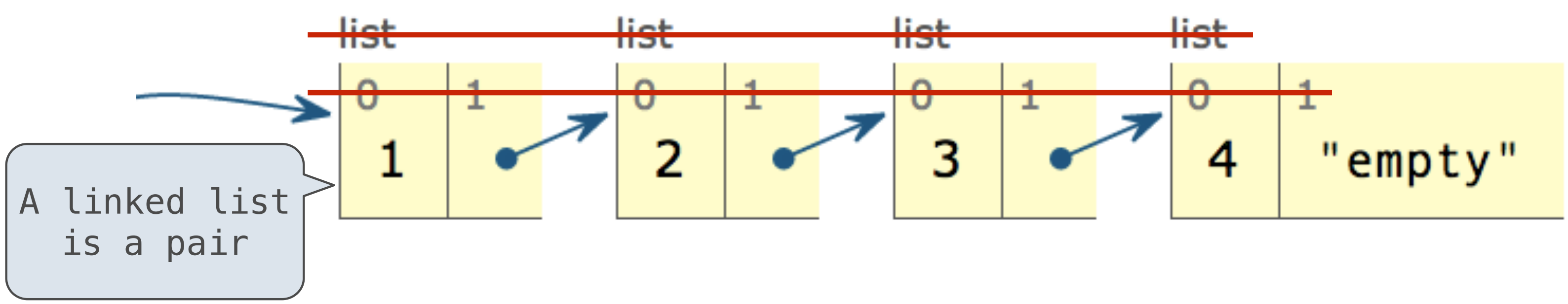
1 2 3 4



Linked Lists as Objects

Linked list idea: Pairs are sufficient to represent sequences of arbitrary length

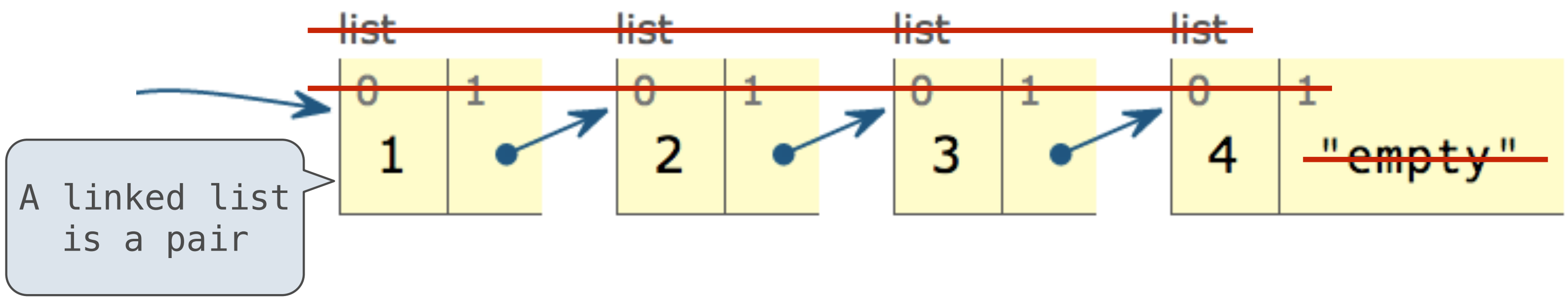
1 2 3 4



Linked Lists as Objects

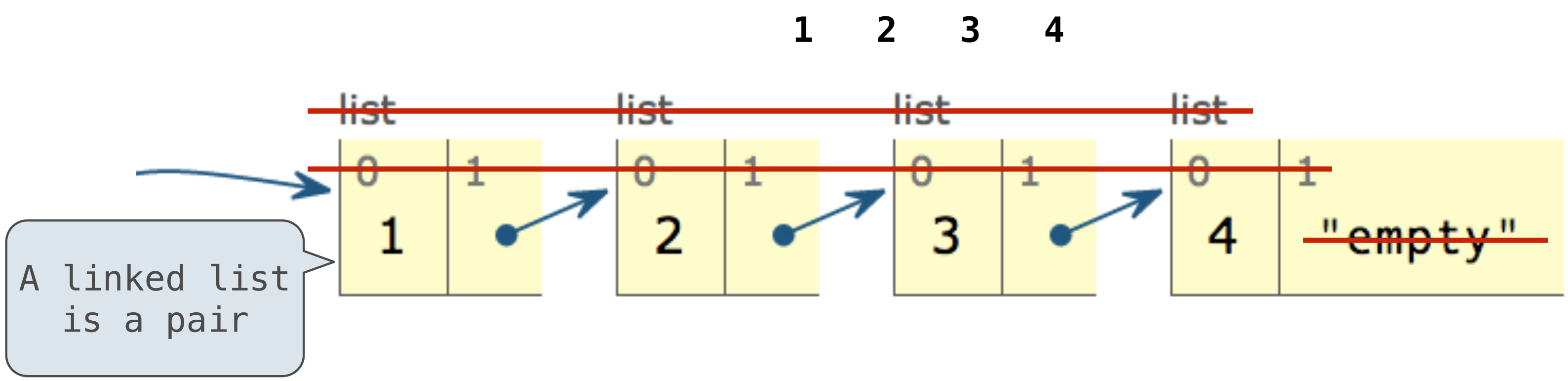
```
Linked list idea: Pairs are sufficient to represent sequences of arbitrary length
```

1 2 3 4



Linked Lists as Objects

Linked list idea: Pairs are sufficient to represent sequences of arbitrary length

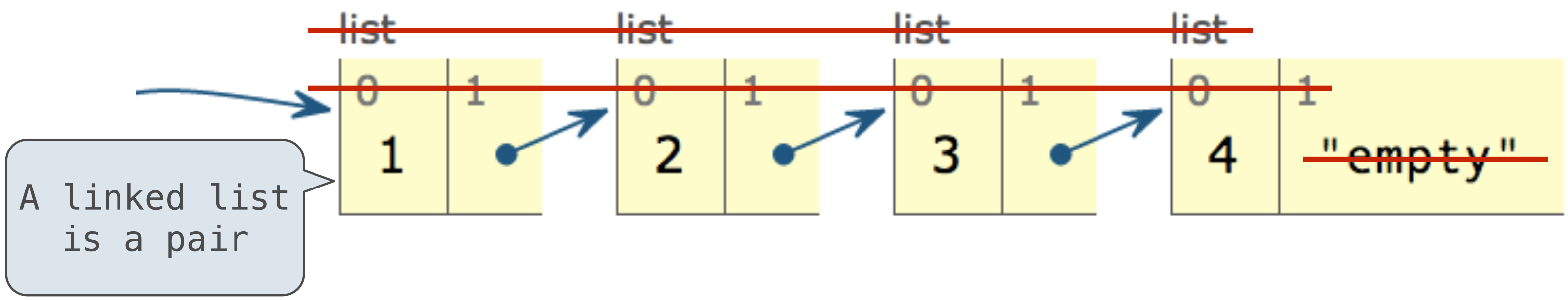


Data abstraction (old way):

Linked Lists as Objects

Linked list idea: Pairs are sufficient to represent sequences of arbitrary length

1 2 3 4



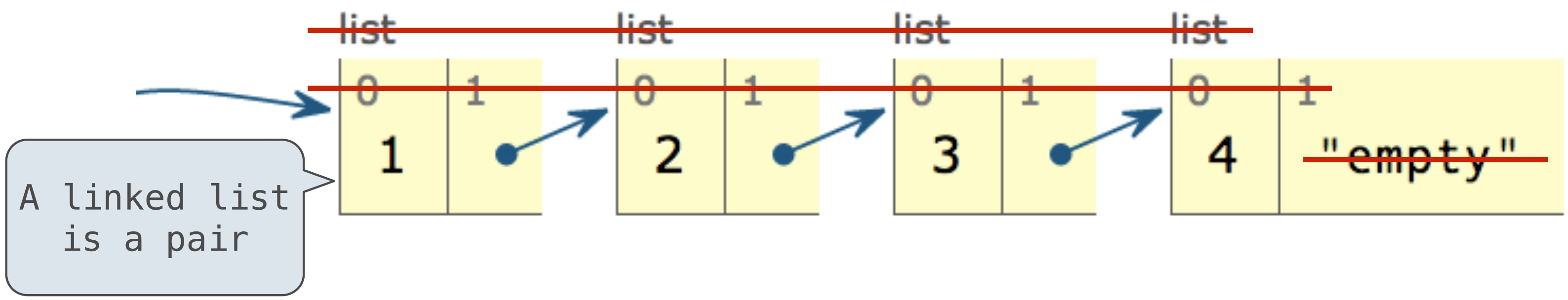
Data abstraction (old way):

Link class (new way):

Linked Lists as Objects

Linked list idea: Pairs are sufficient to represent sequences of arbitrary length

1 2 3 4



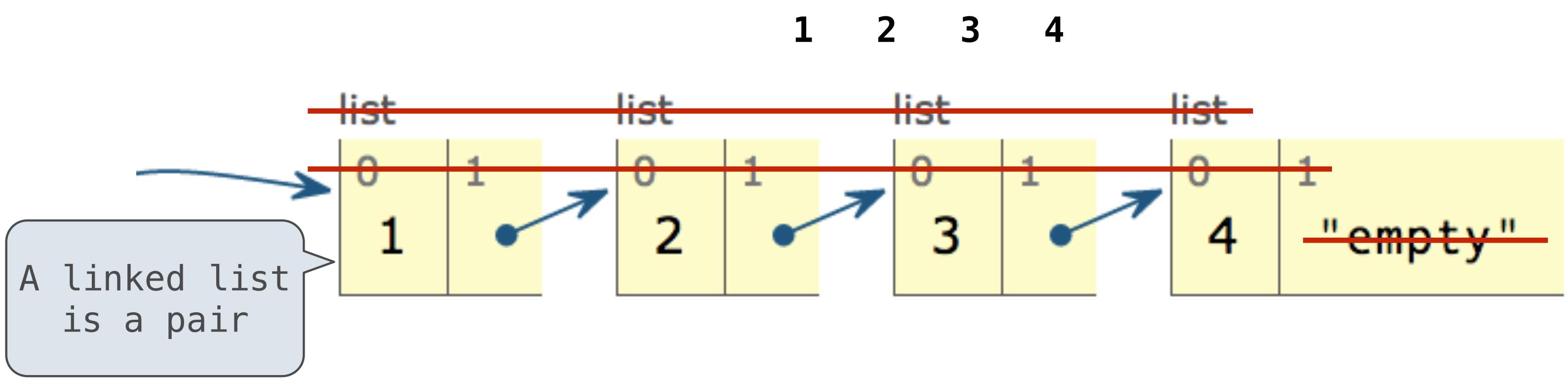
Data abstraction (old way):

```
>>> s = link(1, link(2, link(3, link(4, empty))))
```

Link class (new way):

Linked Lists as Objects

Linked list idea: Pairs are sufficient to represent sequences of arbitrary length



Data abstraction (old way):

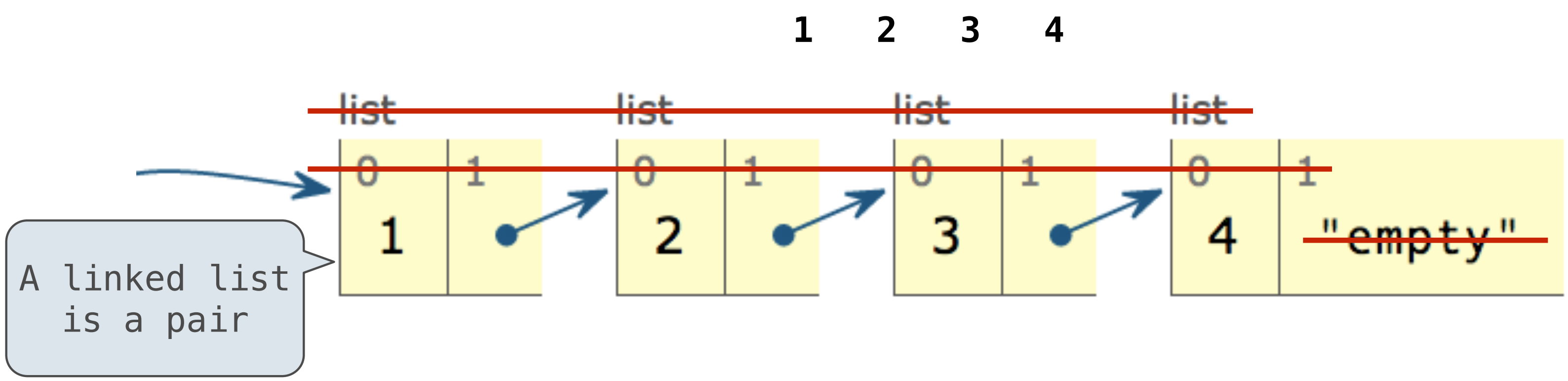
```
>>> s = link(1, link(2, link(3, link(4, empty))))
```

Link class (new way):

```
>>> s = Link(1, Link(2, Link(3, Link(4))))
```

Linked Lists as Objects

Linked list idea: Pairs are sufficient to represent sequences of arbitrary length



Data abstraction (old way):

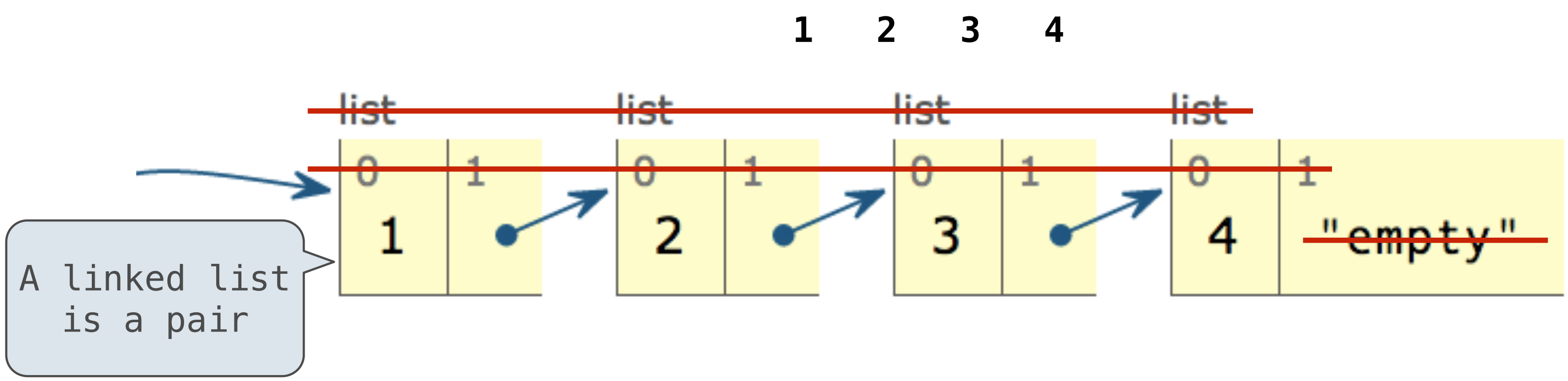
```
>>> s = link(1, link(2, link(3, link(4, empty))))
>>> len_link(s)
4
```

Link class (new way):

```
>>> s = Link(1, Link(2, Link(3, Link(4))))
```

Linked Lists as Objects

Linked list idea: Pairs are sufficient to represent sequences of arbitrary length



Data abstraction (old way):

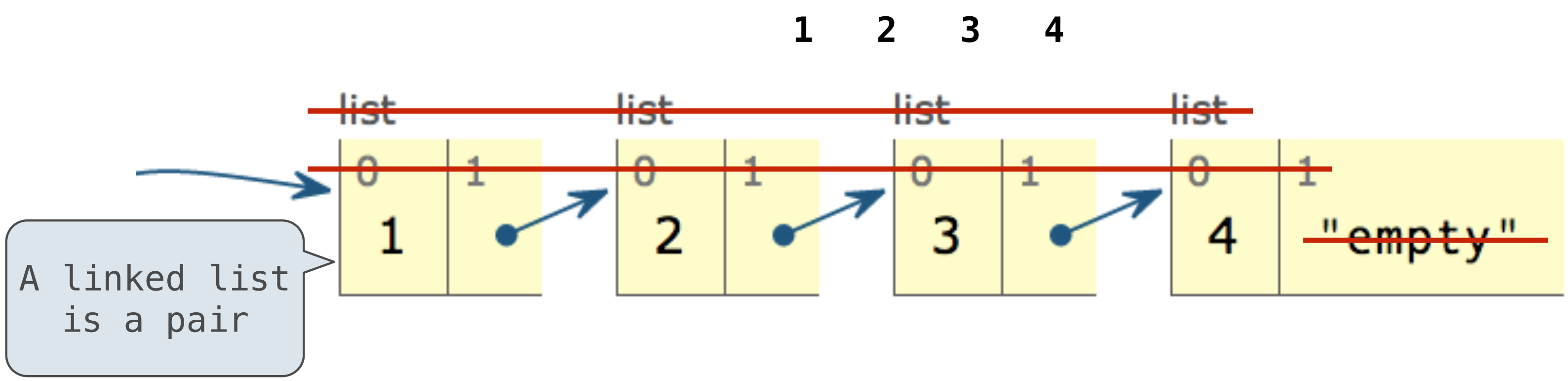
```
>>> s = link(1, link(2, link(3, link(4, empty))))
>>> len_link(s)
4
```

Link class (new way):

```
>>> s = Link(1, Link(2, Link(3, Link(4))))
>>> len(s)
4
```

Linked Lists as Objects

```
Linked list idea: Pairs are sufficient to represent sequences of arbitrary length
```



Data abstraction (old way):

```
>>> s = link(1, link(2, link(3, link(4, empty))))
>>> len_link(s)
4
>>> getitem_link(s, 2)
3
```

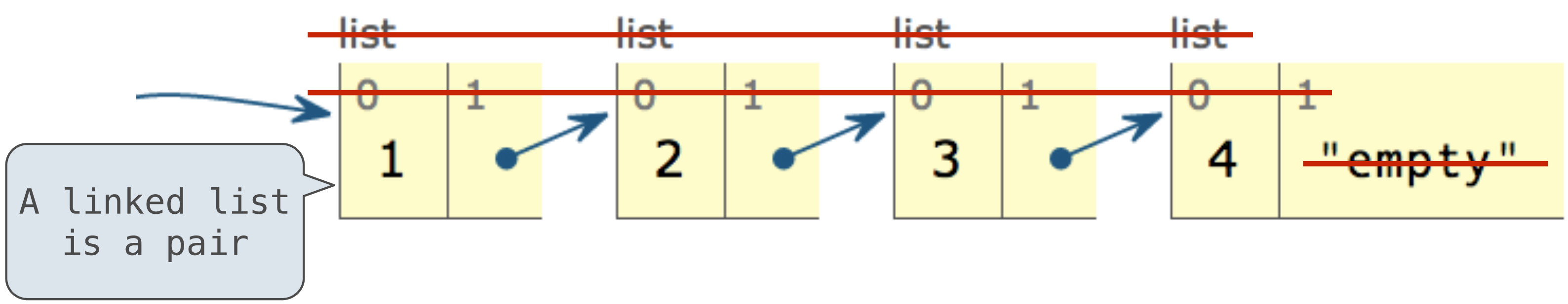
Link class (new way):

```
>>> s = Link(1, Link(2, Link(3, Link(4))))
>>> len(s)
4
```

Linked Lists as Objects

Linked list idea: Pairs are sufficient to represent sequences of arbitrary length

1 2 3 4



Data abstraction (old way):

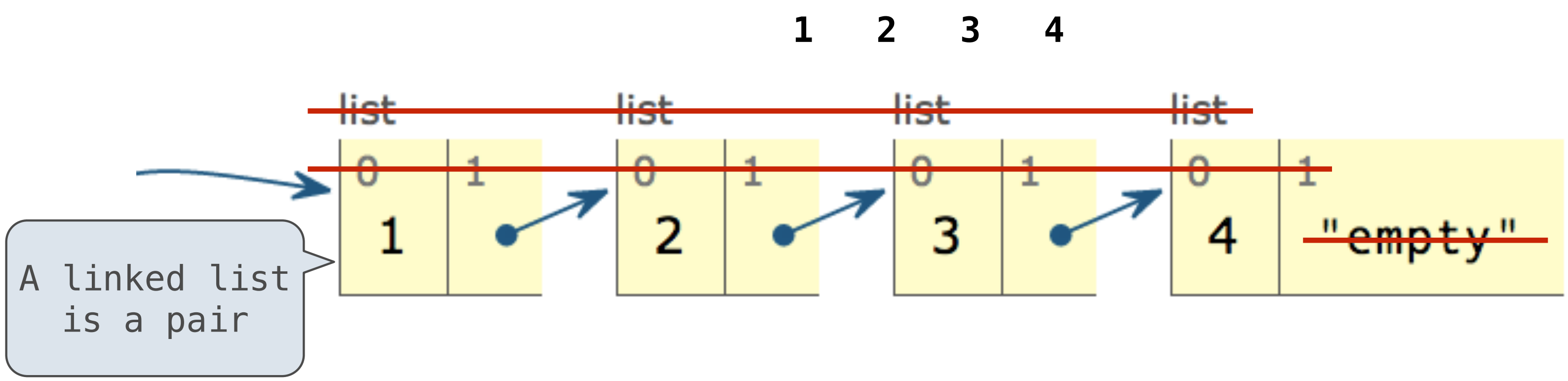
```
>>> s = link(1, link(2, link(3, link(4, empty))))
>>> len_link(s)
4
>>> getitem_link(s, 2)
3
```

Link class (new way):

```
>>> s = Link(1, Link(2, Link(3, Link(4))))
>>> len(s)
4
>>> s[2]
3
```

Linked Lists as Objects

Linked list idea: Pairs are sufficient to represent sequences of arbitrary length



Data abstraction (old way):

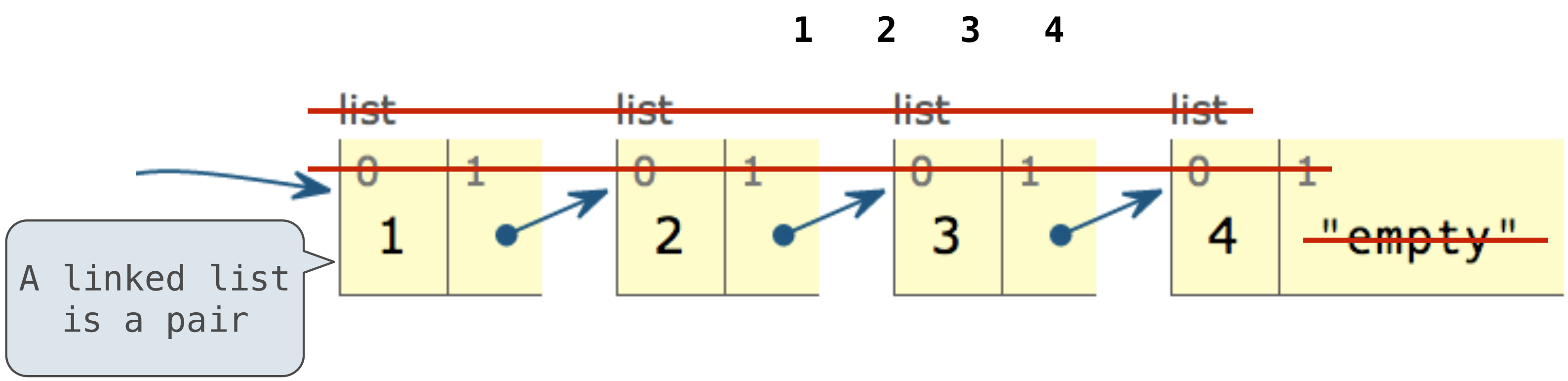
```
>>> s = link(1, link(2, link(3, link(4, empty))))
>>> len_link(s)
4
>>> getitem_link(s, 2)
3
>>> s
[1, [2, [3, [4, 'empty']]]]
```

Link class (new way):

```
>>> s = Link(1, Link(2, Link(3, Link(4))))
>>> len(s)
4
>>> s[2]
3
```

Linked Lists as Objects

Linked list idea: Pairs are sufficient to represent sequences of arbitrary length



Data abstraction (old way):

```
>>> s = link(1, link(2, link(3, link(4, empty))))
>>> len_link(s)
4
>>> getitem_link(s, 2)
3
>>> s
[1, [2, [3, [4, 'empty']]]]
```

Link class (new way):

```
>>> s = Link(1, Link(2, Link(3, Link(4))))
>>> len(s)
4
>>> s[2]
3
>>> s
Link(1, Link(2, Link(3, Link(4))))
```

Linked List Class

More special method names:

`__getitem__`	Element selection []
`__len__`	Built-in len function

Linked List Class

Linked list class: pairs are two-attribute objects

More special method names:

`__getitem__`	Element selection []
`__len__`	Built-in len function

Linked List Class

Linked list class: pairs are two-attribute objects

```
class Link:
```

More special method names:

`__getitem__`	Element selection []
`__len__`	Built-in len function

Linked List Class

Linked list class: pairs are two-attribute objects

```
class Link:

    def __init__(self, first, rest=empty):
        self.first = first
        self.rest = rest
```

More special method names:

`__getitem__`	Element selection []
`__len__`	Built-in len function

Linked List Class

Linked list class: pairs are two-attribute objects

```
class Link:


    def __init__(self, first, rest=empty):
        self.first = first
        self.rest = rest

    def __getitem__(self, i):
        if i == 0:
            return self.first
        else:
            return self.rest[i-1]
```

More special method names:

`__getitem__`	Element selection []
`__len__`	Built-in len function

Linked List Class

Linked list class: pairs are two-attribute objects

```
class Link:

    def __init__(self, first, rest=empty):
        self.first = first
        self.rest = rest

    def __getitem__(self, i):
        if i == 0:
            return self.first
        else:
            return self.rest[i-1]
```

This element selection syntax

More special method names:

`__getitem__`	Element selection []
`__len__`	Built-in len function

Linked List Class

Linked list class: pairs are two-attribute objects

```
class Link:

    def __init__(self, first, rest=empty):
        self.first = first
        self.rest = rest

    def __getitem__(self, i):
        if i == 0:
            return self.first
        else:
            return self.rest[i-1]
```

Calls this method

This element selection syntax

More special method names:

__getitem__	Element selection []
__len__	Built-in len function

Linked List Class

Linked list class: pairs are two-attribute objects

```
class Link:

    def __init__(self, first, rest=empty):
        self.first = first
        self.rest = rest

    def __getitem__(self, i):
        if i == 0:
            return self.first
        else:
            return self.rest[i-1]

    def __len__(self):
        return 1 + len(self.rest)
```

Calls this method

This element selection syntax

More special method names:

`__getitem__`	Element selection []
`__len__`	Built-in len function

Linked List Class

Linked list class: pairs are two-attribute objects

```
class Link:

    def __init__(self, first, rest=empty):
        self.first = first
        self.rest = rest

    def __getitem__(self, i):
        if i == 0:
            return self.first
        else:
            return self.rest[i-1]

    def __len__(self):
        return 1 + len(self.rest)
```

Calls this method

This element selection syntax

Yes, this call is recursive too

More special method names:

`__getitem__`	Element selection []
`__len__`	Built-in len function

Linked List Class

Linked list class: pairs are two-attribute objects

```
class Link:

    empty = ()

    def __init__(self, first, rest=empty):
        self.first = first
        self.rest = rest

    def __getitem__(self, i):
        if i == 0:
            return self.first
        else:
            return self.rest[i-1]

    def __len__(self):
        return 1 + len(self.rest)
```

Calls this method

This element selection syntax

Yes, this call is recursive too

More special method names:

`__getitem__`	Element selection []
`__len__`	Built-in len function

Linked List Class

Linked list class: pairs are two-attribute objects

```
class Link:

    empty = ()

    def __init__(self, first, rest=empty):
        self.first = first
        self.rest = rest

    def __getitem__(self, i):
        if i == 0:
            return self.first
        else:
            return self.rest[i-1]

    def __len__(self):
        return 1 + len(self.rest)
```

Some zero length sequence

Calls this method

This element selection syntax

Yes, this call is recursive too

More special method names:

`__getitem__`	Element selection []
`__len__`	Built-in len function

Linked List Class

Linked list class: pairs are two-attribute objects

```
class Link:

    empty = ()

    def __init__(self, first, rest=empty):
        self.first = first
        self.rest = rest

    def __getitem__(self, i):
        if i == 0:
            return self.first
        else:
            return self.rest[i-1]

    def __len__(self):
        return 1 + len(self.rest)
```

Some zero length sequence

Calls this method

This element selection syntax

Yes, this call is recursive too

More special method names:

`__getitem__`	Element selection []
`__len__`	Built-in len function

Methods can be recursive too!

(Demo)

Tree Class

Tree Class

A Tree has an entry (any value) at its root and a list of branches

Tree Class

A Tree has an entry (any value) at its root and a list of branches

```
class Tree:
```

Tree Class

A Tree has an entry (any value) at its root and a list of branches

```
class Tree:
    def __init__(self, entry, branches=()):
```

Tree Class

A Tree has an entry (any value) at its root and a list of branches

```
class Tree:
    def __init__(self, entry, branches=()):
        self.entry = entry
```

Tree Class

A Tree has an entry (any value) at its root and a list of branches

```
class Tree:
    def __init__(self, entry, branches=()):
        self.entry = entry
        for branch in branches:
            assert isinstance(branch, Tree)
```

Tree Class

A Tree has an entry (any value) at its root and a list of branches

```
class Tree:
    def __init__(self, entry, branches=()):
        self.entry = entry
        for branch in branches:
            assert isinstance(branch, Tree)
```

Built-in isinstance function:
returns True if branch has a class
that *is* **or** *inherits from* Tree

Tree Class

A Tree has an entry (any value) at its root and a list of branches

```
class Tree:
    def __init__(self, entry, branches=()):
        self.entry = entry
        for branch in branches:
            assert isinstance(branch, Tree)
        self.branches = list(branches)
```

Built-in isinstance function: returns True if branch has a class that *is* **or** *inherits from* Tree

Tree Class

A Tree has an entry (any value) at its root and a list of branches

```
class Tree:
    def __init__(self, entry, branches=()):
        self.entry = entry
        for branch in branches:
            assert isinstance(branch, Tree)
        self.branches = list(branches)
```

Built-in isinstance function:
returns True if branch has a class
that *is* **or** *inherits from* Tree

```
def fib_tree(n):
```

Tree Class

A Tree has an entry (any value) at its root and a list of branches

```
class Tree:
    def __init__(self, entry, branches=()):
        self.entry = entry
        for branch in branches:
            assert isinstance(branch, Tree)
        self.branches = list(branches)
```

Built-in isinstance function: returns True if branch has a class that *is* **or** *inherits from* Tree

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

Tree Class

```
A Tree has an entry (any value) at its root and a list of branches
```

```
class Tree:
    def __init__(self, entry, branches=()):
        self.entry = entry
        for branch in branches:
            assert isinstance(branch, Tree)
        self.branches = list(branches)
```

```
Built-in isinstance function:
returns True if branch has a class
that is or inherits from Tree
```

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

Tree Class

A Tree has an entry (any value) at its root and a list of branches

```
class Tree:
    def __init__(self, entry, branches=()):
        self.entry = entry
        for branch in branches:
            assert isinstance(branch, Tree)
        self.branches = list(branches)
```

Built-in isinstance function: returns True if branch has a class that *is* **or** *inherits from* Tree

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

Tree Class

A Tree has an entry (any value) at its root and a list of branches

```
class Tree:
    def __init__(self, entry, branches=()):
        self.entry = entry
        for branch in branches:
            assert isinstance(branch, Tree)
        self.branches = list(branches)
```

Built-in isinstance function:
returns True if branch has a class
that *is* **or** *inherits from* Tree

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

(Demo)

Example: Hailstone Trees

Example: Hailstone Trees

```
Pick a positive integer n as the start
```

Example: Hailstone Trees

`Pick a positive integer n as the start`

`If n is even, divide it by 2`

Example: Hailstone Trees

```
Pick a positive integer n as the start

If n is even, divide it by 2

If n is odd, multiply it by 3 and add 1
```

Example: Hailstone Trees

```
Pick a positive integer n as the start

If n is even, divide it by 2

If n is odd, multiply it by 3 and add 1

Continue this process until n is 1
```

Example: Hailstone Trees

```
Pick a positive integer n as the start

If n is even, divide it by 2

If n is odd, multiply it by 3 and add 1

Continue this process until n is 1
```

1

Example: Hailstone Trees

Pick a positive integer n as the start

If n is even, divide it by 2

If n is odd, multiply it by 3 and add 1

Continue this process until n is 1

1

2

Example: Hailstone Trees

```
Pick a positive integer n as the start

If n is even, divide it by 2

If n is odd, multiply it by 3 and add 1

Continue this process until n is 1
```

1

2

4

Example: Hailstone Trees

```
Pick a positive integer n as the start

If n is even, divide it by 2

If n is odd, multiply it by 3 and add 1

Continue this process until n is 1
```

1

2

4

8

Example: Hailstone Trees

```
Pick a positive integer n as the start

If n is even, divide it by 2

If n is odd, multiply it by 3 and add 1

Continue this process until n is 1
```

1

2

4

8

16

Example: Hailstone Trees

Pick a positive integer n as the start

If n is even, divide it by 2

If n is odd, multiply it by 3 and add 1

Continue this process until n is 1

1

2

4

8

16

32

Example: Hailstone Trees

```
Pick a positive integer n as the start

If n is even, divide it by 2

If n is odd, multiply it by 3 and add 1

Continue this process until n is 1
```

```
1
2
4
8
16
32
64
```

Example: Hailstone Trees

```
Pick a positive integer n as the start

If n is even, divide it by 2

If n is odd, multiply it by 3 and add 1

Continue this process until n is 1
```

1

2

4

8

16

32

64

128

Example: Hailstone Trees

```
Pick a positive integer n as the start

If n is even, divide it by 2

If n is odd, multiply it by 3 and add 1

Continue this process until n is 1
```

```
  1
  |
  2
  |
  4
  |
  8
  |
 16
  |
 32
  |
 64
  |
128
```

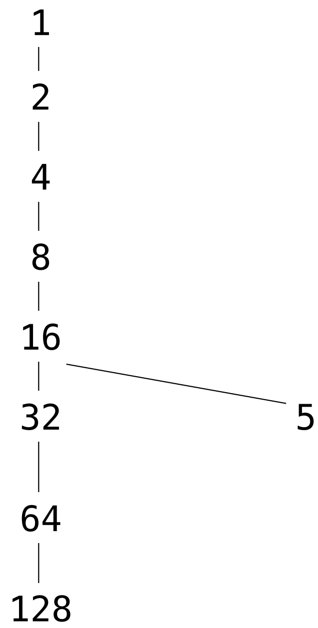
Example: Hailstone Trees

```
Pick a positive integer n as the start

If n is even, divide it by 2

If n is odd, multiply it by 3 and add 1

Continue this process until n is 1
```



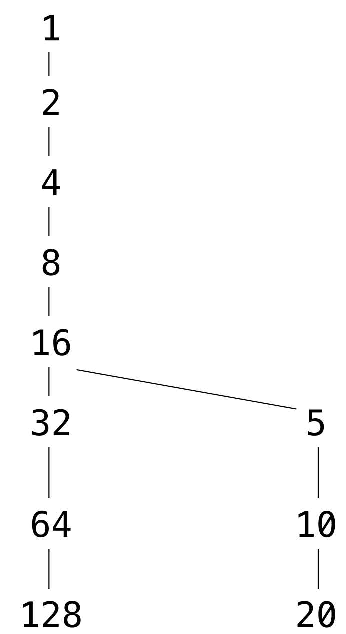
Example: Hailstone Trees

```
Pick a positive integer n as the start

If n is even, divide it by 2

If n is odd, multiply it by 3 and add 1

Continue this process until n is 1
```



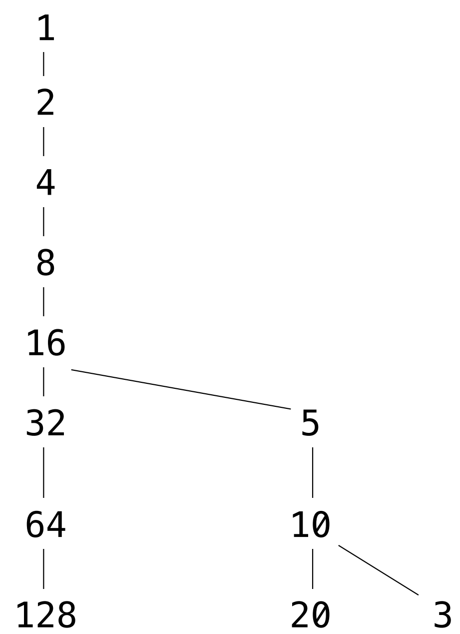
Example: Hailstone Trees

Pick a positive integer n as the start

If n is even, divide it by 2

If n is odd, multiply it by 3 and add 1

Continue this process until n is 1



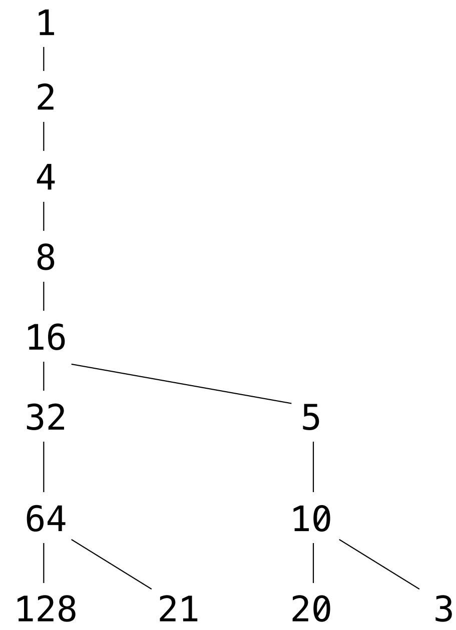
Example: Hailstone Trees

`Pick a positive integer n as the start`

`If n is even, divide it by 2`

`If n is odd, multiply it by 3 and add 1`

`Continue this process until n is 1`



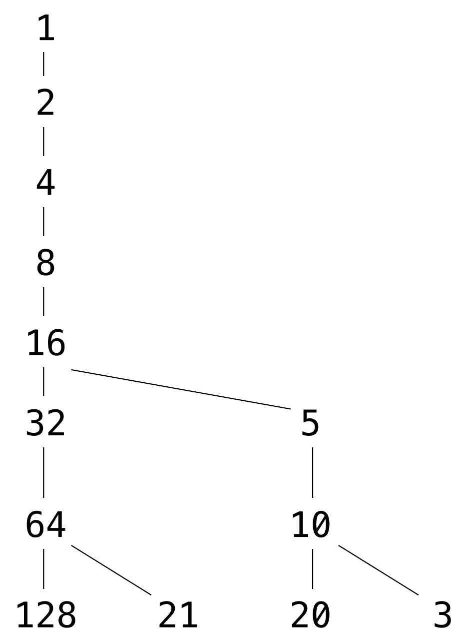
Example: Hailstone Trees

```
Pick a positive integer n as the start

If n is even, divide it by 2

If n is odd, multiply it by 3 and add 1

Continue this process until n is 1
```




```
All starting n that give an
8-number-long hailstone sequence
```

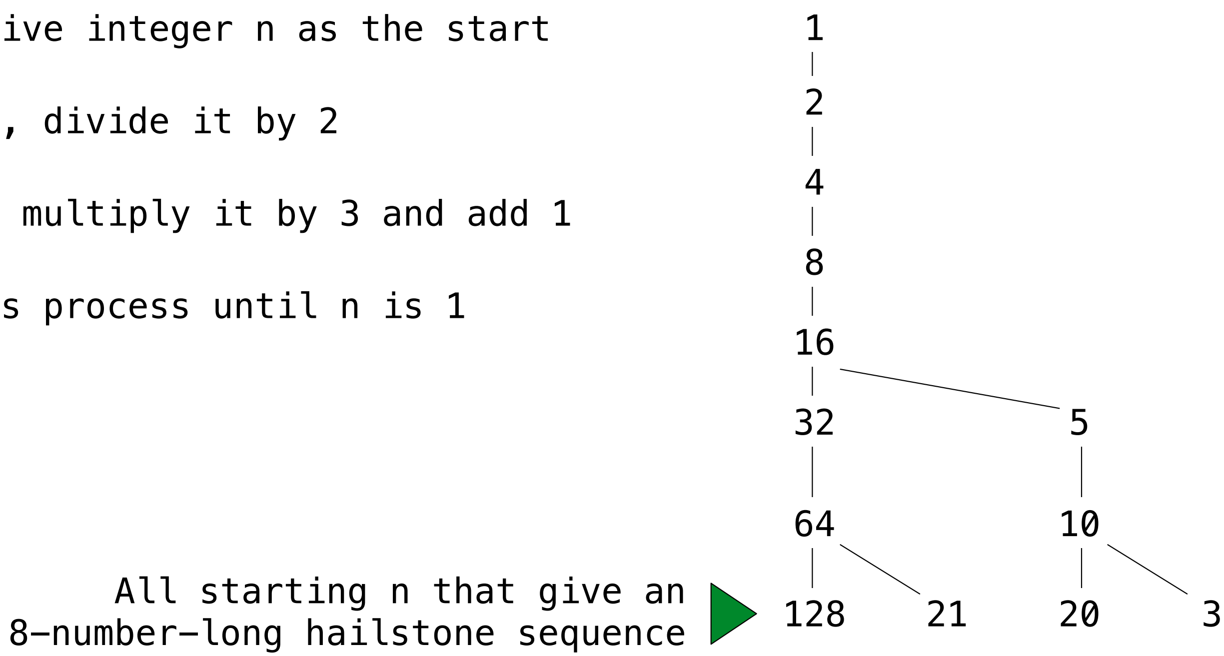
Example: Hailstone Trees

```
Pick a positive integer n as the start

If n is even, divide it by 2

If n is odd, multiply it by 3 and add 1

Continue this process until n is 1
```



(Demo)