61A Lecture 17

Monday, October 14

Announcements

Announcements

- Homework 5 is due Tuesday 10/15 @ 11:59pm

Announcements

- Homework 5 is due Tuesday 10/15 @ 11:59pm
- Project 3 is due Thursday 10/24 @ 11:59pm

Announcements

- Homework 5 is due Tuesday 10/15 @ 11:59pm
- Project 3 is due Thursday 10/24 @ 11:59pm
- Midterm 2 is on Monday 10/28 7pm–9pm

Special Method Names

Special Method Names in Python

Special Method Names in Python

Certain names are special (or "magic") because they have built-in behavior.

Special Method Names in Python

Certain names are special (or "magic") because they have built-in behavior.

These names always start and end with two underscores.

Special Method Names in Python

Certain names are special (or "magic") because they have built-in behavior.

These names always start and end with two underscores.

`__init__`

Special Method Names in Python

Certain names are special (or "magic") because they have built-in behavior.

These names always start and end with two underscores.

`__init__`	Method invoked automatically when an object is constructed.

Special Method Names in Python

Certain names are special (or "magic") because they have built-in behavior.

These names always start and end with two underscores.

`__init__`	Method invoked automatically when an object is constructed.
`__len__`	

Special Method Names in Python

Certain names are special (or "magic") because they have built-in behavior.

These names always start and end with two underscores.

__init__	Method invoked automatically when an object is constructed.
__len__	Method invoked by the built-in len function.

Special Method Names in Python

Certain names are special (or "magic") because they have built-in behavior.

These names always start and end with two underscores.

`__init__`	Method invoked automatically when an object is constructed.
`__len__`	Method invoked by the built-in len function.

```
>>> s = (3, 4, 5)
```

```
>>> s = (3, 4, 5)
```

Special Method Names in Python

Certain names are special (or "magic") because they have built-in behavior.

These names always start and end with two underscores.

`__init__`	Method invoked automatically when an object is constructed.
`__len__`	Method invoked by the built-in len function.

```
>>> s = (3, 4, 5)
>>> len(s)
3
```

```
>>> s = (3, 4, 5)
```

Special Method Names in Python

Certain names are special (or "magic") because they have built-in behavior.

These names always start and end with two underscores.

__init__	Method invoked automatically when an object is constructed.
__len__	Method invoked by the built-in len function.

```
>>> s = (3, 4, 5)
>>> len(s)
3
```

Same behavior using methods

```
>>> s = (3, 4, 5)
```

Special Method Names in Python

Certain names are special (or "magic") because they have built-in behavior.

These names always start and end with two underscores.

__init__	Method invoked automatically when an object is constructed.
__len__	Method invoked by the built-in len function.

```
>>> s = (3, 4, 5)
>>> len(s)
3
```

Same behavior using methods

```
>>> s = (3, 4, 5)
>>> s.__len__()
3
```

Special Method Names in Python

Certain names are special (or "magic") because they have built-in behavior.

These names always start and end with two underscores.

`__init__`	Method invoked automatically when an object is constructed.
`__len__`	Method invoked by the built-in len function.
`__getitem__`	

```
>>> s = (3, 4, 5)
>>> len(s)
3
```

Same behavior using methods

```
>>> s = (3, 4, 5)
>>> s.__len__()
3
```

Special Method Names in Python

Certain names are special (or "magic") because they have built-in behavior.

These names always start and end with two underscores.

__init__	Method invoked automatically when an object is constructed.
__len__	Method invoked by the built-in len function.
__getitem__	Method invoked for element selection: sequence[index]

```
>>> s = (3, 4, 5)
>>> len(s)
3
```

Same behavior using methods

```
>>> s = (3, 4, 5)
>>> s.__len__()
3
```

Special Method Names in Python

Certain names are special (or "magic") because they have built-in behavior.

These names always start and end with two underscores.

__init__	Method invoked automatically when an object is constructed.
__len__	Method invoked by the built-in len function.
__getitem__	Method invoked for element selection: sequence[index]

```
>>> s = (3, 4, 5)
>>> len(s)
3
>>> s[2]
5
```

Same behavior using methods

```
>>> s = (3, 4, 5)
>>> s.__len__()
3
```

Special Method Names in Python

Certain names are special (or "magic") because they have built-in behavior.

These names always start and end with two underscores.

`__init__`	Method invoked automatically when an object is constructed.
`__len__`	Method invoked by the built-in len function.
`__getitem__`	Method invoked for element selection: sequence[index]

```
>>> s = (3, 4, 5)
>>> len(s)
3
>>> s[2]
5
```

Same behavior using methods

```
>>> s = (3, 4, 5)
>>> s.__len__()
3
>>> s.__getitem__(2)
5
```

Special Method Names in Python

Certain names are special (or "magic") because they have built-in behavior.

These names always start and end with two underscores.

`__init__`	Method invoked automatically when an object is constructed.
`__len__`	Method invoked by the built-in len function.
`__getitem__`	Method invoked for element selection: sequence[index]
`__repr__`	

```
>>> s = (3, 4, 5)
>>> len(s)
3
>>> s[2]
5
```

Same behavior using methods

```
>>> s = (3, 4, 5)
>>> s.__len__()
3
>>> s.__getitem__(2)
5
```

Special Method Names in Python

Certain names are special (or "magic") because they have built-in behavior.

These names always start and end with two underscores.

`__init__`	Method invoked automatically when an object is constructed.
`__len__`	Method invoked by the built-in len function.
`__getitem__`	Method invoked for element selection: sequence[index]
`__repr__`	Method invoked to display an object as a string.

```
>>> s = (3, 4, 5)
>>> len(s)
3
>>> s[2]
5
```

Same behavior using methods

```
>>> s = (3, 4, 5)
>>> s.__len__()
3
>>> s.__getitem__(2)
5
```

Special Method Names in Python

Certain names are special (or "magic") because they have built-in behavior.

These names always start and end with two underscores.

__init__	Method invoked automatically when an object is constructed.
__len__	Method invoked by the built-in len function.
__getitem__	Method invoked for element selection: sequence[index]
__repr__	Method invoked to display an object as a string.

```
>>> s = (3, 4, 5)
>>> len(s)
3
>>> s[2]
5
>>> s
(3, 4, 5)
```

Same behavior using methods

```
>>> s = (3, 4, 5)
>>> s.__len__()
3
>>> s.__getitem__(2)
5
```

Special Method Names in Python

Certain names are special (or "magic") because they have built-in behavior.

These names always start and end with two underscores.

`__init__`	Method invoked automatically when an object is constructed.
`__len__`	Method invoked by the built-in len function.
`__getitem__`	Method invoked for element selection: sequence[index]
`__repr__`	Method invoked to display an object as a string.

```
>>> s = (3, 4, 5)
>>> len(s)
3
>>> s[2]
5
>>> s
(3, 4, 5)
```

Same behavior using methods

```
>>> s = (3, 4, 5)
>>> s.__len__()
3
>>> s.__getitem__(2)
5
>>> print(s.__repr__())
(3, 4, 5)
```

Recursive List Class

Closure Property of Data

Closure Property of Data

A tuple can contain another tuple as an element.

Closure Property of Data

A tuple can contain another tuple as an element.

Pairs are sufficient to represent sequences of arbitrary length.

Closure Property of Data

A tuple can contain another tuple as an element.

Pairs are sufficient to represent sequences of arbitrary length.

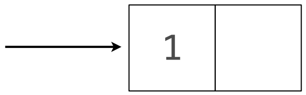
Recursive list representation of the sequence 1, 2, 3, 4:

Closure Property of Data

A tuple can contain another tuple as an element.

Pairs are sufficient to represent sequences of arbitrary length.

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

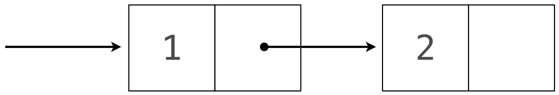


Closure Property of Data

A tuple can contain another tuple as an element.

Pairs are sufficient to represent sequences of arbitrary length.

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

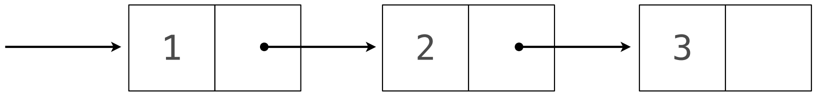


Closure Property of Data

A tuple can contain another tuple as an element.

Pairs are sufficient to represent sequences of arbitrary length.

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

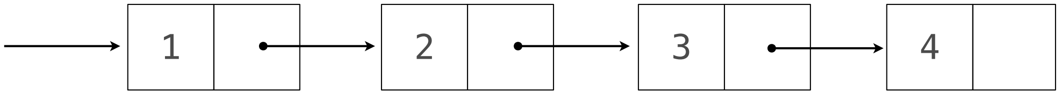


Closure Property of Data

A tuple can contain another tuple as an element.

Pairs are sufficient to represent sequences of arbitrary length.

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

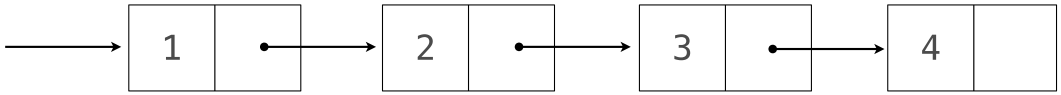


Closure Property of Data

A tuple can contain another tuple as an element.

Pairs are sufficient to represent sequences of arbitrary length.

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



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

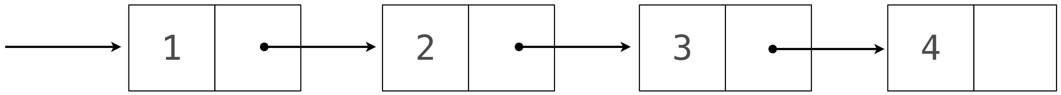
Now, we can implement the same behavior using a class called Rlist:

Closure Property of Data

A tuple can contain another tuple as an element.

Pairs are sufficient to represent sequences of arbitrary length.

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



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

Now, we can implement the same behavior using a class called Rlist:

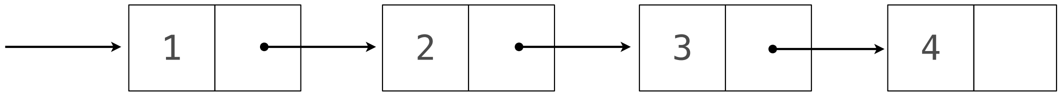
Abstract data type (old):

Closure Property of Data

A tuple can contain another tuple as an element.

Pairs are sufficient to represent sequences of arbitrary length.

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



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

Now, we can implement the same behavior using a class called Rlist:

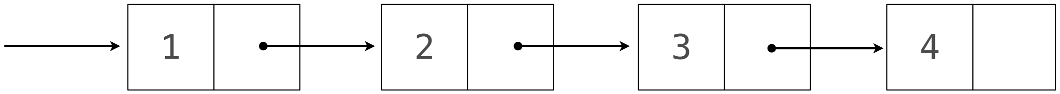
Abstract data type (old): `rlist(1, rlist(2, rlist(3, rlist(4, empty_rlist))))`

Closure Property of Data

A tuple can contain another tuple as an element.

Pairs are sufficient to represent sequences of arbitrary length.

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



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

Now, we can implement the same behavior using a class called Rlist:

Abstract data type (old): `rlist(1, rlist(2, rlist(3, rlist(4, empty_rlist))))`

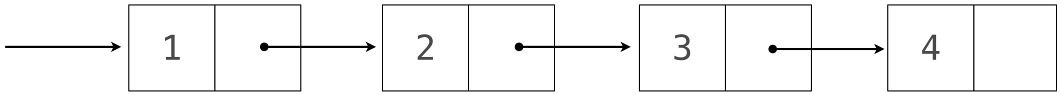
Rlist class (new):

Closure Property of Data

A tuple can contain another tuple as an element.

Pairs are sufficient to represent sequences of arbitrary length.

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



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

Now, we can implement the same behavior using a class called Rlist:

Abstract data type (old): `rlist(1, rlist(2, rlist(3, rlist(4, empty_rlist))))`

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

Recursive List Class

Recursive List Class

```
class Rlist:
```

Recursive List Class

```
class Rlist:
    class EmptyList:
        def __len__(self):
            return 0

    empty = EmptyList()
```

Recursive List Class

```
class Rlist:
    class EmptyList:
        def __len__(self):
            return 0

    empty = EmptyList()

    def __init__(self, first, rest=empty):
        assert type(rest) is Rlist or rest is Rlist.empty
        self.first = first
        self.rest = rest
```

Recursive List Class

```
class Rlist:
    class EmptyList:
        def __len__(self):
            return 0

    empty = EmptyList()

    def __init__(self, first, rest=empty):
        assert type(rest) is Rlist or rest is Rlist.empty
        self.first = first
        self.rest = rest

    def __getitem__(self, index):
```

Recursive List Class

```
class Rlist:
    class EmptyList:
        def __len__(self):
            return 0

    empty = EmptyList()

    def __init__(self, first, rest=empty):
        assert type(rest) is Rlist or rest is Rlist.empty
        self.first = first
        self.rest = rest

    def __getitem__(self, index):
        if index == 0:
```

Recursive List Class

```
class Rlist:
    class EmptyList:
        def __len__(self):
            return 0

    empty = EmptyList()

    def __init__(self, first, rest=empty):
        assert type(rest) is Rlist or rest is Rlist.empty
        self.first = first
        self.rest = rest

    def __getitem__(self, index):
        if index == 0:
            return self.first
```

Recursive List Class

```
class Rlist:
    class EmptyList:
        def __len__(self):
            return 0

    empty = EmptyList()

    def __init__(self, first, rest=empty):
        assert type(rest) is Rlist or rest is Rlist.empty
        self.first = first
        self.rest = rest

    def __getitem__(self, index):
        if index == 0:
            return self.first
        else:
```

Recursive List Class

```
class Rlist:
    class EmptyList:
        def __len__(self):
            return 0

    empty = EmptyList()

    def __init__(self, first, rest=empty):
        assert type(rest) is Rlist or rest is Rlist.empty
        self.first = first
        self.rest = rest

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

Recursive List Class

```
class Rlist:
    class EmptyList:
        def __len__(self):
            return 0

    empty = EmptyList()

    def __init__(self, first, rest=empty):
        assert type(rest) is Rlist or rest is Rlist.empty
        self.first = first
        self.rest = rest

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

This element selection syntax

Recursive List Class

```
class Rlist:
    class EmptyList:
        def __len__(self):
            return 0

    empty = EmptyList()

    def __init__(self, first, rest=empty):
        assert type(rest) is Rlist or rest is Rlist.empty
        self.first = first
        self.rest = rest

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

Calls this method with a special name

This element selection syntax

Recursive List Class

```
class Rlist:
    class EmptyList:
        def __len__(self):
            return 0

    empty = EmptyList()

    def __init__(self, first, rest=empty):
        assert type(rest) is Rlist or rest is Rlist.empty
        self.first = first
        self.rest = rest

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

    def __len__(self):
```

Calls this method with a special name

This element selection syntax

Recursive List Class

```
class Rlist:
    class EmptyList:
        def __len__(self):
            return 0

    empty = EmptyList()

    def __init__(self, first, rest=empty):
        assert type(rest) is Rlist or rest is Rlist.empty
        self.first = first
        self.rest = rest

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

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

Calls this method with a special name

This element selection syntax

Recursive List Class

```
class Rlist:
    class EmptyList:
        def __len__(self):
            return 0

    empty = EmptyList()

    def __init__(self, first, rest=empty):
        assert type(rest) is Rlist or rest is Rlist.empty
        self.first = first
        self.rest = rest

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

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

Calls this method with a special name

This element selection syntax

Yes, this call is recursive

Recursive List Class

```
class Rlist:
    class EmptyList:
        def __len__(self):
            return 0
    empty = EmptyList()

    def __init__(self, first, rest=empty):
        assert type(rest) is Rlist or rest is Rlist.empty
        self.first = first
        self.rest = rest

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

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

There's the base case!

Calls this method with a special name

This element selection syntax

Yes, this call is recursive

Recursive List Class

```
class Rlist:
    class EmptyList:
        def __len__(self):
            return 0
    empty = EmptyList()

    def __init__(self, first, rest=empty):
        assert type(rest) is Rlist or rest is Rlist.empty
        self.first = first
        self.rest = rest

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

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

There's the base case!

Calls this method with a special name

This element selection syntax

Yes, this call is recursive

Methods can be recursive too!

(Demo)

Recursive List Processing

Recursive Operations on Recursive Lists

Recursive Operations on Recursive Lists

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

Recursive Operations on Recursive Lists

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

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

Recursive Operations on Recursive Lists

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

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

Recursive Operations on Recursive Lists

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

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

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

Recursive Operations on Recursive Lists

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

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

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

>>> extend_rlist(s.rest, s)
```

Recursive Operations on Recursive Lists

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

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

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

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

Recursive Operations on Recursive Lists

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

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

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

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

```
def extend_rlist(s1, s2):
```

Recursive Operations on Recursive Lists

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

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

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

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

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

Recursive Operations on Recursive Lists

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

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

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

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

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

Recursive Operations on Recursive Lists

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

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

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

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

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

Recursive Operations on Recursive Lists

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

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

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

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

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

Higher-Order Functions on Recursive Lists

Higher-Order Functions on Recursive Lists

`We want operations on all elements of a list, not just an element at a time.`

Higher-Order Functions on Recursive Lists

We want operations on all elements of a list, not just an element at a time.

```
double_rlist(s)
```

Higher-Order Functions on Recursive Lists

We want operations on all elements of a list, not just an element at a time.

`double_rlist(s)`   Double `s.first`, then `double_rlist(s.rest)`

Higher-Order Functions on Recursive Lists

We want operations on all elements of a list, not just an element at a time.

`double_rlist(s)`   Double `s.first`, then `double_rlist(s.rest)`

`map_rlist(s, fn)`

Higher-Order Functions on Recursive Lists

We want operations on all elements of a list, not just an element at a time.

`double_rlist(s)`	Double `s.first`, then `double_rlist(s.rest)`
`map_rlist(s, fn)`	Apply `fn` to `s.first`, then `map_rlist(s.rest, fn)`

Higher-Order Functions on Recursive Lists

We want operations on all elements of a list, not just an element at a time.

`double_rlist(s)`	Double `s.first`, then `double_rlist(s.rest)`
`map_rlist(s, fn)`	Apply `fn` to `s.first`, then `map_rlist(s.rest, fn)`
`filter_rlist(s, fn)`	

Higher-Order Functions on Recursive Lists

`We want operations on all elements of a list, not just an element at a time.`

`double_rlist(s)`	`Double s.first, then double_rlist(s.rest)`
`map_rlist(s, fn)`	`Apply fn to s.first, then map_rlist(s.rest, fn)`
`filter_rlist(s, fn)`	`Either keep s.first or not, then filter_rlist(s.rest, fn)`

Higher-Order Functions on Recursive Lists

We want operations on all elements of a list, not just an element at a time.

`double_rlist(s)`	Double `s.first`, then `double_rlist(s.rest)`
`map_rlist(s, fn)`	Apply `fn` to `s.first`, then `map_rlist(s.rest, fn)`
`filter_rlist(s, fn)`	Either keep `s.first` or not, then `filter_rlist(s.rest, fn)`

In all of these functions, the base case is the empty list.

Higher-Order Functions on Recursive Lists

We want operations on all elements of a list, not just an element at a time.

`double_rlist(s)`	Double `s.first`, then `double_rlist(s.rest)`
`map_rlist(s, fn)`	Apply `fn` to `s.first`, then `map_rlist(s.rest, fn)`
`filter_rlist(s, fn)`	Either keep `s.first` or not, then `filter_rlist(s.rest, fn)`

In all of these functions, the base case is the empty list.

(Demo)

Trees

Tree Structured Data

Tree Structured Data

Nested sequences form hierarchical structures: tree-structured data

Tree Structured Data

Nested sequences form hierarchical structures: tree-structured data

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

Tree Structured Data

Nested sequences form hierarchical structures: tree-structured data

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

Tree Structured Data

Nested sequences form hierarchical structures: tree-structured data

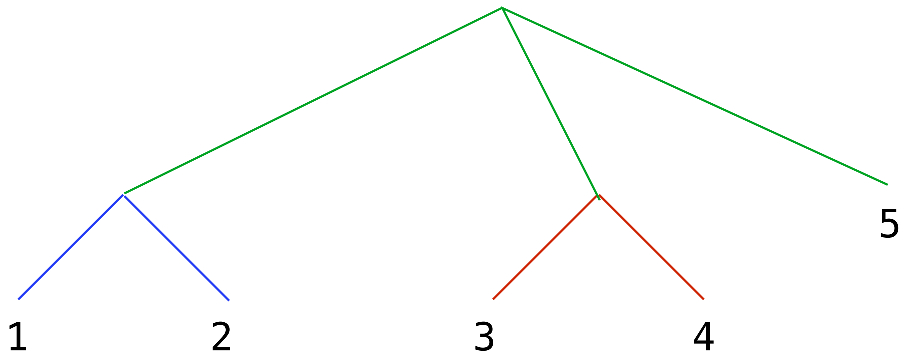
((1, 2), (3, 4), 5)

In every tree, a vast forest

Tree Structured Data

Nested sequences form hierarchical structures: tree-structured data

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

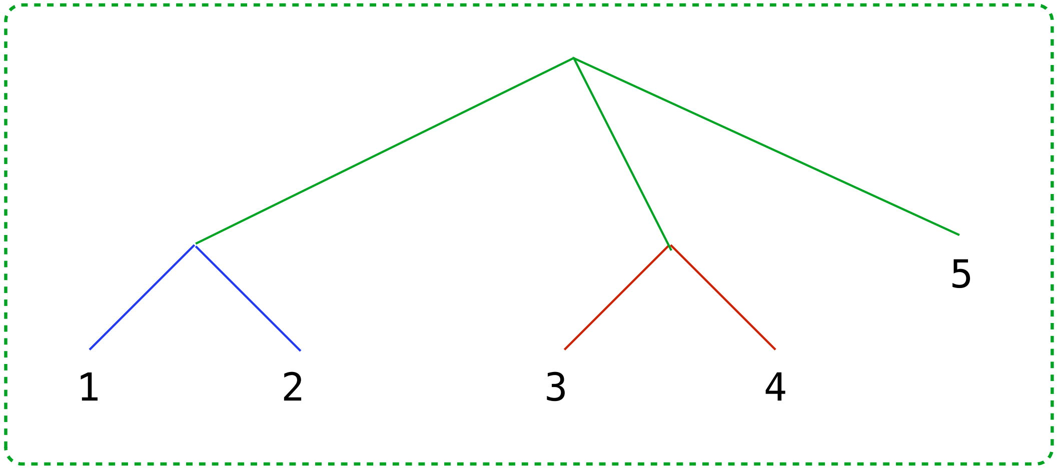


In every tree, a vast forest

Tree Structured Data

`Nested sequences form hierarchical structures: tree-structured data`

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

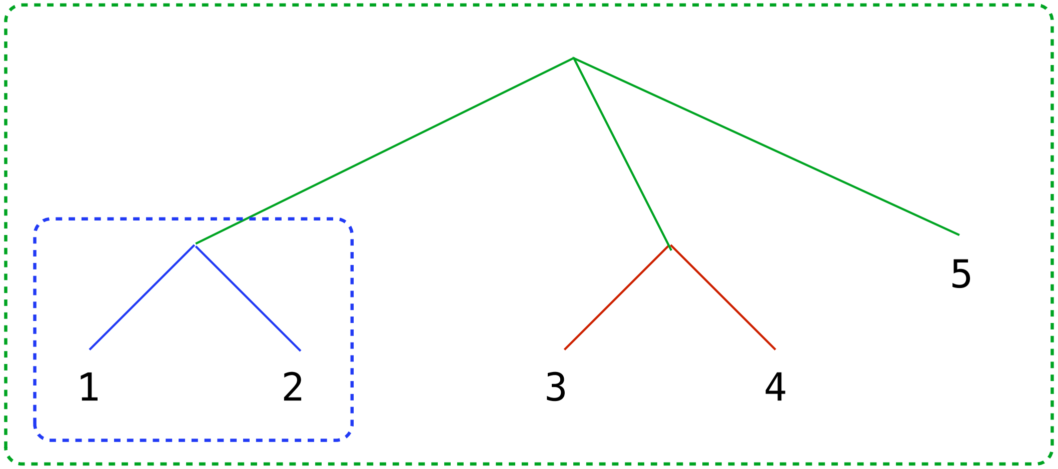


`In every tree, a vast forest`

Tree Structured Data

`Nested sequences form hierarchical structures: tree-structured data`

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

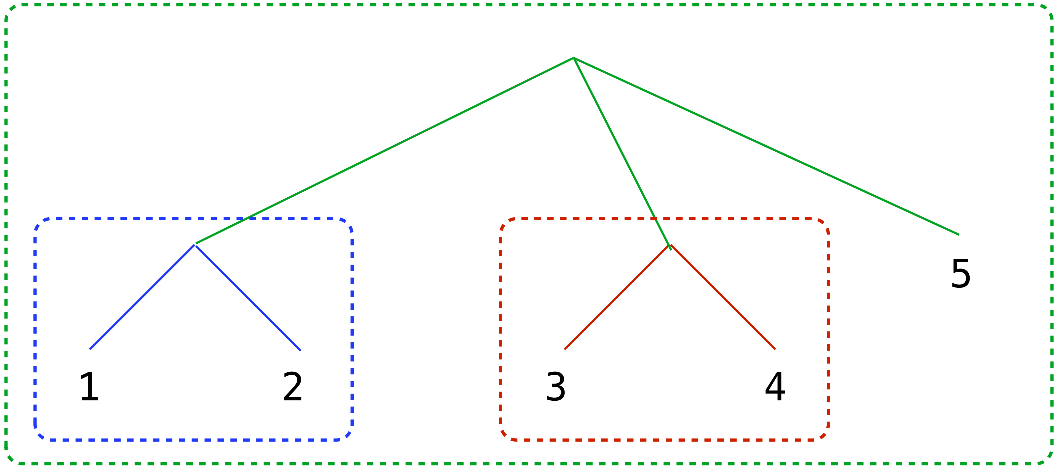


`In every tree, a vast forest`

Tree Structured Data

Nested sequences form hierarchical structures: tree-structured data

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

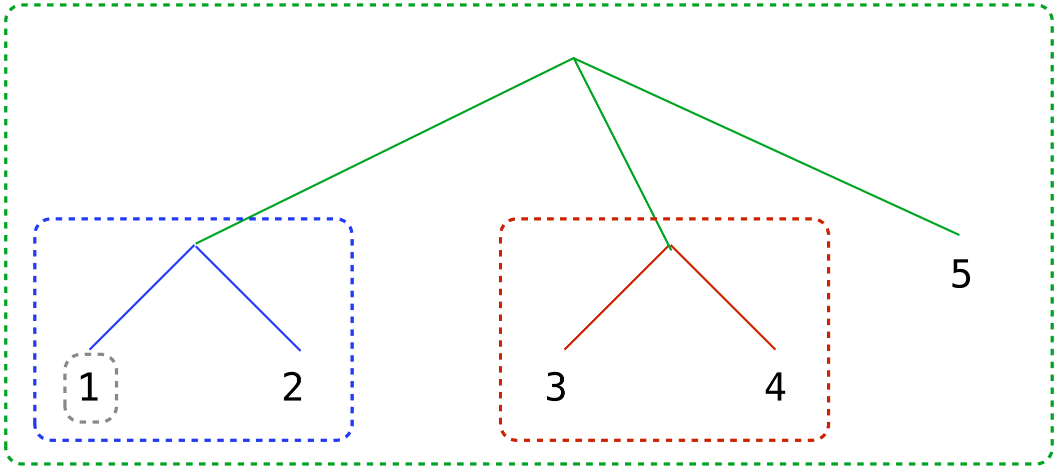


In every tree, a vast forest

Tree Structured Data

Nested sequences form hierarchical structures: tree-structured data

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

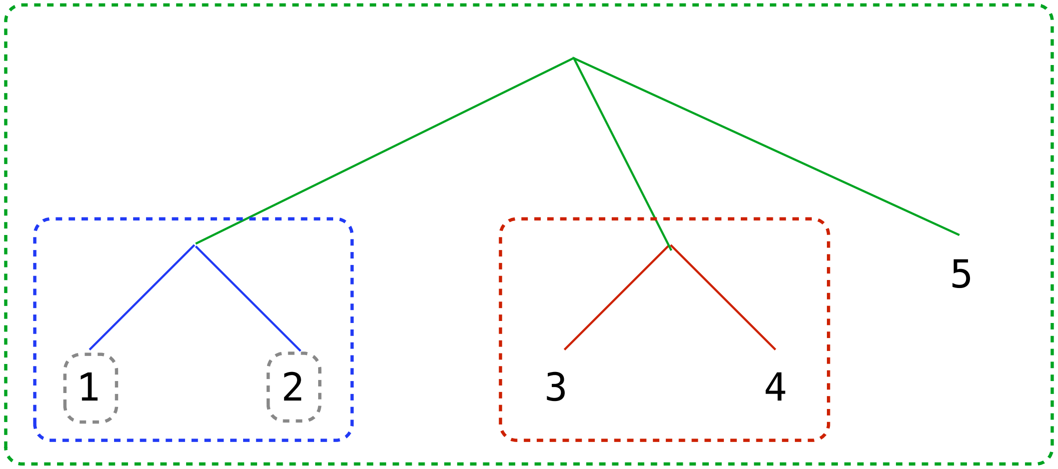


In every tree, a vast forest

Tree Structured Data

`Nested sequences form hierarchical structures: tree-structured data`

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

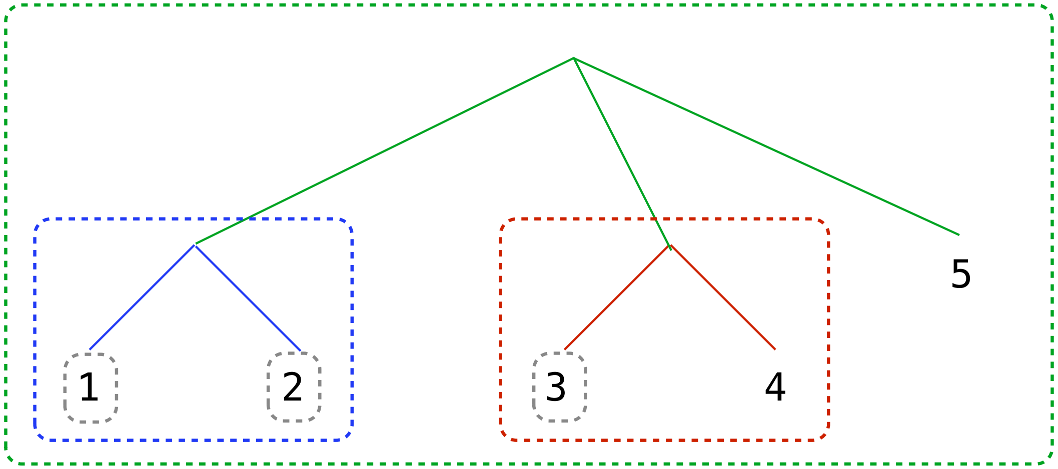


`In every tree, a vast forest`

Tree Structured Data

`Nested sequences form hierarchical structures: tree-structured data`

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

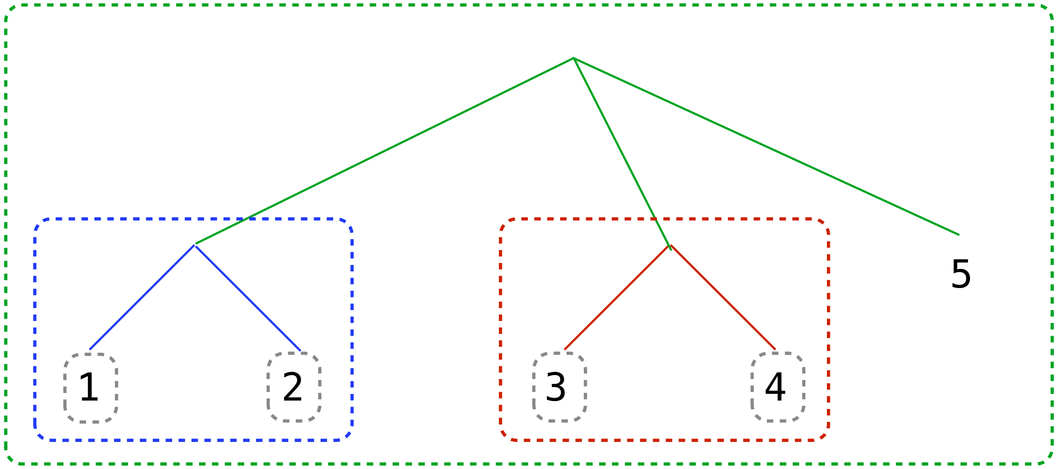


`In every tree, a vast forest`

Tree Structured Data

Nested sequences form hierarchical structures: tree-structured data

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

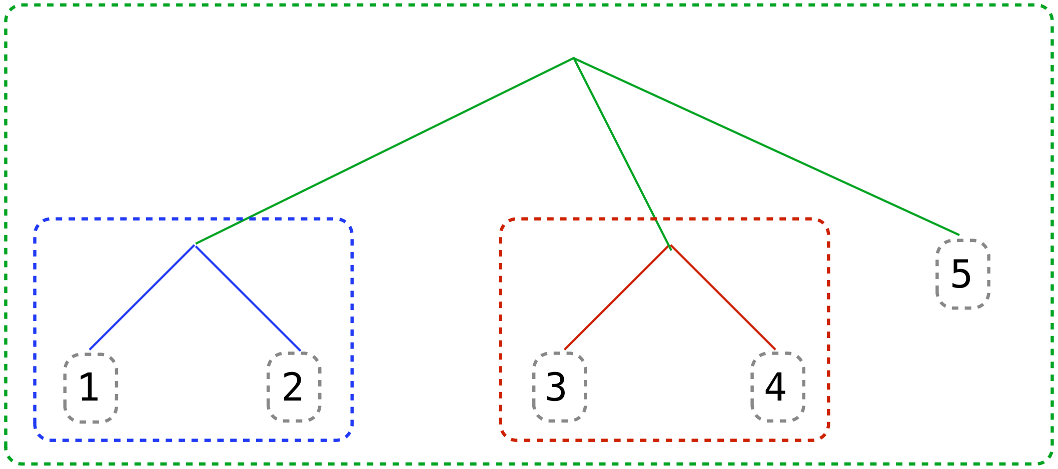


In every tree, a vast forest

Tree Structured Data

Nested sequences form hierarchical structures: tree-structured data

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



In every tree, a vast forest

Recursive Tree Processing

Tree operations typically make recursive calls on branches.

Recursive Tree Processing

Tree operations typically make recursive calls on branches.

```
count_leaves(t)
```

Recursive Tree Processing

Tree operations typically make recursive calls on branches.

`count_leaves(t)` **1** if t is a leaf, otherwise sum `count_leaves(branch)`

Recursive Tree Processing

Tree operations typically make recursive calls on branches.

`count_leaves(t)` **1** if t is a leaf, otherwise sum `count_leaves(branch)`

`map_tree(t, fn)`

Recursive Tree Processing

Tree operations typically make recursive calls on branches.

count_leaves(t)	**1** if t is a leaf, otherwise sum count_leaves(branch)
map_tree(t, fn)	**fn(t)** if t is a leaf, otherwise combine map_tree(branch, fn)

Recursive Tree Processing

Tree operations typically make recursive calls on branches.

count_leaves(t)	**1** if t is a leaf, otherwise sum count_leaves(branch)
map_tree(t, fn)	**fn(t)** if t is a leaf, otherwise combine map_tree(branch, fn)

In these functions, the base case is a leaf.

Recursive Tree Processing

Tree operations typically make recursive calls on branches.

`count_leaves(t)`	**1** if t is a leaf, otherwise sum `count_leaves(branch)`
`map_tree(t, fn)`	**fn(t)** if t is a leaf, otherwise combine `map_tree(branch, fn)`

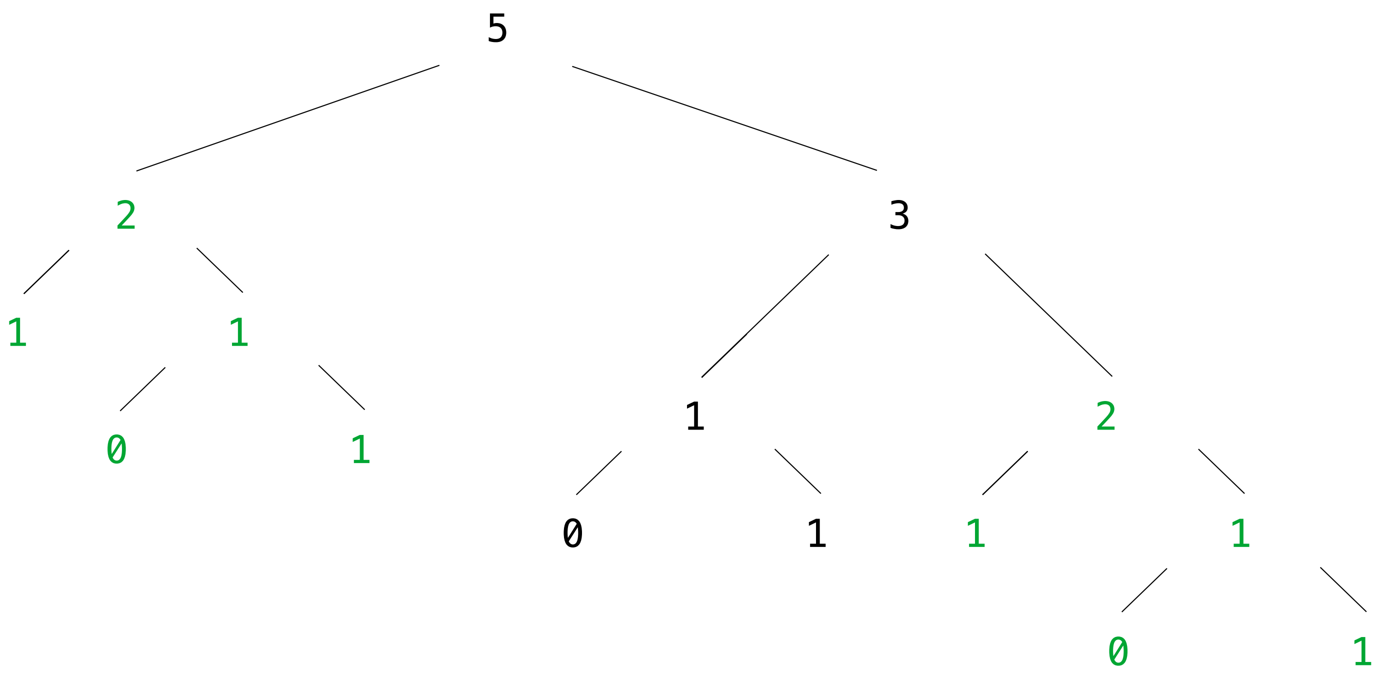
In these functions, the base case is a leaf.

(Demo)

Trees with Internal Entries

Trees with Internal Entries

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



Trees with Internal Entries

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

Trees with Internal Entries

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

```
class Tree:
```

Trees with Internal Entries

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

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

Trees with Internal Entries

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

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

Trees with Internal Entries

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

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

Trees with Internal Entries

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

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

Trees with Internal Entries

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

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

def fib_tree(n):
```

Trees with Internal Entries

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

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

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

Trees with Internal Entries

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

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

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

Trees with Internal Entries

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

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

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

Trees with Internal Entries

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

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

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

Trees with Internal Entries

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

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

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

Trees with Internal Entries

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

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

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

Trees with Internal Entries

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

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

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

Trees with Internal Entries

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

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

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

(Demo)

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

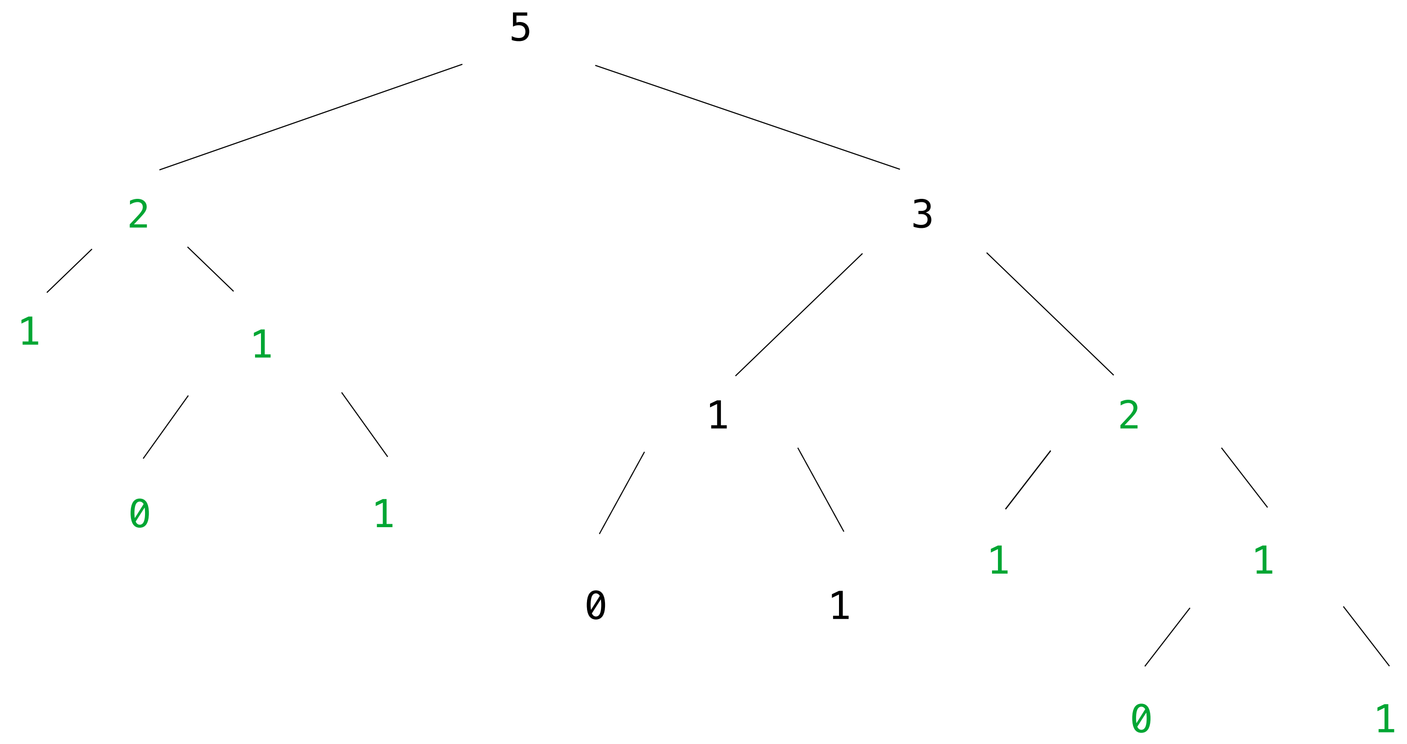
```
def memo(f):
    cache = {}
    def memoized(n):
        if n not in cache:
            cache[n] = f(n)
        return cache[n]
    return memoized
```

Keys are arguments that map to return values

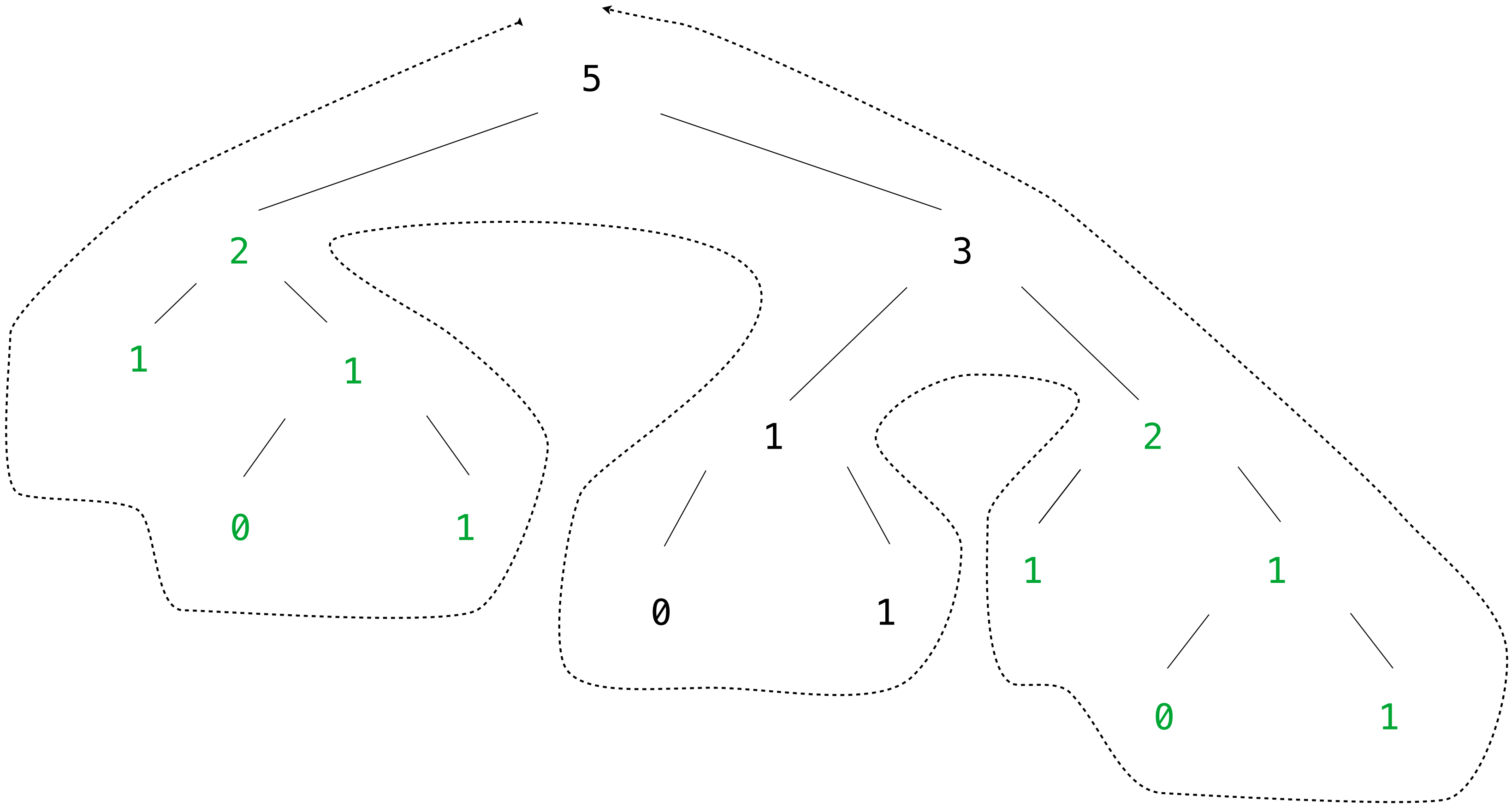
Same behavior as f, if f is a pure function

(Demo)

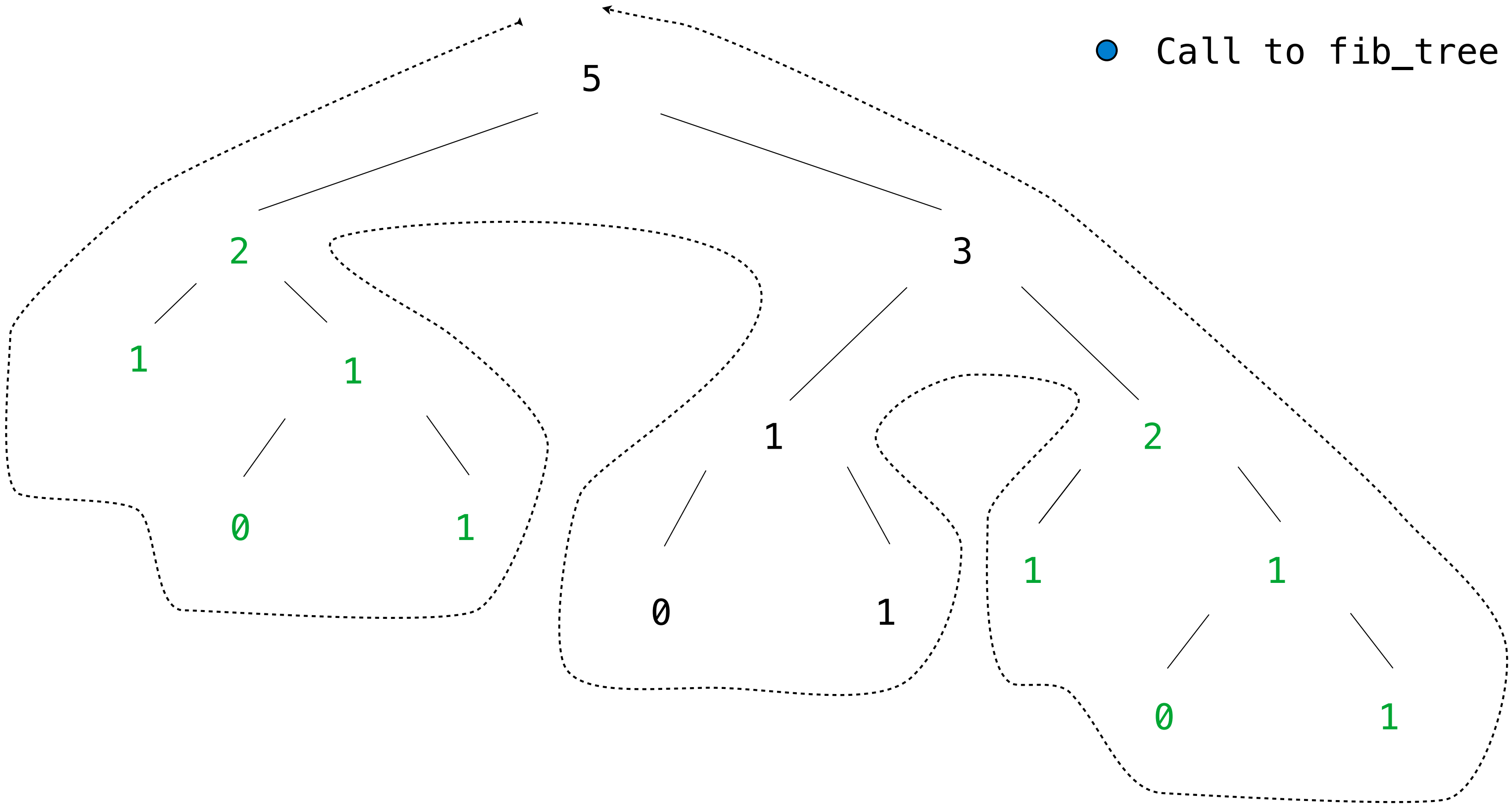
Memoized Tree Recursion



Memoized Tree Recursion

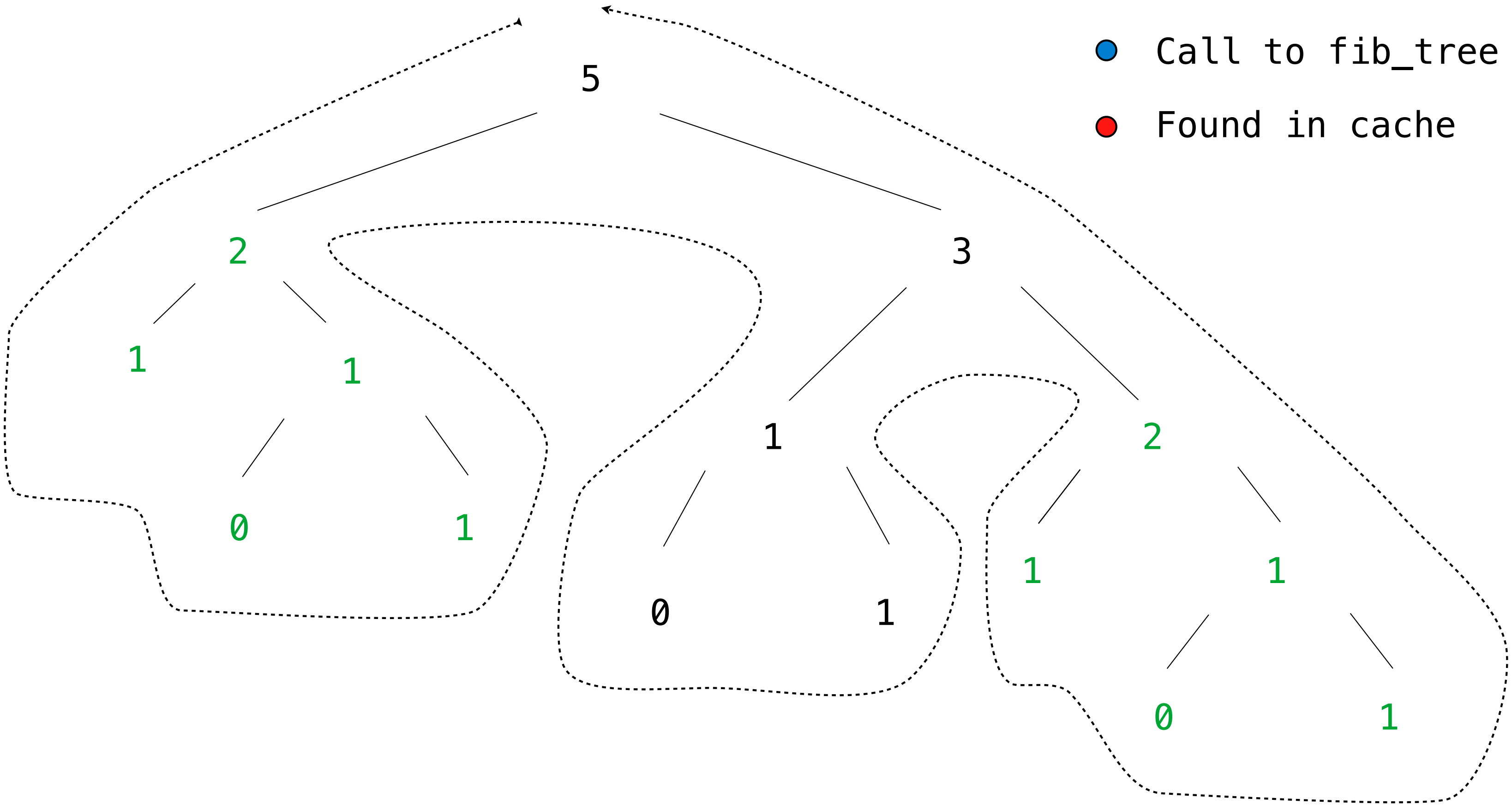


Memoized Tree Recursion



● `Call to fib_tree`

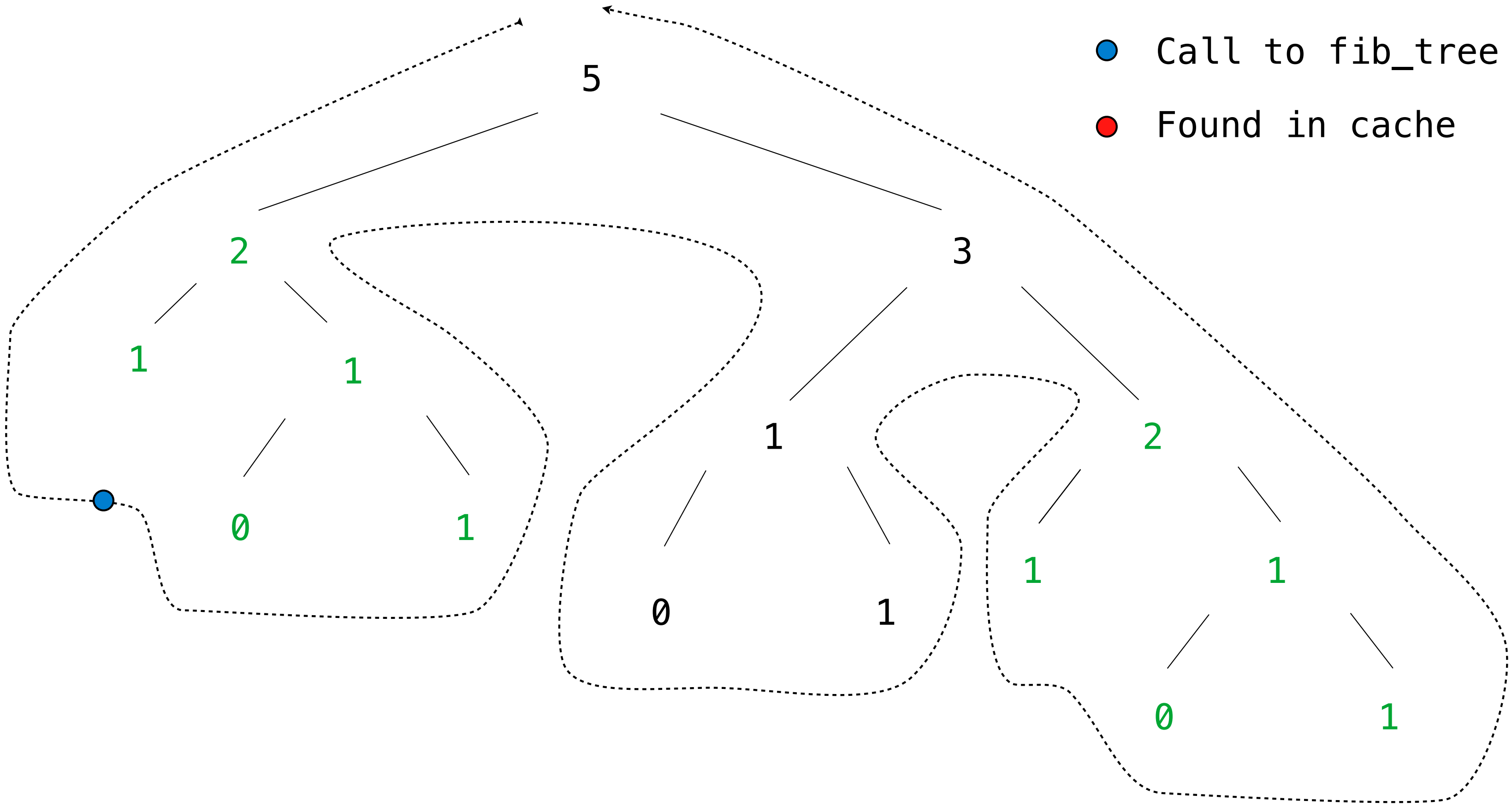
Memoized Tree Recursion



● Call to fib_tree

🔴 Found in cache

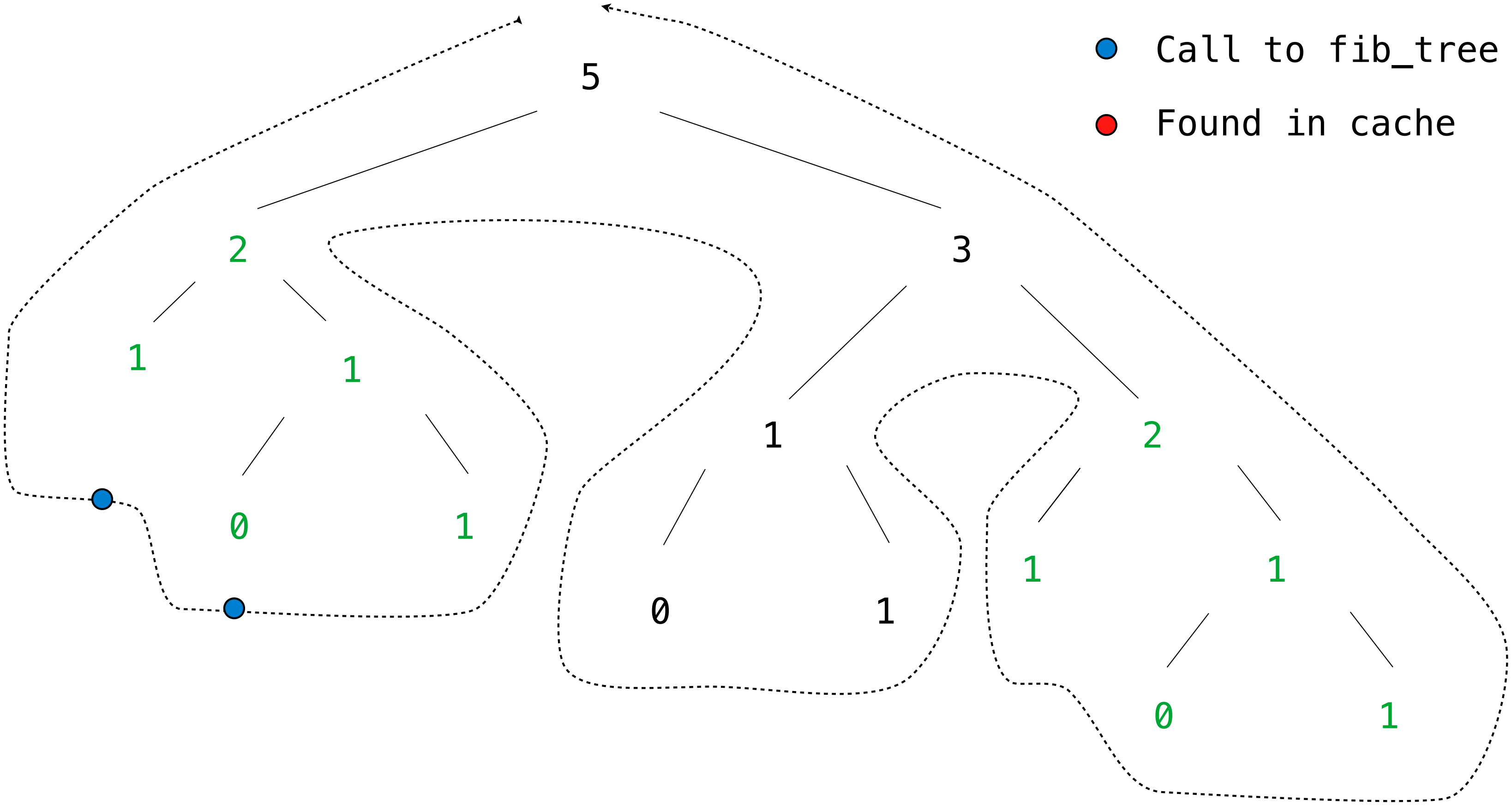
Memoized Tree Recursion



● Call to fib_tree

🟥 Found in cache

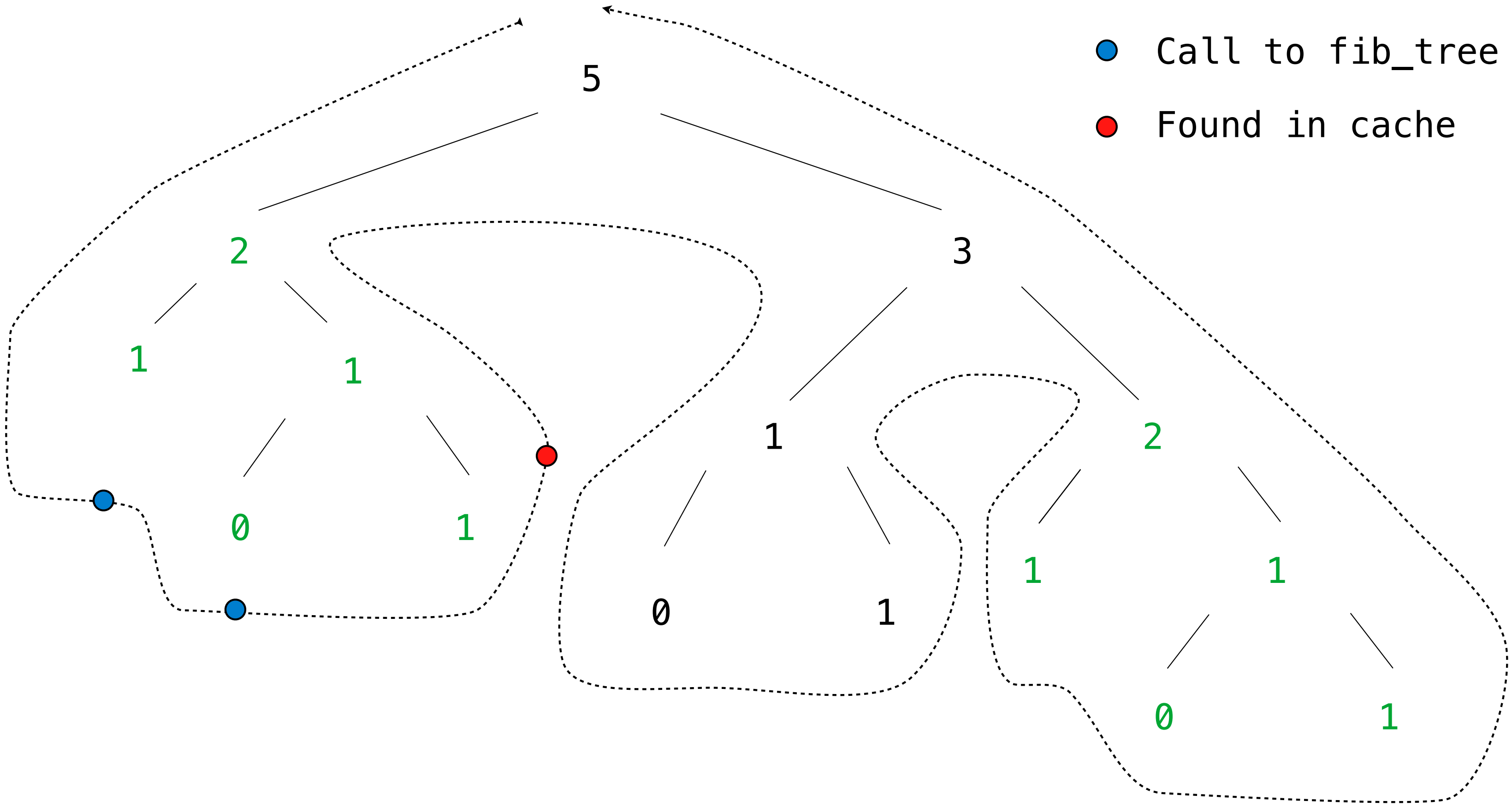
Memoized Tree Recursion



🔵 Call to fib_tree

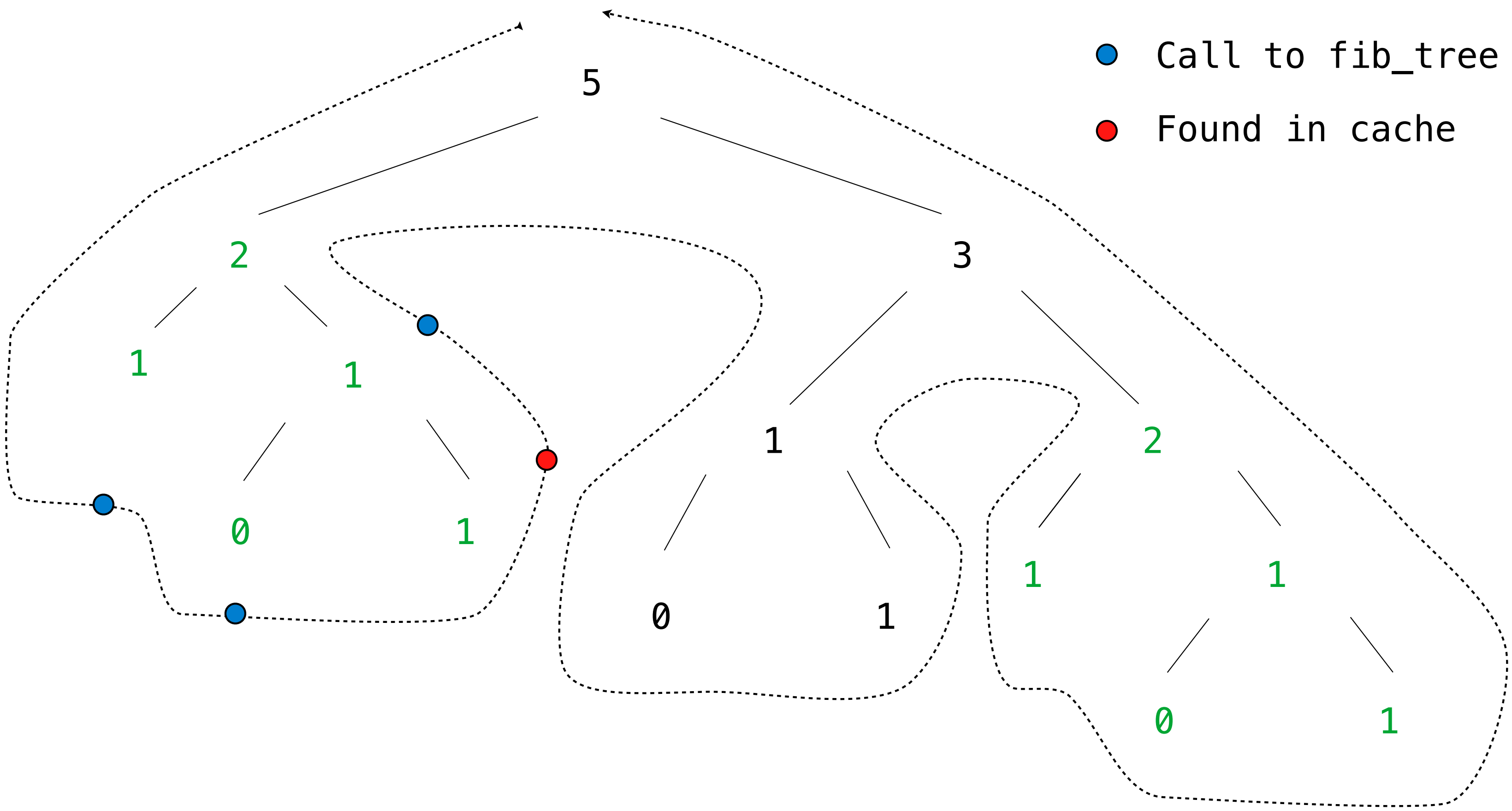
🔴 Found in cache

Memoized Tree Recursion



- 🔵 Call to fib_tree
- 🔴 Found in cache

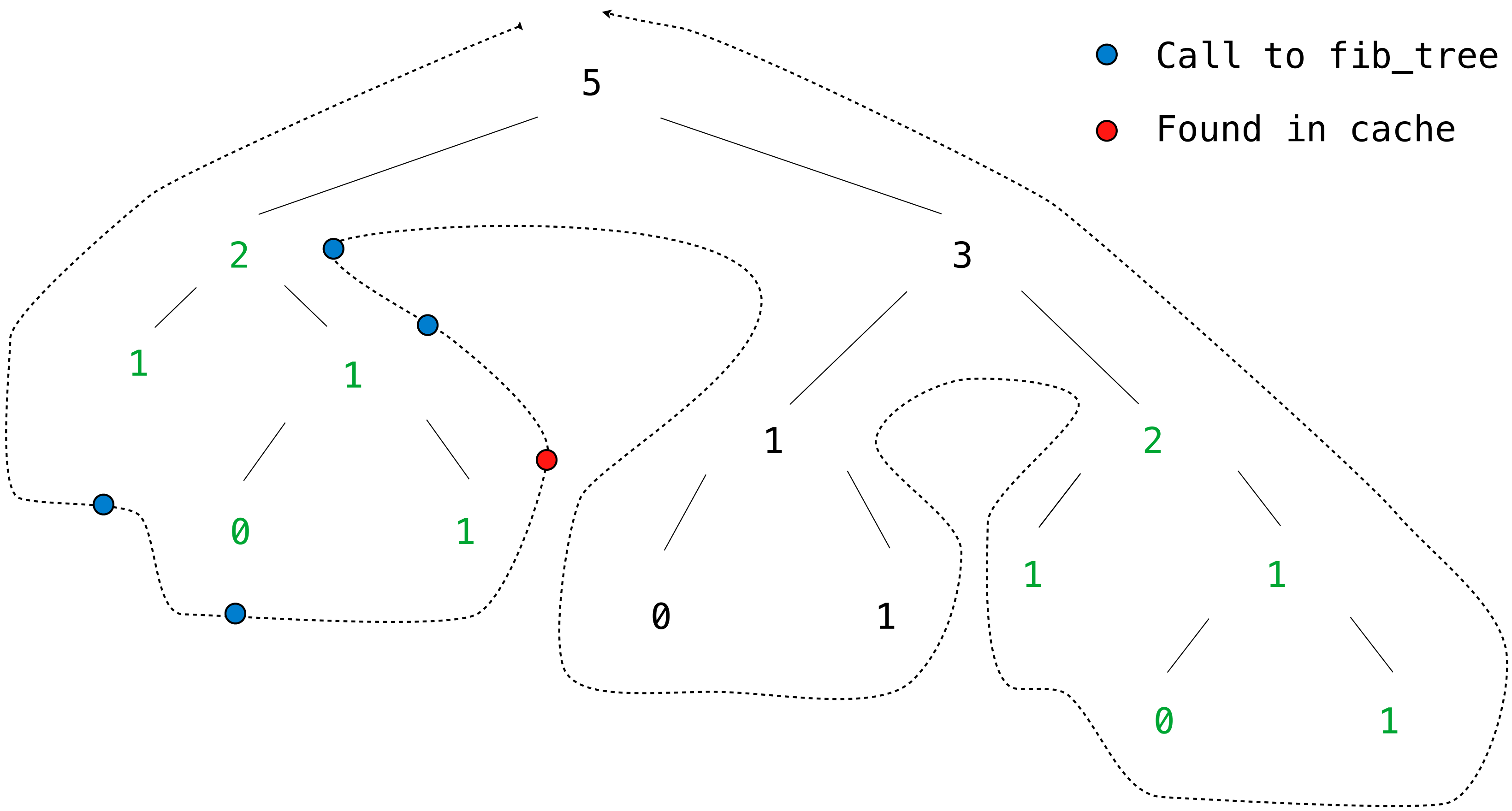
Memoized Tree Recursion



● Call to fib_tree

● Found in cache

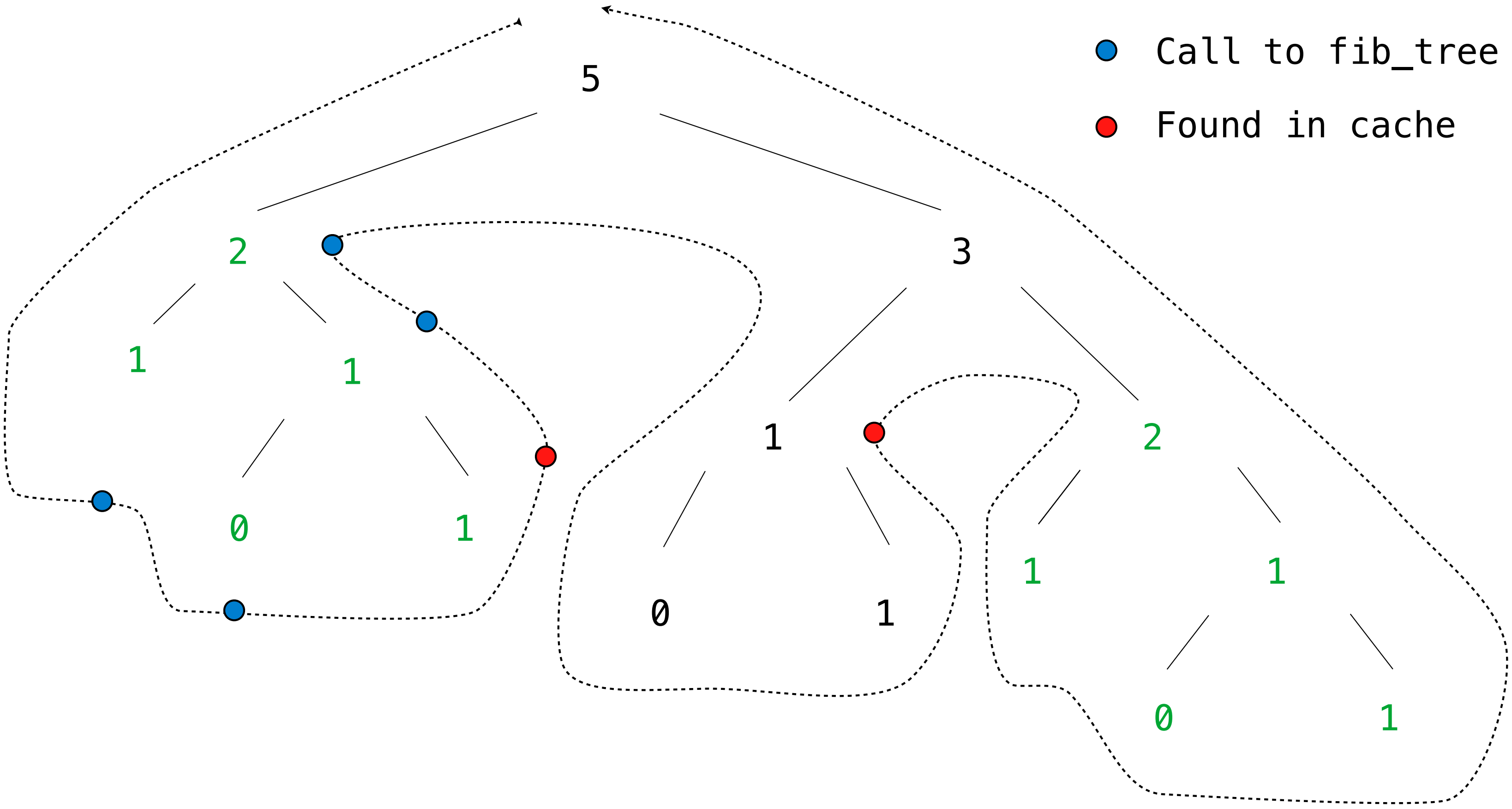
Memoized Tree Recursion



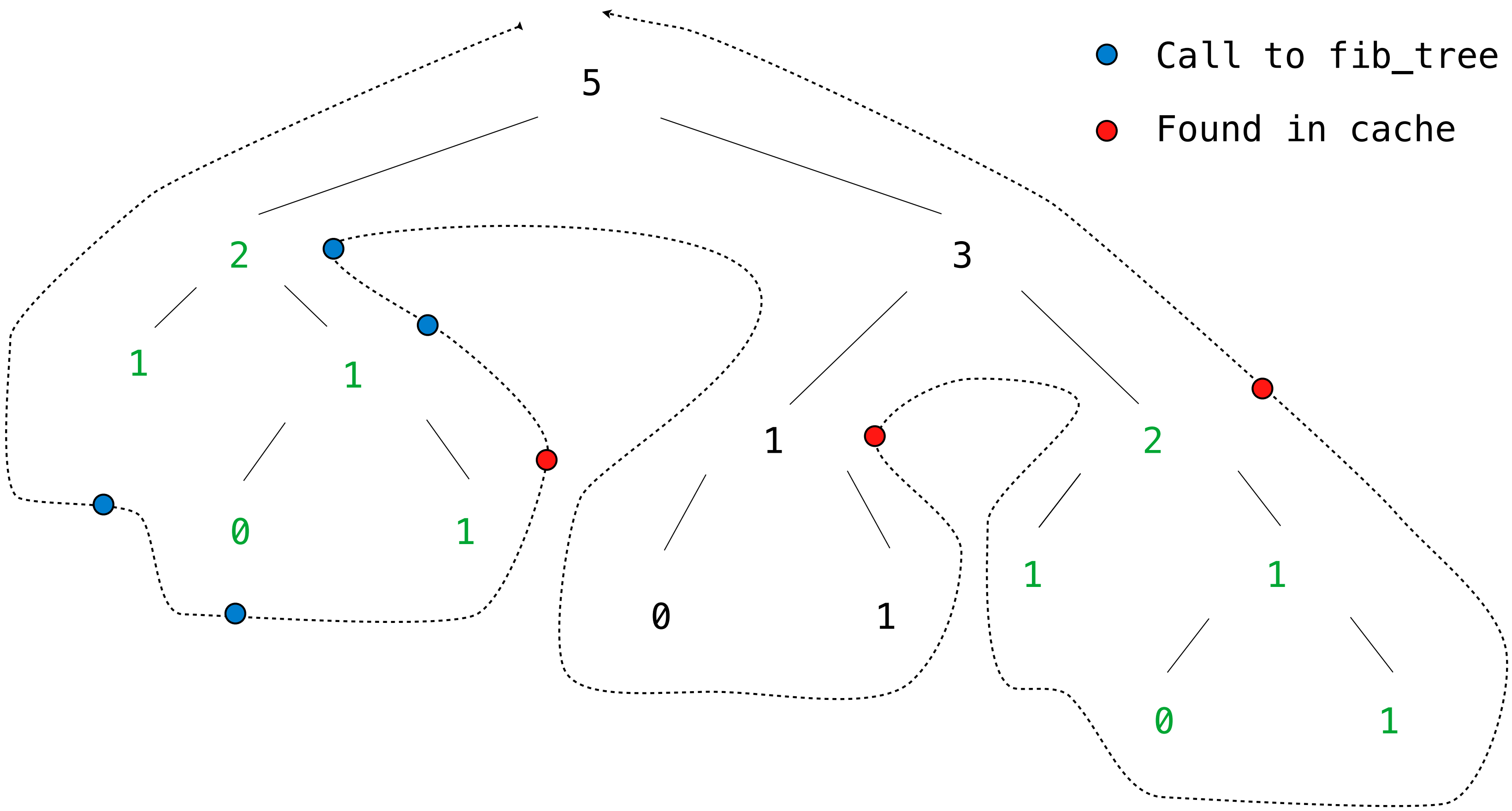
● Call to fib_tree

🔴 Found in cache

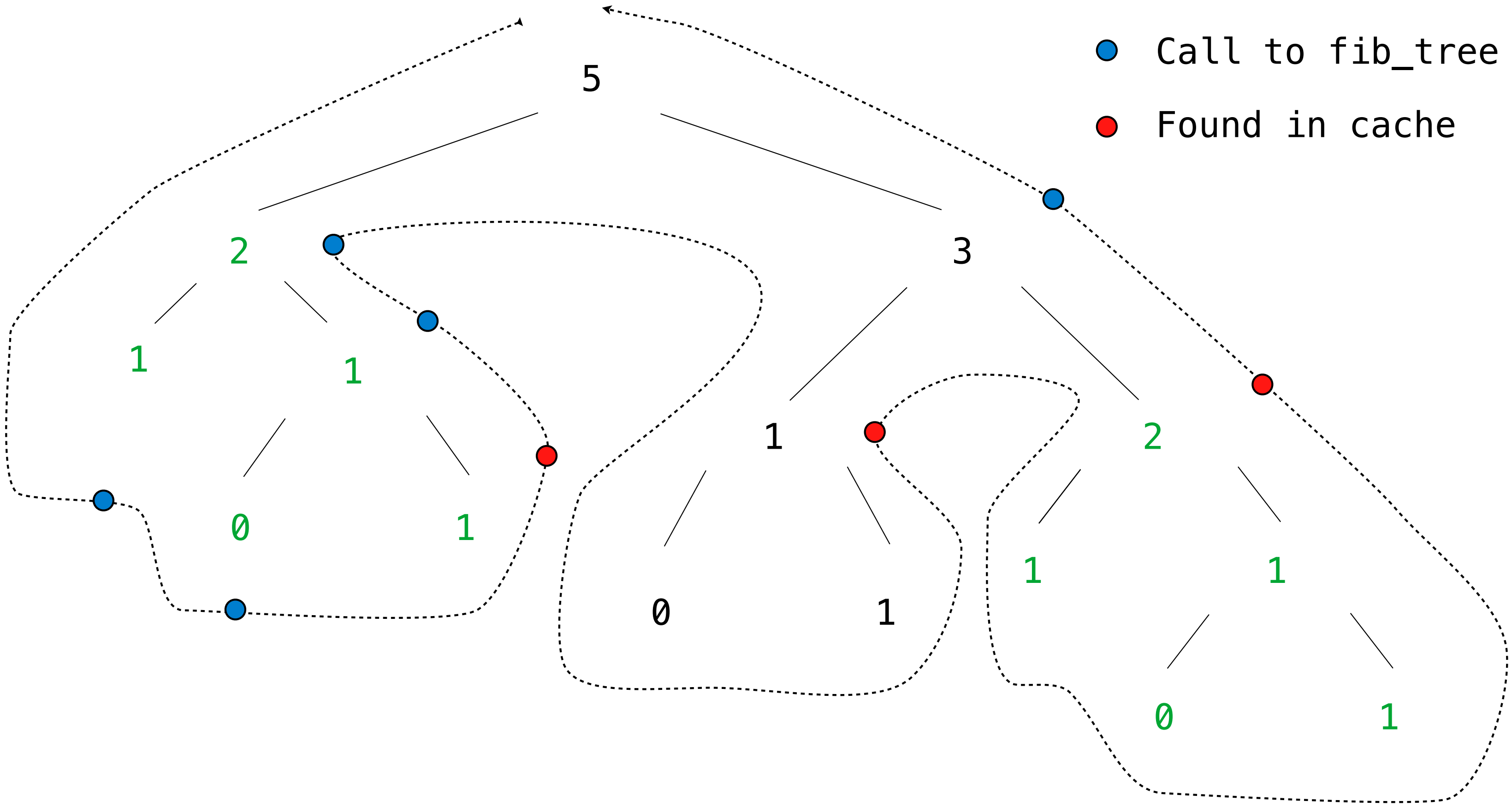
Memoized Tree Recursion



Memoized Tree Recursion



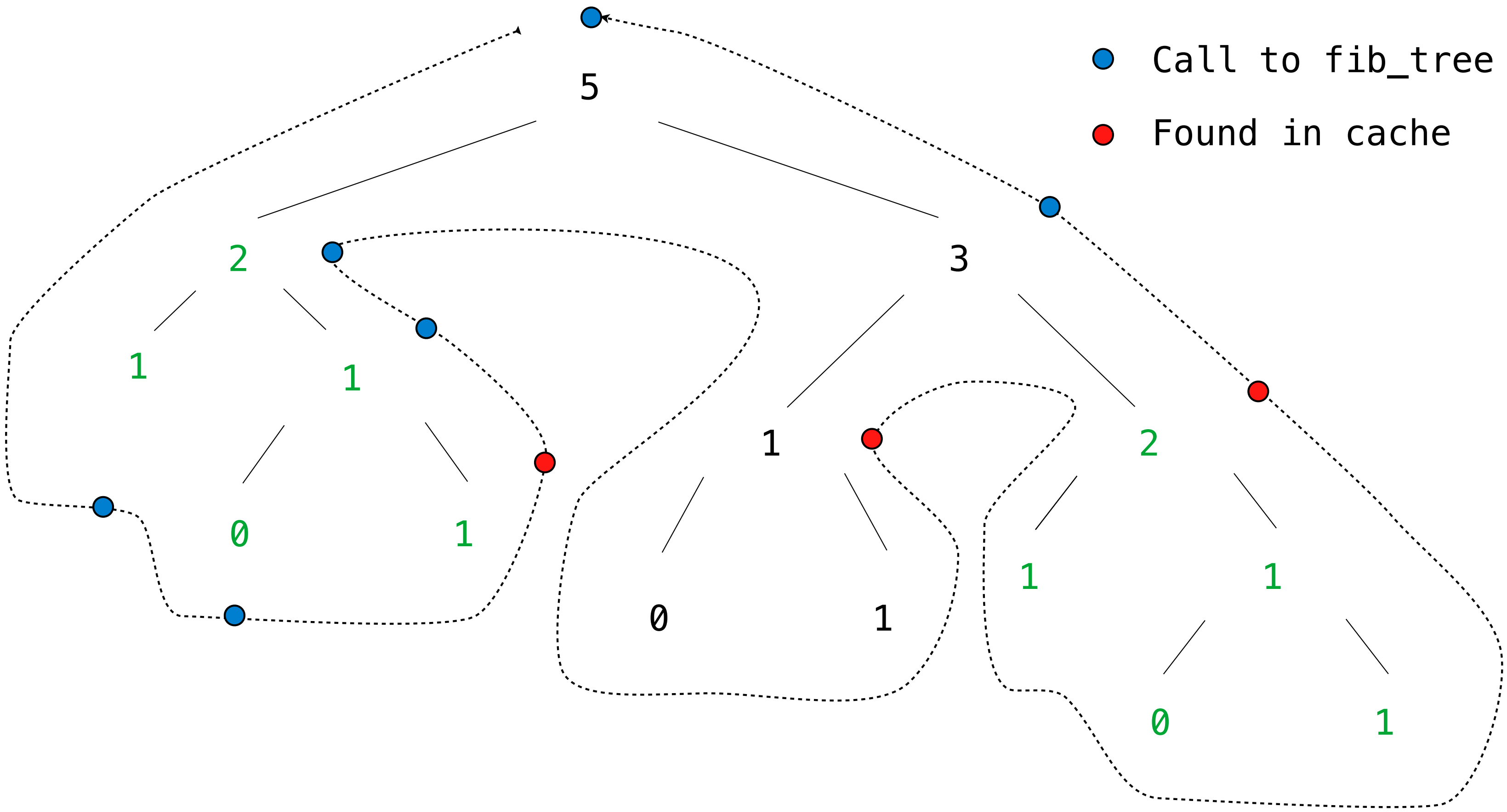
Memoized Tree Recursion



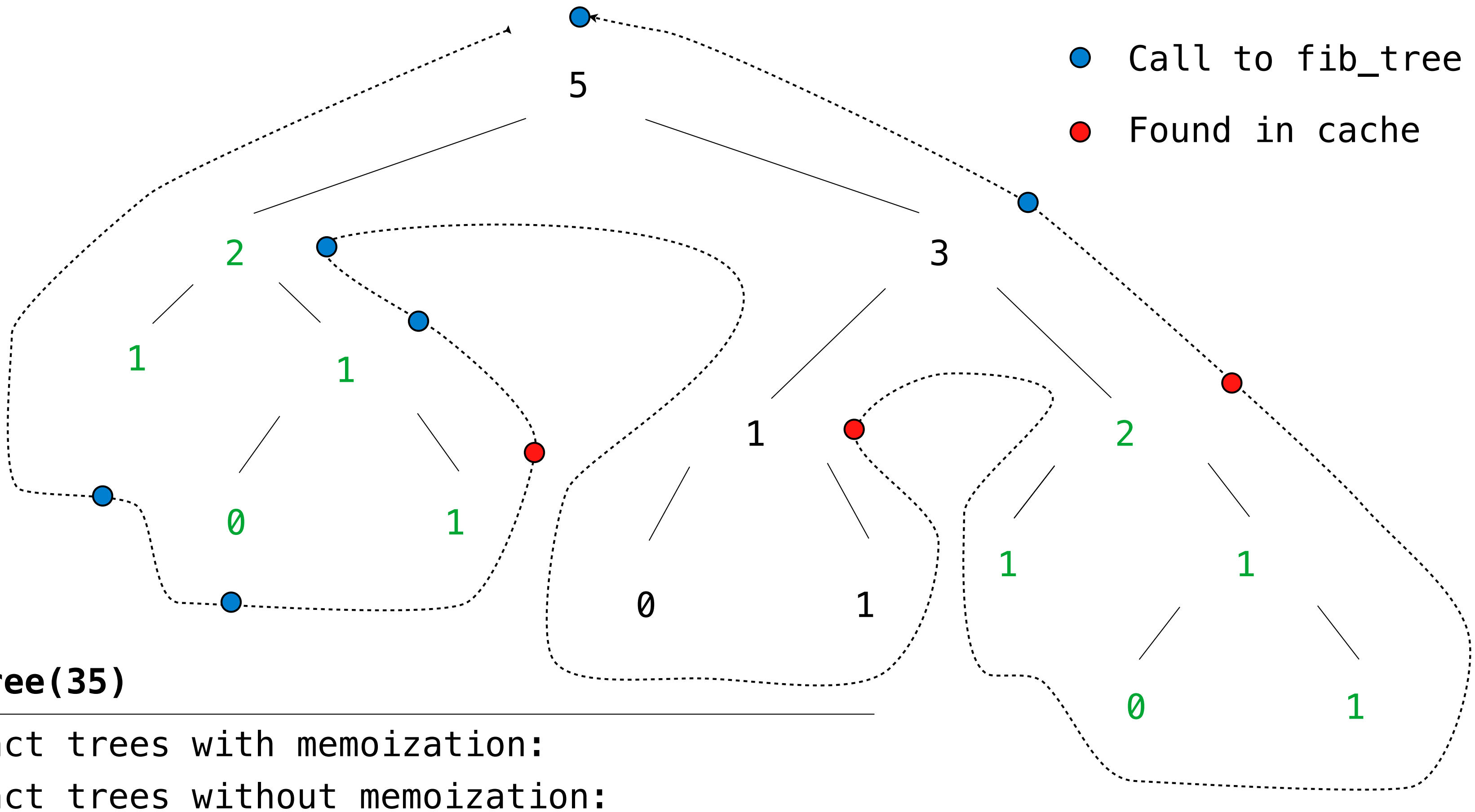
● `Call to fib_tree`

🟥 `Found in cache`

Memoized Tree Recursion



Memoized Tree Recursion

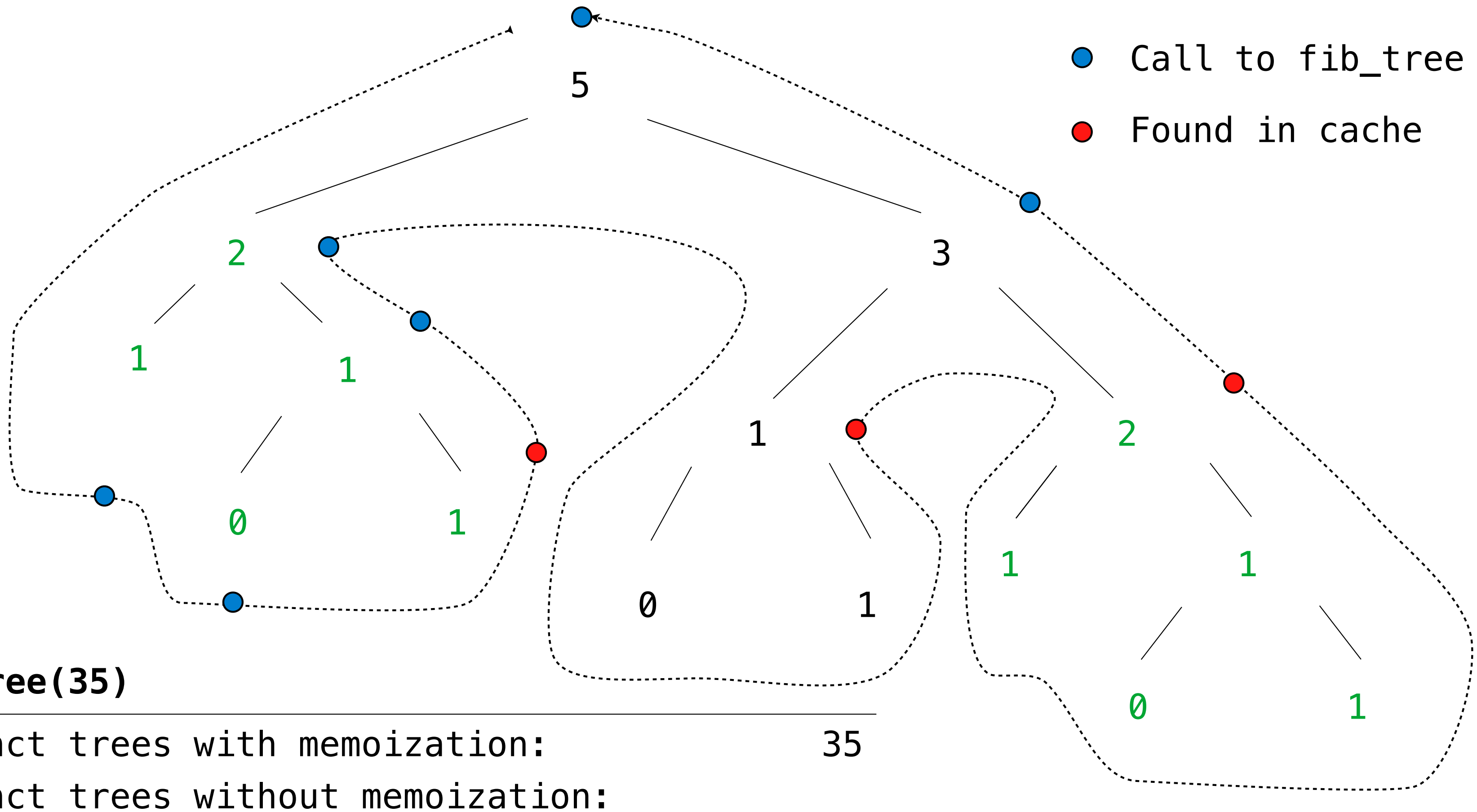


fib_tree(35)

Distinct trees with memoization:

Distinct trees without memoization:

Memoized Tree Recursion

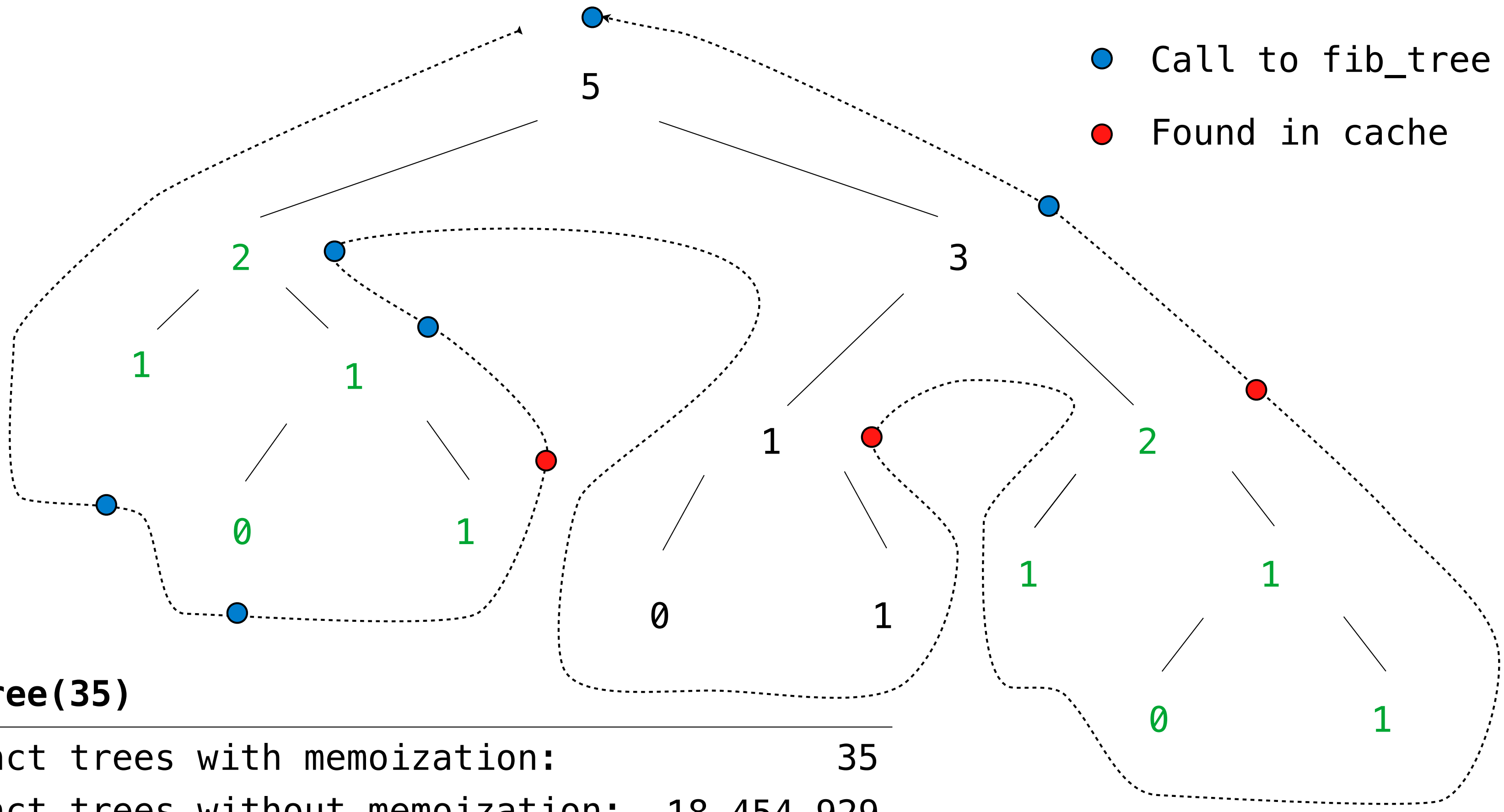


fib_tree(35)

Distinct trees with memoization: 35

Distinct trees without memoization:

Memoized Tree Recursion



fib_tree(35)

Distinct trees with memoization:	35
Distinct trees without memoization:	18,454,929