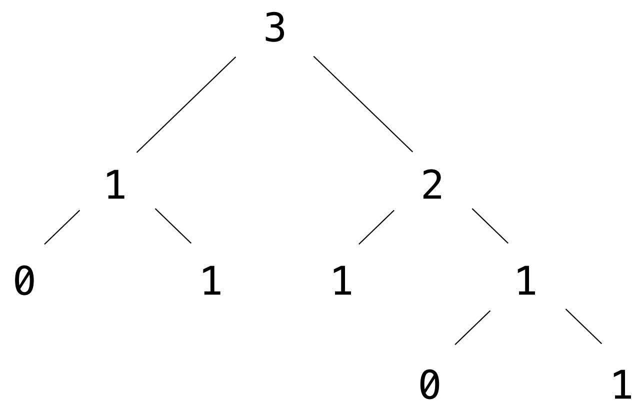
61A Lecture 19

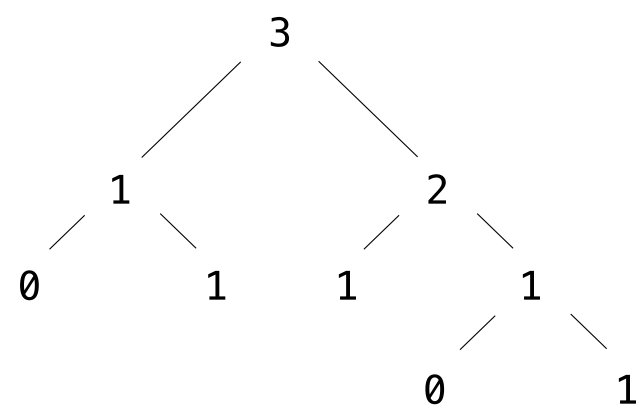
Announcements

Tree Class

Tree Review



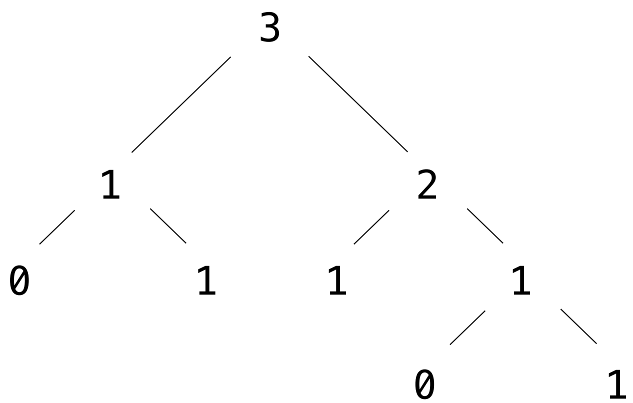
Tree Review



Recursive description (wooden trees):

Relative description (family trees):

Tree Review

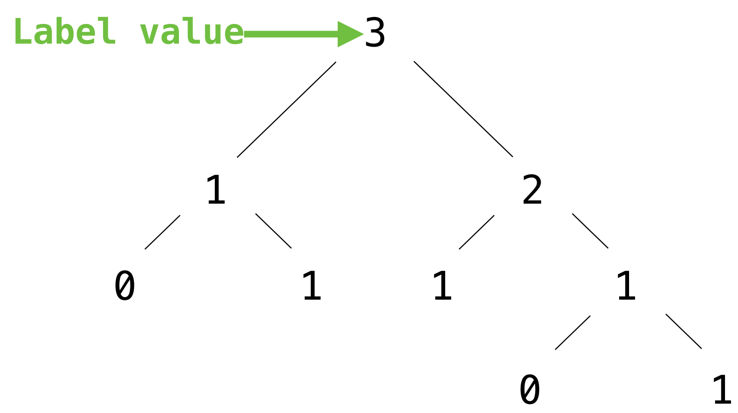


Recursive description **(wooden trees):**

A **tree** has a **label** value and a list of **branches**

Relative description **(family trees):**

Tree Review

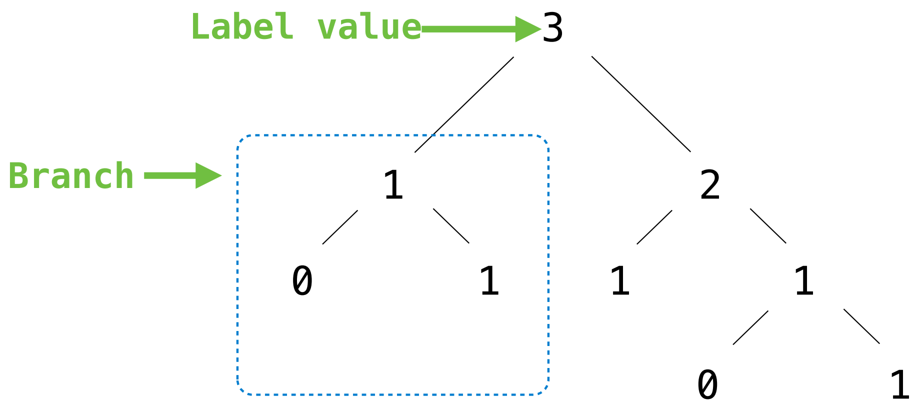


Recursive description (wooden trees):

A **tree** has a **label** value and a list of **branches**

Relative description (family trees):

Tree Review

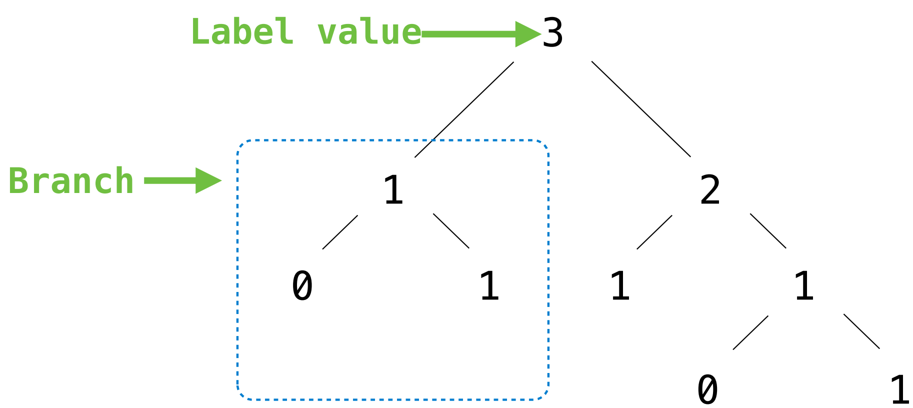


Recursive description **(wooden trees):**

A **tree** has a **label** value and a list of **branches**

Relative description **(family trees):**

Tree Review



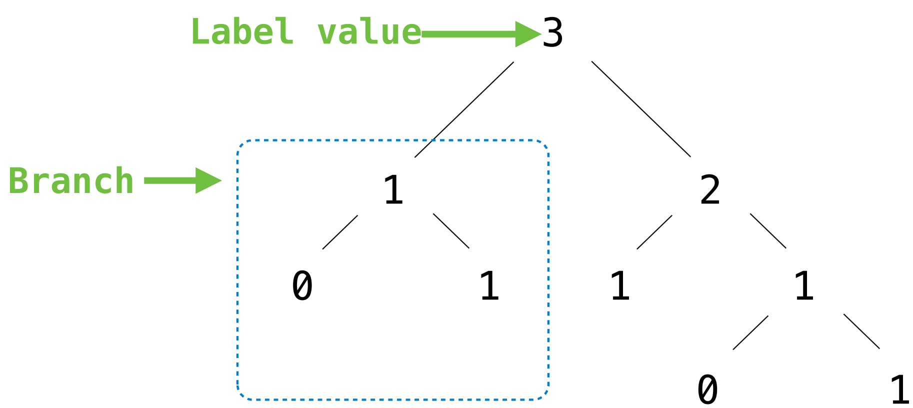
Recursive description **(wooden trees):**

A **tree** has a **label** value and a list of **branches**

Each branch is a **tree**

Relative description **(family trees):**

Tree Review



Recursive description **(wooden trees):**

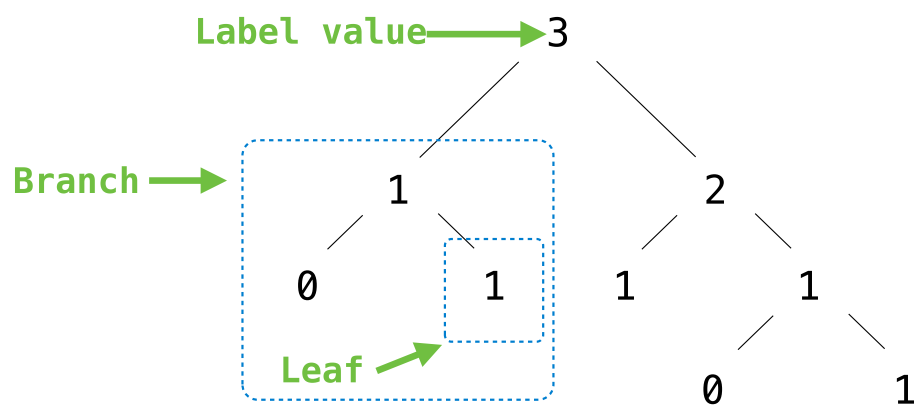
A **tree** has a **label** value and a list of **branches**

Each branch is a **tree**

A tree with zero branches is called a **leaf**

Relative description **(family trees):**

Tree Review



Recursive description (wooden trees):

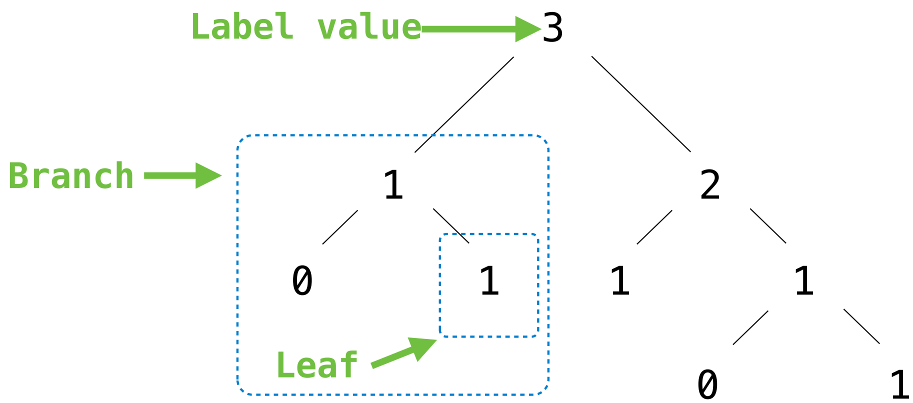
A **tree** has a **label** value and a list of **branches**

Each branch is a **tree**

A tree with zero branches is called a **leaf**

Relative description (family trees):

Tree Review



Recursive description (wooden trees):

A **tree** has a **label** value and a list of **branches**

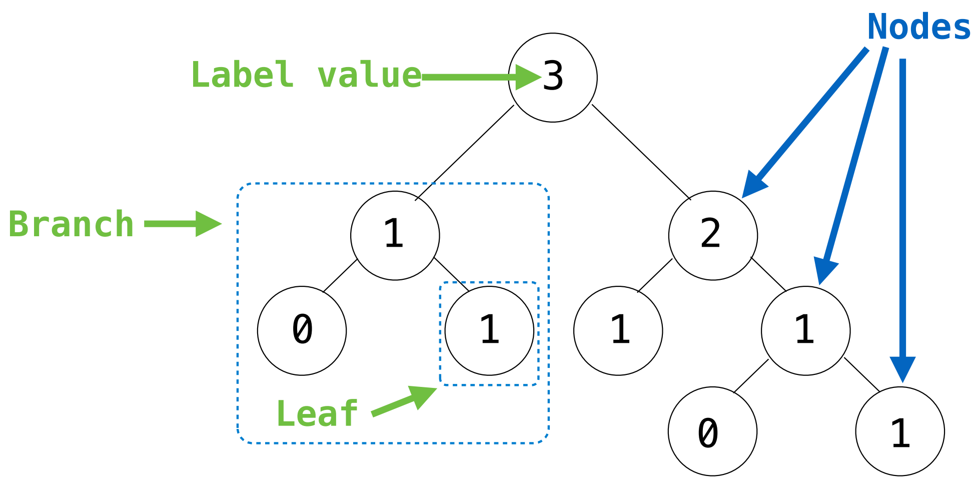
Each branch is a **tree**

A tree with zero branches is called a **leaf**

Relative description (family trees):

Each location in a tree is called a **node**

Tree Review



Recursive description **(wooden trees):**

A **tree** has a **label** value and a list of **branches**

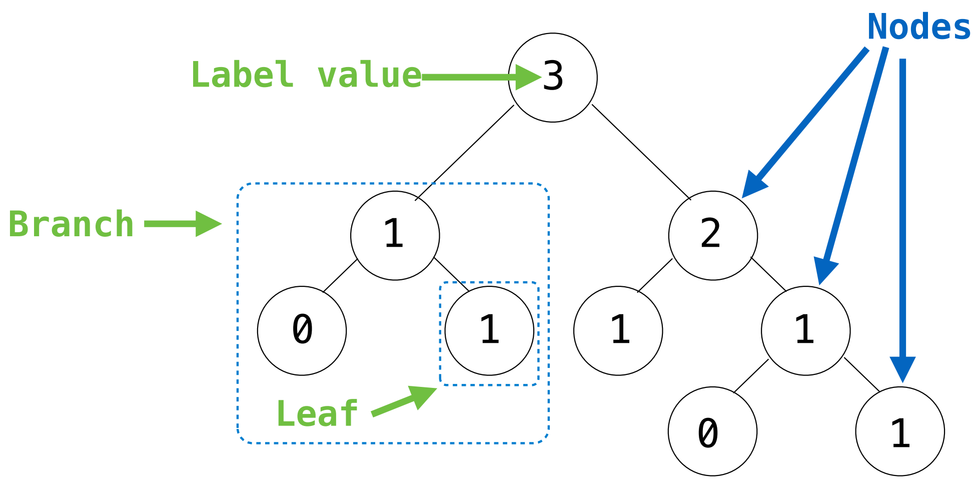
Each branch is a **tree**

A tree with zero branches is called a **leaf**

Relative description **(family trees):**

Each location in a tree is called a **node**

Tree Review



Recursive description (wooden trees):

A **tree** has a **label** value and a list of **branches**

Each branch is a **tree**

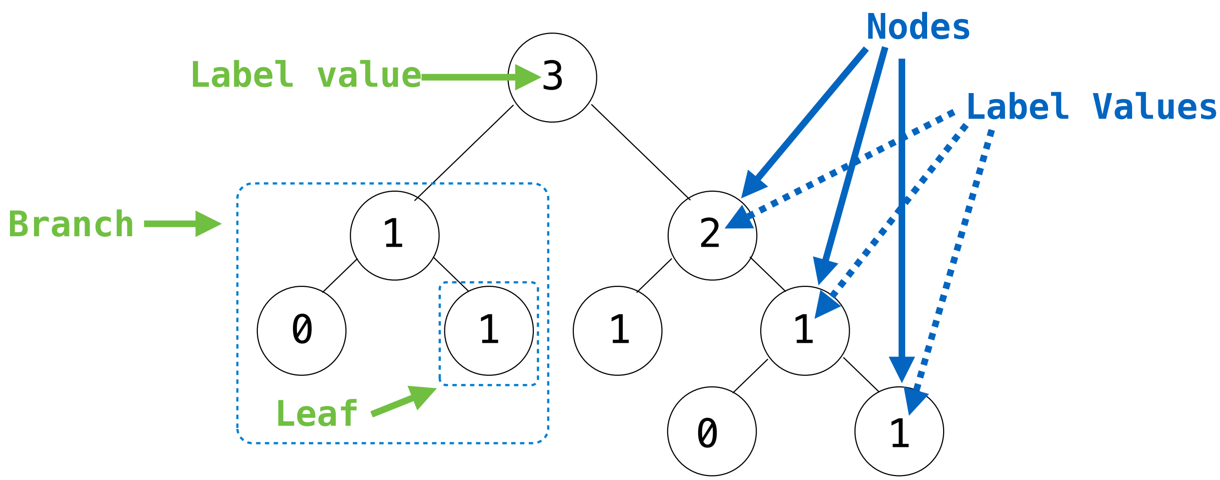
A tree with zero branches is called a **leaf**

Relative description (family trees):

Each location in a tree is called a **node**

Each **node** has a **value**

Tree Review



Recursive description (wooden trees):

A **tree** has a **label** value and a list of **branches**

Each branch is a **tree**

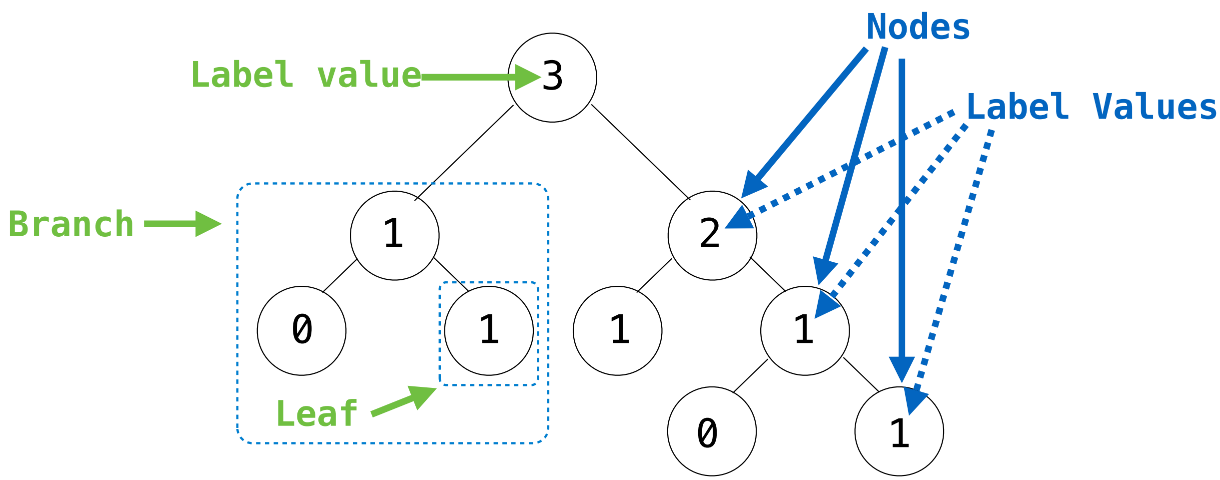
A tree with zero branches is called a **leaf**

Relative description (family trees):

Each location in a tree is called a **node**

Each **node** has a **value**

Tree Review



Recursive description (wooden trees):

A **tree** has a **label** value and a list of **branches**

Each branch is a **tree**

A tree with zero branches is called a **leaf**

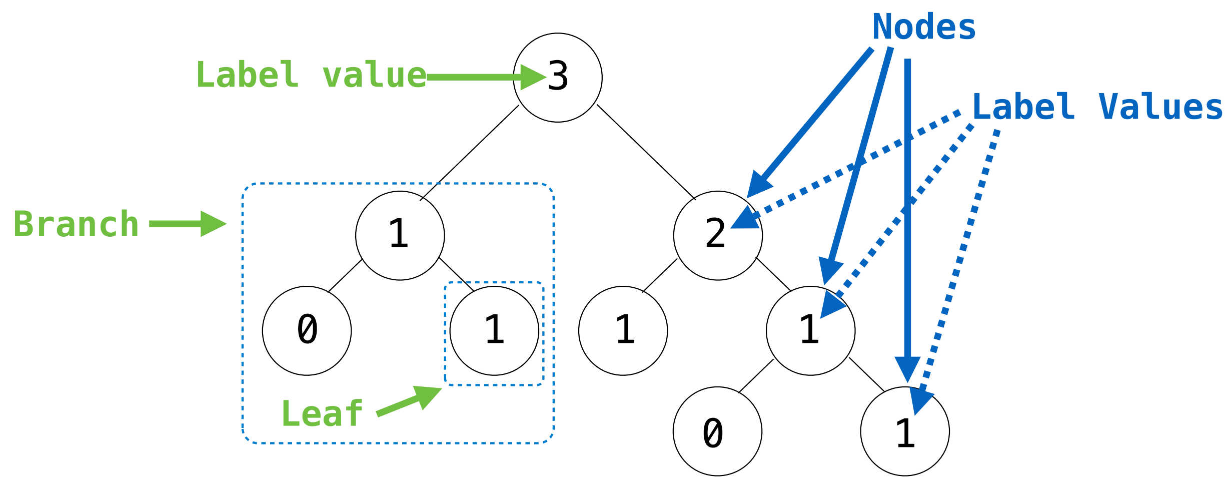
Relative description (family trees):

Each location in a tree is called a **node**

Each **node** has a **value**

One node can be the **parent/child** of another

Tree Review



Recursive description (wooden trees):

A **tree** has a **label** value and a list of **branches**

Each branch is a **tree**

A tree with zero branches is called a **leaf**

Relative description (family trees):

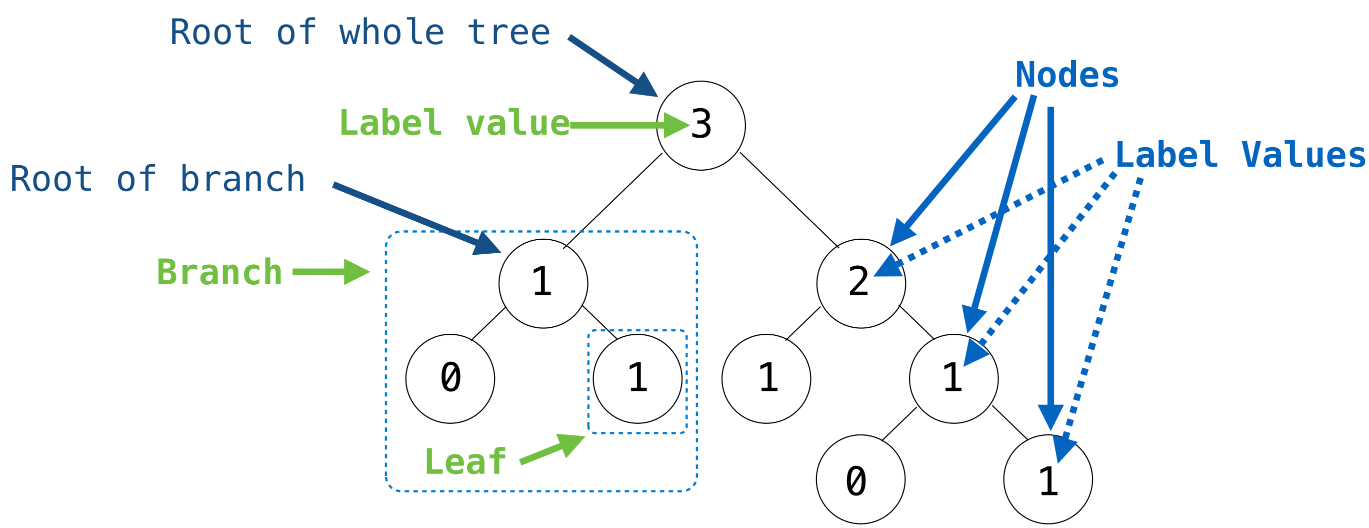
Each location in a tree is called a **node**

Each **node** has a **value**

One node can be the **parent/child** of another

Top node of tree is its **root**

Tree Review



Recursive description (wooden trees):

A **tree** has a **label** value and a list of **branches**

Each branch is a **tree**

A tree with zero branches is called a **leaf**

Relative description (family trees):

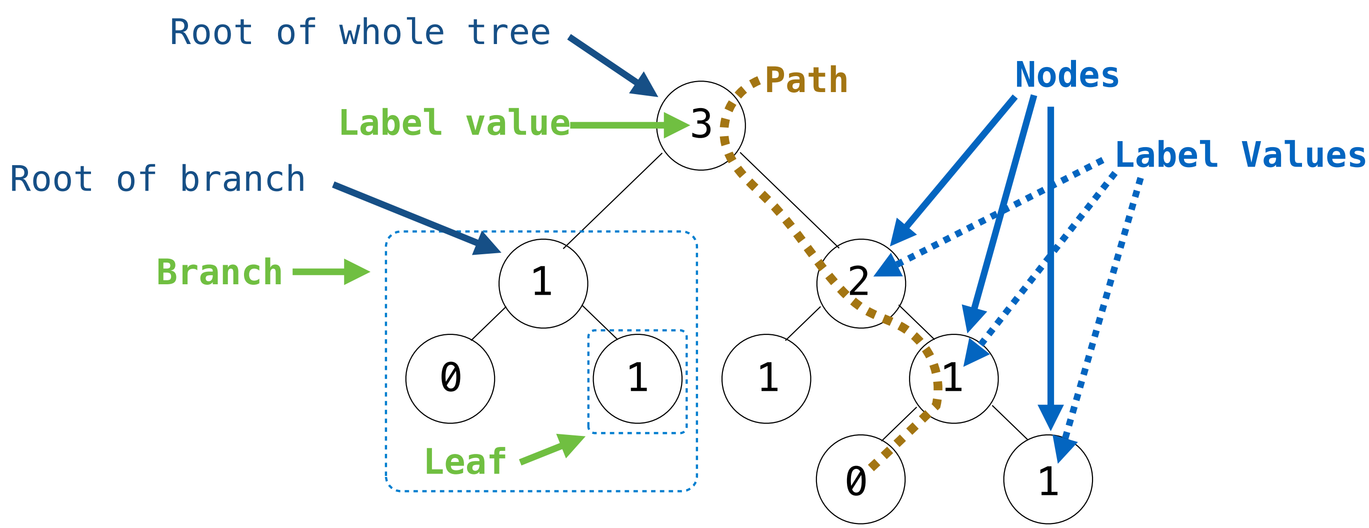
Each location in a tree is called a **node**

Each **node** has a **value**

One node can be the **parent/child** of another

Top node of tree is its **root**

Tree Review



Recursive description (wooden trees):

A **tree** has a **label** value and a list of **branches**

Each branch is a **tree**

A tree with zero branches is called a **leaf**

Relative description (family trees):

Each location in a tree is called a **node**

Each **node** has a **value**

One node can be the **parent/child** of another

Top node of tree is its **root**

Tree Class

```
A Tree has a label value and a list of branches; each branch is a Tree
```

Tree Class

A Tree has a label value and a list of branches; each branch is a Tree

```
class Tree:
```

Tree Class

A Tree has a label value and a list of branches; each branch is a Tree

```
class Tree:
    def __init__(self, label, branches=[]):
```

Tree Class

A Tree has a label value and a list of branches; each branch is a Tree

```
class Tree:
    def __init__(self, label, branches=[]):
        self.label = label
```

Tree Class

A Tree has a label value and a list of branches; each branch is a Tree

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

Tree Class

A Tree has a label value and a list of branches; each branch is a Tree

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

Tree Class

A Tree has a label value and a list of branches; each branch is a Tree

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

```
def tree(label, branches=[]):
    for branch in branches:
        assert is_tree(branch)
    return [label] + list(branches)
def label(tree):
    return tree[0]
def branches(tree):
    return tree[1:]
```

Tree Class

A Tree has a label value and a list of branches; each branch is a Tree

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

```
def tree(label, branches=[]):
    for branch in branches:
        assert is_tree(branch)
    return [label] + list(branches)
def label(tree):
    return tree[0]
def branches(tree):
    return tree[1:]
```

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

Tree Class

A Tree has a label value and a list of branches; each branch is a Tree

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

```
def tree(label, branches=[]):
    for branch in branches:
        assert is_tree(branch)
    return [label] + list(branches)
def label(tree):
    return tree[0]
def branches(tree):
    return tree[1:]
```

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

```
def fib_tree(n):
    if n == 0 or n == 1:
        return tree(n)
    else:
        left = fib_tree(n-2)
        right = fib_tree(n-1)
        fib_n = label(left) + label(right)
        return tree(fib_n, [left, right])
```

Tree Class

A Tree has a label value and a list of branches; each branch is a Tree

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

```
def tree(label, branches=[]):
    for branch in branches:
        assert is_tree(branch)
    return [label] + list(branches)
def label(tree):
    return tree[0]
def branches(tree):
    return tree[1:]
```

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

```
def fib_tree(n):
    if n == 0 or n == 1:
        return tree(n)
    else:
        left = fib_tree(n-2)
        right = fib_tree(n-1)
        fib_n = label(left) + label(right)
        return tree(fib_n, [left, right])
```

(Demo)

Side Excursion: Equality

If x and y are two objects, the equality test, x == y, does not automatically mean what you want it to mean.

For example, `Tree(4) != Tree(4)` but after performing x = Tree(4), we do have `x == x`

The reason for this is that in Python,

- All values (conceptually, at least) are in fact *pointers* to objects, and
- By default, == on pointers compares the pointers themselves ("are these pointing at exactly the same object?").
- That is, by default == and != are the same as the **is** and **is not** operators.
- That can be changed on a class-by-class basis. For example, == on numbers, lists, tuples, strings, sets, and dictionaries means what we expect: the *contents* are the same.

Tree Mutation

Example: Pruning Trees

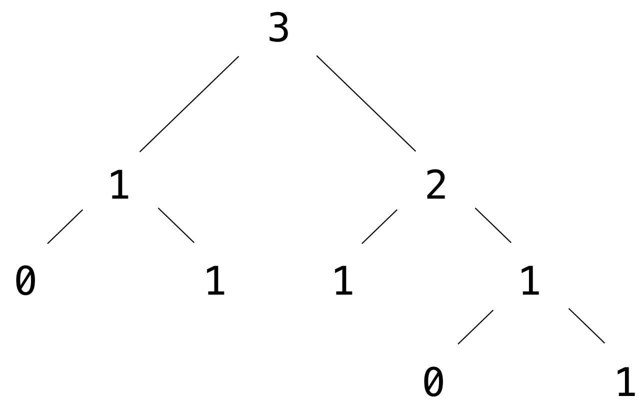
Removing subtrees from a tree is called *pruning*

Prune branches before recursive processing

Example: Pruning Trees

Removing subtrees from a tree is called *pruning*

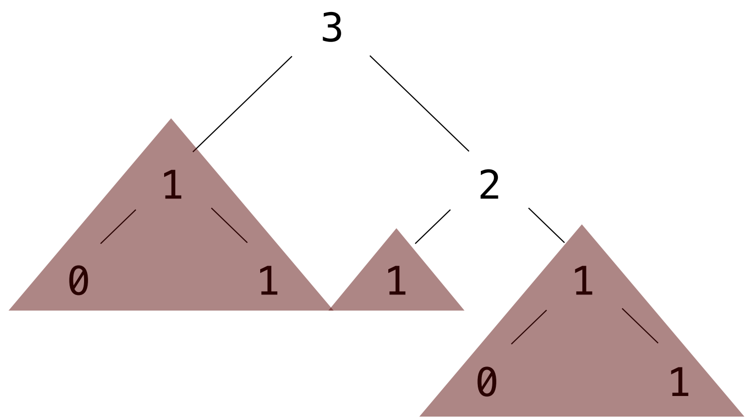
Prune branches before recursive processing



Example: Pruning Trees

Removing subtrees from a tree is called *pruning*

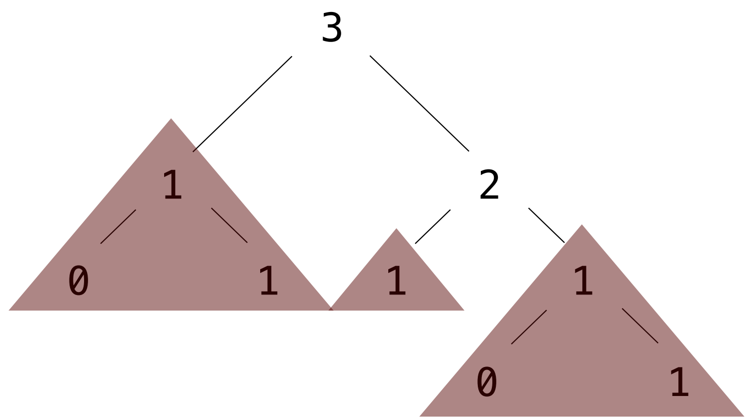
Prune branches before recursive processing



Example: Pruning Trees

Removing subtrees from a tree is called *pruning*

Prune branches before recursive processing

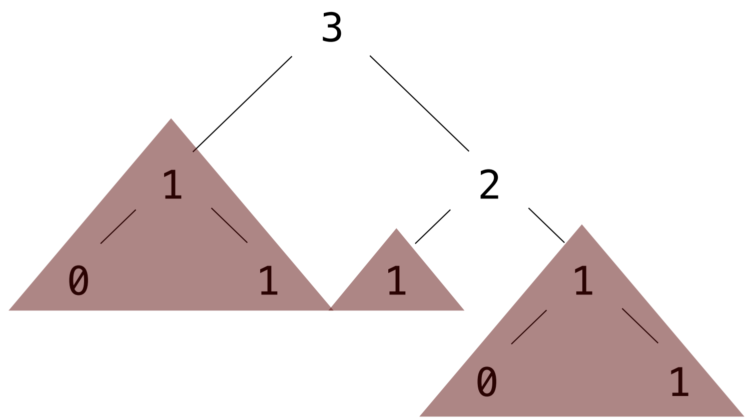



```
def prune(t, n):
    """Prune sub-trees whose label value is n."""
    t.branches = [______________ for b in t.branches if ____________________]
    for b in t.branches:
        prune(_______________________________, _______________________________)
```

Example: Pruning Trees

Removing subtrees from a tree is called *pruning*

Prune branches before recursive processing

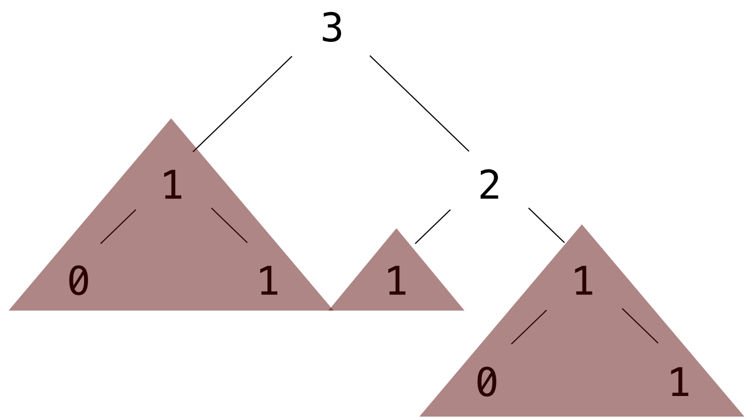



```
def prune(t, n):
    """Prune sub-trees whose label value is n."""
    t.branches = [______b______ for b in t.branches if ___b.label != n___]
    for b in t.branches:
        prune(_______________________________, _______________________________)
```

Example: Pruning Trees

Removing subtrees from a tree is called *pruning*

Prune branches before recursive processing

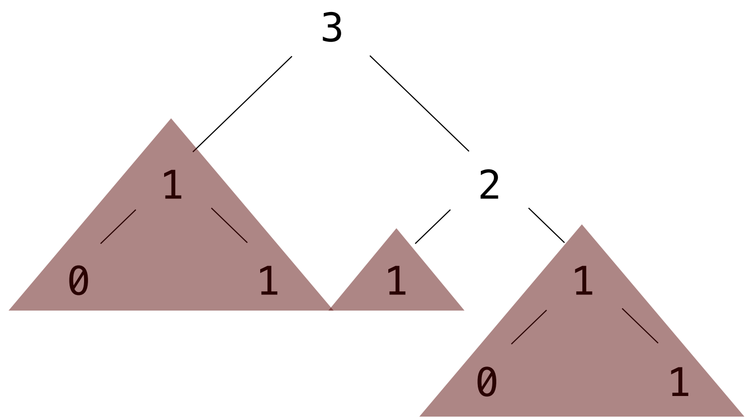



```
def prune(t, n):
    """Prune sub-trees whose label value is n."""
    t.branches = [______b______ for b in t.branches if ____b.label != n____]
    for b in t.branches:
        prune(_______b_______, _______n_______)
```

Example: Pruning Trees

Removing subtrees from a tree is called *pruning*

Prune branches before recursive processing




```
def prune(t, n):
    """Prune sub-trees whose label value is n."""
    t.branches = [____b____ for b in t.branches if ____b.label != n____]
    for b in t.branches:
        prune(________b________, ________n________)
```

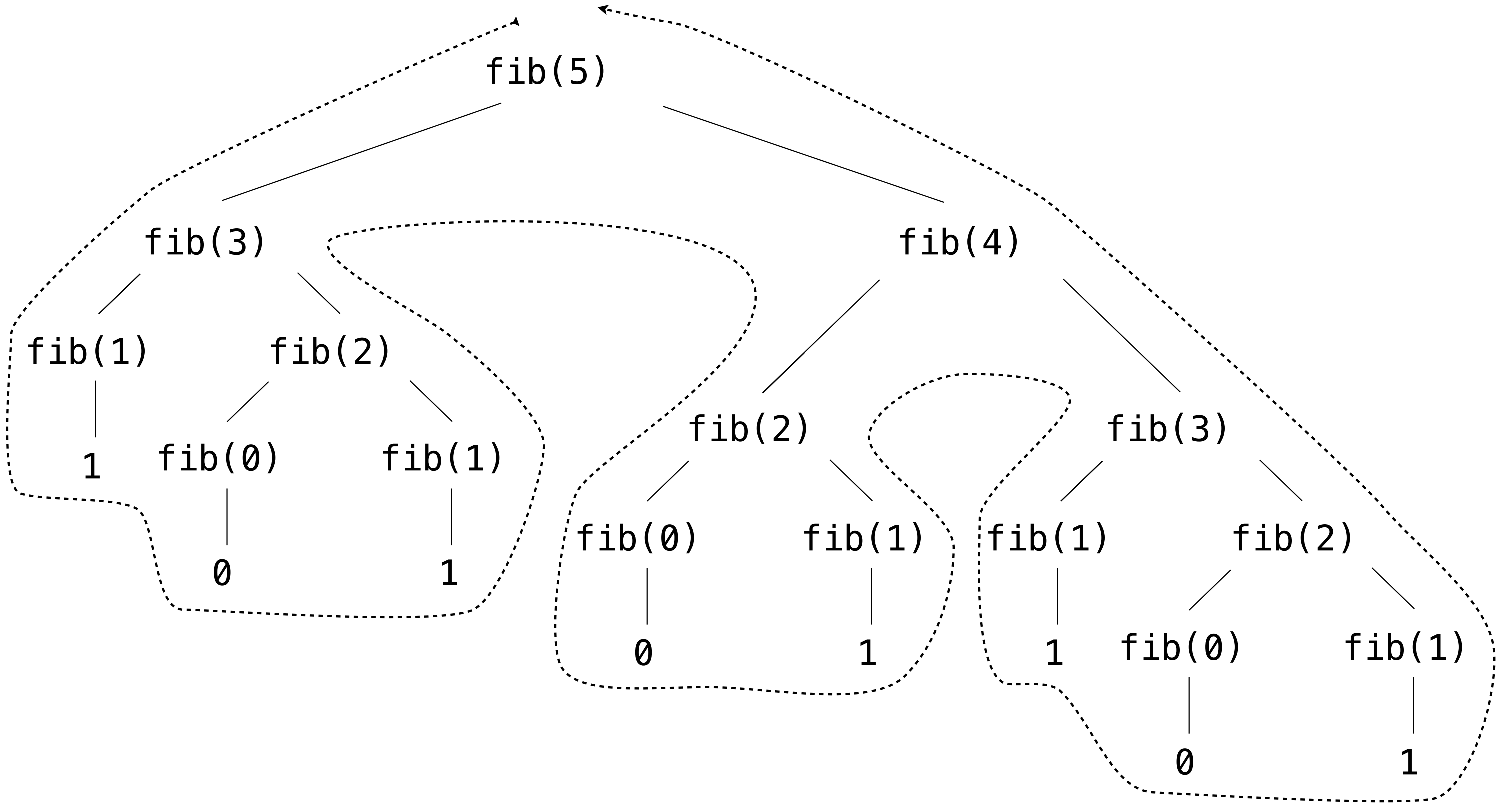
(Demo)

Example: Pruning Trees

Removing subtrees from a tree is called *pruning*

Prune branches before recursive processing

E.g., want to prune cached (previously memorized) values.



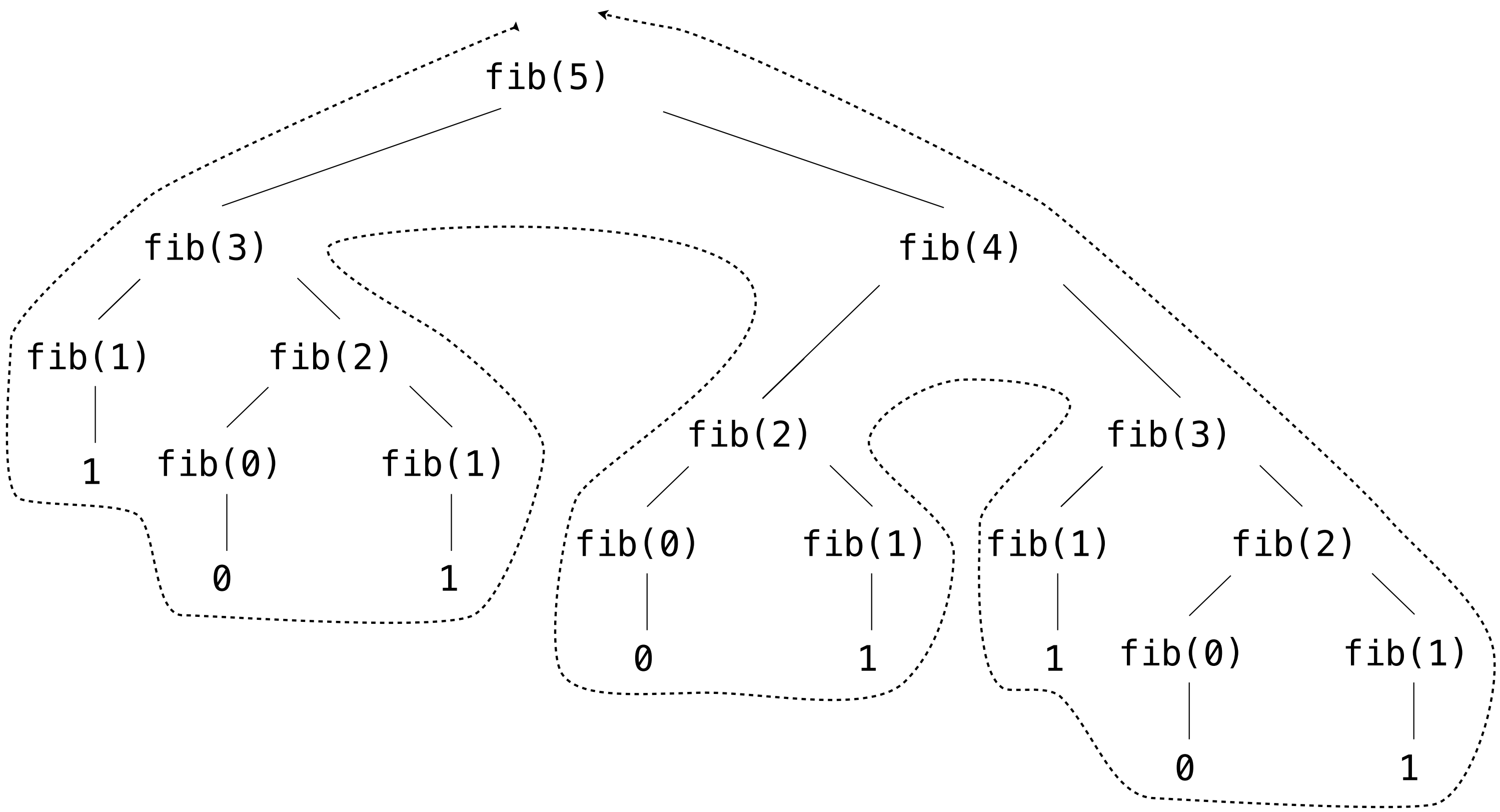
Example: Pruning Trees

Removing subtrees from a tree is called *pruning*

Prune branches before recursive processing

E.g., want to prune cached (previously memorized) values.

Memoization:



Example: Pruning Trees

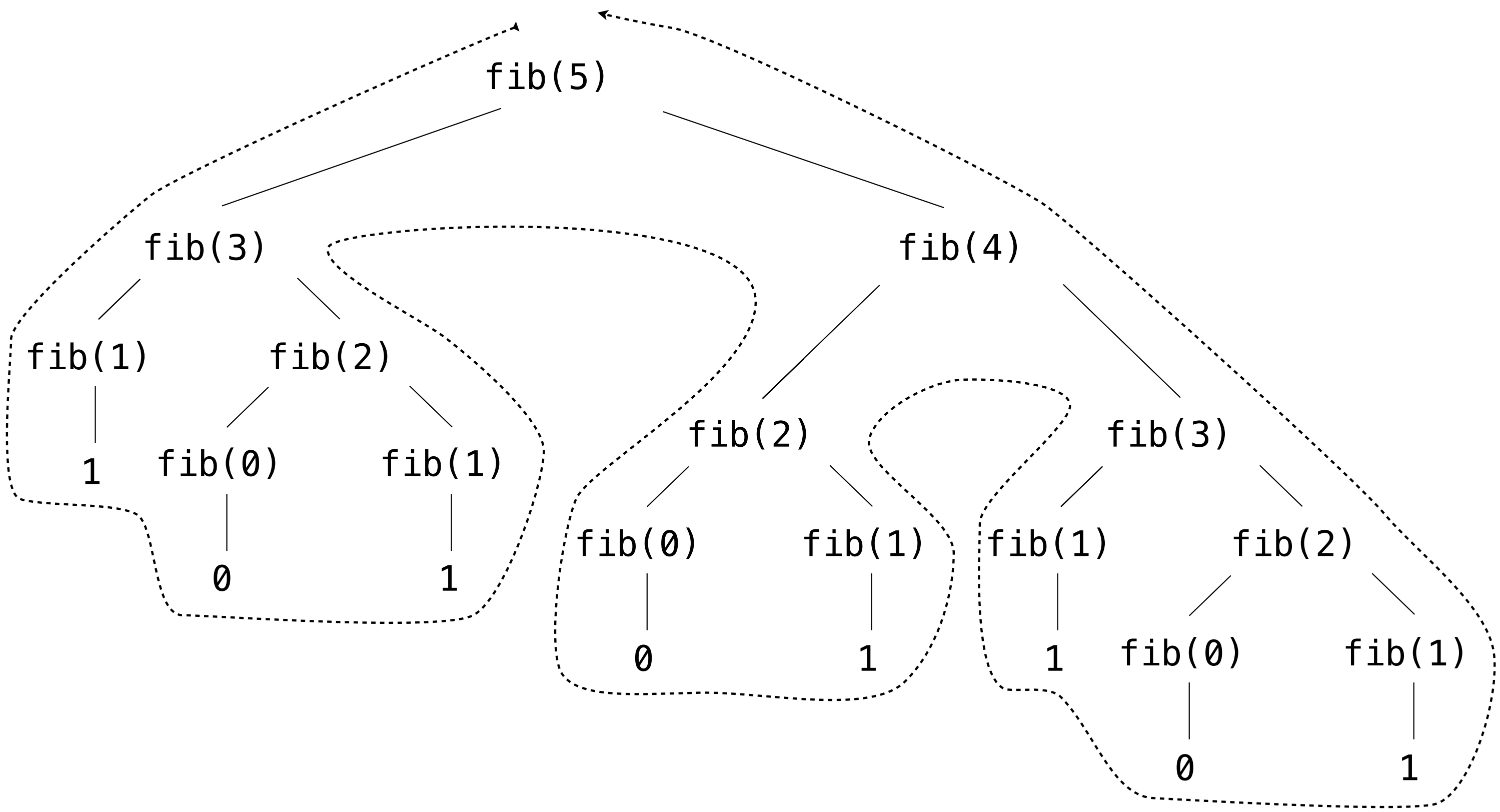
Removing subtrees from a tree is called *pruning*

Prune branches before recursive processing

E.g., want to prune cached (previously memorized) values.

Memoization:

- Returned by fib



Example: Pruning Trees

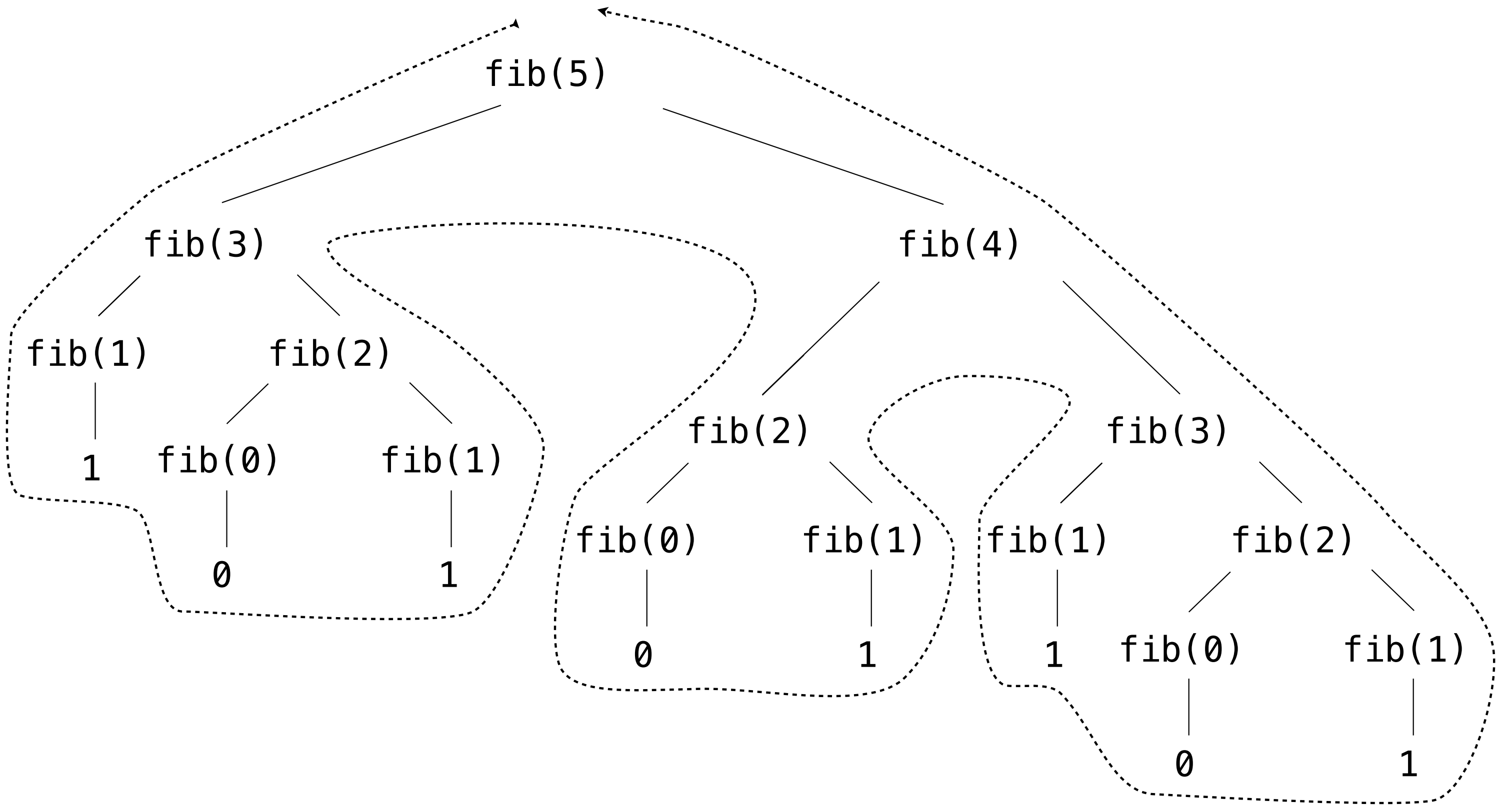
Removing subtrees from a tree is called *pruning*

Prune branches before recursive processing

E.g., want to prune cached (previously memorized) values.

Memoization:

- Returned by fib
- Found in cache



Example: Pruning Trees

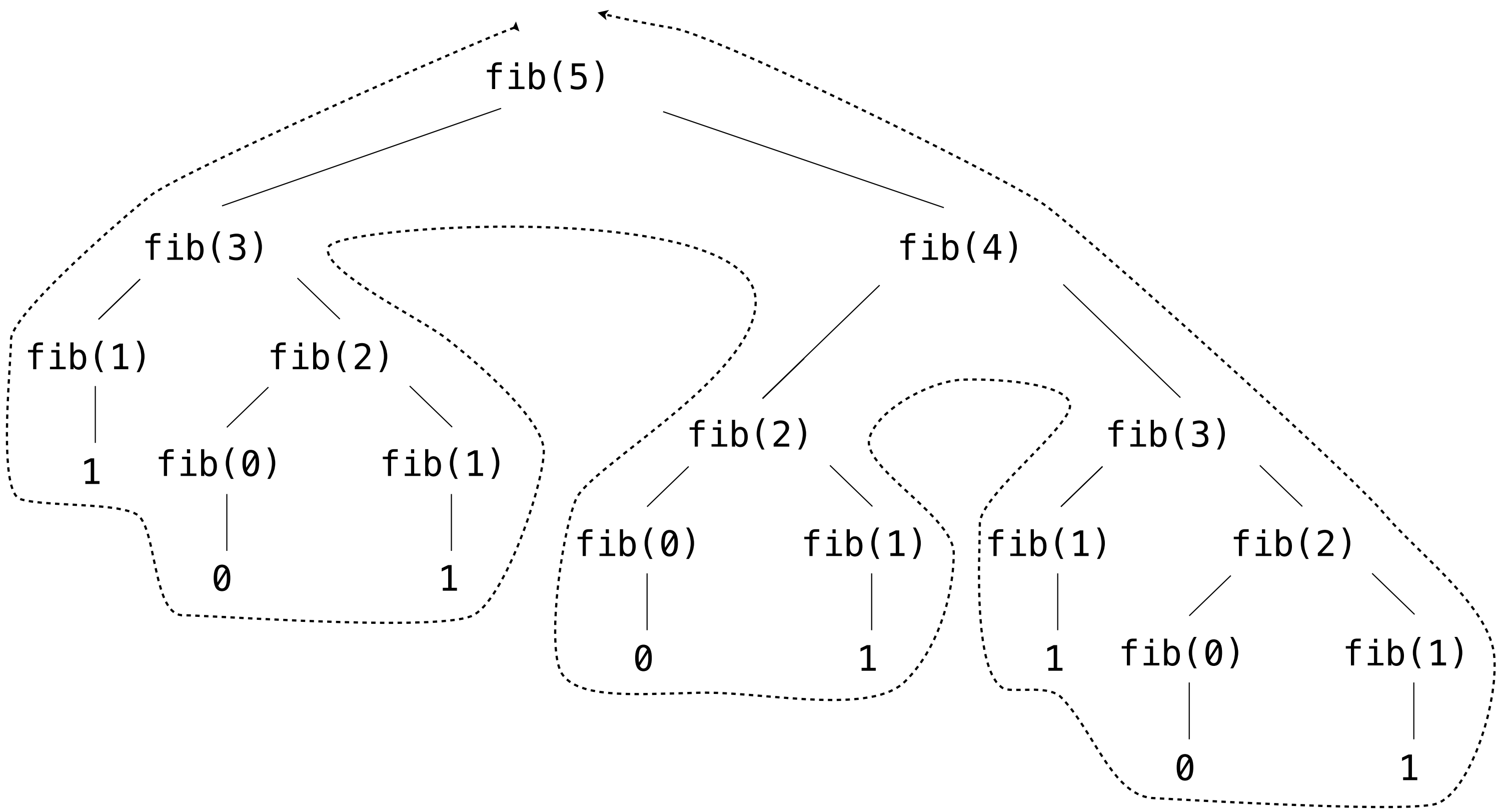
Removing subtrees from a tree is called *pruning*

Prune branches before recursive processing

E.g., want to prune cached (previously memorized) values.

Memoization:

- Returned by fib
- Found in cache
- Skipped



Example: Pruning Trees

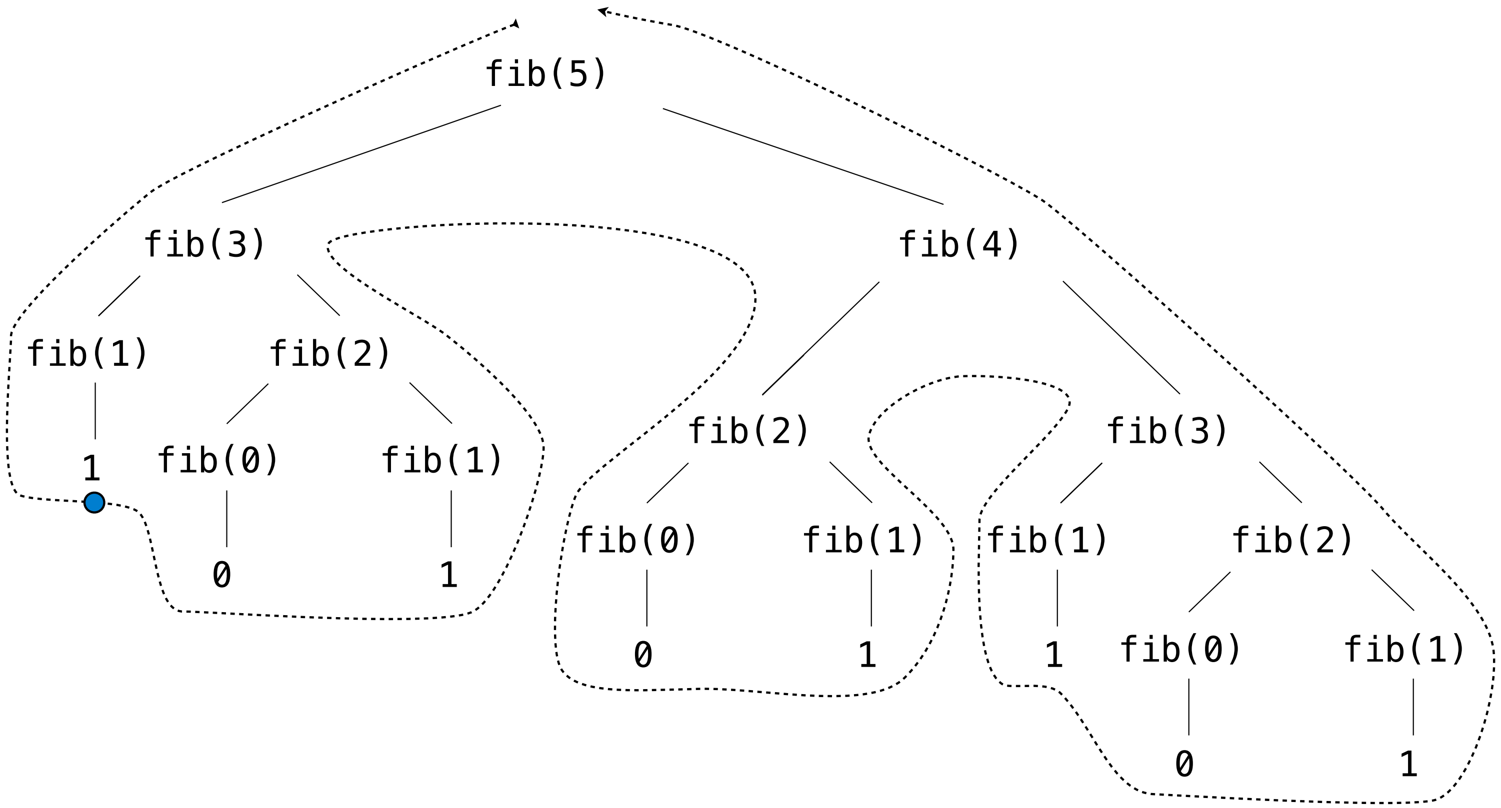
Removing subtrees from a tree is called *pruning*

Prune branches before recursive processing

E.g., want to prune cached (previously memorized) values.

Memoization:

- Returned by fib
- Found in cache
- Skipped



Example: Pruning Trees

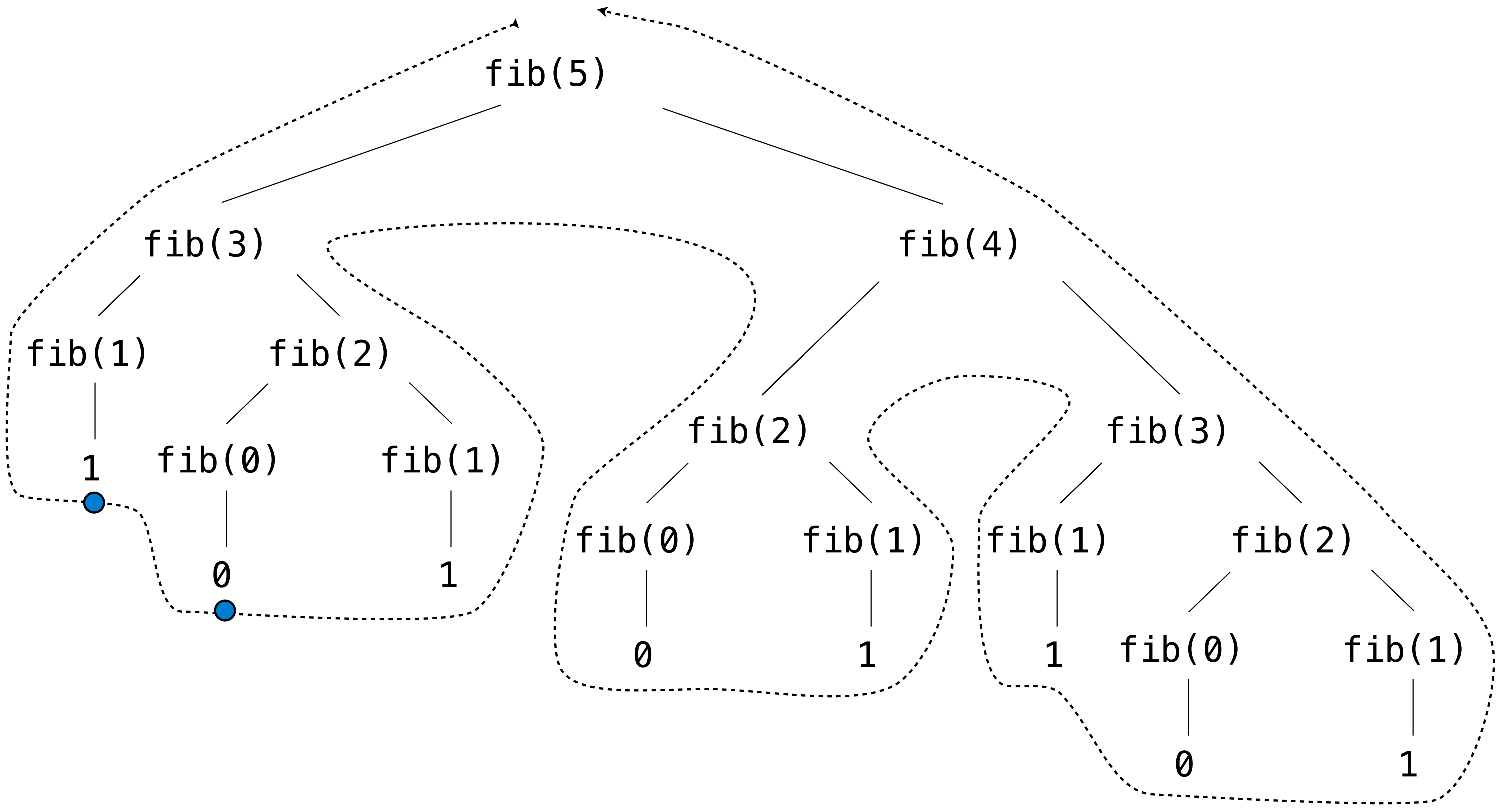
Removing subtrees from a tree is called *pruning*

Prune branches before recursive processing

E.g., want to prune cached (previously memorized) values.

Memoization:

- Returned by fib
- Found in cache
- Skipped



Example: Pruning Trees

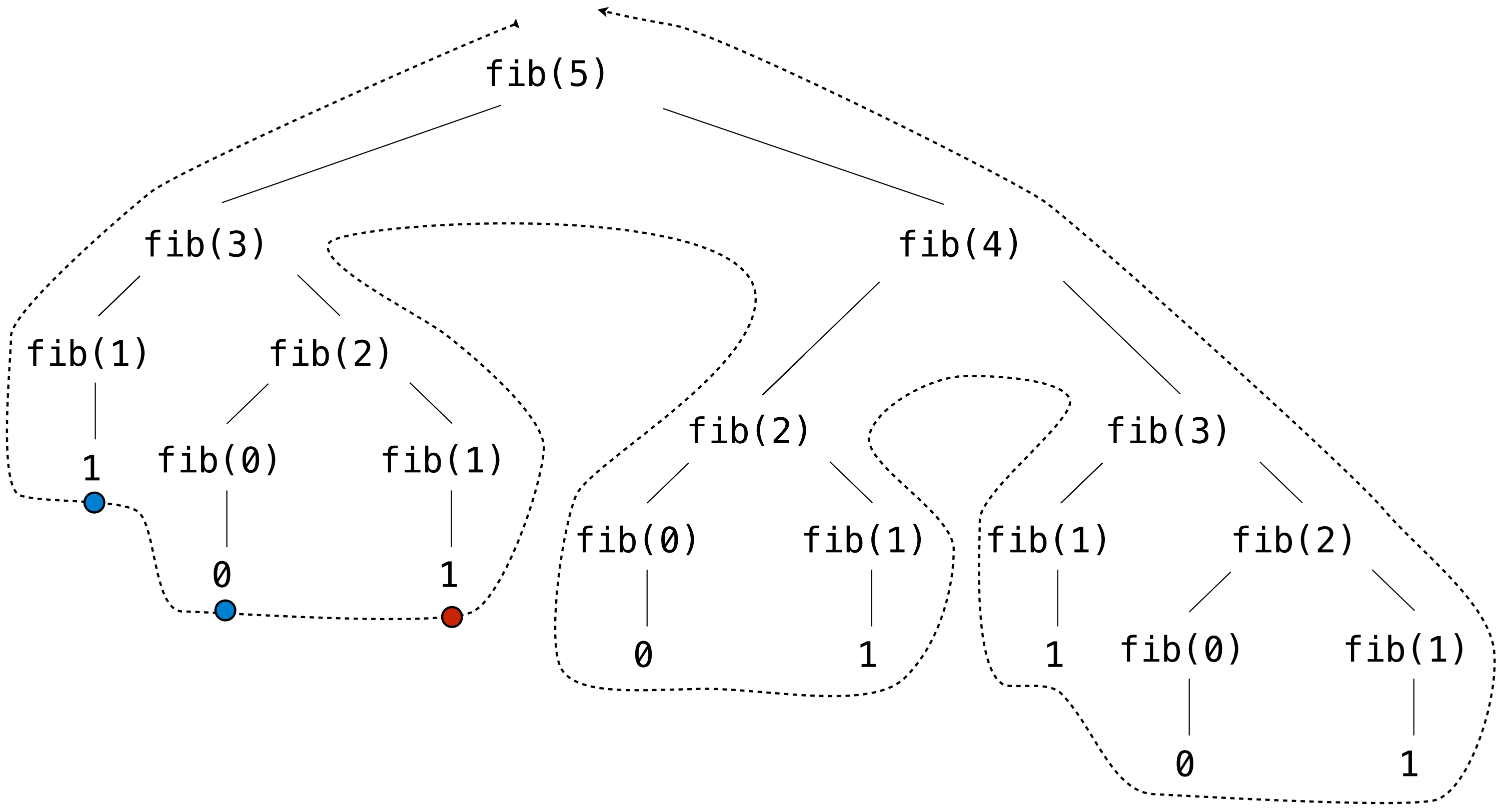
Removing subtrees from a tree is called *pruning*

Prune branches before recursive processing

E.g., want to prune cached (previously memorized) values.

Memoization:

- Returned by fib
- Found in cache
- Skipped



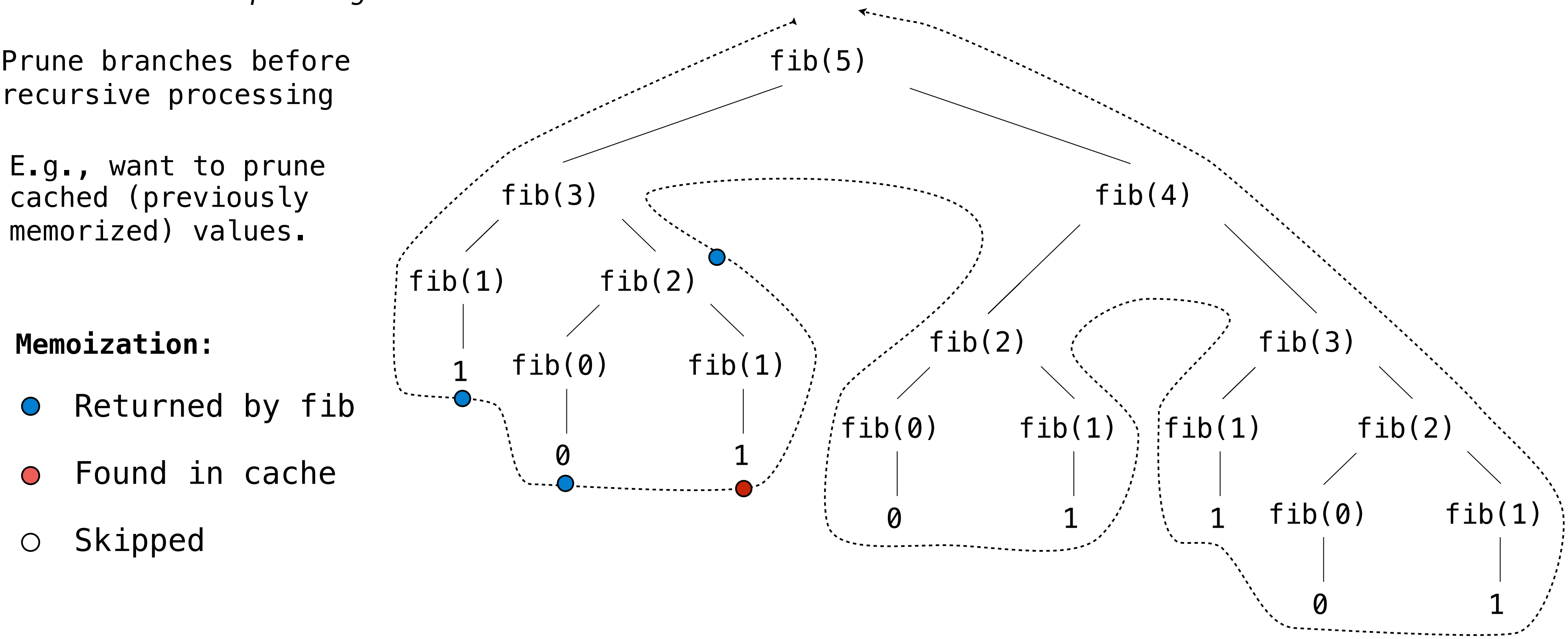
Example: Pruning Trees

Removing subtrees from a tree is called *pruning*

Prune branches before recursive processing

E.g., want to prune cached (previously memorized) values.

Memoization:



Example: Pruning Trees

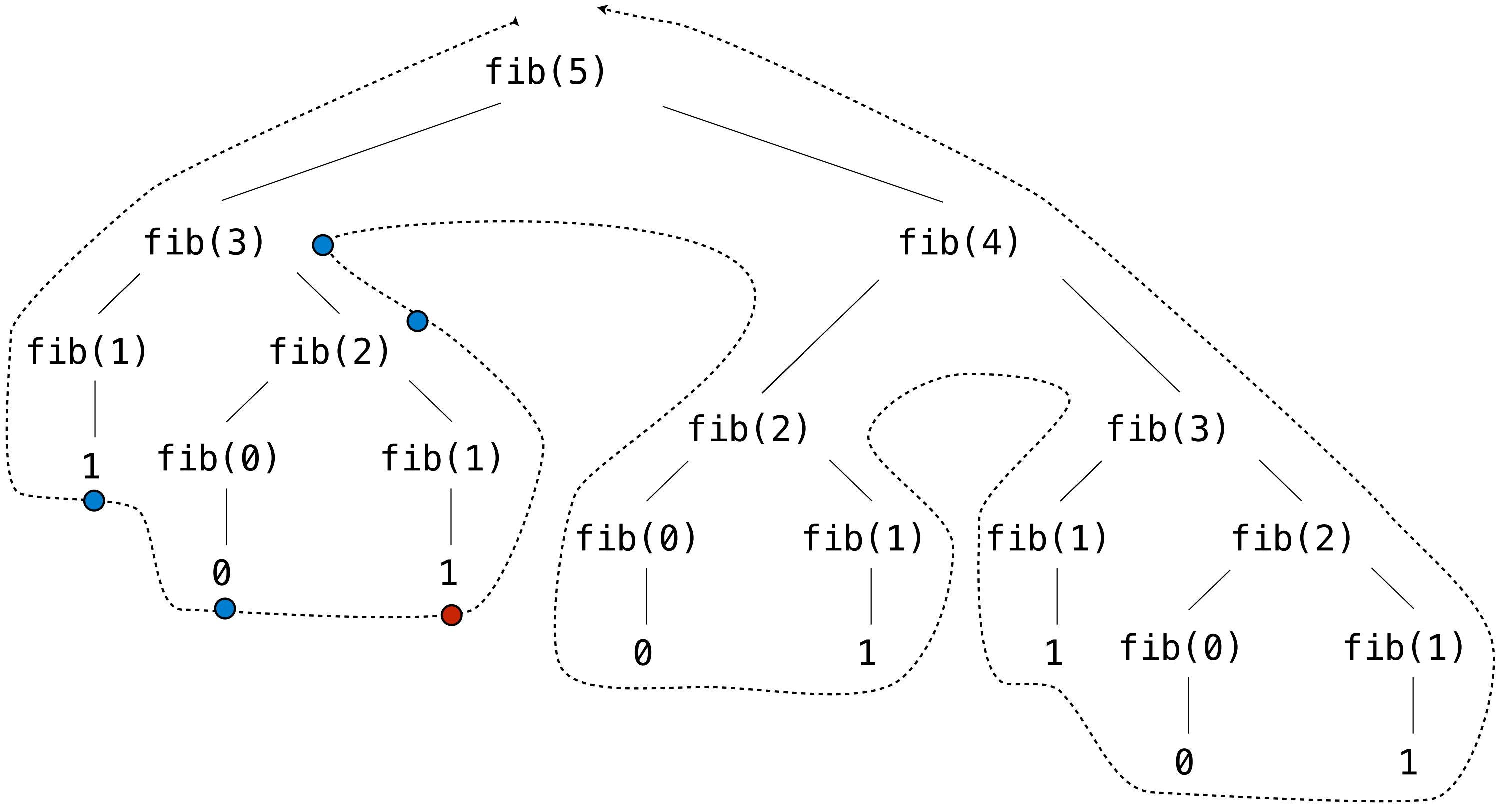
Removing subtrees from a tree is called *pruning*

Prune branches before recursive processing

E.g., want to prune cached (previously memorized) values.

Memoization:

- Returned by fib
- Found in cache
- Skipped



Example: Pruning Trees

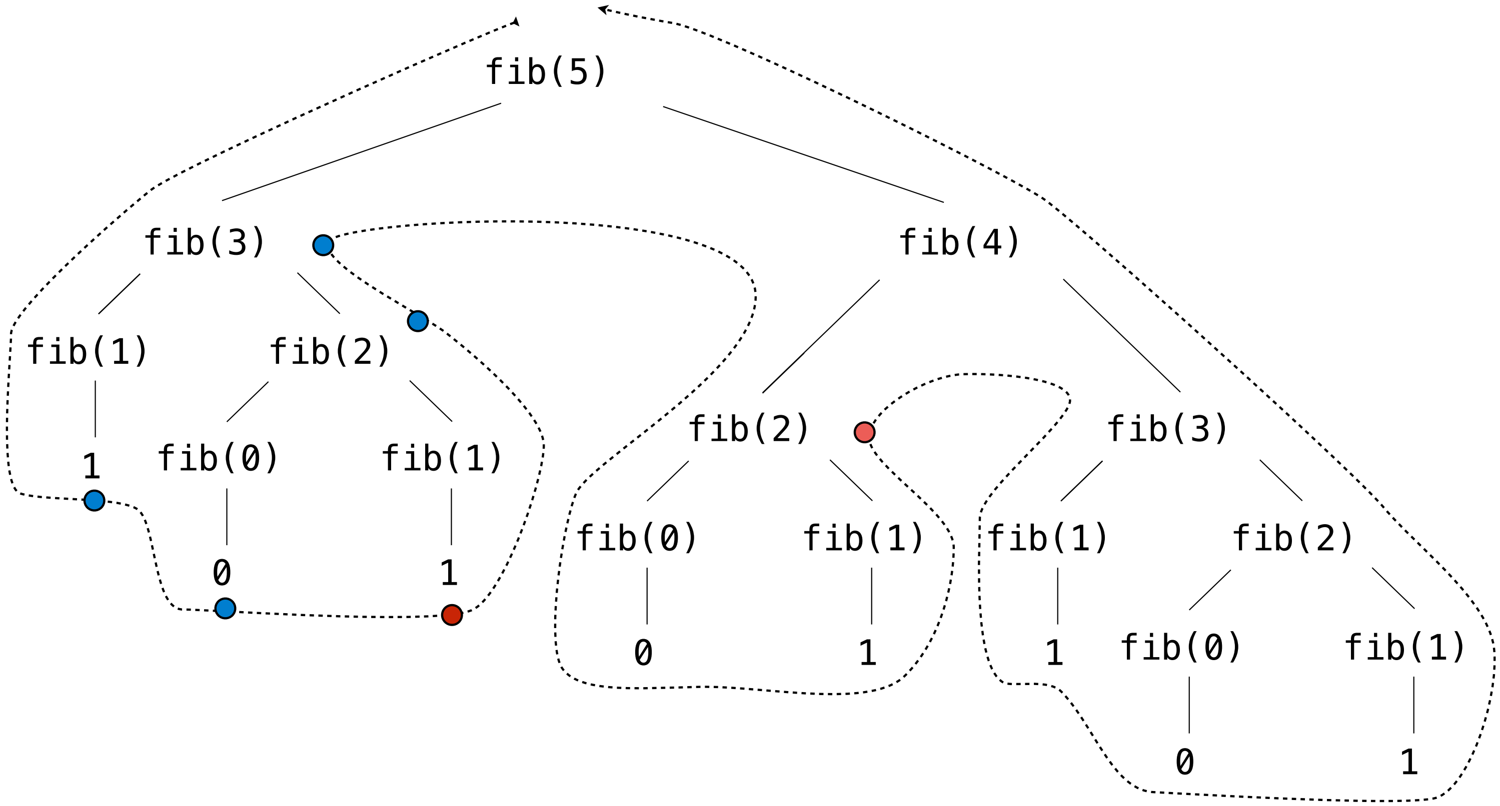
Removing subtrees from a tree is called *pruning*

Prune branches before recursive processing

E.g., want to prune cached (previously memorized) values.

Memoization:

- Returned by fib
- Found in cache
- Skipped



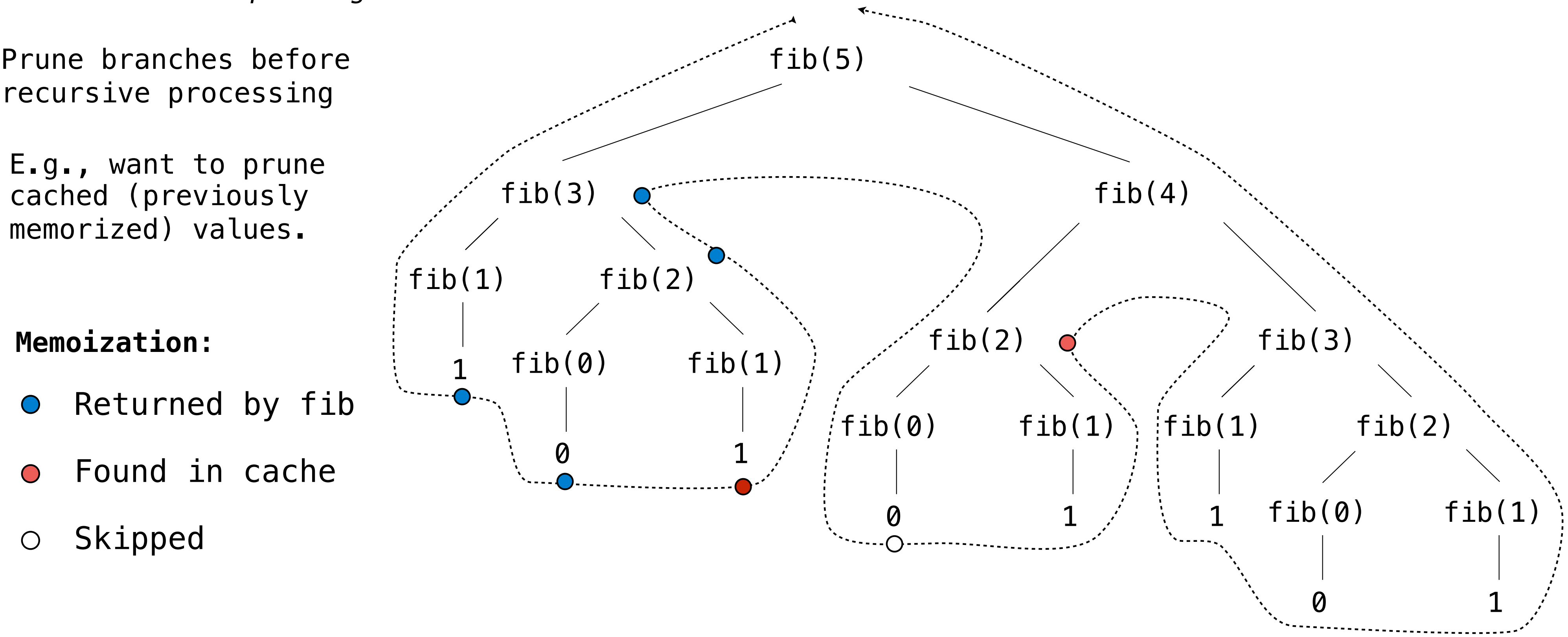
Example: Pruning Trees

Removing subtrees from a tree is called *pruning*

Prune branches before recursive processing

E.g., want to prune cached (previously memorized) values.

Memoization:



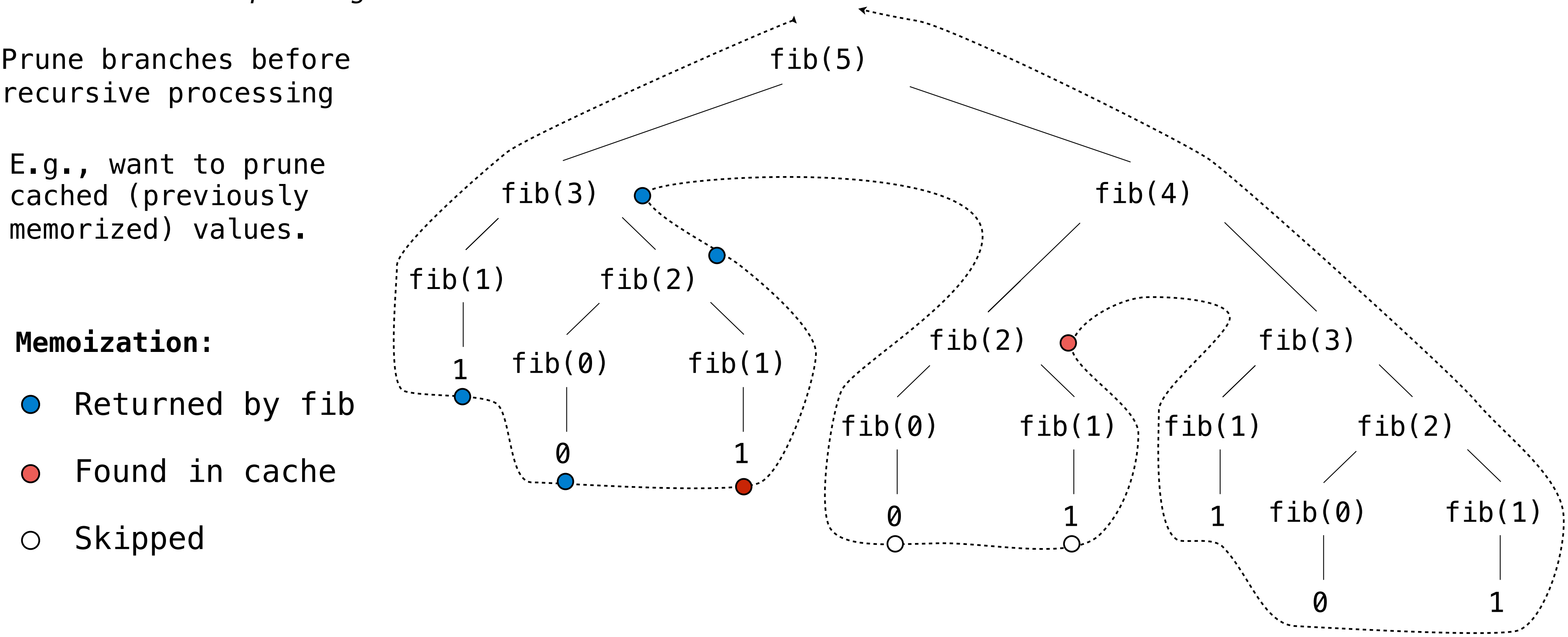
Example: Pruning Trees

Removing subtrees from a tree is called *pruning*

Prune branches before recursive processing

E.g., want to prune cached (previously memorized) values.

Memoization:



Example: Pruning Trees

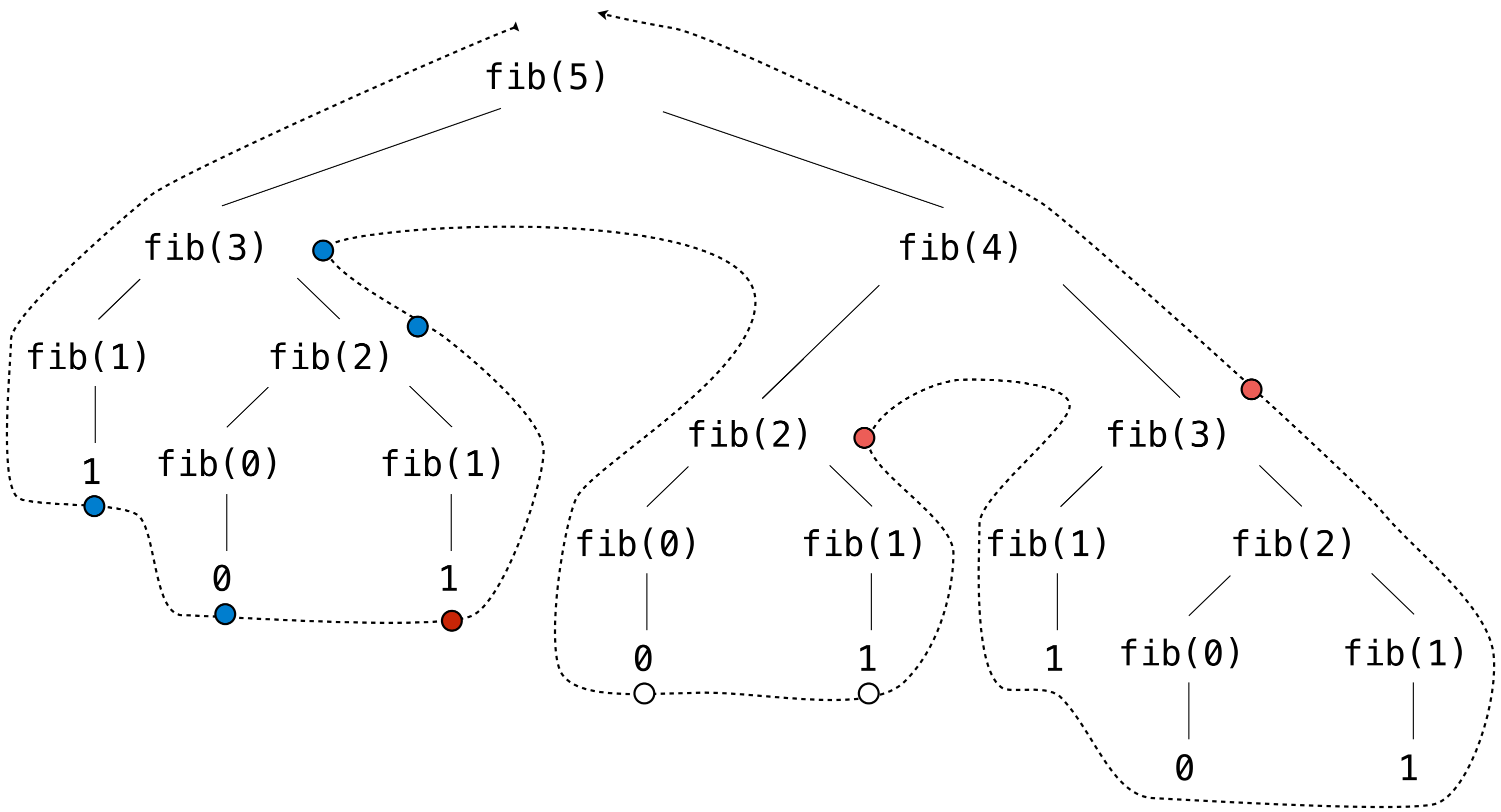
Removing subtrees from a tree is called *pruning*

Prune branches before recursive processing

E.g., want to prune cached (previously memorized) values.

Memoization:

- Returned by fib
- Found in cache
- Skipped



Example: Pruning Trees

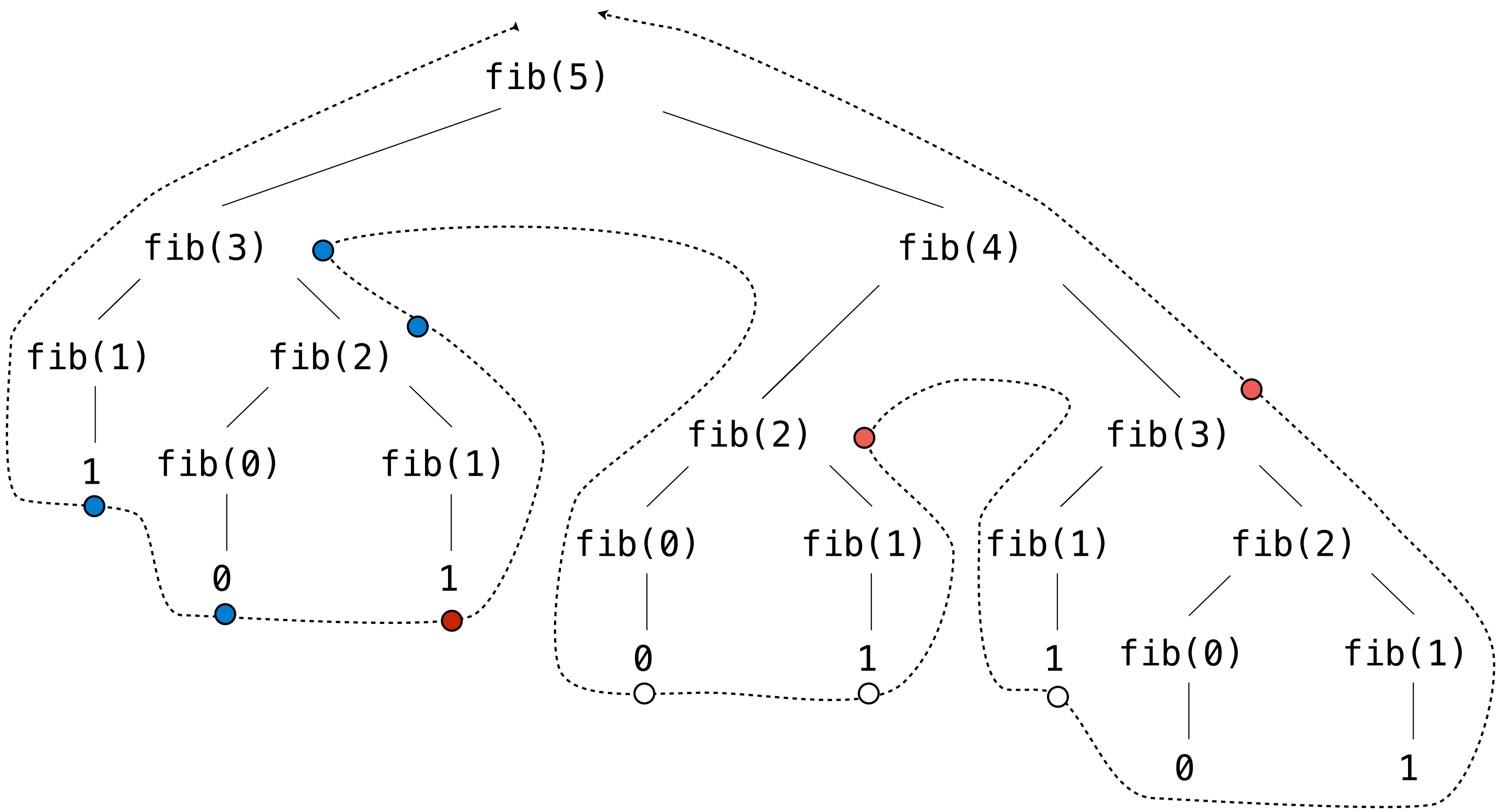
Removing subtrees from a tree is called *pruning*

Prune branches before recursive processing

E.g., want to prune cached (previously memorized) values.

Memoization:

- Returned by fib
- Found in cache
- Skipped

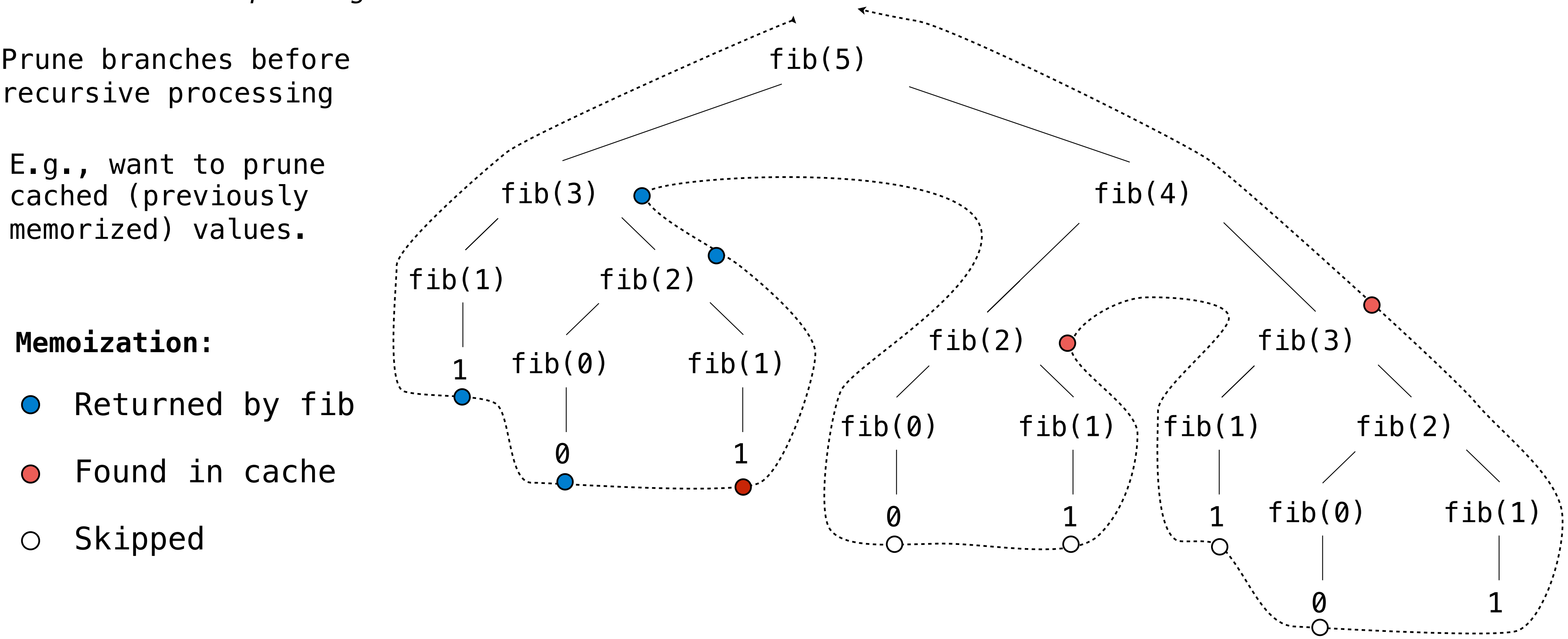


Example: Pruning Trees

Removing subtrees from a tree is called *pruning*

Prune branches before recursive processing

E.g., want to prune cached (previously memorized) values.



Example: Pruning Trees

Removing subtrees from a tree is called *pruning*

Prune branches before recursive processing

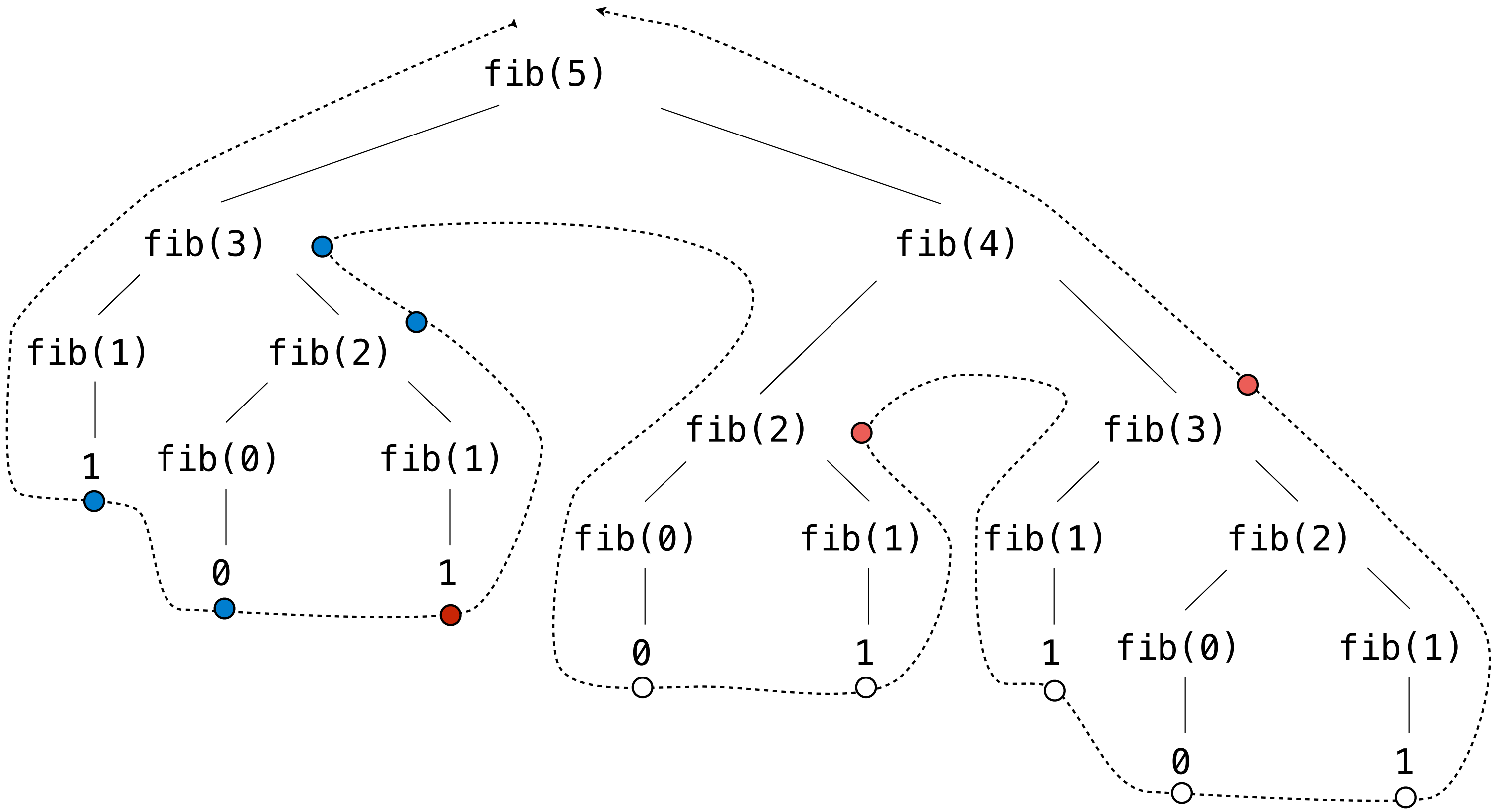
E.g., want to prune cached (previously memorized) values.

Memoization:

Returned by fib

Found in cache

Skipped



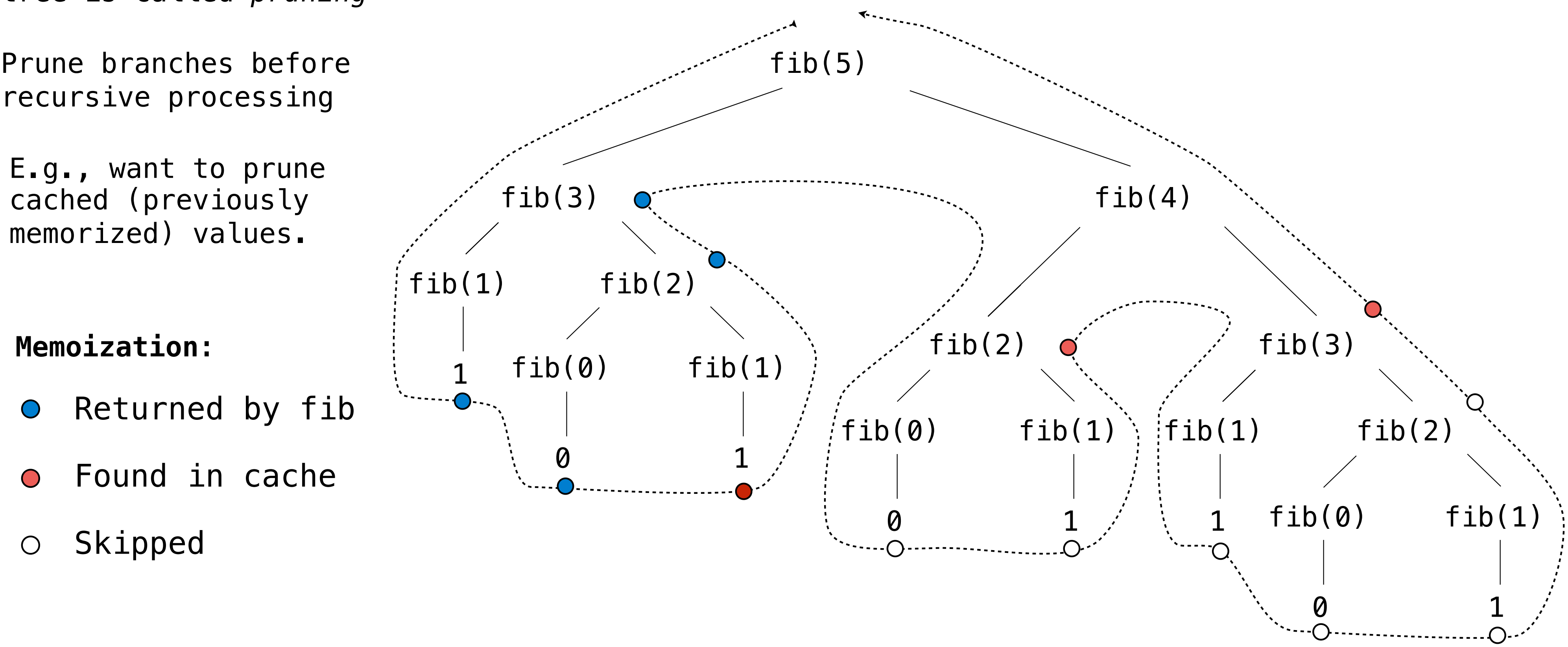
Example: Pruning Trees

Removing subtrees from a tree is called *pruning*

Prune branches before recursive processing

E.g., want to prune cached (previously memorized) values.

Memoization:



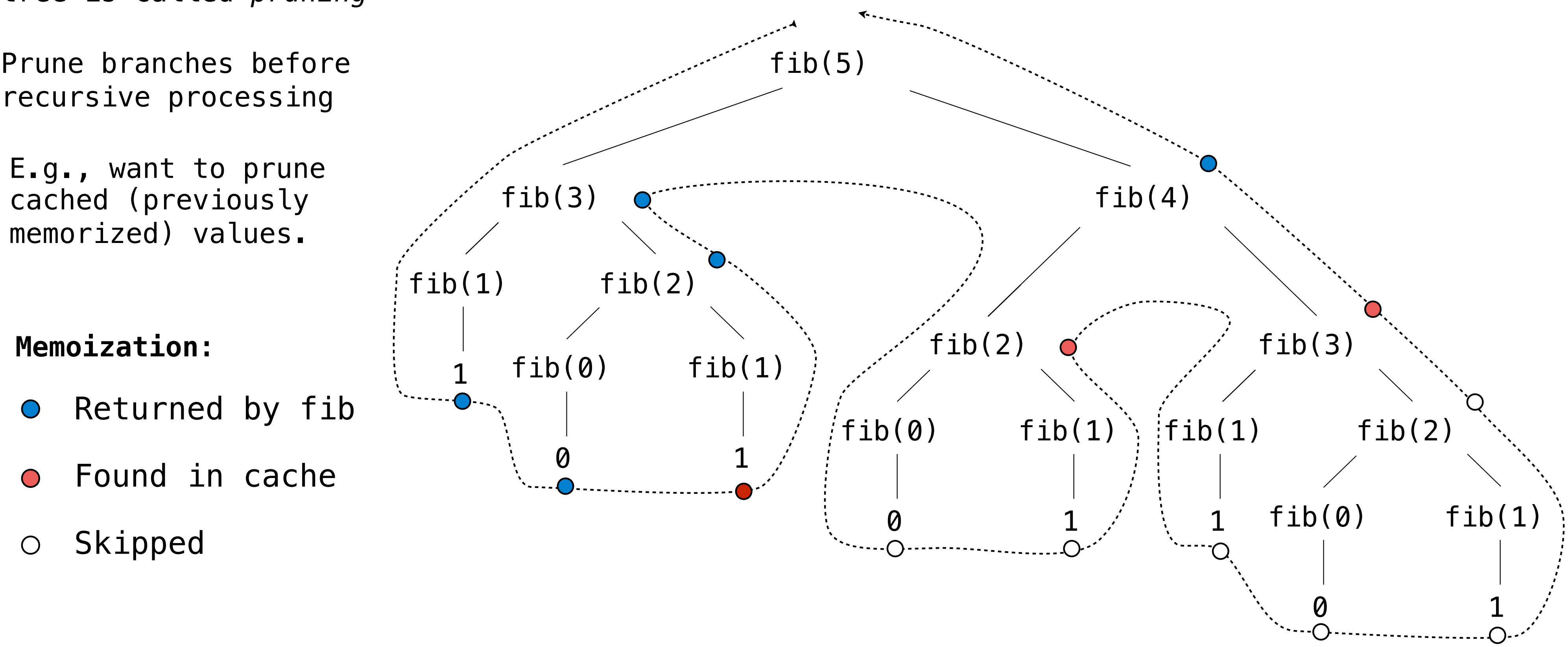
Example: Pruning Trees

Removing subtrees from a tree is called *pruning*

Prune branches before recursive processing

E.g., want to prune cached (previously memorized) values.

Memoization:

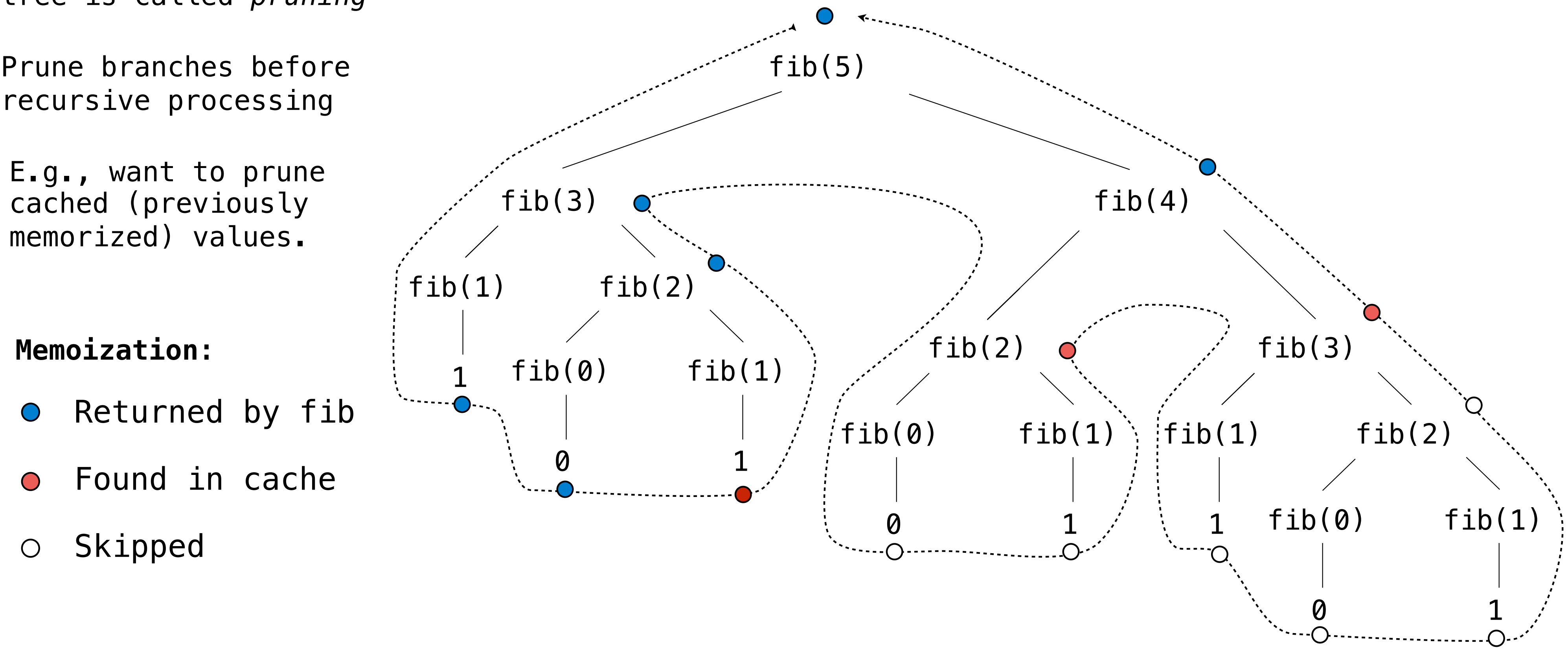


Example: Pruning Trees

Removing subtrees from a tree is called *pruning*

Prune branches before recursive processing

E.g., want to prune cached (previously memorized) values.



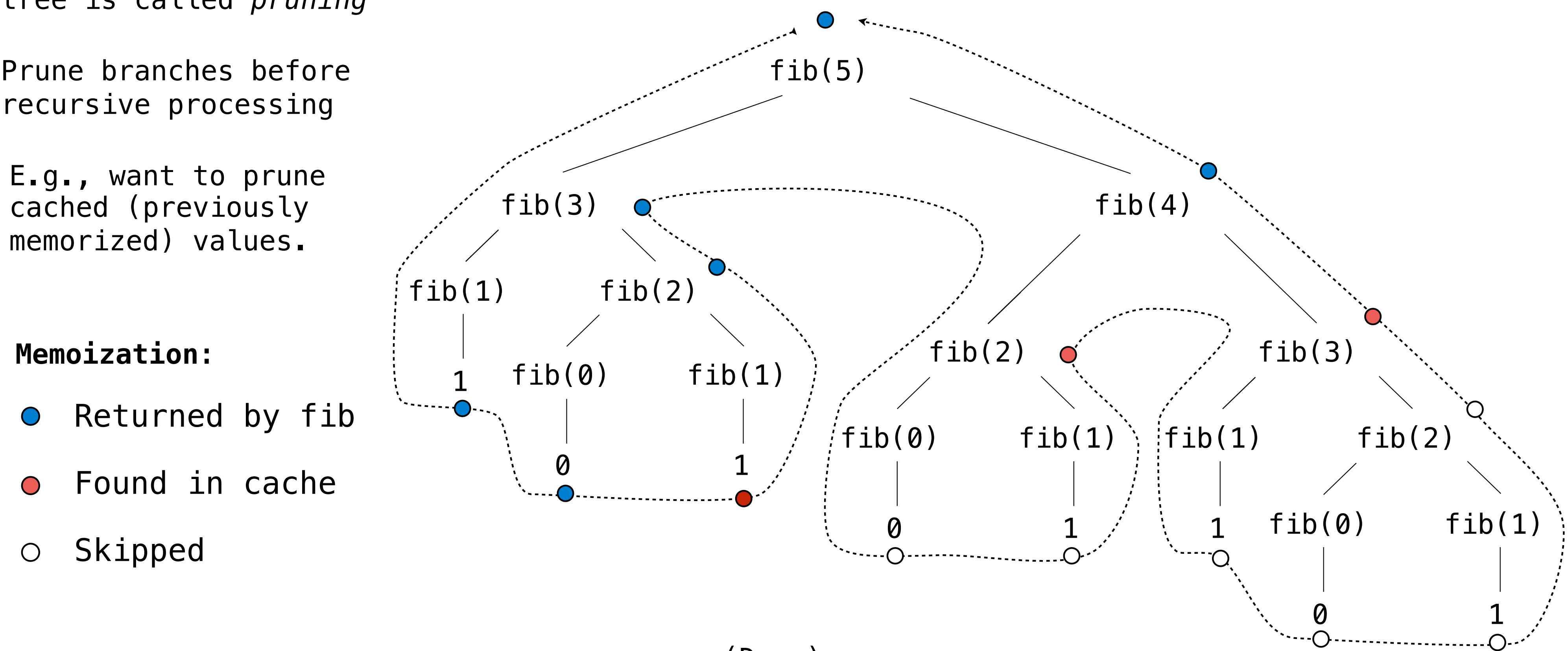
Example: Pruning Trees

Removing subtrees from a tree is called *pruning*

Prune branches before recursive processing

E.g., want to prune cached (previously memorized) values.

Memoization:



(Demo)

Hailstone Trees

Hailstone Trees

Hailstone Trees

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

Hailstone Trees

```
Pick a positive integer n as the start

If n is even, divide it by 2
```

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
```

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
```

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)
```

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

(Demo)

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)

1

2

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)
```

1

2

4

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)

1

2

4

8

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)
```

1

2

4

8

16

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)

1

2

4

8

16

32

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)

1

2

4

8

16

32

64

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)

1

2

4

8

16

32

64

128

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

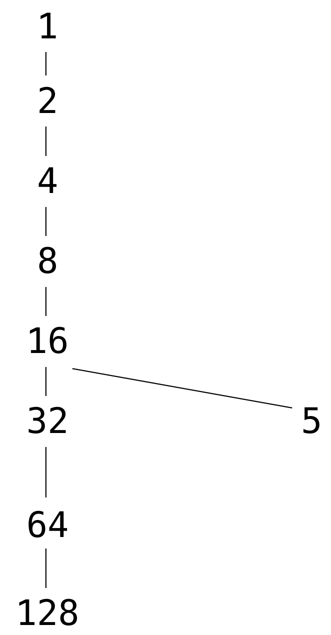
(Demo)

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

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)



Hailstone Trees

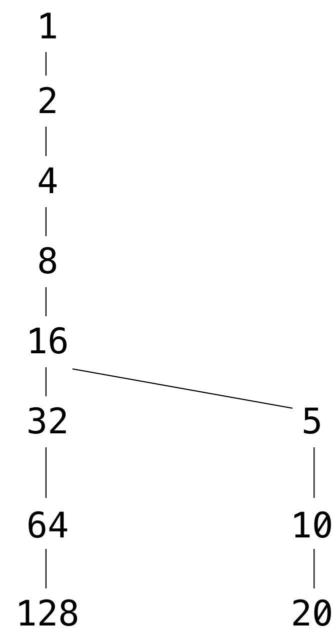
Pick a positive integer n as the start

If n is even, divide it by 2

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

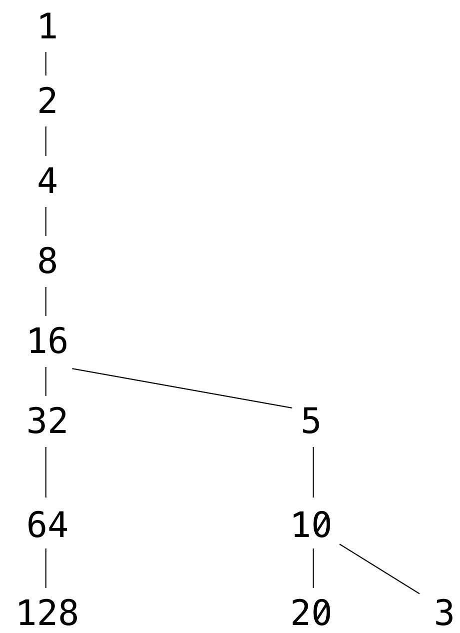
Continue this process until n is 1

(Demo)



Hailstone Trees

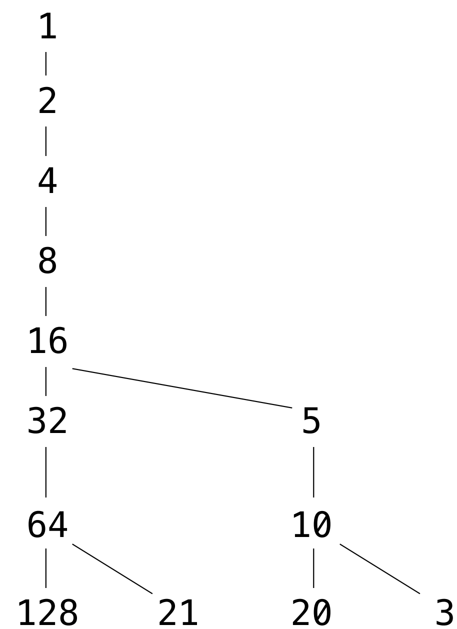
```
Pick a positive integer n as the start

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)
```



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
```

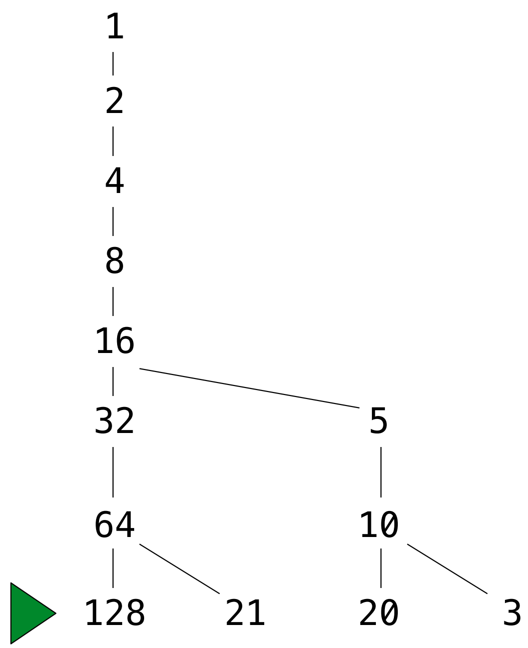
(Demo)



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)
```

All possible n that start a length-8 hailstone sequence



Hailstone Trees

Pick a positive integer n as the start

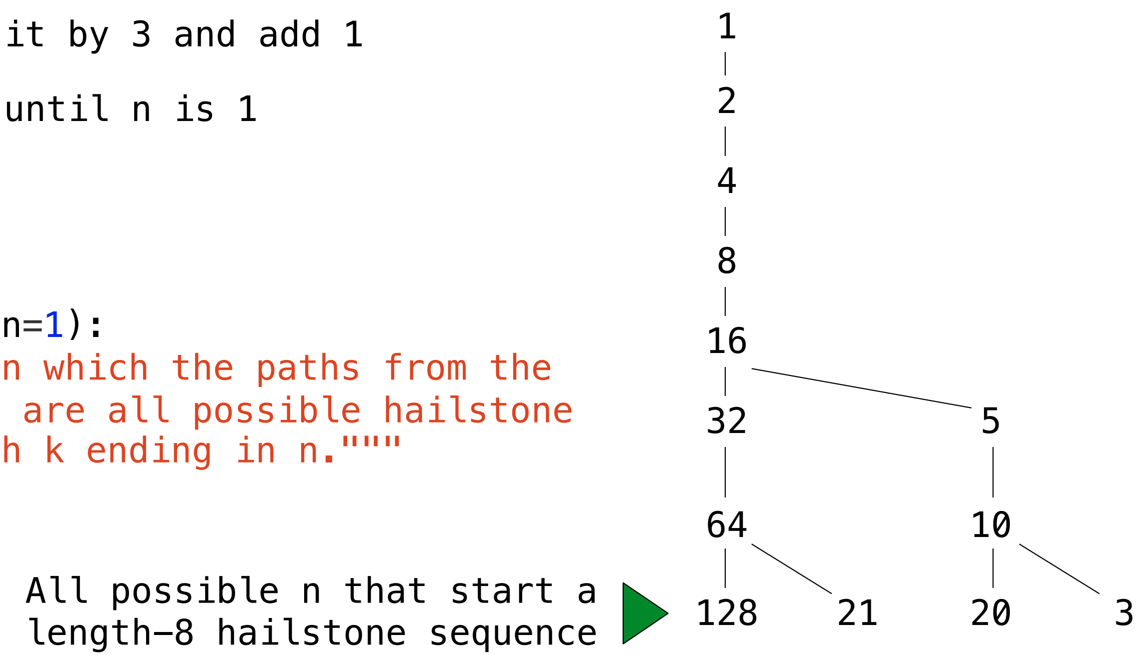
If n is even, divide it by 2

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

Continue this process until n is 1

(Demo)

```
def hailstone_tree(k, n=1):
    """Return a Tree in which the paths from the
    leaves to the root are all possible hailstone
    sequences of length k ending in n."""
```



Hailstone Trees

Pick a positive integer n as the start

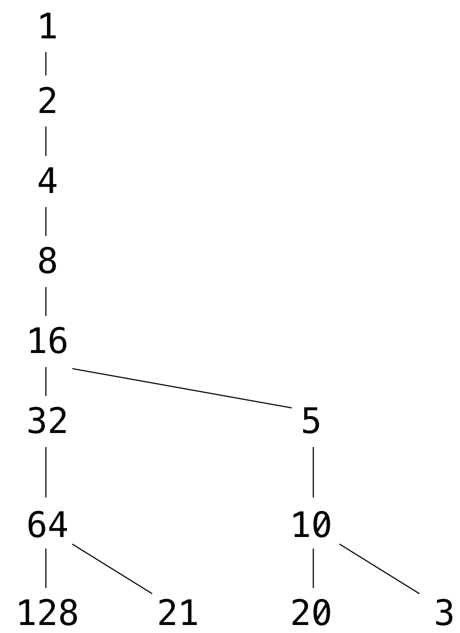
If n is even, divide it by 2

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

Continue this process until n is 1

(Demo)

```
def hailstone_tree(k, n=1):
    """Return a Tree in which the paths from the
    leaves to the root are all possible hailstone
    sequences of length k ending in n."""
```



All possible n that start a length-8 hailstone sequence

(Demo)