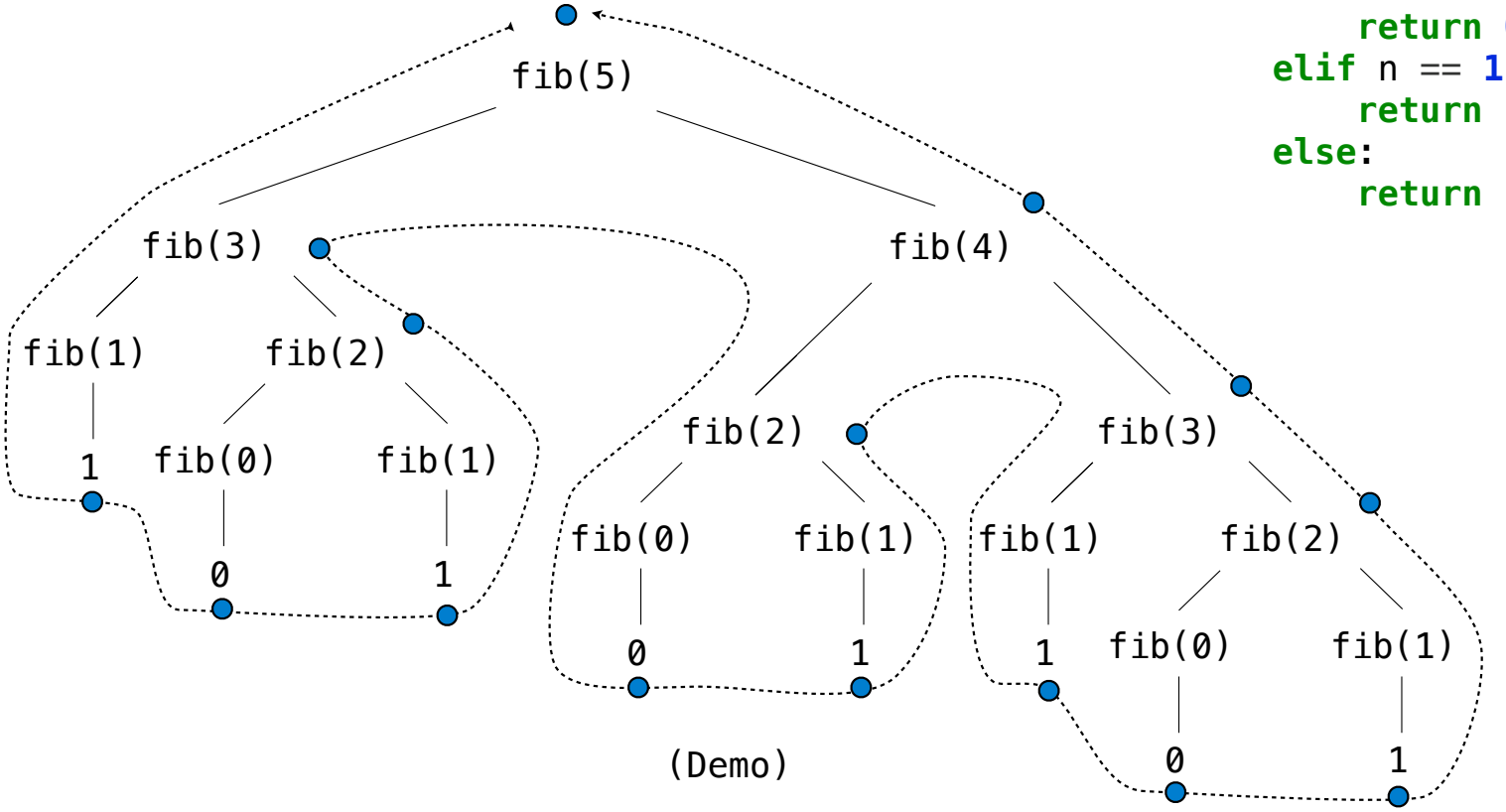
Efficiency

Announcements

Measuring Efficiency

Recursive Computation of the Fibonacci Sequence

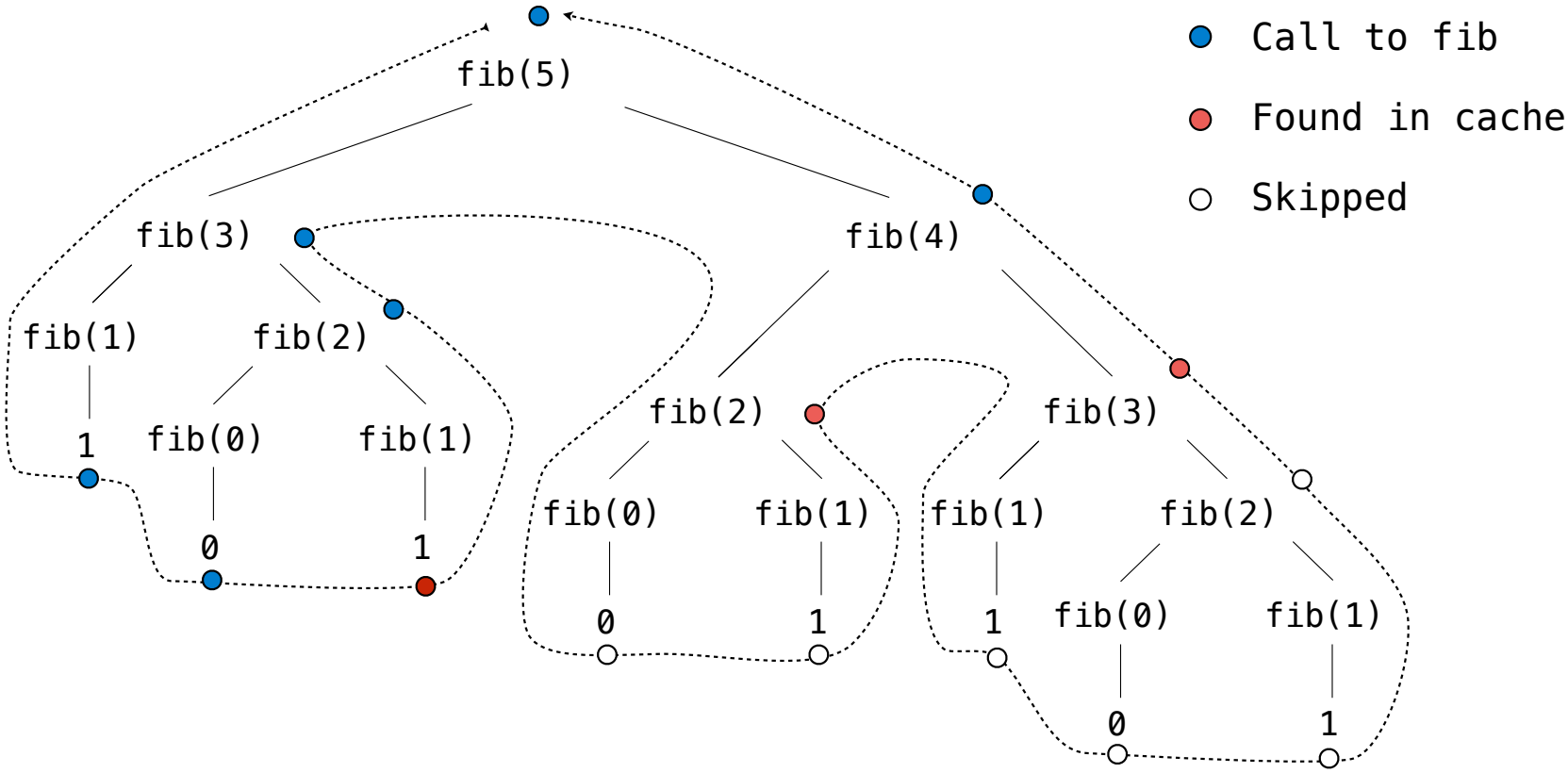
Our first example of tree recursion:




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

Memoization

Memoized Tree Recursion



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)

List Efficiency

List Efficiency

Appending, assigning, and list comprehensions are fast:



Inserting (beginning/middle), slicing, and adding lists are slow:



Example: building long lists of perfect squares (numbers that are an integer times itself)

```
def using_list_comprehension(n):
    return [k * k for k in range(n)]

def using_append(n):
    s = []
    for k in range(n):
        s.append(k * k)
    return s

def using_assign(n):
    s = [0 for k in range(n)]
    for k in range(n):
        s[k] = k * k
    return s
```

```
def using_insert(n):
    s = []
    for k in range(n):
        s.insert(0, k * k)
    return s

def using_add(n):
    s = []
    for k in range(n):
        s = s + [k*k]
    return s
```

```
When n = 100,000
using_list_comprehension:  1.58 ms
using_append:              1.76 ms
using_assign:              2.58 ms
using_insert:          1,470    ms
using_add:             9,210    ms
```

Generator Practice

Example: Powers

A generator function returns a generator which is an iterator

```
def powers(b):
    """Yield the powers of b."""
    n = 0
    __________:
        ________________________
        n += 1

def prefix(t, n):
    """Yield the first n items of an infinite iterator t.

    >>> list(prefix(powers(2), 10))
    [1, 2, 4, 8, 16, 32, 64, 128, 256, 512]
    """
    for i in range(n):
        _____________
```

Example: Powers

A generator function returns a generator which is an iterator

```
def powers(b):
    """Yield the powers of b."""
    n = 0
    while True:
        yield pow(b, n)
        n += 1

def prefix(t, n):
    """Yield the first n items of an infinite iterator t.

    >>> list(prefix(powers(2), 10))
    [1, 2, 4, 8, 16, 32, 64, 128, 256, 512]
    """
    for i in range(n):
        yield next(t)
```

Higher-Order Generator Functions

Generator functions can take functions as arguments

```
def powers(b):
    """Yield the powers of b."""
    n = 0
    while True:
        yield pow(b, n)
        n += 1
```

```
from operator import add

def map_pairs(f, t):
    """Yield the result of calling f on each pair of consecutive items in t.
    (You can assume t has at least two items)

    >>> list(map_pairs(add, iter([3, 4, 5, 6, 7])))
    [7, 9, 11, 13]
    >>> list(prefix(map_pairs(lambda x, y: (x, y), powers(2)), 5))
    [(1, 2), (2, 4), (4, 8), (8, 16), (16, 32)]
    """
    ________________________
    for x in t:
        ____________________
        ____________________
```

Higher-Order Generator Functions

Generator functions can take functions as arguments

```
from operator import add

def map_pairs(f, t):
    """Yield the result of calling f on each pair of consecutive items in t.
    (You can assume t has at least two items)

    >>> list(map_pairs(add, iter([3, 4, 5, 6, 7])))
    [7, 9, 11, 13]
    >>> list(prefix(map_pairs(lambda x, y: (x, y), powers(2)), 5))
    [(1, 2), (2, 4), (4, 8), (8, 16), (16, 32)]
    """
    previous = next(t)
    for x in t:
        yield f(previous, x)
        previous = x
```

```
def powers(b):
    """Yield the powers of b."""
    n = 0
    while True:
        yield pow(b, n)
        n += 1
```

Generators for Linked Lists and Trees

```
def items(s: LinkedList):
    """Yield the items in a linked list.

    >>> nums = Link(3, Link(5, Link(7)))
    >>> sum(items(nums))
    15
    """
    _____________________________:
        yield s.first

        ____________________
```

```
def labels(t: Tree):
    """Yield the labels of a tree.

    >>> list(labels(Tree(1, [Tree(2, [Tree(3), Tree(4)]), Tree(5)])))
    [1, 2, 3, 4, 5]
    """
    _______________
    for b in t.branches:

        ____________________
```

```
type LinkedList[T] = Link[T] | tuple[()]

@dataclass
class Link[T]:
    first: T
    rest: LinkedList[T] = ()  # rest defaults to empty
```

```
@dataclass
class Tree[T]:
    label: T
    branches: list[Tree[T]] = ...  # branches defaults to []
```

Generators for Linked Lists and Trees

```
def items(s: LinkedList):
    """Yield the items in a linked list.

    >>> nums = Link(3, Link(5, Link(7)))
    >>> sum(items(nums))
    15
    """
    while isinstance(s, Link) :
        yield s.first
        s = s.rest
```

```
type LinkedList[T] = Link[T] | tuple[()]

@dataclass
class Link[T]:
    first: T
    rest: LinkedList[T] = ()  # rest defaults to empty
```

```
def labels(t: Tree):
    """Yield the labels of a tree.

    >>> list(labels(Tree(1, [Tree(2, [Tree(3), Tree(4)]), Tree(5)])))
    [1, 2, 3, 4, 5]
    """
    yield t.label
    for b in t.branches:
        yield from labels(b)
```

```
@dataclass
class Tree[T]:
    label: T
    branches: list[Tree[T]] = ...  # branches defaults to []
```