61A Lecture 18

Wednesday, October 16

Announcements

Announcements

- Homework 6 is due Tuesday 10/22 @ 11:59pm

Announcements

- Homework 6 is due Tuesday 10/22 @ 11:59pm
- Project 3 is due Thursday 10/24 @ 11:59pm

Announcements

- Homework 6 is due Tuesday 10/22 @ 11:59pm
- Project 3 is due Thursday 10/24 @ 11:59pm
- Midterm 2 is on Monday 10/28 7pm–9pm

Announcements

- Homework 6 is due Tuesday 10/22 @ 11:59pm
- Project 3 is due Thursday 10/24 @ 11:59pm
- Midterm 2 is on Monday 10/28 7pm–9pm
- Hog strategy contest winners will be announced on Wednesday 10/16 in lecture

Memoization

Memoization

Idea: Remember the results that have been computed before

Memoization

Idea: Remember the results that have been computed before

```
def memo(f):
```

Memoization

Idea: Remember the results that have been computed before

```
def memo(f):
    cache = {}
```

Memoization

Idea: Remember the results that have been computed before

```
def memo(f):
    cache = {}
    def memoized(n):
```

Memoization

Idea: Remember the results that have been computed before

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

Memoization

Idea: Remember the results that have been computed before

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

Memoization

Idea: Remember the results that have been computed before

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

Memoization

Idea: Remember the results that have been computed before

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

Memoization

Idea: Remember the results that have been computed before

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

Keys are arguments that map to return values

Memoization

Idea: Remember the results that have been computed before

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

Keys are arguments that map to return values

Same behavior as f, if f is a pure function

Memoization

Idea: Remember the results that have been computed before

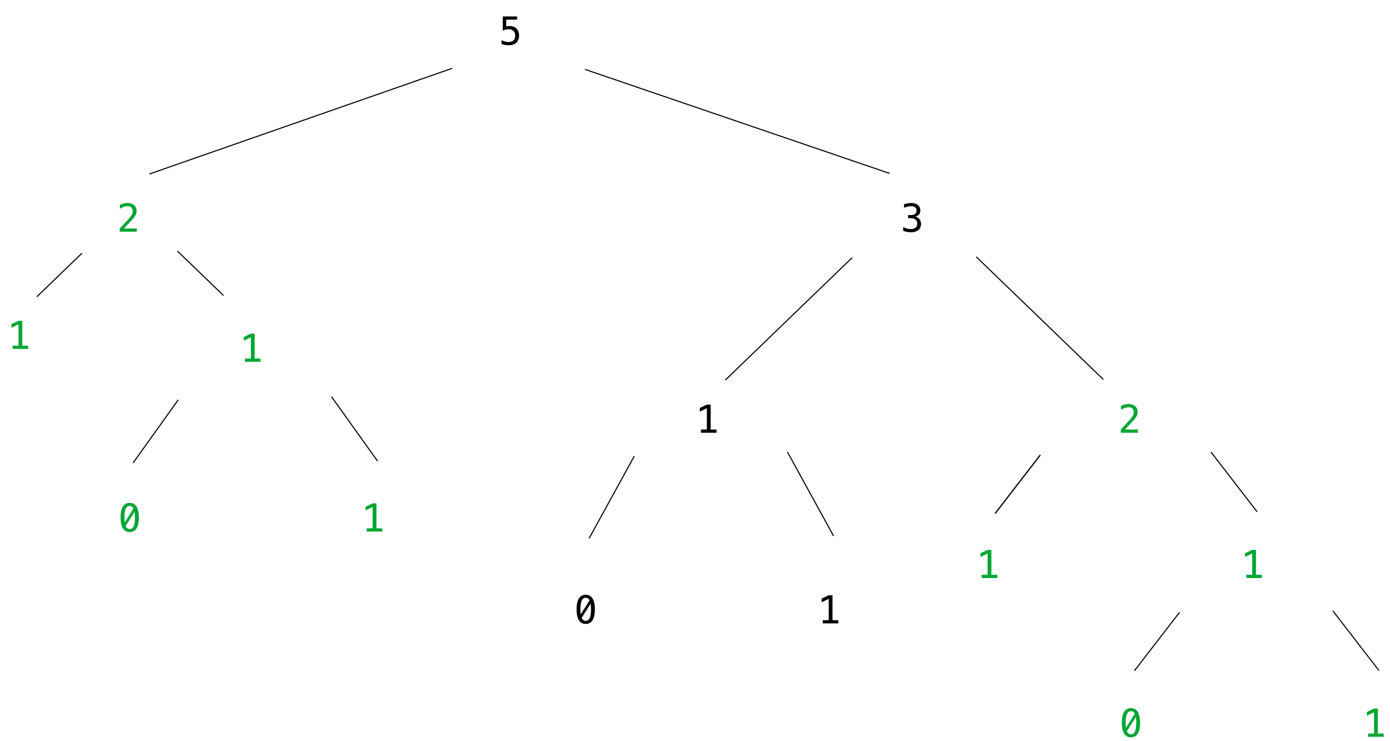
```
def memo(f):
    cache = {}
    def memoized(n):
        if n not in cache:
            cache[n] = f(n)
        return cache[n]
    return memoized
```

Keys are arguments that map to return values

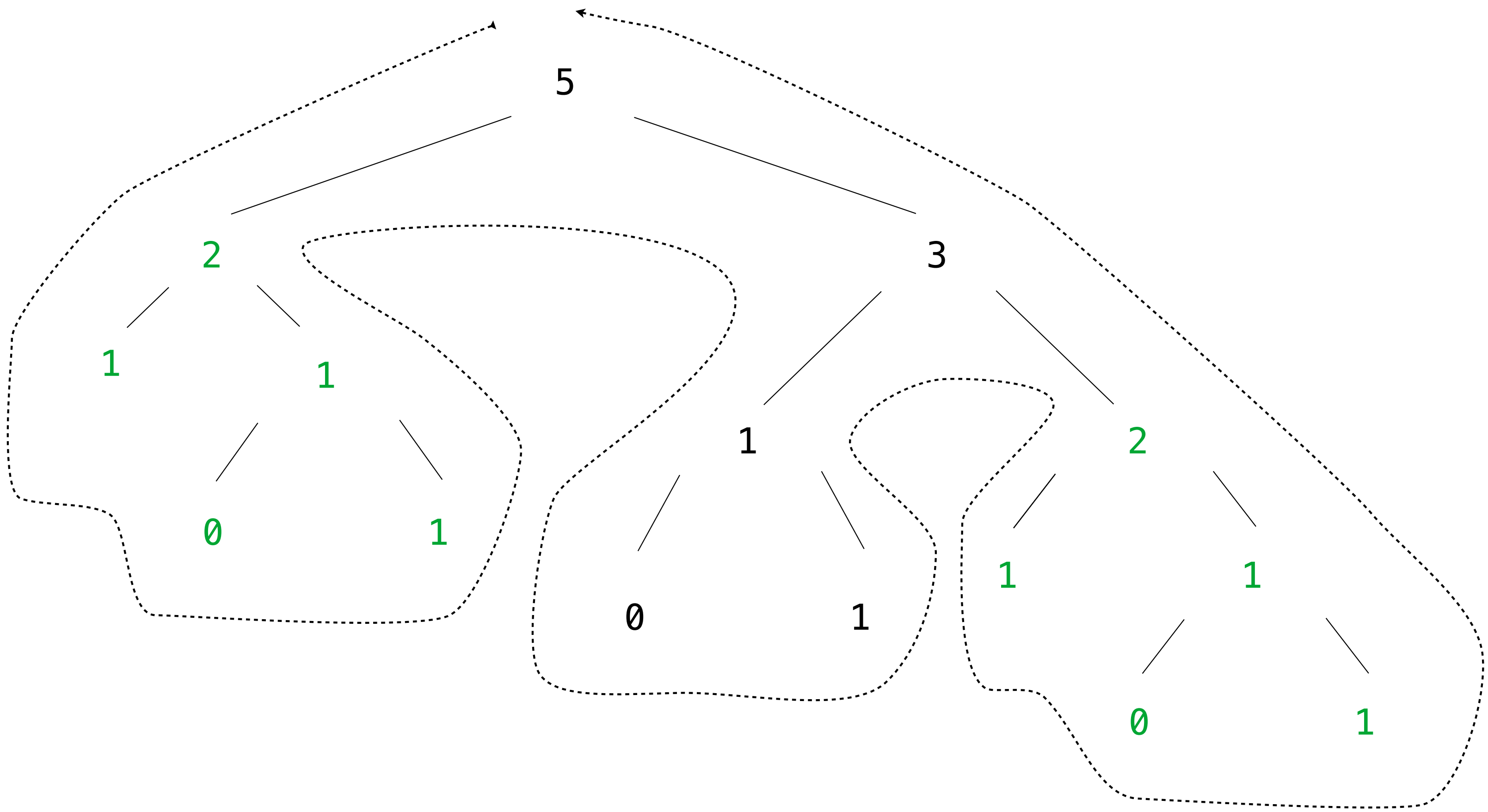
Same behavior as f, if f is a pure function

(Demo)

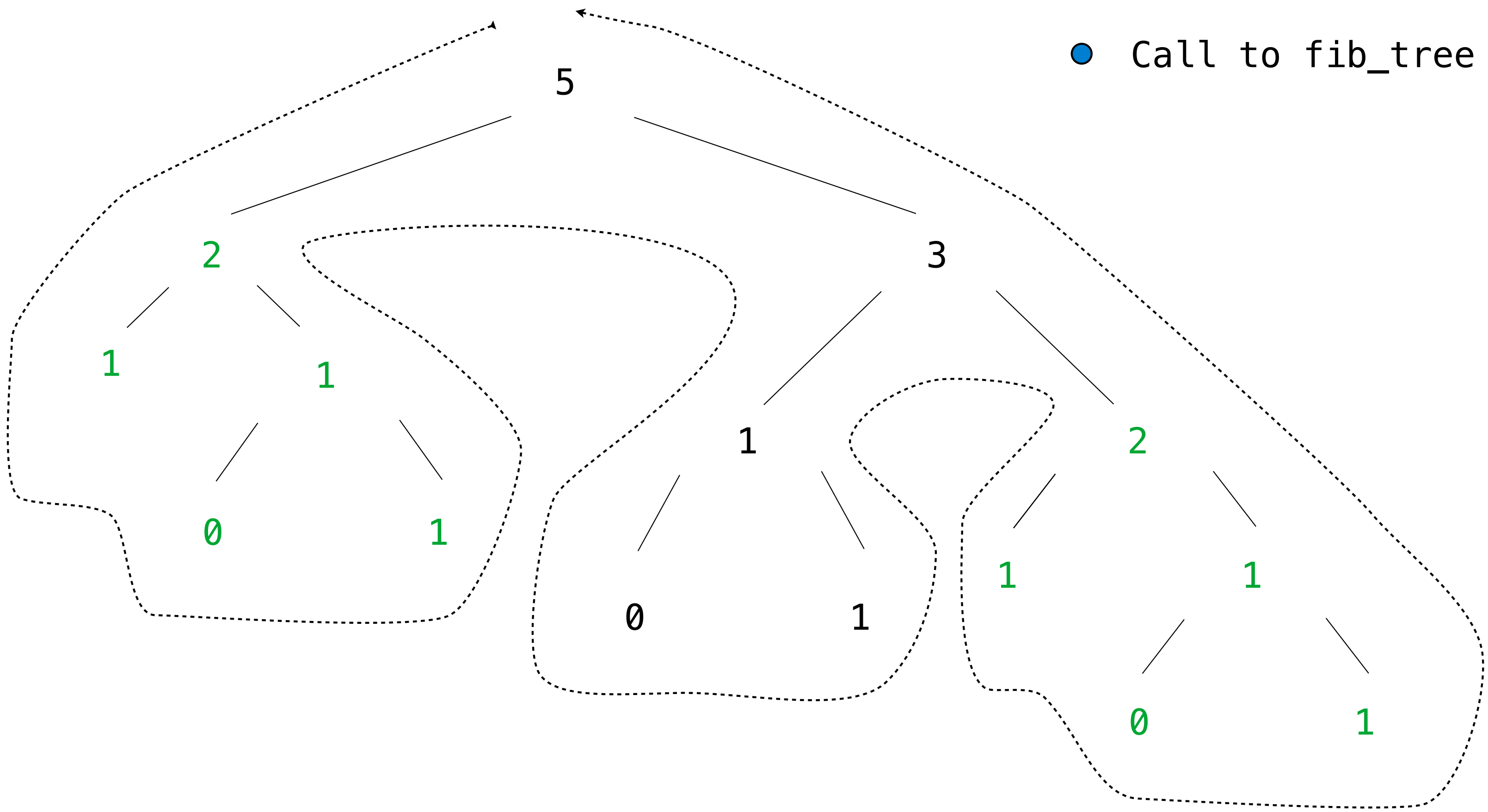
Memoized Tree Recursion



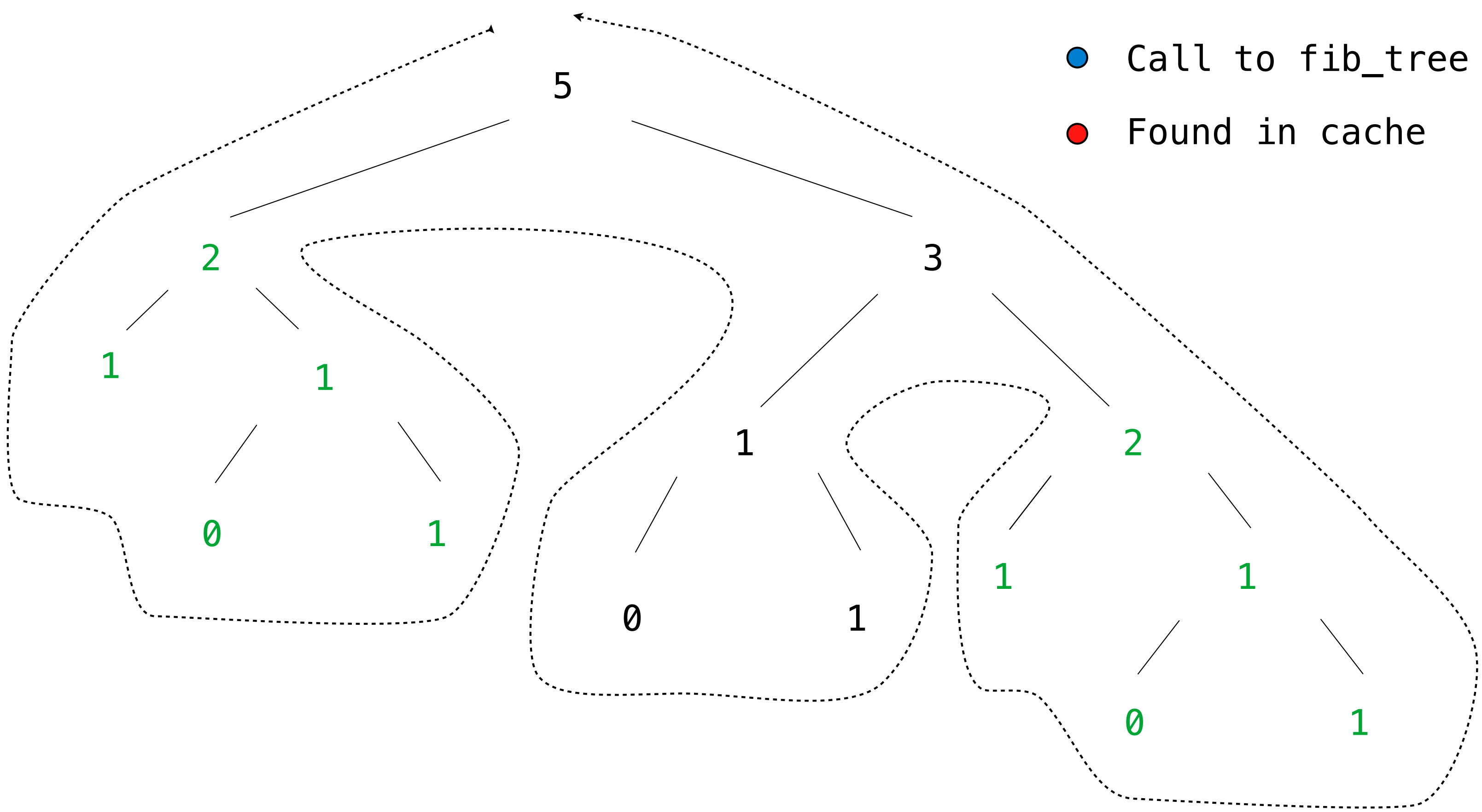
Memoized Tree Recursion



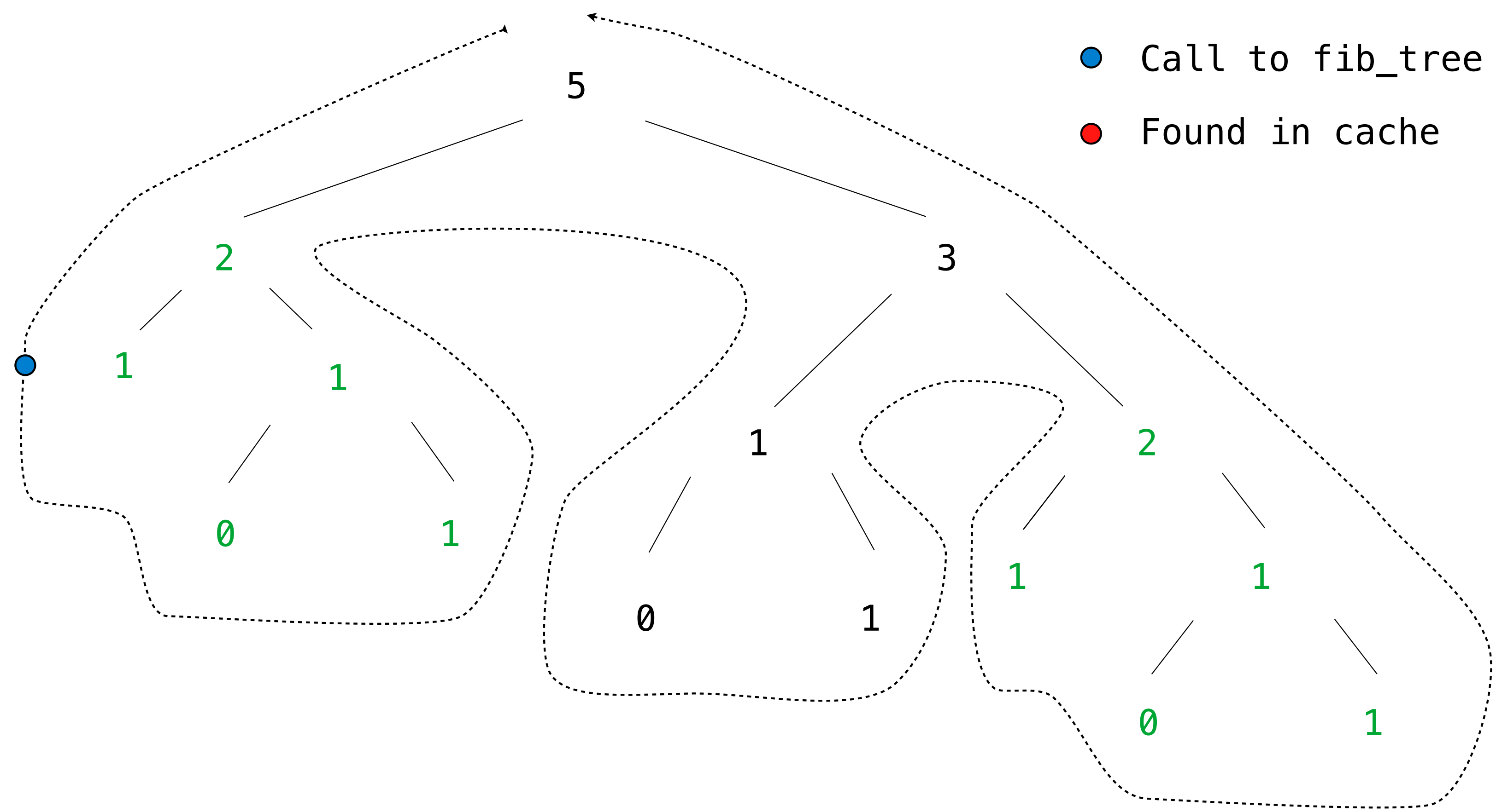
Memoized Tree Recursion



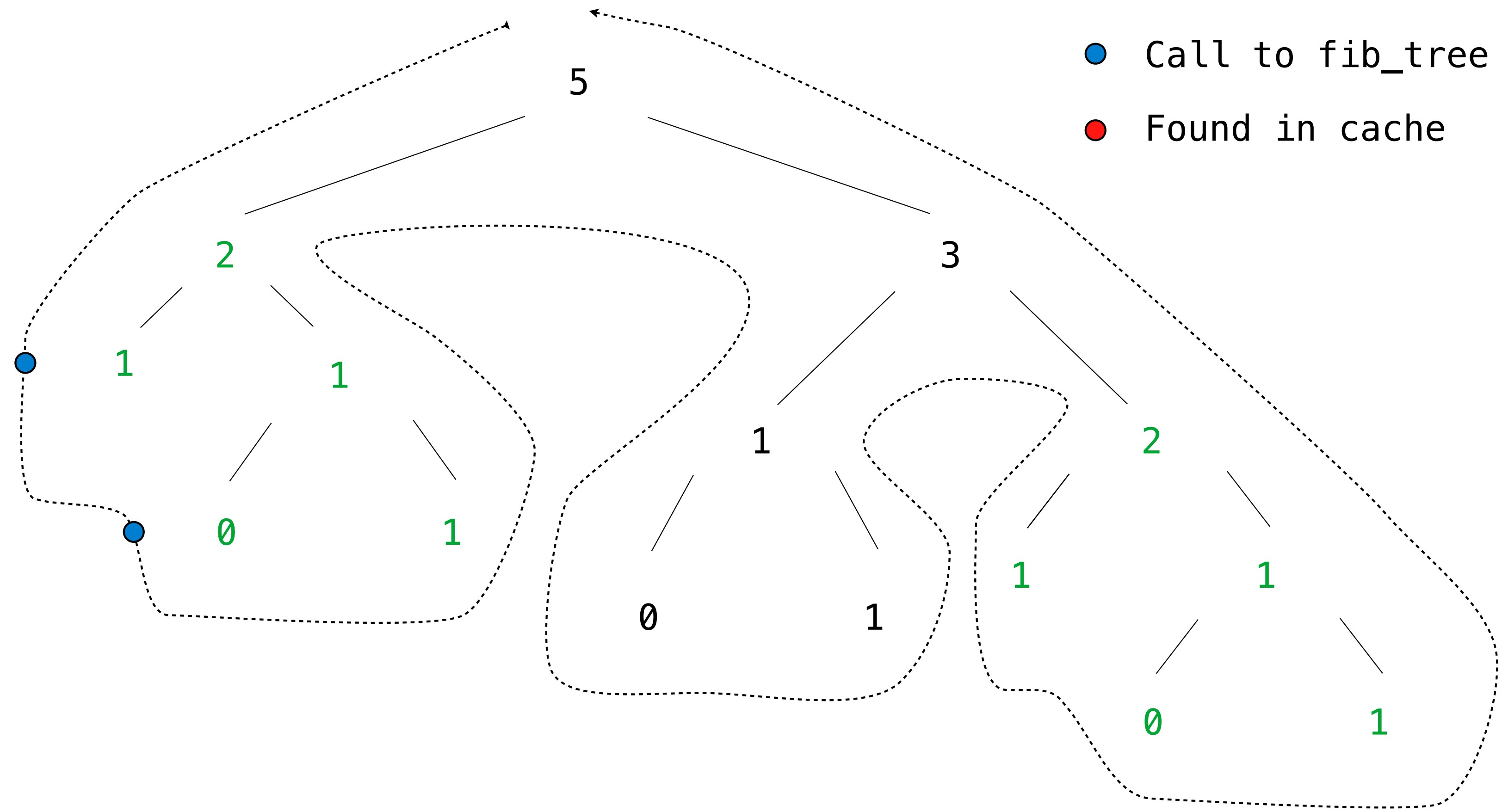
Memoized Tree Recursion



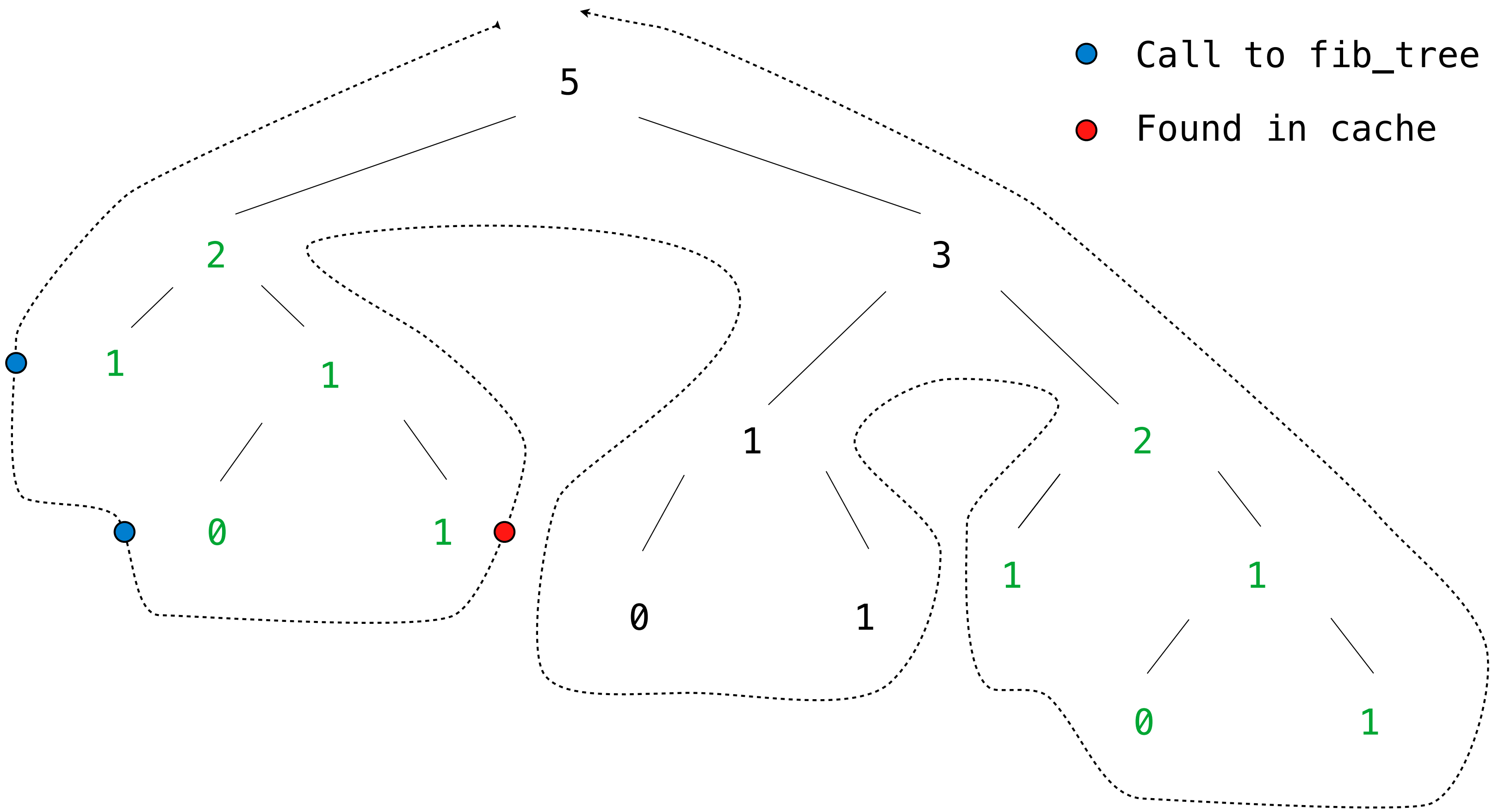
Memoized Tree Recursion



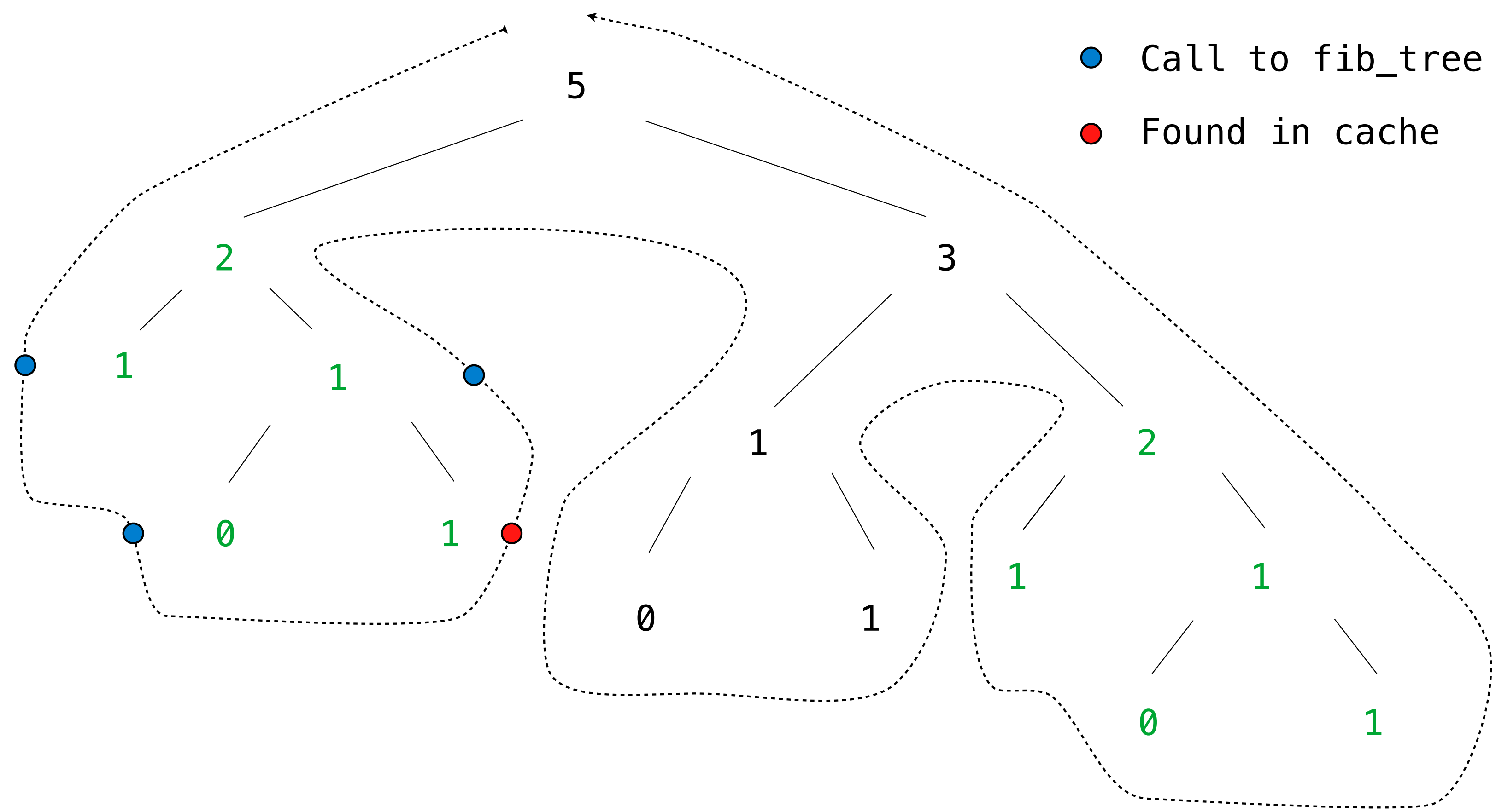
Memoized Tree Recursion



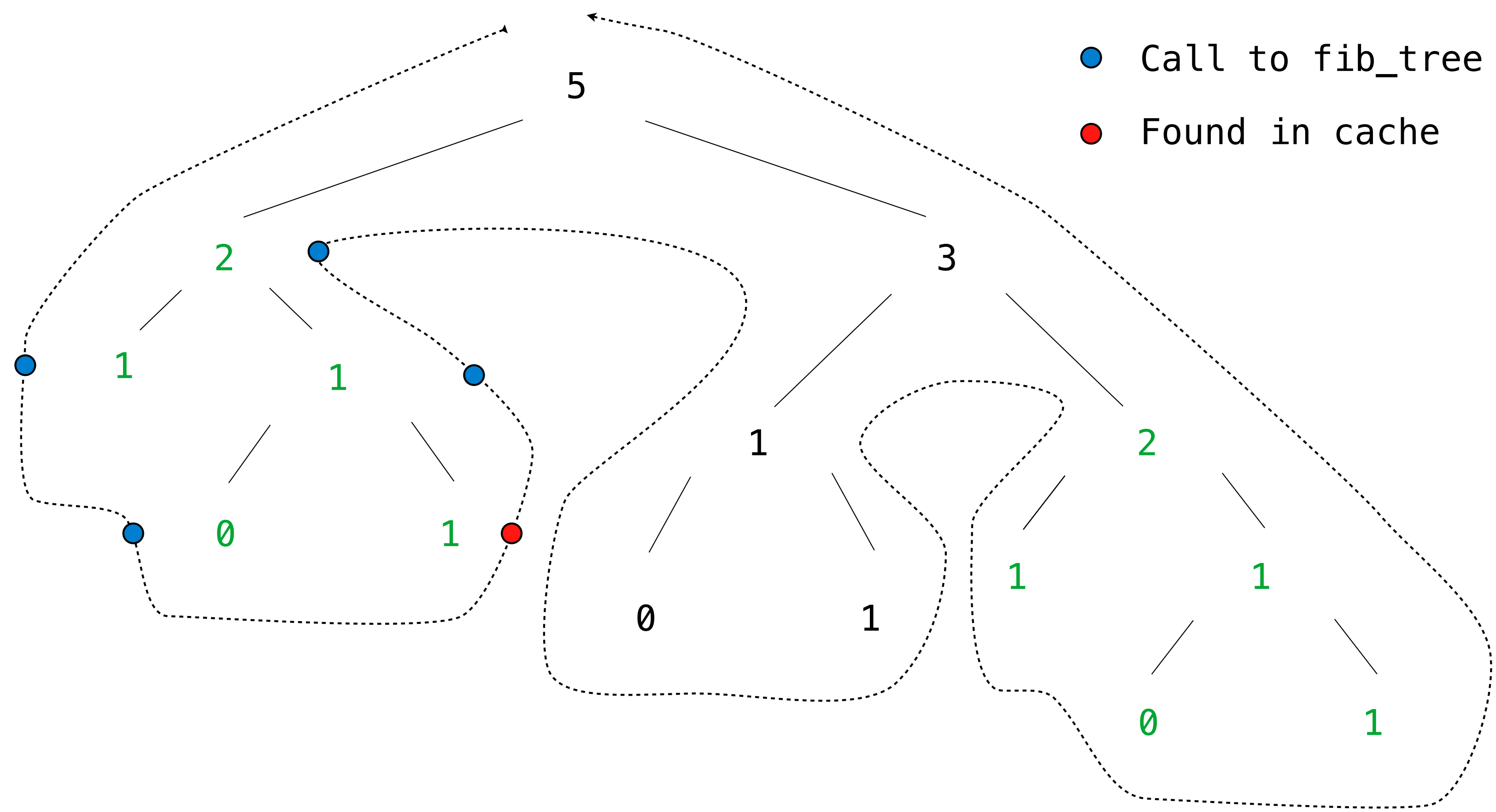
Memoized Tree Recursion



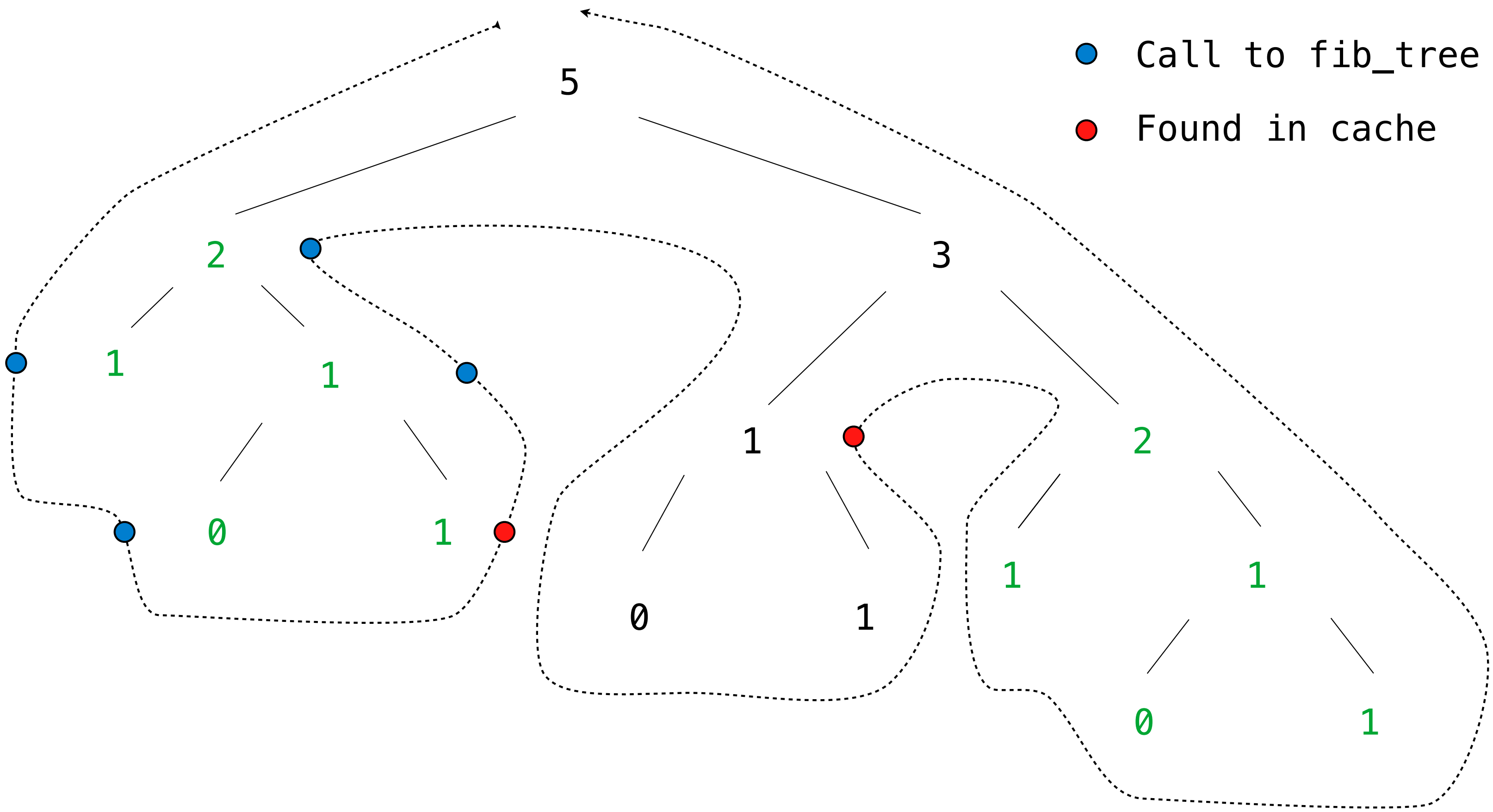
Memoized Tree Recursion



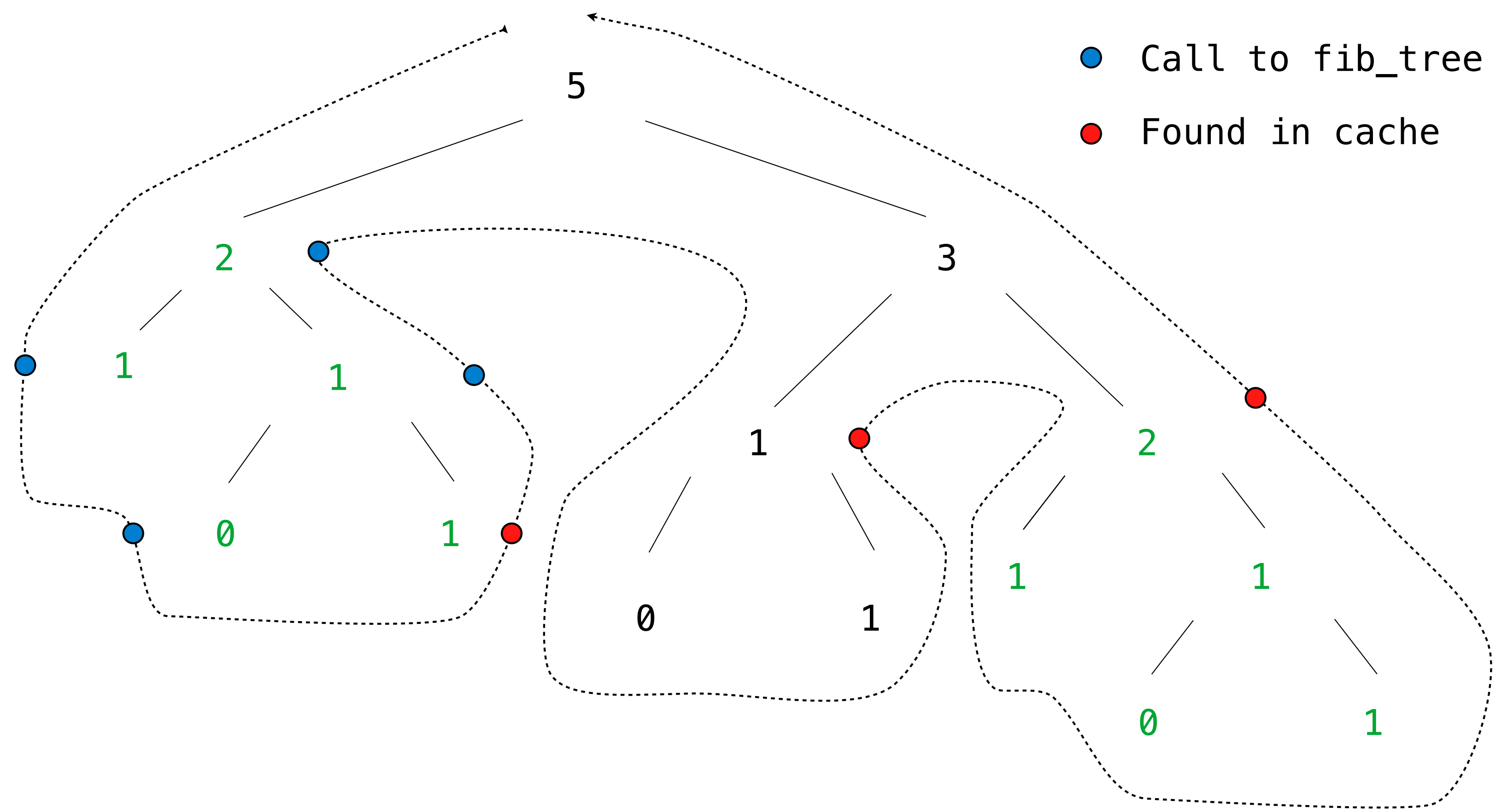
Memoized Tree Recursion



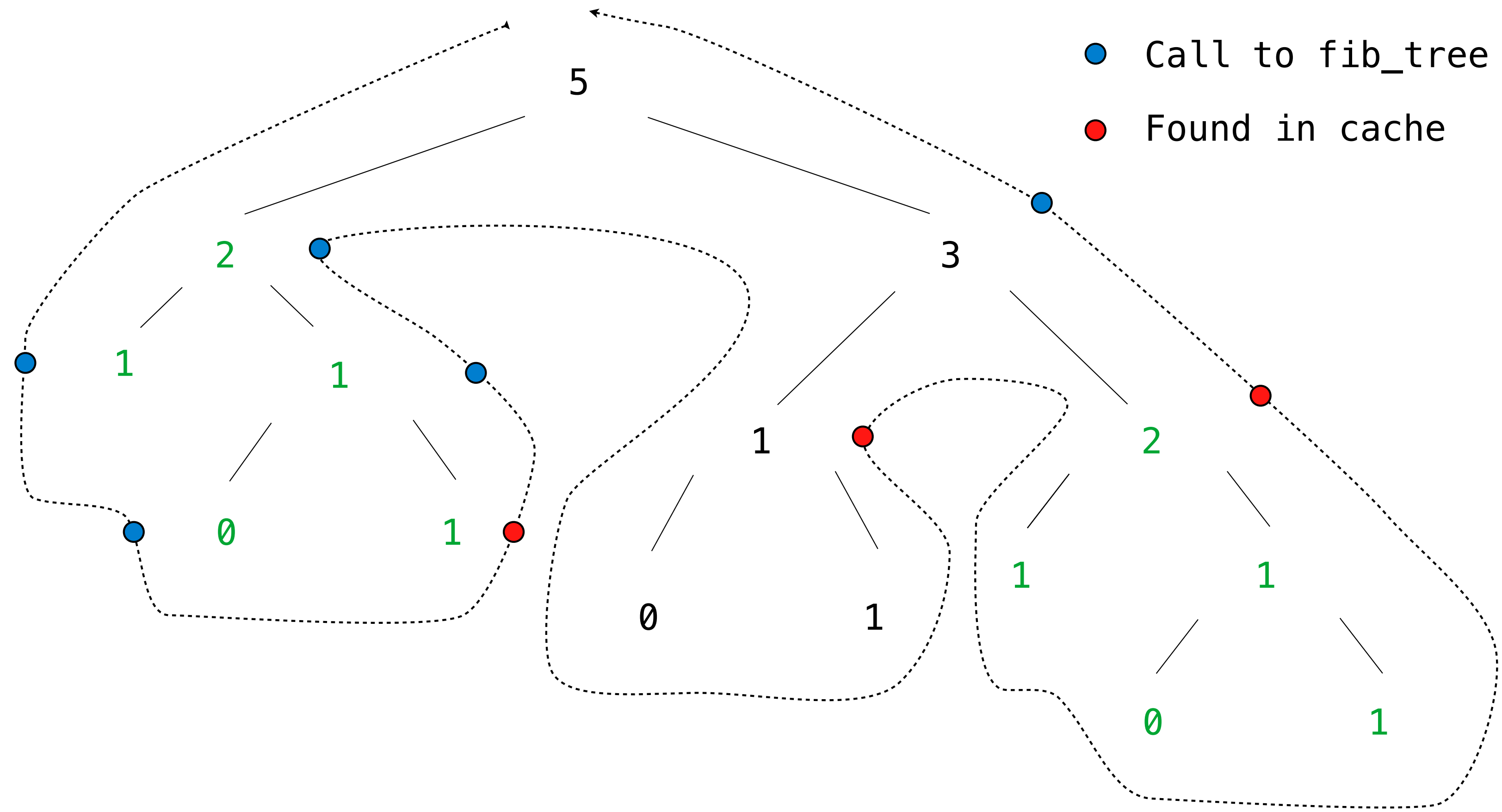
Memoized Tree Recursion



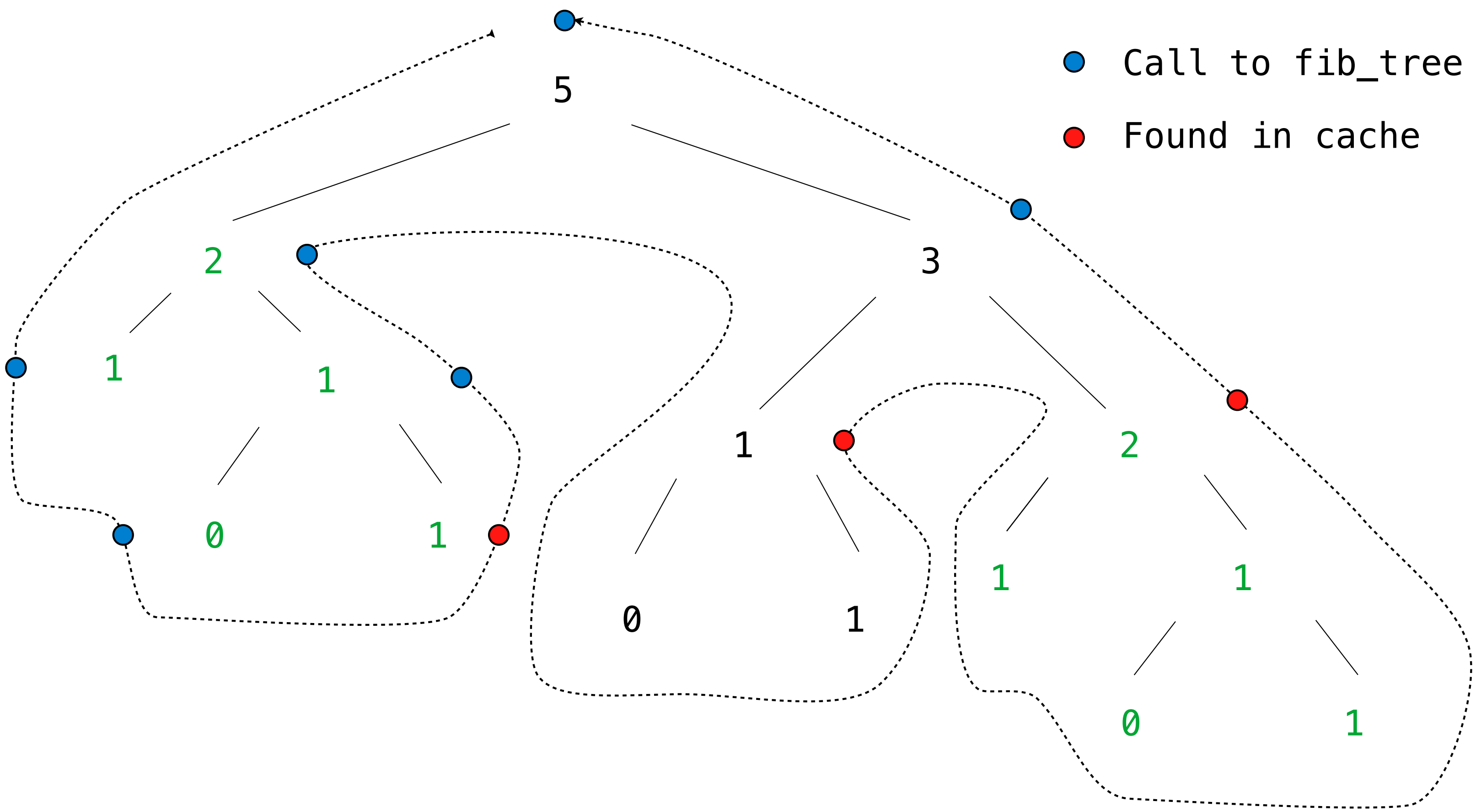
Memoized Tree Recursion



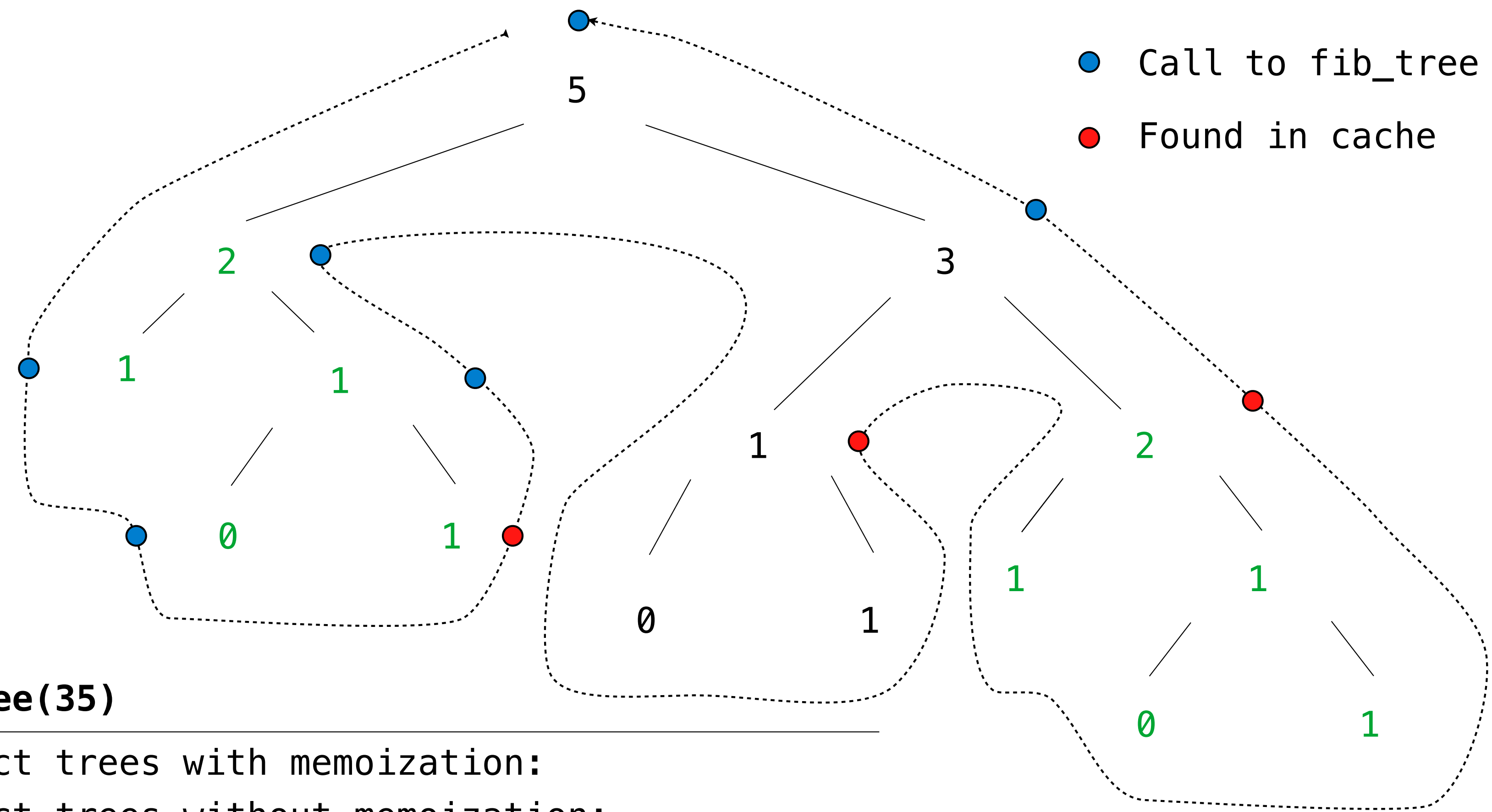
Memoized Tree Recursion



Memoized Tree Recursion



Memoized Tree Recursion

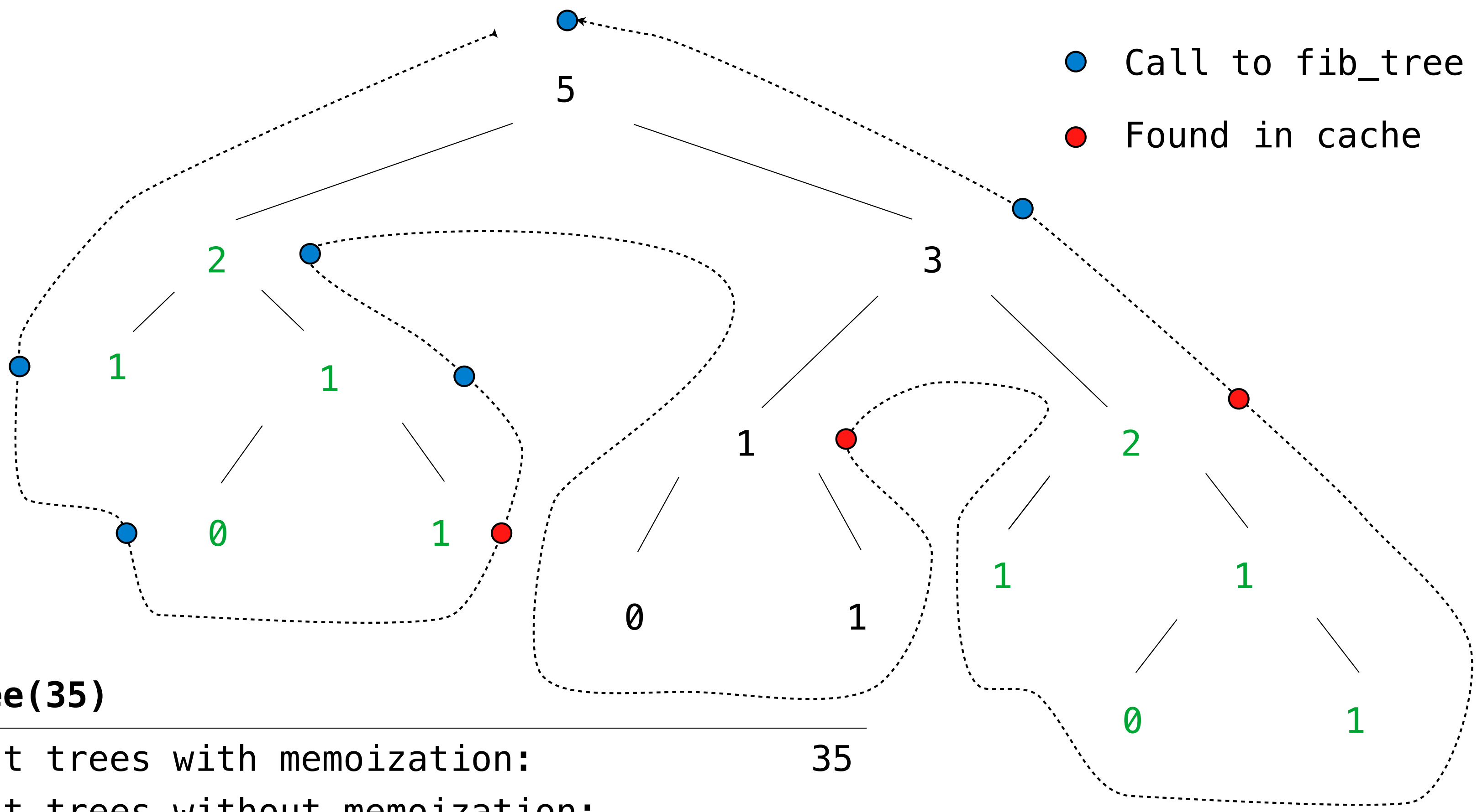


fib_tree(35)

Distinct trees with memoization:

Distinct trees without memoization:

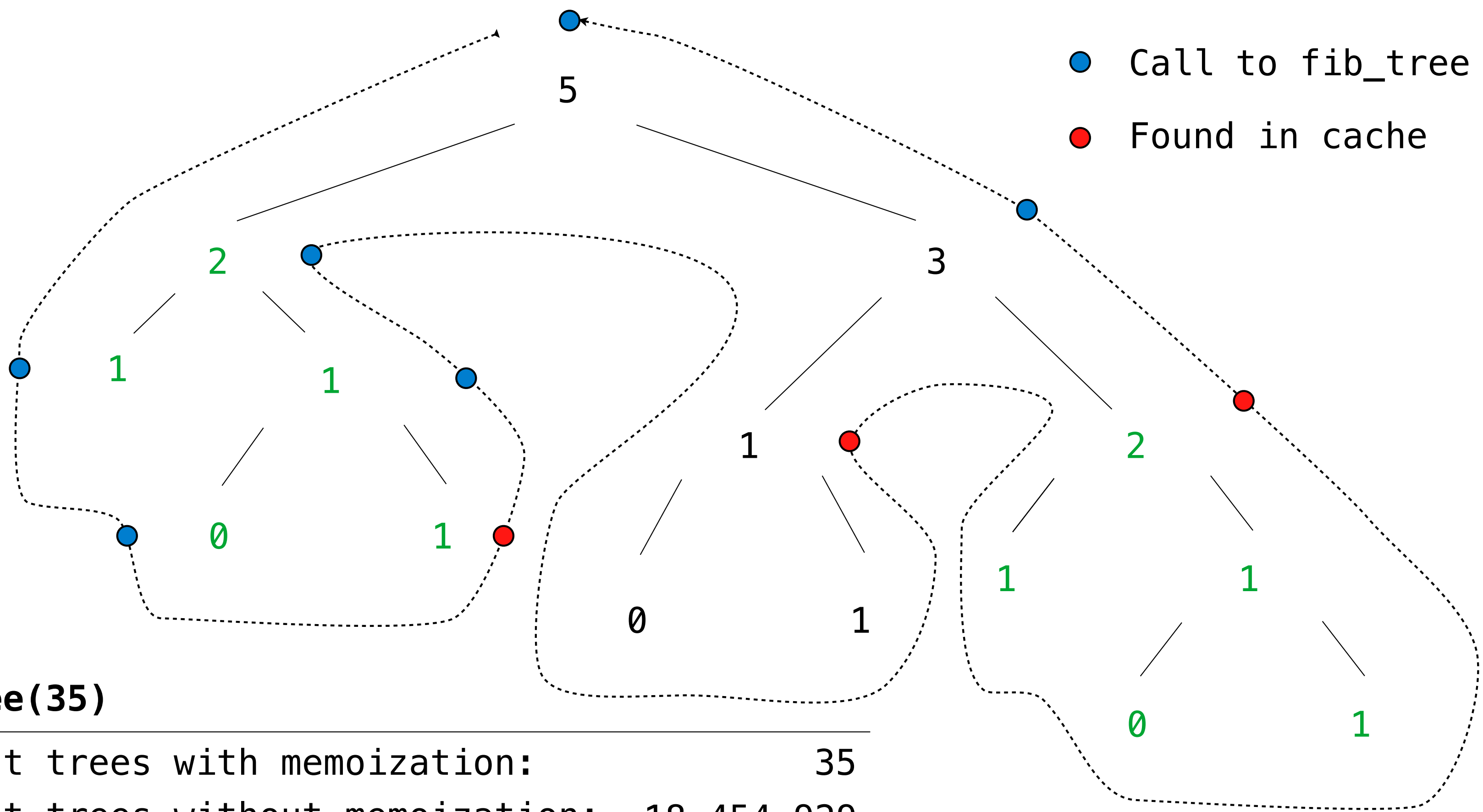
Memoized Tree Recursion



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

Time

The Consumption of Time

Implementations of the same functional abstraction can require different amounts of time to compute their result.

The Consumption of Time

Implementations of the same functional abstraction can require different amounts of time to compute their result.

Problem: How many factors does a positive integer n have?

A factor k of n is a positive integer such that n/k is also a positive integer.

The Consumption of Time

Implementations of the same functional abstraction can require different amounts of time to compute their result.

Problem: How many factors does a positive integer n have?

A factor k of n is a positive integer such that n/k is also a positive integer.

```
def count_factors(n):
```

The Consumption of Time

Implementations of the same functional abstraction can require different amounts of time to compute their result.

Problem: How many factors does a positive integer n have?

A factor k of n is a positive integer such that n/k is also a positive integer.

```
def count_factors(n):
```

Slow: Test each k from 1 through n.

The Consumption of Time

Implementations of the same functional abstraction can require different amounts of time to compute their result.

Problem: How many factors does a positive integer n have?

A factor k of n is a positive integer such that n/k is also a positive integer.

```
def count_factors(n):
```

Slow: Test each k from 1 through n.

Fast: Test each k from 1 to square root n.
For every k, n/k is also a factor!

The Consumption of Time

Implementations of the same functional abstraction can require different amounts of time to compute their result.

Problem: How many factors does a positive integer n have?

A factor k of n is a positive integer such that n/k is also a positive integer.

```
def count_factors(n):
```

Time (number of divisions)

Slow: Test each k from 1 through n.

Fast: Test each k from 1 to square root n.
For every k, n/k is also a factor!

The Consumption of Time

Implementations of the same functional abstraction can require different amounts of time to compute their result.

Problem: How many factors does a positive integer n have?

A factor k of n is a positive integer such that n/k is also a positive integer.

```
def count_factors(n):
```

	Time (number of divisions)
Slow: Test each k from 1 through n.	n
Fast: Test each k from 1 to square root n. For every k, n/k is also a factor!	

The Consumption of Time

Implementations of the same functional abstraction can require different amounts of time to compute their result.

Problem: How many factors does a positive integer n have?

A factor k of n is a positive integer such that n/k is also a positive integer.

```
def count_factors(n):
```

	Time (number of divisions)
Slow: Test each k from 1 through n.	n
Fast: Test each k from 1 to square root n. For every k, n/k is also a factor!	$\lfloor \sqrt{n} \rfloor$

The Consumption of Time

Implementations of the same functional abstraction can require different amounts of time to compute their result.

Problem: How many factors does a positive integer n have?

A factor k of n is a positive integer such that n/k is also a positive integer.

```
def count_factors(n):
```

	Time (number of divisions)
Slow: Test each k from 1 through n.	n
Fast: Test each k from 1 to square root n. For every k, n/k is also a factor!	$\lfloor \sqrt{n} \rfloor$

(Demo)

Space

The Consumption of Space

The Consumption of Space

Which environment frames do we need to keep during evaluation?

The Consumption of Space

Which environment frames do we need to keep during evaluation?

Each step of evaluation has a set of **active** environments.

The Consumption of Space

Which environment frames do we need to keep during evaluation?

Each step of evaluation has a set of **active** environments.

Values and frames in active environments consume memory.

The Consumption of Space

Which environment frames do we need to keep during evaluation?

Each step of evaluation has a set of **active** environments.

Values and frames in active environments consume memory.

Memory used for other values and frames can be recycled.

The Consumption of Space

Which environment frames do we need to keep during evaluation?

Each step of evaluation has a set of **active** environments.

Values and frames in active environments consume memory.

Memory used for other values and frames can be recycled.

Active environments:

The Consumption of Space

Which environment frames do we need to keep during evaluation?

Each step of evaluation has a set of **active** environments.

Values and frames in active environments consume memory.

Memory used for other values and frames can be recycled.

Active environments:

- Environments for any function calls currently being evaluated

The Consumption of Space

Which environment frames do we need to keep during evaluation?

Each step of evaluation has a set of **active** environments.

Values and frames in active environments consume memory.

Memory used for other values and frames can be recycled.

Active environments:

- Environments for any function calls currently being evaluated
- Parent environments of functions named in active environments

The Consumption of Space

Which environment frames do we need to keep during evaluation?

Each step of evaluation has a set of **active** environments.

Values and frames in active environments consume memory.

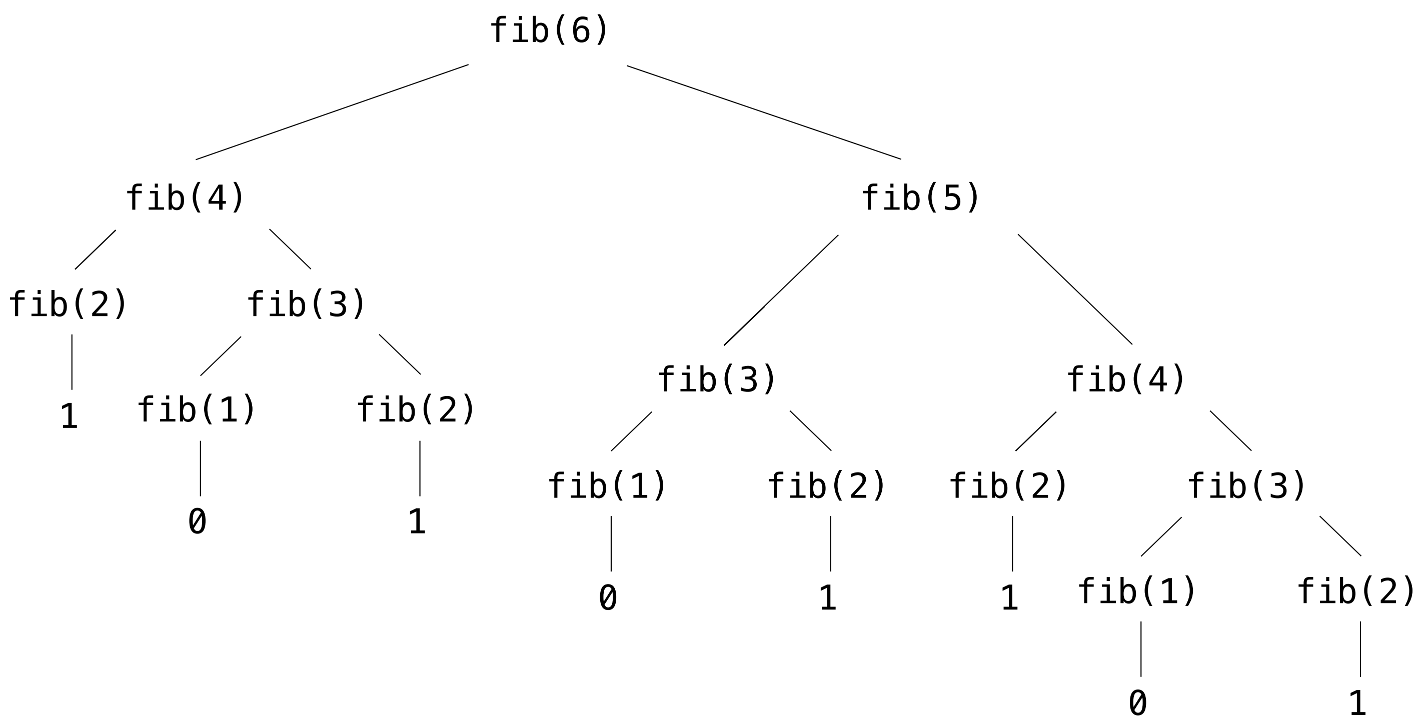
Memory used for other values and frames can be recycled.

Active environments:

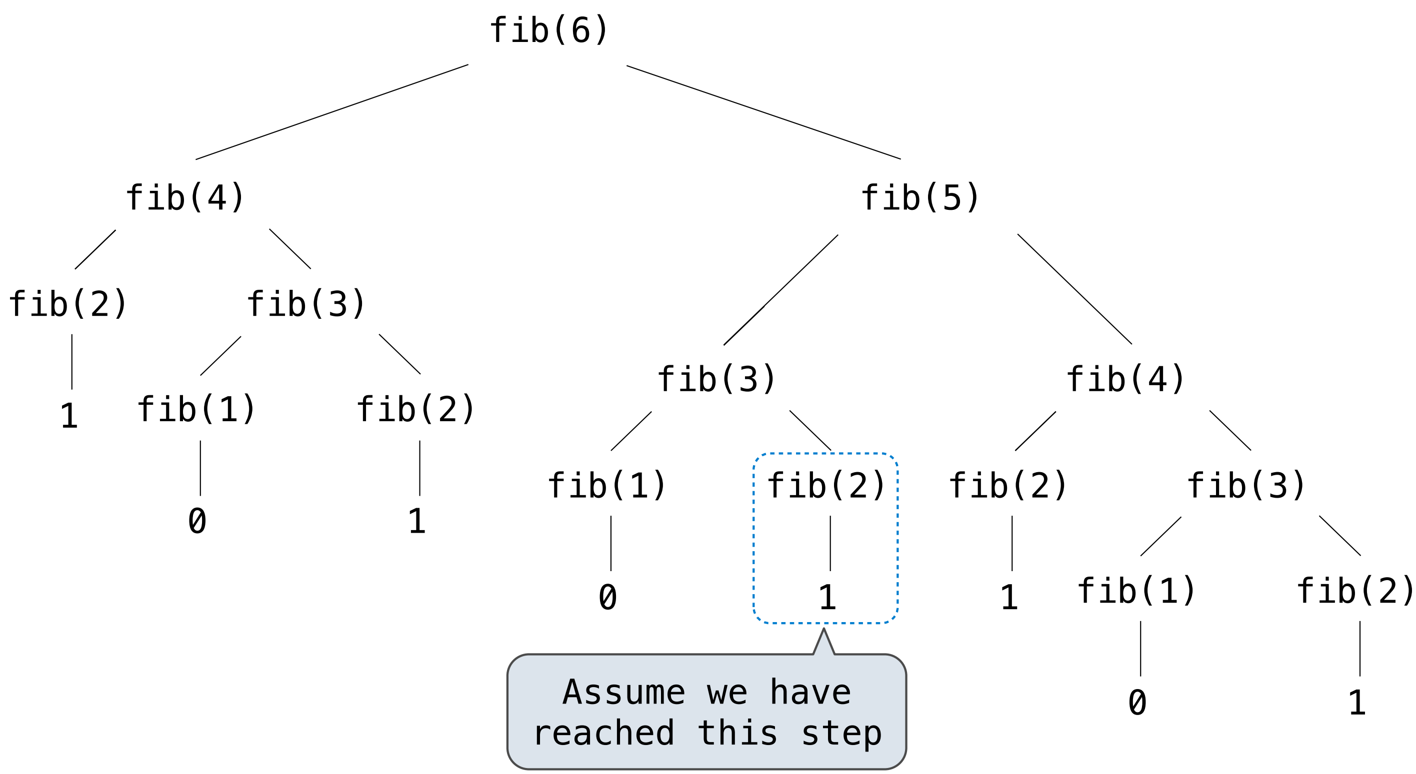
- Environments for any function calls currently being evaluated
- Parent environments of functions named in active environments

(Demo)

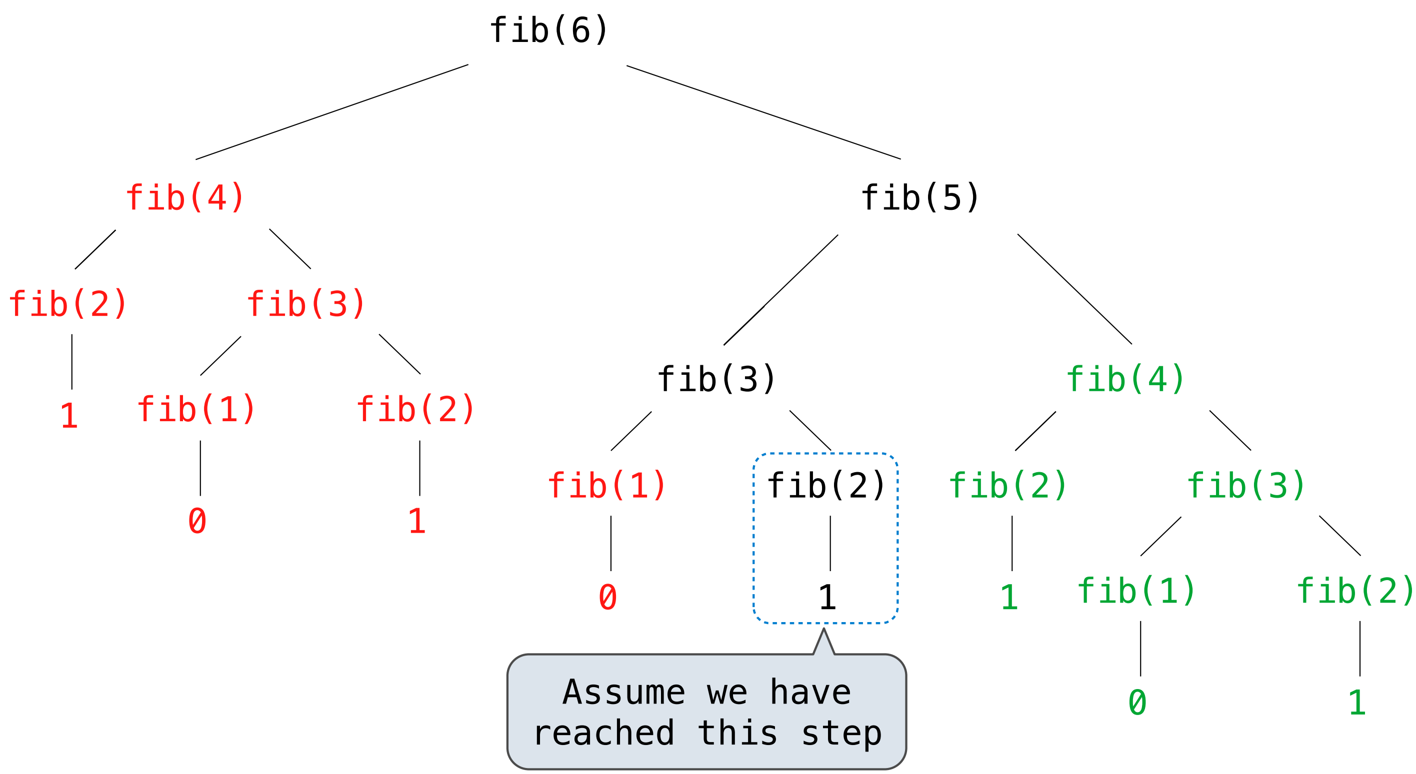
Fibonacci Memory Consumption



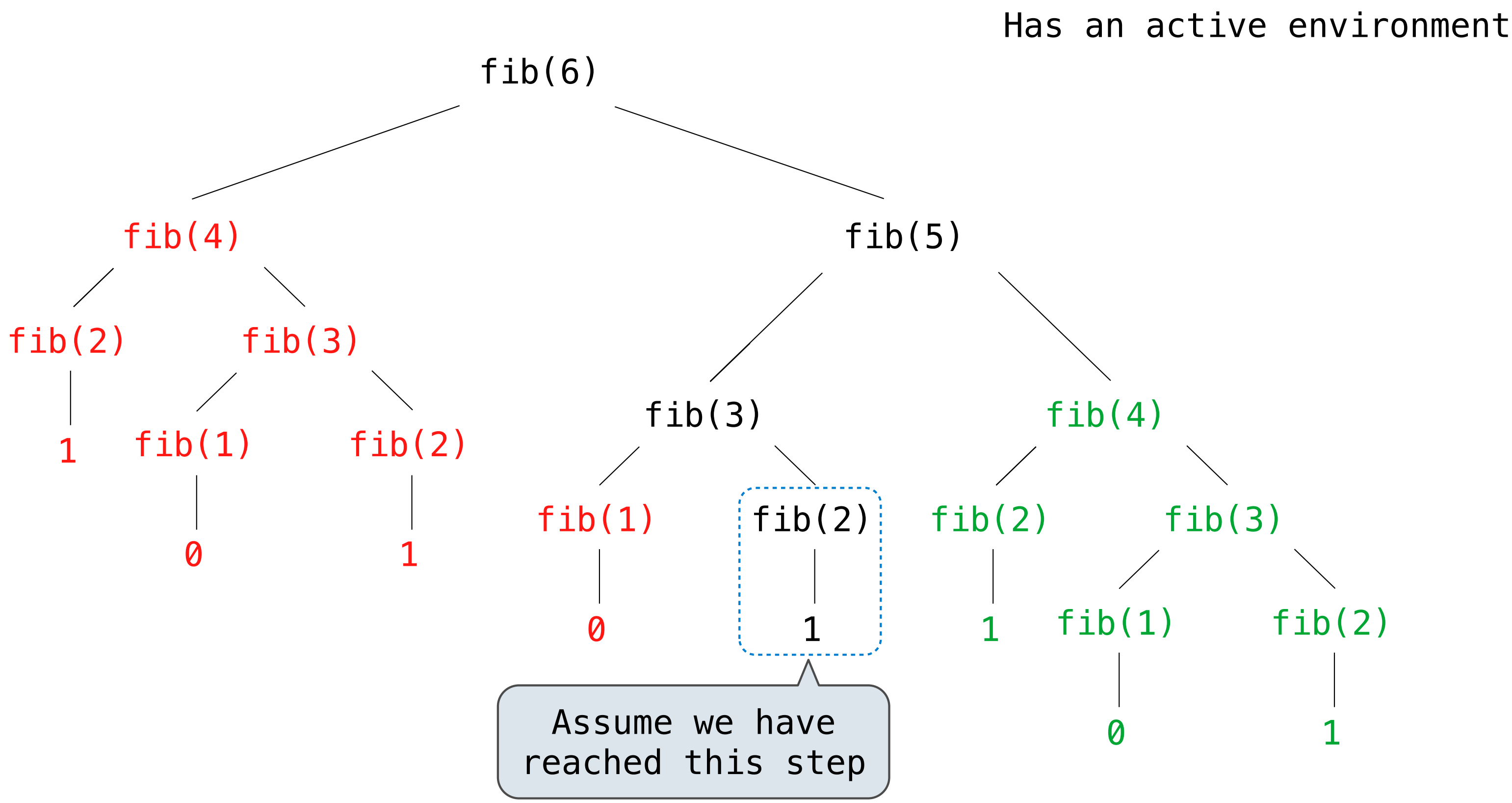
Fibonacci Memory Consumption



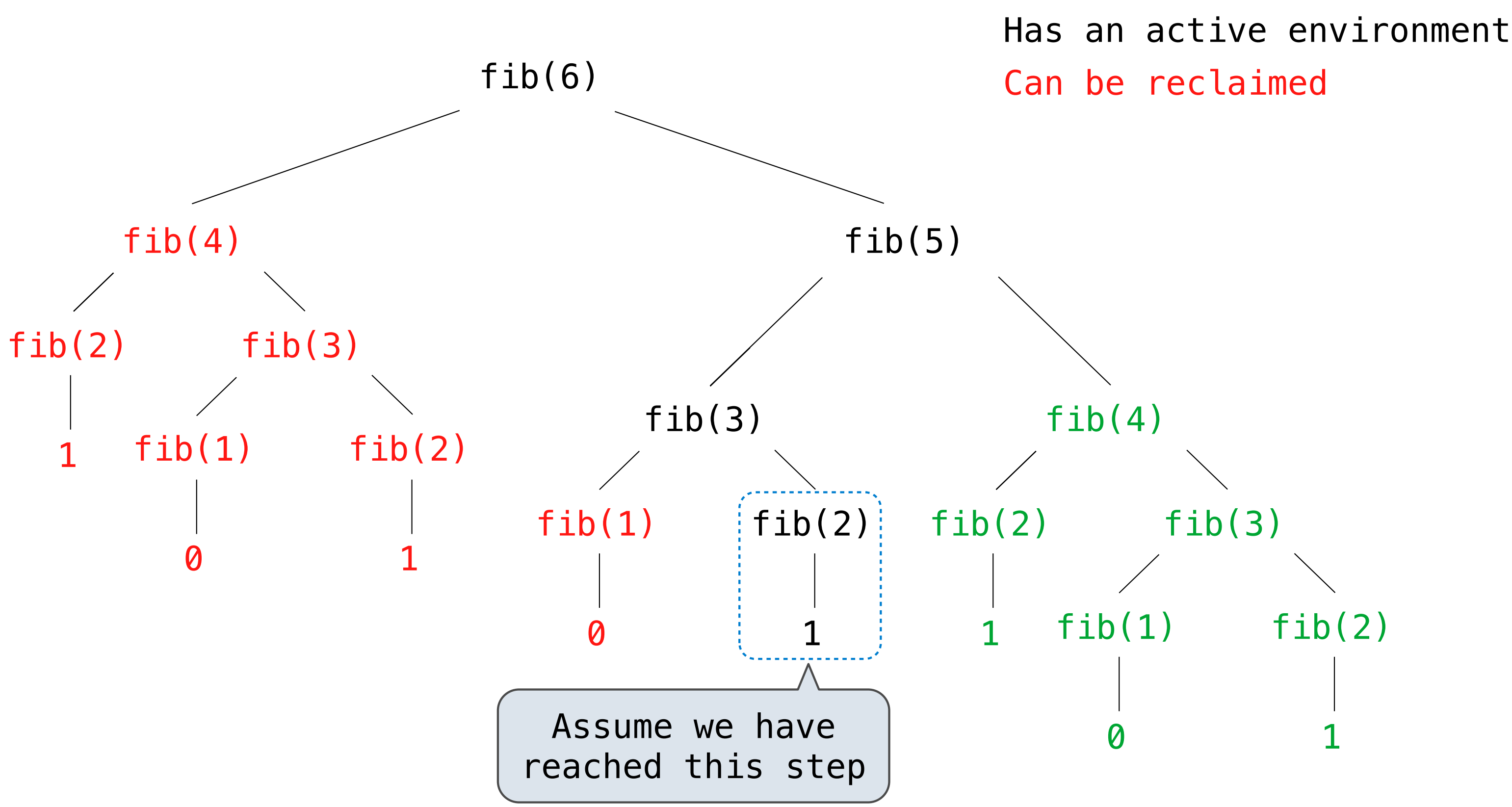
Fibonacci Memory Consumption



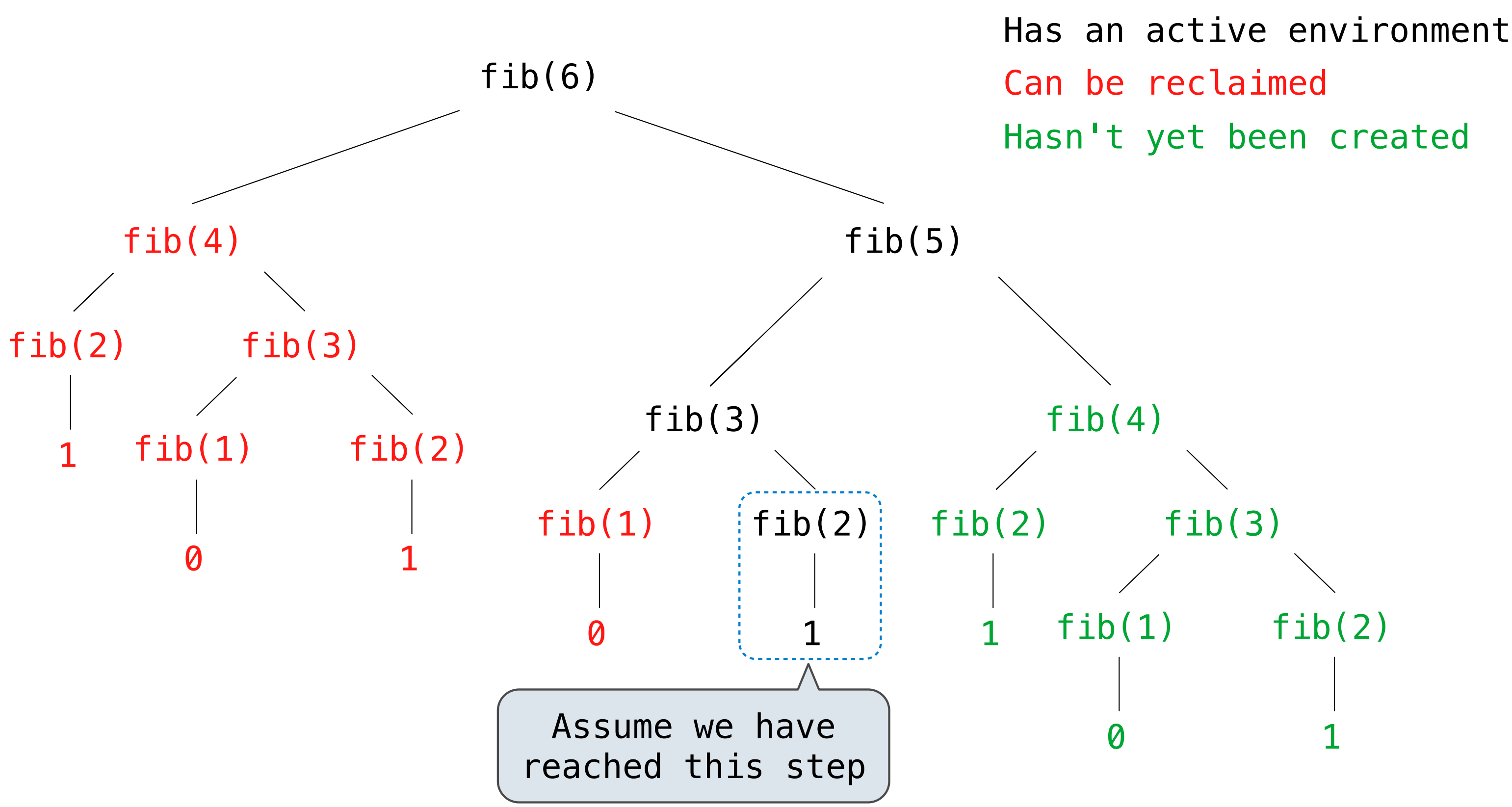
Fibonacci Memory Consumption



Fibonacci Memory Consumption



Fibonacci Memory Consumption



Order of Growth

Order of Growth

Order of Growth

`A method for bounding the resources used by a function by the "size" of a problem`

Order of Growth

A method for bounding the resources used by a function by the "size" of a problem

***n*:** size of the problem

Order of Growth

A method for bounding the resources used by a function by the "size" of a problem

***n*:** size of the problem

***R*(*n*):** Measurement of some resource used (time or space)

Order of Growth

A method for bounding the resources used by a function by the "size" of a problem

***n*:** size of the problem

***R*(*n*):** Measurement of some resource used (time or space)

$$R(n) = \Theta(f(n))$$

Order of Growth

A method for bounding the resources used by a function by the "size" of a problem

n**:** size of the problem

R**(*****n*****):** Measurement of some resource used (time or space)

$$R(n) = \Theta(f(n))$$

means that there are positive constants k_1 and k_2 such that

Order of Growth

A method for bounding the resources used by a function by the "size" of a problem

***n*:** size of the problem

***R*(*n*):** Measurement of some resource used (time or space)

$$R(n) = \Theta(f(n))$$

means that there are positive constants k_1 and k_2 such that

$$k_1 \cdot f(n) \leq R(n) \leq k_2 \cdot f(n)$$

Order of Growth

A method for bounding the resources used by a function by the "size" of a problem

n**:** size of the problem

R**(*****n*****):** Measurement of some resource used (time or space)

$$R(n) = \Theta(f(n))$$

means that there are positive constants k_1 and k_2 such that

$$k_1 \cdot f(n) \leq R(n) \leq k_2 \cdot f(n)$$

for sufficiently large values of ***n*****.**

Iteration vs Memoized Tree Recursion

Iterative and memoized implementations are not the same.

Time	Space

```
def fib_iter(n):
    prev, curr = 1, 0
    for _ in range(n-1):
         prev, curr = curr, prev + curr
    return curr
```

```
@memo
def fib(n):
    if n == 1:
        return 0
    if n == 2:
        return 1
    return fib(n-2) + fib(n-1)
```

Iteration vs Memoized Tree Recursion

Iterative and memoized implementations are not the same.

	Time	Space
`fib_iter`	$\Theta(n)$	
`fib` (memoized)		

```
def fib_iter(n):
    prev, curr = 1, 0
    for _ in range(n-1):
        prev, curr = curr, prev + curr
    return curr
```

```
@memo
def fib(n):
    if n == 1:
        return 0
    if n == 2:
        return 1
    return fib(n-2) + fib(n-1)
```

Iteration vs Memoized Tree Recursion

Iterative and memoized implementations are not the same.

	Time	Space
`fib_iter`	$\Theta(n)$	$\Theta(1)$

```
def fib_iter(n):
    prev, curr = 1, 0
    for _ in range(n-1):
         prev, curr = curr, prev + curr
    return curr
```

```
@memo
def fib(n):
    if n == 1:
        return 0
    if n == 2:
        return 1
    return fib(n-2) + fib(n-1)
```

Iteration vs Memoized Tree Recursion

Iterative and memoized implementations are not the same.

	Time	Space
`fib_iter`	$\Theta(n)$	$\Theta(1)$
`fib` (memoized)	$\Theta(n)$	

```python
def fib_iter(n):
    prev, curr = 1, 0
    for _ in range(n-1):
        prev, curr = curr, prev + curr
    return curr
```

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

Iteration vs Memoized Tree Recursion

Iterative and memoized implementations are not the same.

	Time	Space
`fib_iter`	$\Theta(n)$	$\Theta(1)$
memoized `fib`	$\Theta(n)$	$\Theta(n)$

```python
def fib_iter(n):
    prev, curr = 1, 0
    for _ in range(n-1):
        prev, curr = curr, prev + curr
    return curr
```

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

Counting Factors

Order of growth can still be used, even if we can quantify amounts exactly.

Problem: How many factors does a positive integer n have?

A factor k of n is a positive integer such that n/k is also a positive integer.

```
def count_factors(n)"
```

Time	Space

Slow: Test each k from 1 to n.

Fast: Test each k from 1 to square root n.
For every k, n/k is also a factor!

Counting Factors

Order of growth can still be used, even if we can quantify amounts exactly.

Problem: How many factors does a positive integer n have?

A factor k of n is a positive integer such that n/k is also a positive integer.

```
def count_factors(n)"
```

Slow: Test each k from 1 to n.

Fast: Test each k from 1 to square root n.
For every k, n/k is also a factor!

Time	Space
$\Theta(n)$	$\Theta(1)$

Counting Factors

Order of growth can still be used, even if we can quantify amounts exactly.

Problem: How many factors does a positive integer n have?

A factor k of n is a positive integer such that n/k is also a positive integer.

```
def count_factors(n)"
```

	Time	Space
Slow: Test each k from 1 to n.	$\Theta(n)$	$\Theta(1)$
Fast: Test each k from 1 to square root n. For every k, n/k is also a factor!	$\Theta(\sqrt{n})$	$\Theta(1)$

Exponentiation

Exponentiation

Exponentiation

Goal: one more multiplication lets us double the problem size.

Exponentiation

Goal: one more multiplication lets us double the problem size.

```
def exp(b, n):
    if n == 0:
        return 1
   else:
        return b * exp(b, n-1)
```

Exponentiation

Goal: one more multiplication lets us double the problem size.

```
def exp(b, n):
    if n == 0:
        return 1
   else:
        return b * exp(b, n-1)
```

$$b^n = \begin{cases} 1 & \text{if } n = 0 \\ b \cdot b^{n-1} & \text{otherwise} \end{cases}$$

Exponentiation

Goal: one more multiplication lets us double the problem size.

```
def exp(b, n):
    if n == 0:
        return 1
   else:
        return b * exp(b, n-1)
```

$$b^n = \begin{cases} 1 & \text{if } n = 0 \\ b \cdot b^{n-1} & \text{otherwise} \end{cases}$$

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

Exponentiation

Goal: one more multiplication lets us double the problem size.

```
def exp(b, n):
    if n == 0:
        return 1
   else:
        return b * exp(b, n-1)

def square(x):
    return x*x

def fast_exp(b, n):
    if n == 0:
        return 1
    elif n % 2 == 0:
        return square(fast_exp(b, n//2))
    else:
        return b * fast_exp(b, n-1)
```

$$b^n = \begin{cases} 1 & \text{if } n = 0 \\ b \cdot b^{n-1} & \text{otherwise} \end{cases}$$

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

Exponentiation

Goal: one more multiplication lets us double the problem size.

```
def exp(b, n):
    if n == 0:
        return 1
   else:
        return b * exp(b, n-1)
```

$$b^n = \begin{cases} 1 & \text{if } n = 0 \\ b \cdot b^{n-1} & \text{otherwise} \end{cases}$$

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

def fast_exp(b, n):
    if n == 0:
        return 1
    elif n % 2 == 0:
        return square(fast_exp(b, n//2))
    else:
        return b * fast_exp(b, n-1)
```

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

(Demo)

Exponentiation

Goal: one more multiplication lets us double the problem size.

Time	Space

```
def exp(b, n):
    if n == 0:
        return 1
    else:
        return b * exp(b, n-1)

def square(x):
    return x*x

def fast_exp(b, n):
    if n == 0:
        return 1
    elif n % 2 == 0:
        return square(fast_exp(b, n//2))
    else:
        return b * fast_exp(b, n-1)
```

Exponentiation

Goal: one more multiplication lets us double the problem size.

```
def exp(b, n):
    if n == 0:
        return 1
    else:
        return b * exp(b, n-1)

def square(x):
    return x*x

def fast_exp(b, n):
    if n == 0:
        return 1
    elif n % 2 == 0:
        return square(fast_exp(b, n//2))
    else:
        return b * fast_exp(b, n-1)
```

Time	Space
$\Theta(n)$	$\Theta(n)$

Exponentiation

Goal: one more multiplication lets us double the problem size.

```
def exp(b, n):
    if n == 0:
        return 1
    else:
        return b * exp(b, n-1)

def square(x):
    return x*x

def fast_exp(b, n):
    if n == 0:
        return 1
    elif n % 2 == 0:
        return square(fast_exp(b, n//2))
    else:
        return b * fast_exp(b, n-1)
```

	Time	Space
`exp`	$\Theta(n)$	$\Theta(n)$
`fast_exp`	$\Theta(\log n)$	$\Theta(\log n)$

Comparing Orders of Growth

Comparing orders of growth (n is the problem size)

Comparing orders of growth (n is the problem size)

$$\Theta(b^n)$$

Comparing orders of growth (n is the problem size)

$\Theta(b^n)$ Exponential growth! Recursive fib takes

$\Theta(\phi^n)$ steps, where $\phi = \dfrac{1+\sqrt{5}}{2} \approx 1.61828$

Comparing orders of growth (n is the problem size)

$\Theta(b^n)$ Exponential growth! Recursive fib takes

$\Theta(\phi^n)$ steps, where $\phi = \frac{1 + \sqrt{5}}{2} \approx 1.61828$

Incrementing the problem scales R(n) by a factor.

Comparing orders of growth (n is the problem size)

$\Theta(b^n)$ Exponential growth! Recursive fib takes

$\Theta(\phi^n)$ steps, where $\phi = \frac{1+\sqrt{5}}{2} \approx 1.61828$

Incrementing the problem scales R(n) by a factor.

$\Theta(n^2)$

Comparing orders of growth (n is the problem size)

$\Theta(b^n)$ Exponential growth! Recursive fib takes

$\Theta(\phi^n)$ steps, where $\phi = \frac{1+\sqrt{5}}{2} \approx 1.61828$

Incrementing the problem scales R(n) by a factor.

$\Theta(n^2)$ Quadratic growth. E.g., operations on all pairs.

Comparing orders of growth (n is the problem size)

$\Theta(b^n)$ Exponential growth! Recursive fib takes

$\Theta(\phi^n)$ steps, where $\phi = \frac{1+\sqrt{5}}{2} \approx 1.61828$

Incrementing the problem scales R(n) by a factor.

$\Theta(n^2)$ Quadratic growth. E.g., operations on all pairs.

Incrementing n increases R(n) by the problem size n.

Comparing orders of growth (n is the problem size)

$\Theta(b^n)$ Exponential growth! Recursive fib takes

$\Theta(\phi^n)$ steps, where $\phi = \dfrac{1+\sqrt{5}}{2} \approx 1.61828$

Incrementing the problem scales R(n) by a factor.

$\Theta(n^2)$ Quadratic growth. E.g., operations on all pairs.

Incrementing n increases R(n) by the problem size n.

$\Theta(n)$

Comparing orders of growth (n is the problem size)

$\Theta(b^n)$ Exponential growth! Recursive fib takes

$\Theta(\phi^n)$ steps, where $\phi = \dfrac{1+\sqrt{5}}{2} \approx 1.61828$

Incrementing the problem scales R(n) by a factor.

$\Theta(n^2)$ Quadratic growth. E.g., operations on all pairs.

Incrementing n increases R(n) by the problem size n.

$\Theta(n)$ Linear growth. Resources scale with the problem.

Comparing orders of growth (n is the problem size)

$\Theta(b^n)$ Exponential growth! Recursive fib takes

$\Theta(\phi^n)$ steps, where $\phi = \frac{1+\sqrt{5}}{2} \approx 1.61828$

Incrementing the problem scales R(n) by a factor.

$\Theta(n^2)$ Quadratic growth. E.g., operations on all pairs.

Incrementing n increases R(n) by the problem size n.

$\Theta(n)$ Linear growth. Resources scale with the problem.

$\Theta(\log n)$

Comparing orders of growth (n is the problem size)

$\Theta(b^n)$ Exponential growth! Recursive fib takes

$\Theta(\phi^n)$ steps, where $\phi = \frac{1+\sqrt{5}}{2} \approx 1.61828$

Incrementing the problem scales R(n) by a factor.

$\Theta(n^2)$ Quadratic growth. E.g., operations on all pairs.

Incrementing n increases R(n) by the problem size n.

$\Theta(n)$ Linear growth. Resources scale with the problem.

$\Theta(\log n)$ Logarithmic growth. These processes scale well.

Comparing orders of growth (n is the problem size)

$\Theta(b^n)$ Exponential growth! Recursive fib takes

$\Theta(\phi^n)$ steps, where $\phi = \frac{1+\sqrt{5}}{2} \approx 1.61828$

Incrementing the problem scales R(n) by a factor.

$\Theta(n^2)$ Quadratic growth. E.g., operations on all pairs.

Incrementing n increases R(n) by the problem size n.

$\Theta(n)$ Linear growth. Resources scale with the problem.

$\Theta(\log n)$ Logarithmic growth. These processes scale well.

Doubling the problem only increments R(n).

Comparing orders of growth (n is the problem size)

$\Theta(b^n)$ Exponential growth! Recursive fib takes

$\Theta(\phi^n)$ steps, where $\phi = \frac{1+\sqrt{5}}{2} \approx 1.61828$

Incrementing the problem scales R(n) by a factor.

$\Theta(n^2)$ Quadratic growth. E.g., operations on all pairs.

Incrementing n increases R(n) by the problem size n.

$\Theta(n)$ Linear growth. Resources scale with the problem.

$\Theta(\log n)$ Logarithmic growth. These processes scale well.

Doubling the problem only increments R(n).

$\Theta(1)$

Comparing orders of growth (n is the problem size)

$\Theta(b^n)$ Exponential growth! Recursive fib takes

$\Theta(\phi^n)$ steps, where $\phi = \frac{1+\sqrt{5}}{2} \approx 1.61828$

Incrementing the problem scales R(n) by a factor.

$\Theta(n^2)$ Quadratic growth. E.g., operations on all pairs.

Incrementing n increases R(n) by the problem size n.

$\Theta(n)$ Linear growth. Resources scale with the problem.

$\Theta(\log n)$ Logarithmic growth. These processes scale well.

Doubling the problem only increments R(n).

$\Theta(1)$ Constant. The problem size doesn't matter.

Comparing orders of growth (n is the problem size)

$\Theta(b^n)$ Exponential growth! Recursive fib takes

$\Theta(\phi^n)$ steps, where $\phi = \dfrac{1+\sqrt{5}}{2} \approx 1.61828$

Incrementing the problem scales R(n) by a factor.

$\Theta(n^2)$ Quadratic growth. E.g., operations on all pairs.

Incrementing n increases R(n) by the problem size n.

$\Theta(n)$ Linear growth. Resources scale with the problem.

$\Theta(\log n)$ Logarithmic growth. These processes scale well.

Doubling the problem only increments R(n).

$\Theta(1)$ Constant. The problem size doesn't matter.

Comparing orders of growth (n is the problem size)

$\Theta(b^n)$	Exponential growth! Recursive fib takes $\Theta(\phi^n)$ steps, where $\phi = \frac{1+\sqrt{5}}{2} \approx 1.61828$
$\Theta(n^6)$	Incrementing the problem scales R(n) by a factor.
$\Theta(n^2)$	Quadratic growth. E.g., operations on all pairs. Incrementing n increases R(n) by the problem size n.
$\Theta(n)$	Linear growth. Resources scale with the problem.
$\Theta(\log n)$	Logarithmic growth. These processes scale well. Doubling the problem only increments R(n).
$\Theta(1)$	Constant. The problem size doesn't matter.

Comparing orders of growth (n is the problem size)

$\Theta(b^n)$	`Exponential growth!  Recursive fib takes`
	$\Theta(\phi^n)$ `steps, where` $\phi = \frac{1+\sqrt{5}}{2} \approx 1.61828$
$\Theta(n^6)$	`Incrementing the problem scales R(n) by a factor.`
$\Theta(n^2)$	`Quadratic growth.  E.g., operations on all pairs.`
	`Incrementing n increases R(n) by the problem size n.`
$\Theta(n)$	`Linear growth.  Resources scale with the problem.`
$\Theta(\sqrt{n})$	
$\Theta(\log n)$	`Logarithmic growth. These processes scale well.`
	`Doubling the problem only increments R(n).`
$\Theta(1)$	`Constant. The problem size doesn't matter.`