Efficiency

July 12, 2022

Laryn Qi

Announcements

Efficiency

Efficiency

A measure of how much resource consumption a computational task takes.

An analysis of computer programs rather than a technique for writing them.

In computer science, we are concerned with time and space efficiency.

The time efficiency of could determine how long a user has to wait for a webpage to load.

The space efficiency of your algorithm could determine how much memory running your application takes.

We are going down a layer of abstraction – opening up the black box.

Exponentiation

Exponentiation

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

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

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

How many calls to `exp` are required to calculate `exp(2, 16)`?

Orders of Growth

Common Orders of Growth

One way to describe the efficiency of an algorithm is according to its **order of growth**, a description of how the number of steps needed grows with respect to a growing input size.

Order of growth	Description
Constant growth	Always the same # of steps, regardless of input size.
Logarithmic growth	# of steps increases proportionally to the logarithm of the input size.
Linear growth	# of steps increases in direct proportion to the input size.
Quadratic growth	# of steps increases in proportion to the square of the input size.
Exponential growth	# of steps increases faster than a polynomial function of the input size.

Why consider number of steps/operations instead of number of seconds/milliseconds?

Prepend

```
def prepend(lst, val):
    """Add VAL to the front of LST."""
    lst.insert(0, val)
```

How many operations will this require for lists of increasing size?

List size	Operations
1	1
10	1
100	1
1000	1

Constant Growth

An algorithm that takes **constant time** always executes a fixed number of operations regardless of the input size.

(Demo)

List size	Operations
1	1
10	1
100	1
1000	1

Fast Exponentiation

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

How many operations will this require for lists of increasing values of n?

n	Operations
0	1
8	5
16	6
1024	12

Logarithmic Growth

An algorithm that takes **logarithmic time**, always executes a fixed number of operations regardless of the input size.

(Demo)

n	Operations
0	1
8	5
16	6
1024	12

Contains

```
def contains(lst, val):
    """Return True if LST contains VAL. Else, False."""
    if not lst:
      return False
    return lst[0] == val or contains(lst[1:], val)
```

How many operations will this require for lists of increasing size?

List size	Best Case: Operations	Worst Case: Operations
1	1	1
10	1	10
100	1	100
1000	1	1000

Slow Exponentiation

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

How many operations will this require for increasing values of n?

n	Operations
1	1
10	10
100	100
1000	1000

Exponentiation Comparison

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

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

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

Linear time:

- Doubling the input **doubles** the time
- 1024x the input takes 1024x as much time

Logarithmic time:

- Doubling the input **increases** the time by a constant C
- 1024x the input increases the time by only 10 times C

Linear Growth

When an algorithm takes **linear time**, its number of operations increases in direct proportion to the input size.

(Demo)

List size/n	(Worst Case:) Operations
1	1
10	10
100	100
1000	1000

Overlap

```python
def overlap(a, b):
    """Return the number of overlapping values in A and B.
    >>> overlap([3, 5, 7, 6], [4, 5, 6, 5])
    3
    """
    count = 0
    for item in a:
      for other in b:
        if item == other:
          count += 1
    return count
```

	3	5	7	6
4	0	0	0	0
5	0	1	0	0
6	0	0	0	1
5	0	1	0	0

How many operations will this require for lists of increasing size?

List Size	Operations
1	1
10	100
100	10000
1000	1000000

Quadratic Growth

When an algorithm grows in **quadratic time**, its number of operations increases in proportion to the square of the input size.

(Demo)

List size	Operations
1	1
10	100
100	10000
1000	1000000

Virahanka-Fibonacci Numbers

```
def vir_fib(n):
    if n <= 1:
      return n
    return vir_fib(n - 2) + vir_fib(n - 1)
```

How many operations will this require for increasing values of n?

n	Operations
1	1
2	3
3	5
4	9
7	41
8	67
20	21891

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

Exponential Growth

When an algorithm grows in **exponential time**, its number of operations increases faster than a polynomial function of the input size.

(Demo)

n	Operations
1	1
2	3
3	5
4	9
7	41
8	67
20	21891

Mathematical View of Growth

Time for n+n | Time for input n+1 | Time for input n

Exponential growth. E.g., recursive `fib`

Incrementing *n* multiplies *time* by a constant

$$\boxed{a \cdot b^{n+1}} = \boxed{(a \cdot b^n)} \cdot b$$

Quadratic growth. E.g., `overlap`

Incrementing *n* increases *time* by *n* times a constant

$$\boxed{a \cdot (n+1)^2} = \boxed{(a \cdot n^2)} + a \cdot (2n+1)$$

Linear growth. E.g., slow `exp`

Incrementing *n* increases *time* by a constant

$$\boxed{a \cdot (n+1)} = \boxed{(a \cdot n)} + a$$

Logarithmic growth. E.g., `exp_fast`

Doubling *n* only increments *time* by a constant

$$\boxed{a \cdot \ln(2 \cdot n)} = \boxed{(a \cdot \ln n)} + a \cdot \ln 2$$

Constant growth. Increasing *n* doesn't affect time

Break

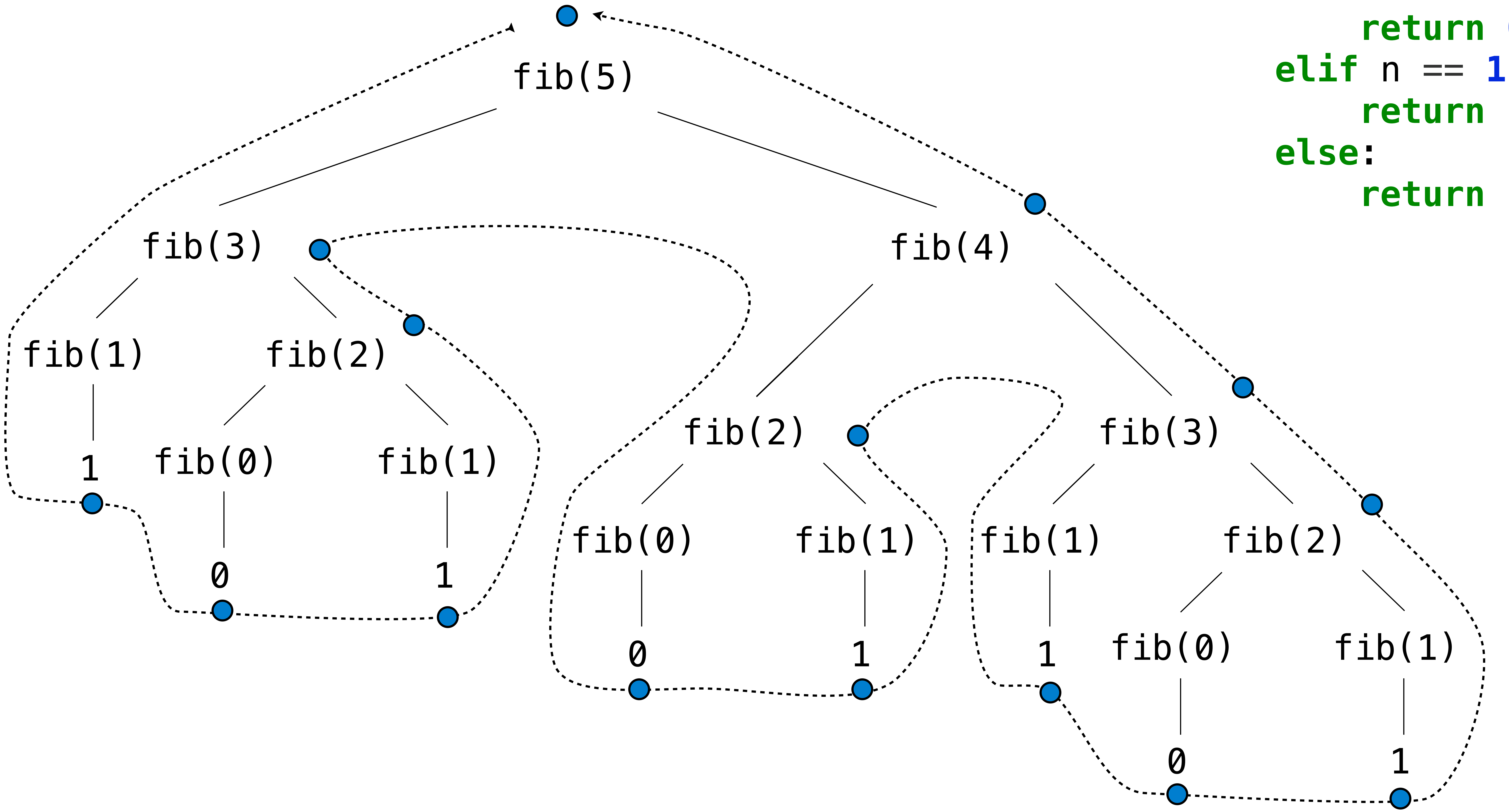

lambda x: x // 2
lambda y: y // 2
lambda z: 999999999
Wait, it's all
Constant Growth?
Always has been

Memoization Revisited

Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:

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



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

Memoization

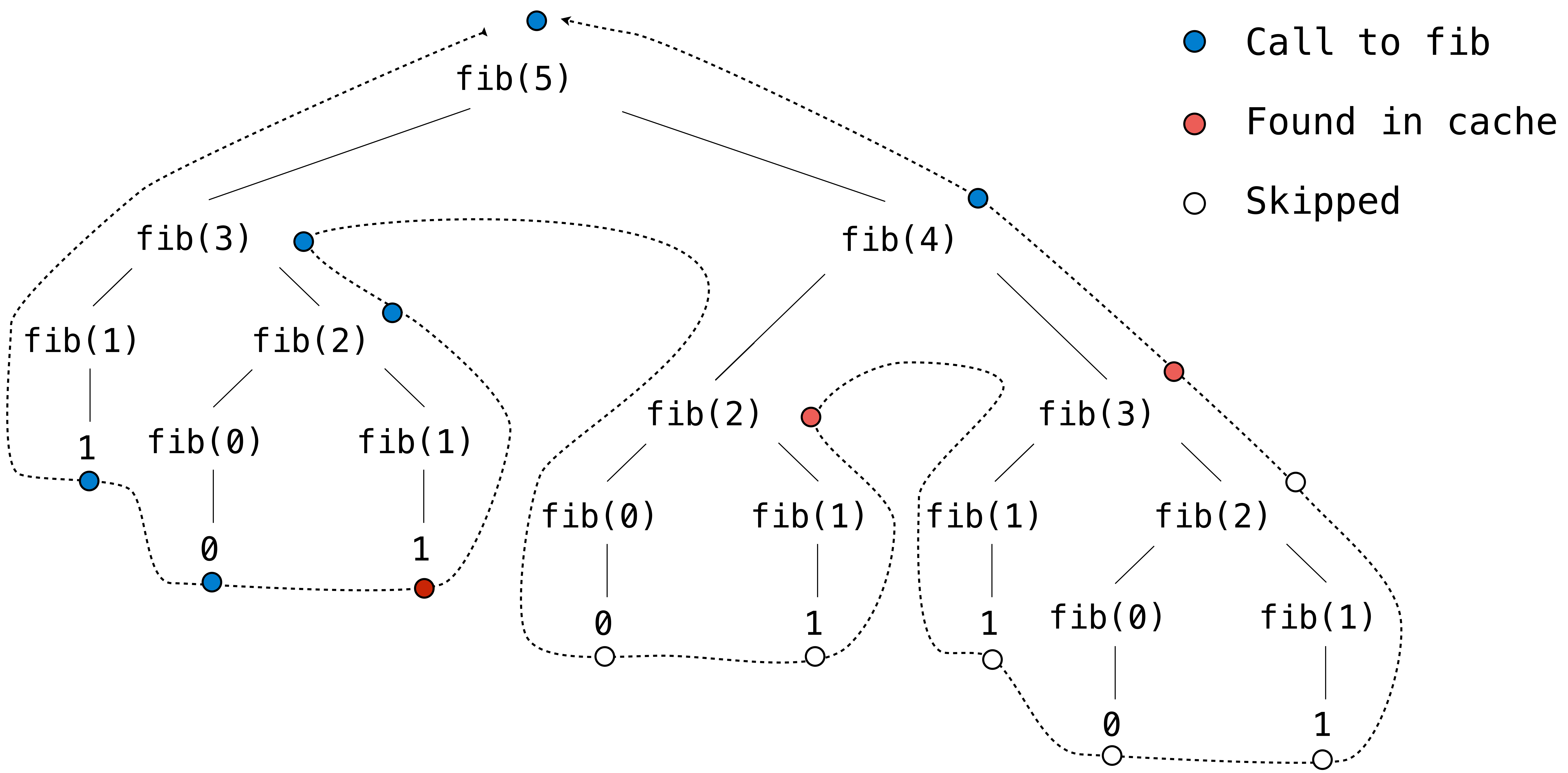
Idea: Remember the results that have been computed before

```
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

Memoized Tree Recursion



(Demo)

Revisiting Functions

(Demo)

Sum Digits, Count Partitions, Palindrome

Efficiency Practice

Space

Space and Environments

Which environment frames do we need to keep during evaluation?

At any moment there is a set of active environments

Values and frames in active environments consume memory

Memory that is used for other values and frames can be recycled

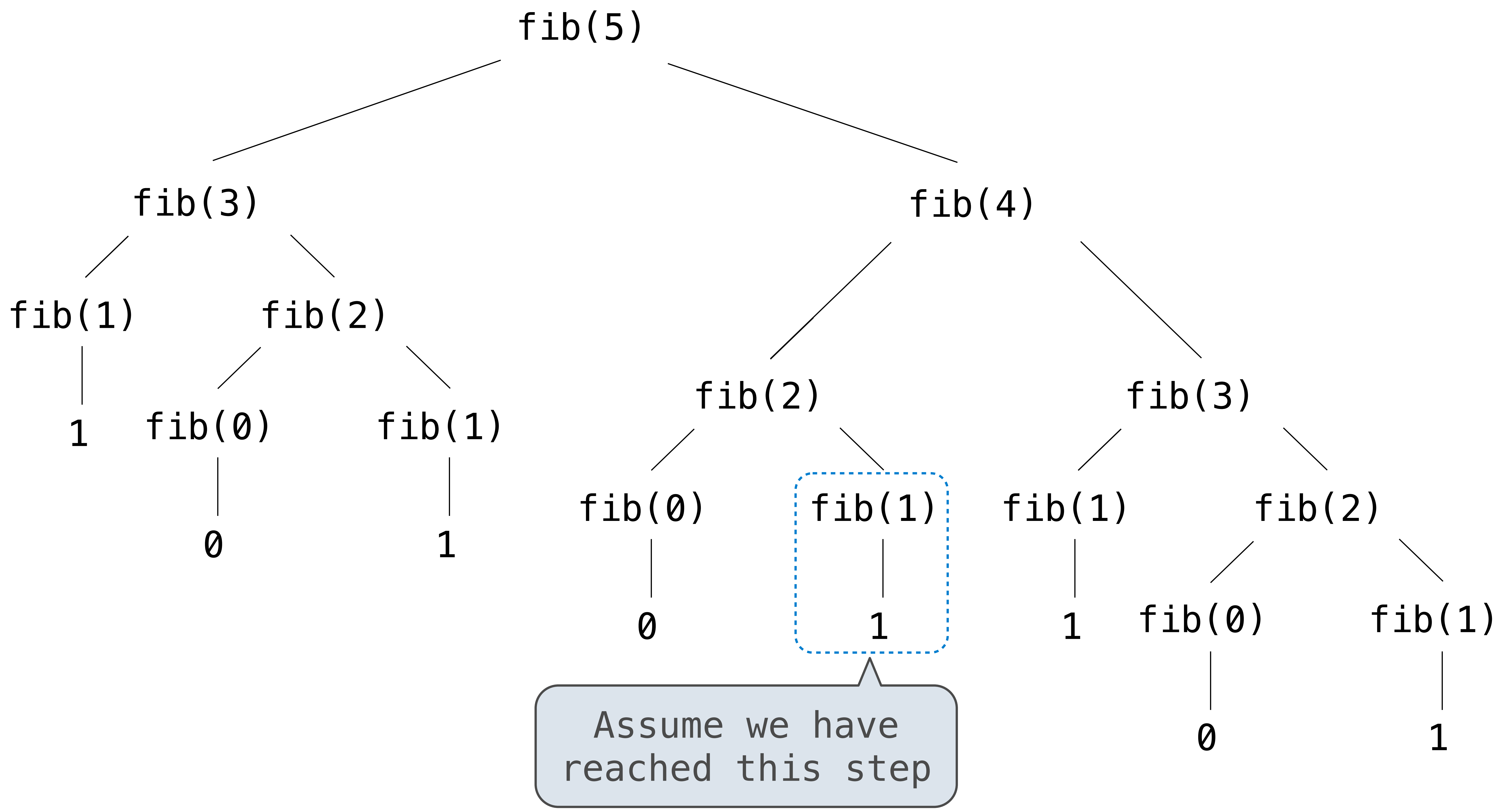
Active environments:

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

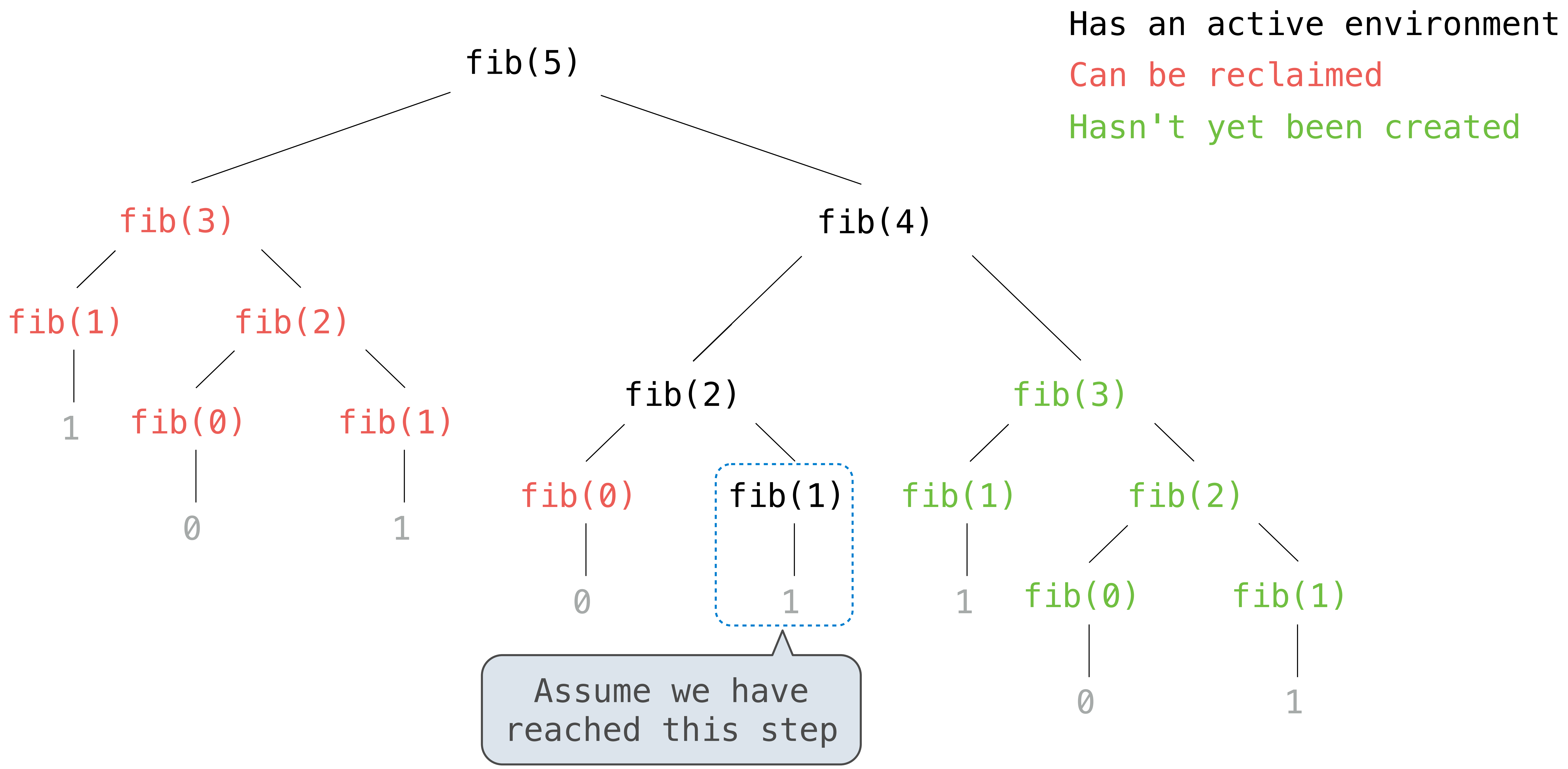
(Demo)

pythontutor.com/composingprograms.html#code=def%20fib%28n%29%3A%0A%20%20%20%20if%20n%20%3D%3D%200%20or%20n%20%3D%3D%201%3A%0A%20%20%20%20%20%20%20%20return%20n%0A%20%20%20%20else%3A%0A%20%20%20%20%20%20%20%20return%20fib%28n-2%29%20%2B%20fib%28n-1%29%0A%20%20%20%20%20%20%20%20%0Afib%286%29&mode=display&origin=composingprograms.js&cumulative=false&py=3&rawInputLstJSON=[]&curInstr=1

Fibonacci Space Consumption



Fibonacci Space Consumption



`fib` takes **linear space.**

(Demo)