Google Looks to Leave Passwords Behind for a Billion Android Devices

https://www.cnet.com/news/google-looks-to-leave-passwords-behind-for-a-billion-android-devices/

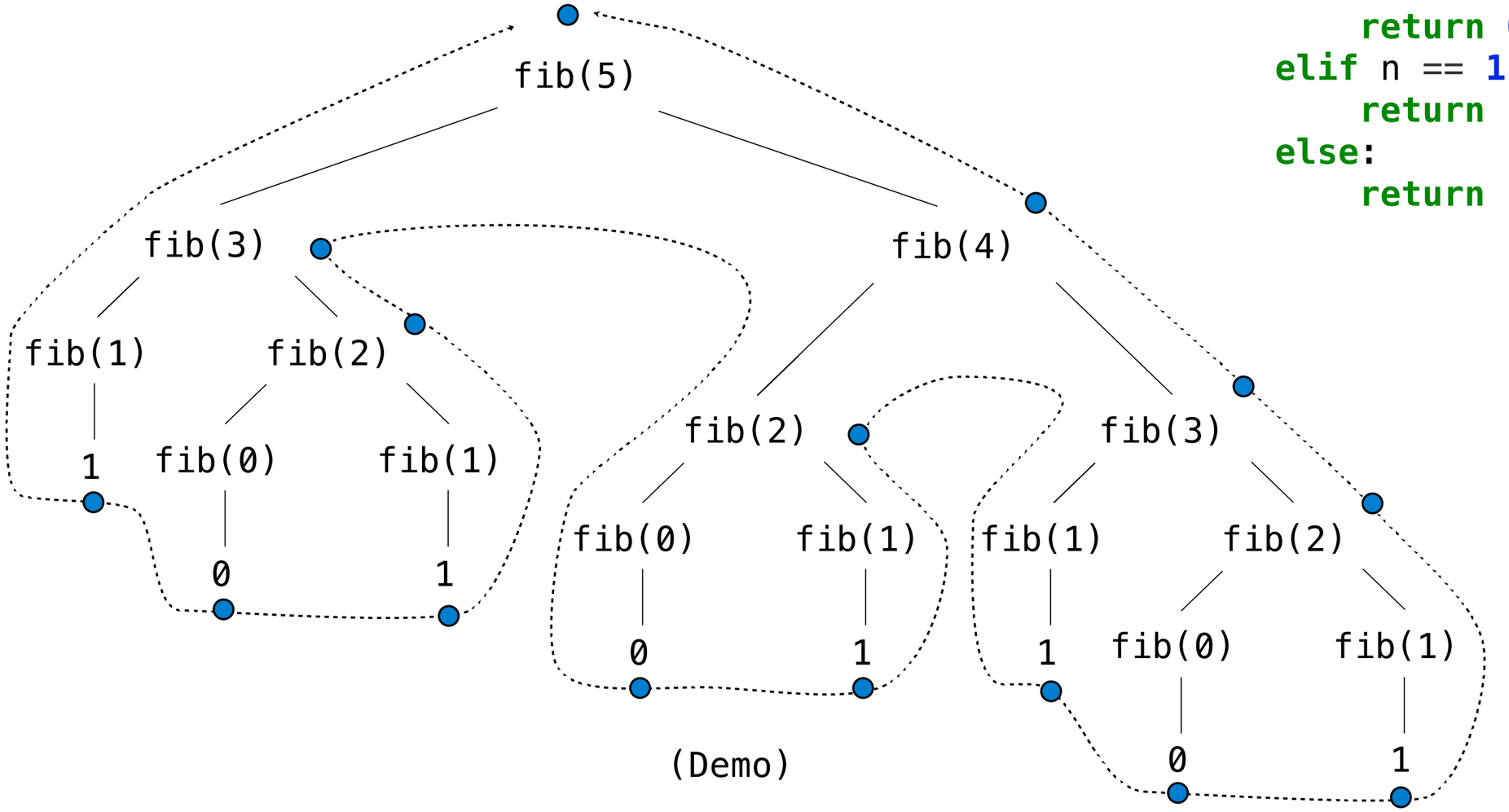
"Google and the Fast Identity Online (FIDO) Alliance announced at Mobile World Congress 2019 that Android is now FIDO2-certified, meaning Android device users can log in to their accounts with fingerprints and security keys instead of passwords. Only devices running Android 7 and up will require updates to incorporate FIDO Authentication capabilities. This development will make security features accessible to any Android developer, enabling password-free logins on the operating system's mobile browser and apps. Fingerprints and security keys are less prone to online theft than passwords, and the FIDO2 standard also shields against phishing attacks. The standard checks when someone logs in to authenticate a Web page against spoofing."

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)
```



(Demo)

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

Memoization

Idea: Remember the results that have been computed before

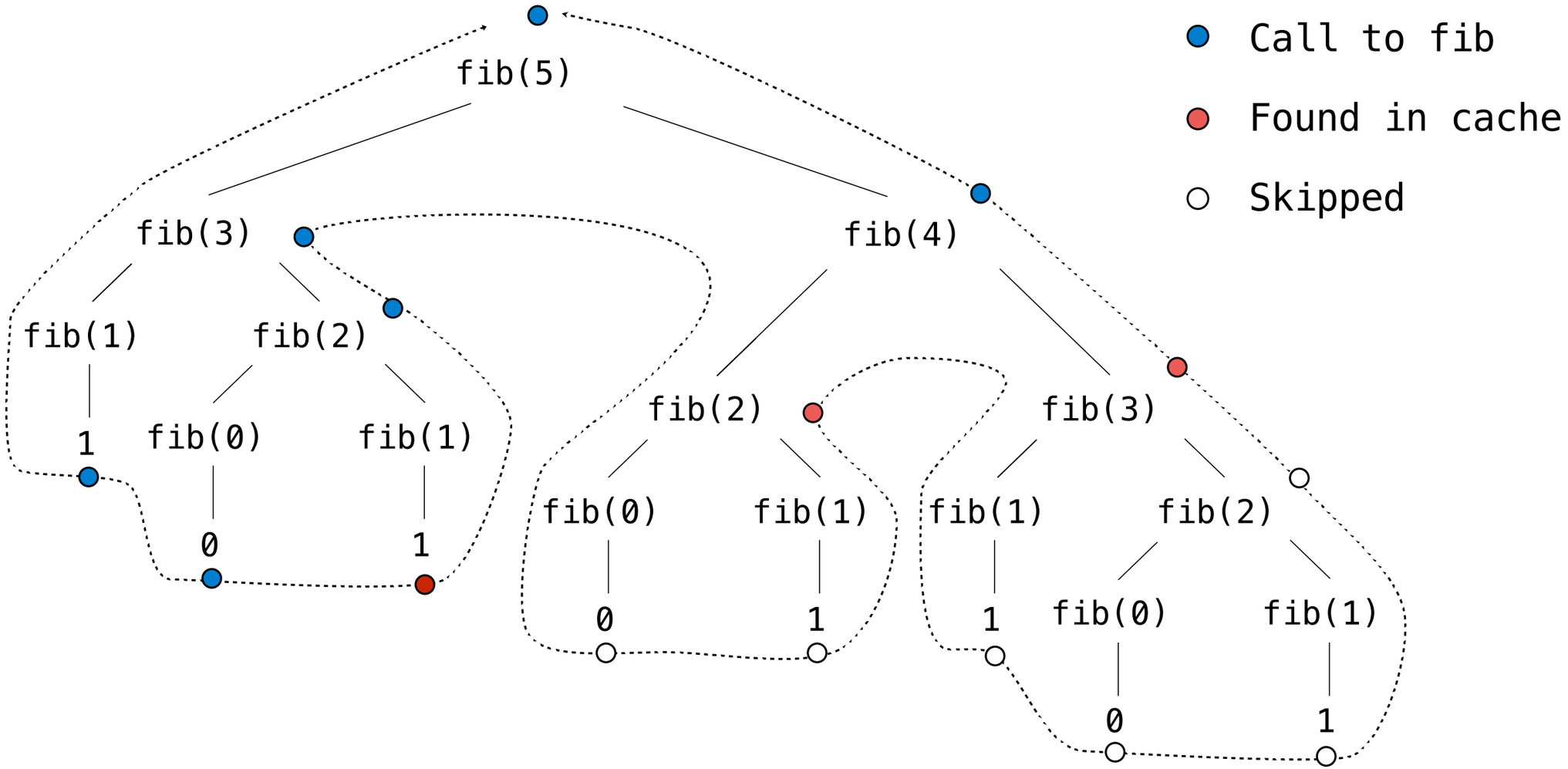
```
def memo(f):
    cache = {}
    def memoized(n):
        if n not in cache:
            cache[n] = f(n)
        return cache[n]
    return memoized
```

Keys are arguments that map to return values

Same behavior as f, if f is a pure function

(Demo)

Memoized Tree Recursion



Space

The Consumption of Space

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

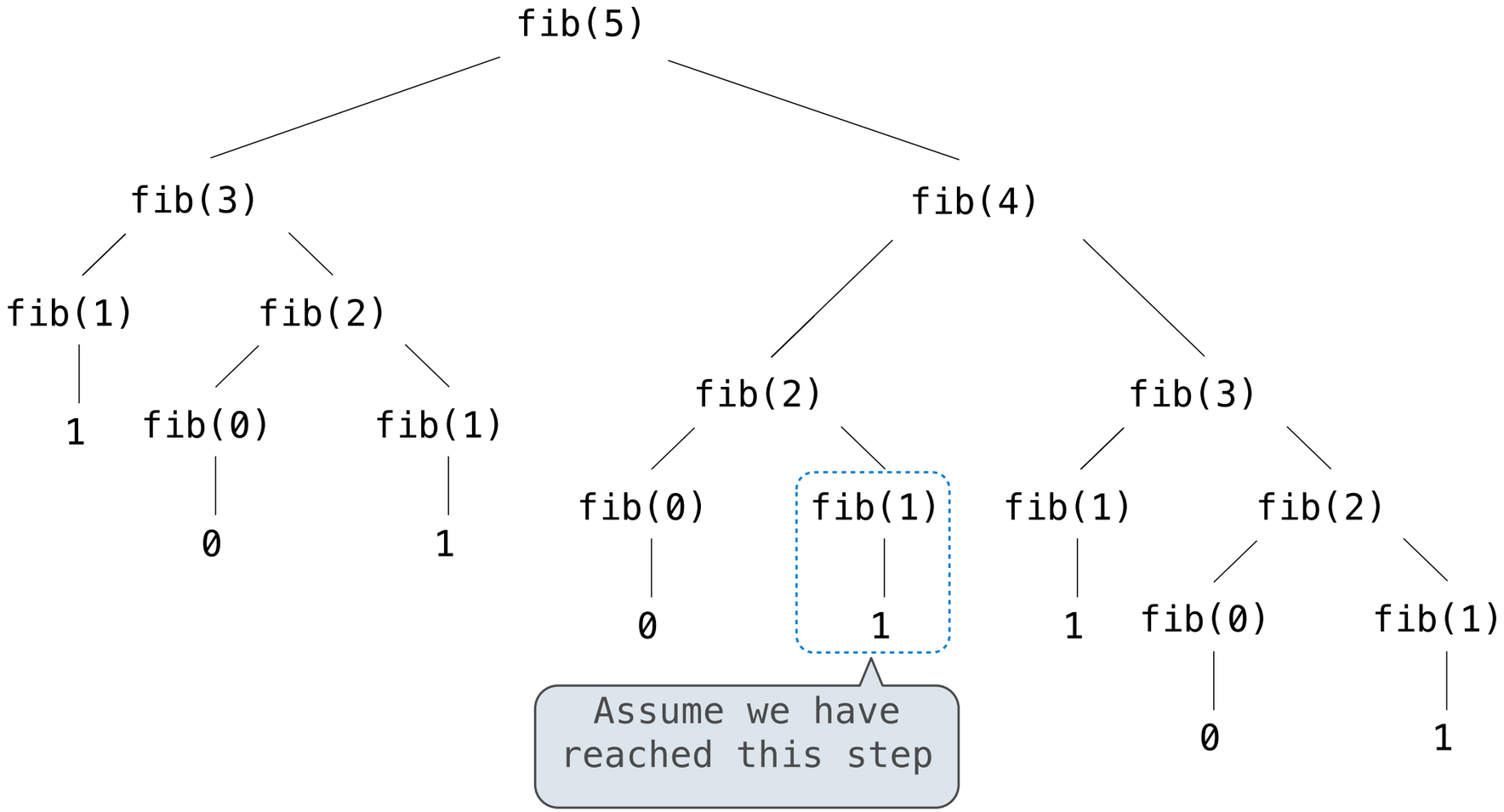
Active environments:

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

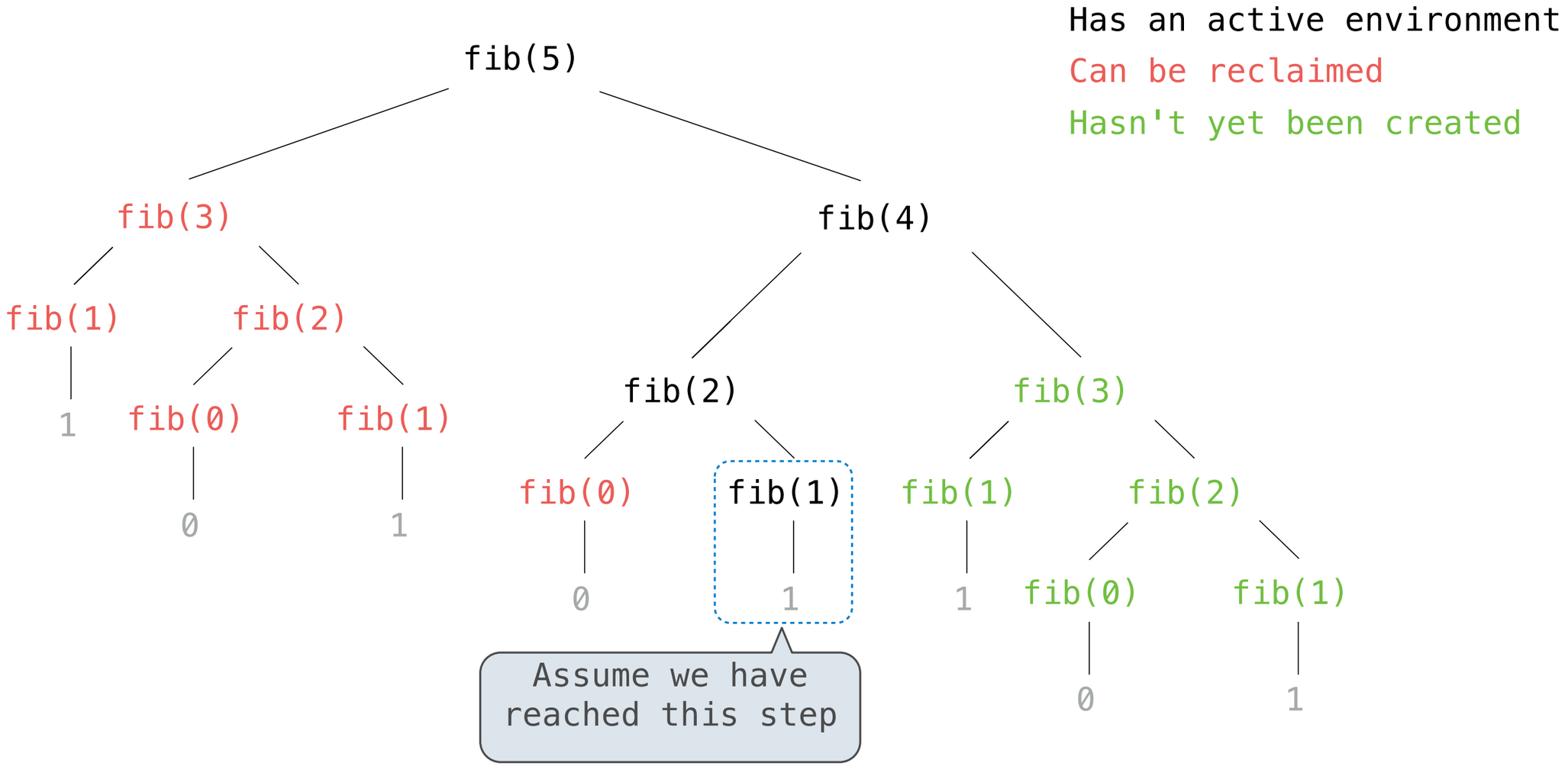
(Demo)

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%2

Fibonacci Space Consumption



Fibonacci Space Consumption



Comparing Implementations

Implementations of the same functional abstraction can require different resources

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

A factor k of n is a positive integer that evenly divides n

```
def 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!	Greatest integer less than $\sqrt{n}$

Question: How many time does each implementation use division? (Demo)

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 all **n** larger than some minimum **m**

Order of Growth of Counting Factors

Implementations of the same functional abstraction can require different amounts of time

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

A factor k of n is a positive integer that evenly divides n

```
def 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!

	Time	**Space**
Slow:	$\Theta(n)$	$\Theta(1)$
Fast:	$\Theta(\sqrt{n})$	$\Theta(1)$

Assumption: integers occupy a fixed amount of space

(Demo)

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 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)
```

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

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

(Demo)

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 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)
```

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

Properties of Orders of Growth

Constants: Constant terms do not affect the order of growth of a process

$$\Theta(n) \qquad \Theta(500 \cdot n) \qquad \Theta(\frac{1}{500} \cdot n)$$

Logarithms: The base of a logarithm does not affect the order of growth of a process

$$\Theta(\log_2 n) \qquad \Theta(\log_{10} n) \qquad \Theta(\ln n)$$

Nesting: When an inner process is repeated for each step in an outer process, multiply the steps in the outer and inner processes to find the total number of steps

```
def overlap(a, b):
    count = 0
    for item in a:
        if item in b:
            count += 1
    return count
```

Outer: length of a

Inner: length of b

If a and b are both length **n**, then overlap takes $\Theta(n^2)$ steps

Lower-order terms: The fastest-growing part of the computation dominates the total

$$\Theta(n^2) \qquad \Theta(n^2 + n) \qquad \Theta(n^2 + 500 \cdot n + \log_2 n + 1000)$$

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., overlap

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

$\Theta(n)$ Linear growth. E.g., slow factors or exp

$\Theta(\sqrt{n})$ Square root growth. E.g., factors_fast

$\Theta(\log n)$ Logarithmic growth. E.g., exp_fast

Doubling the problem only increments R(n).

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