Growth

Announcements

Measuring Efficiency

same as @count

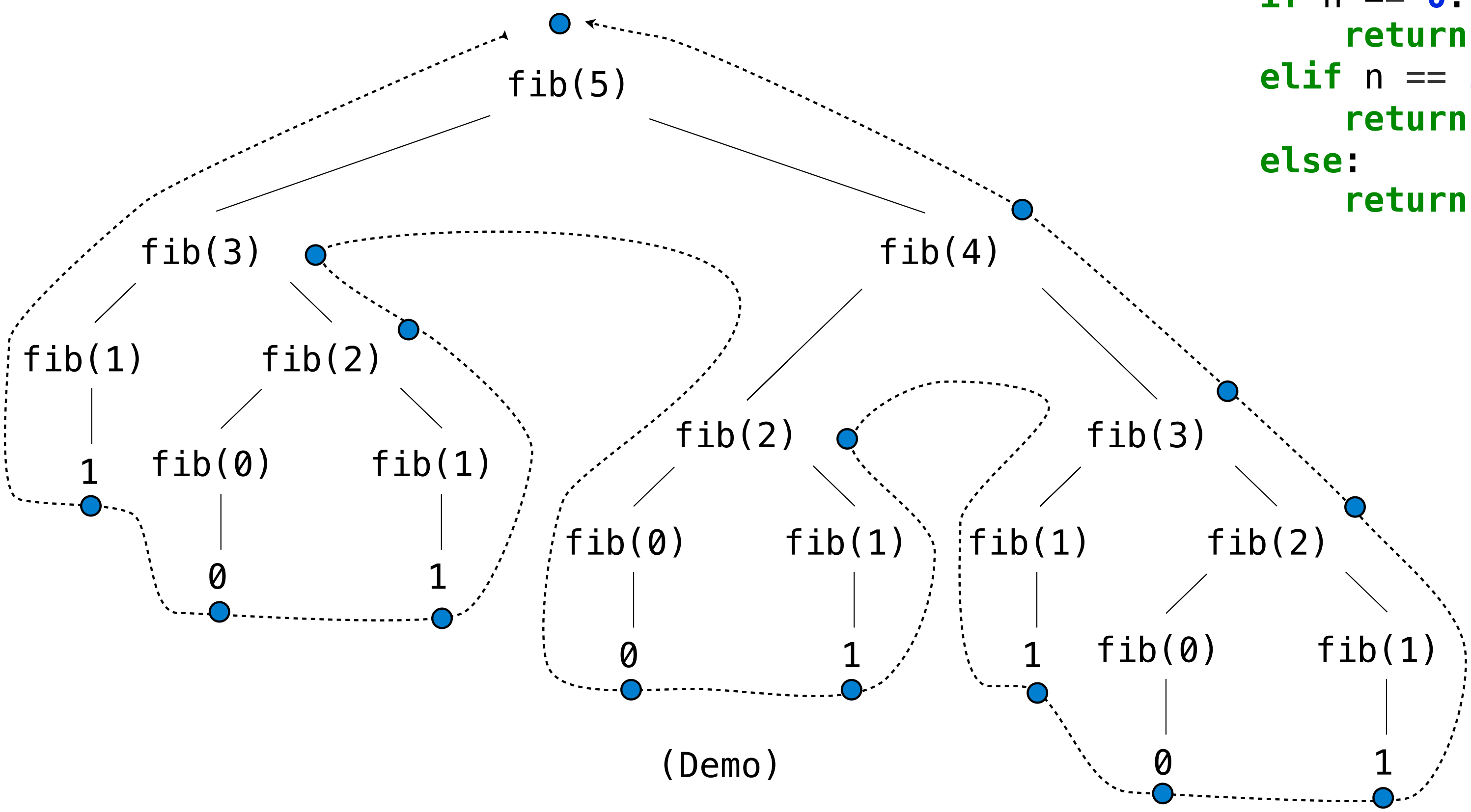
```
~/lec$ python3 -i ex.py
>>> fib = count(fib)
>>> fib(5)
5
>>> fib.call_count
15
>>> fib(5)
5
>>> fib.call_count
30
>>> fib(30)
832040
>>> fib.call_count
2692567
```

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

def count(f):
    def counted(n):
        counted.call_count += 1
        return f(n)
    counted.call_count = 0
    return counted
```

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)
```

(Demo)

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

Memoization

Memoization

Idea: Remember the results that have been computed before

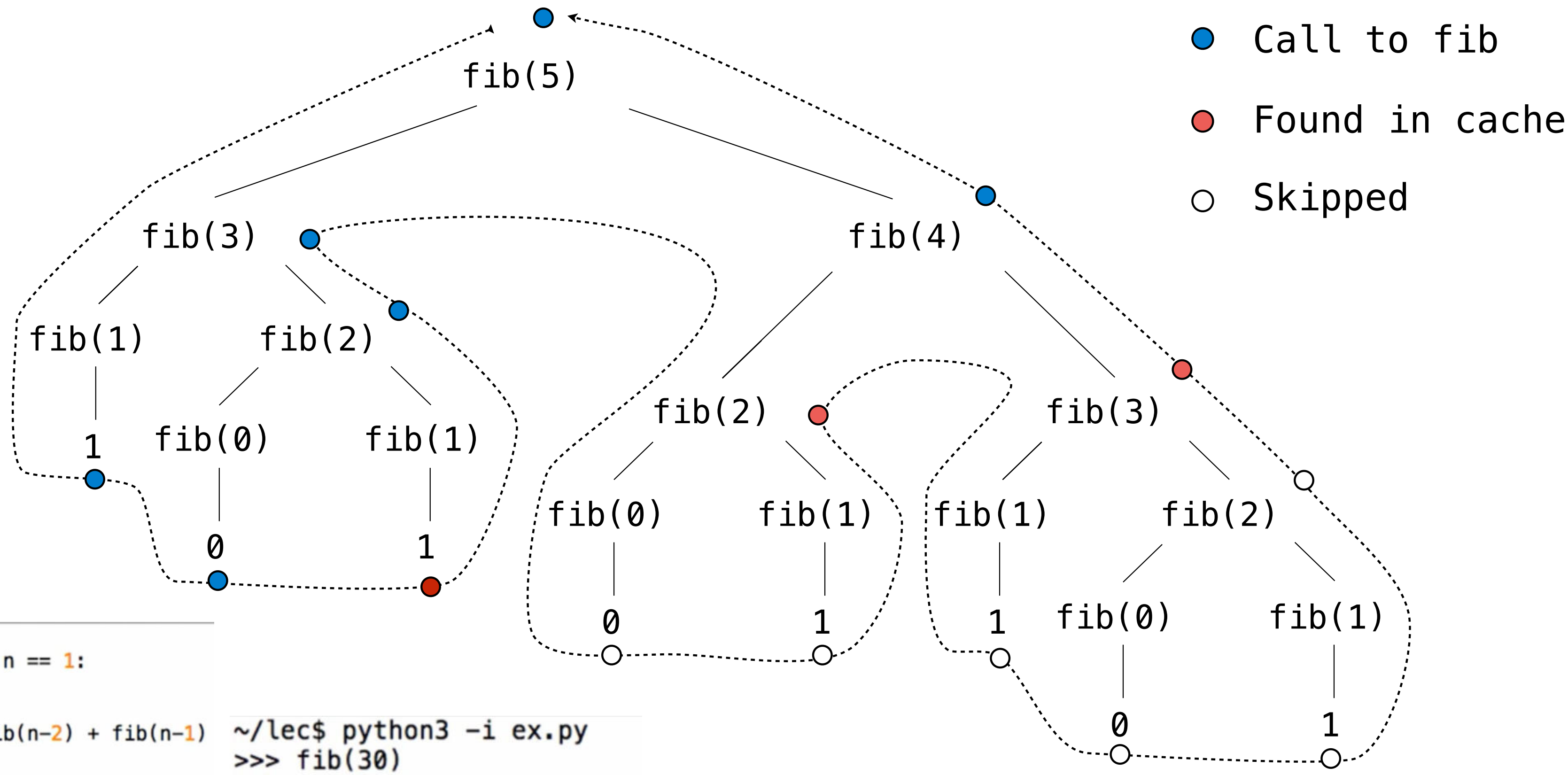
```
def memo(f):
    cache = {}
    def memoized(n):
        if n not in cache:
            cache[n] = f(n)
        return cache[n]
    return memoized
```

Keys are arguments that map to return values

Same behavior as f, if f is a pure function

(Demo)

Memoized Tree Recursion




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

def count(f):
    def counted(n):
        counted.call_count += 1
        return f(n)
    counted.call_count = 0
    return counted

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

```
~/lec$ python3 -i ex.py
>>> fib(30)
832040
>>> fib = count(fib)
>>> counted_fib = fib
>>> fib = memo(fib)
>>> fib = count(fib)
>>> fib(30)
832040
>>> fib.call_count
59
>>> counted_fib.call_count
31
```

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)

Interactive Diagram

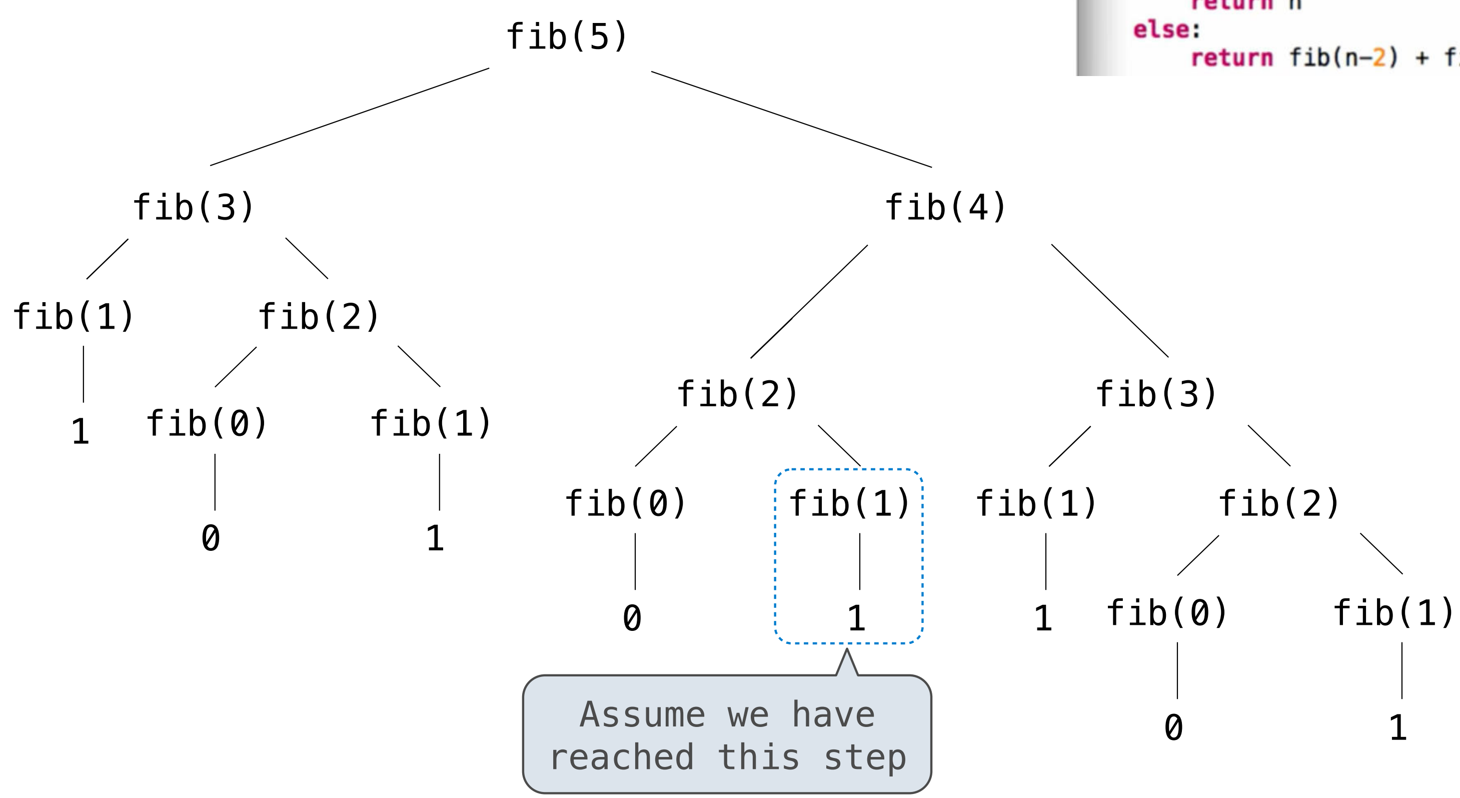
Fibonacci Space Consumption

```
~/lec$ python3 -i ex.py
>>> fib = count_frames(fib)
>>> fib(20)
6765
>>> fib.open_count
0
>>> fib.max_count
20
```

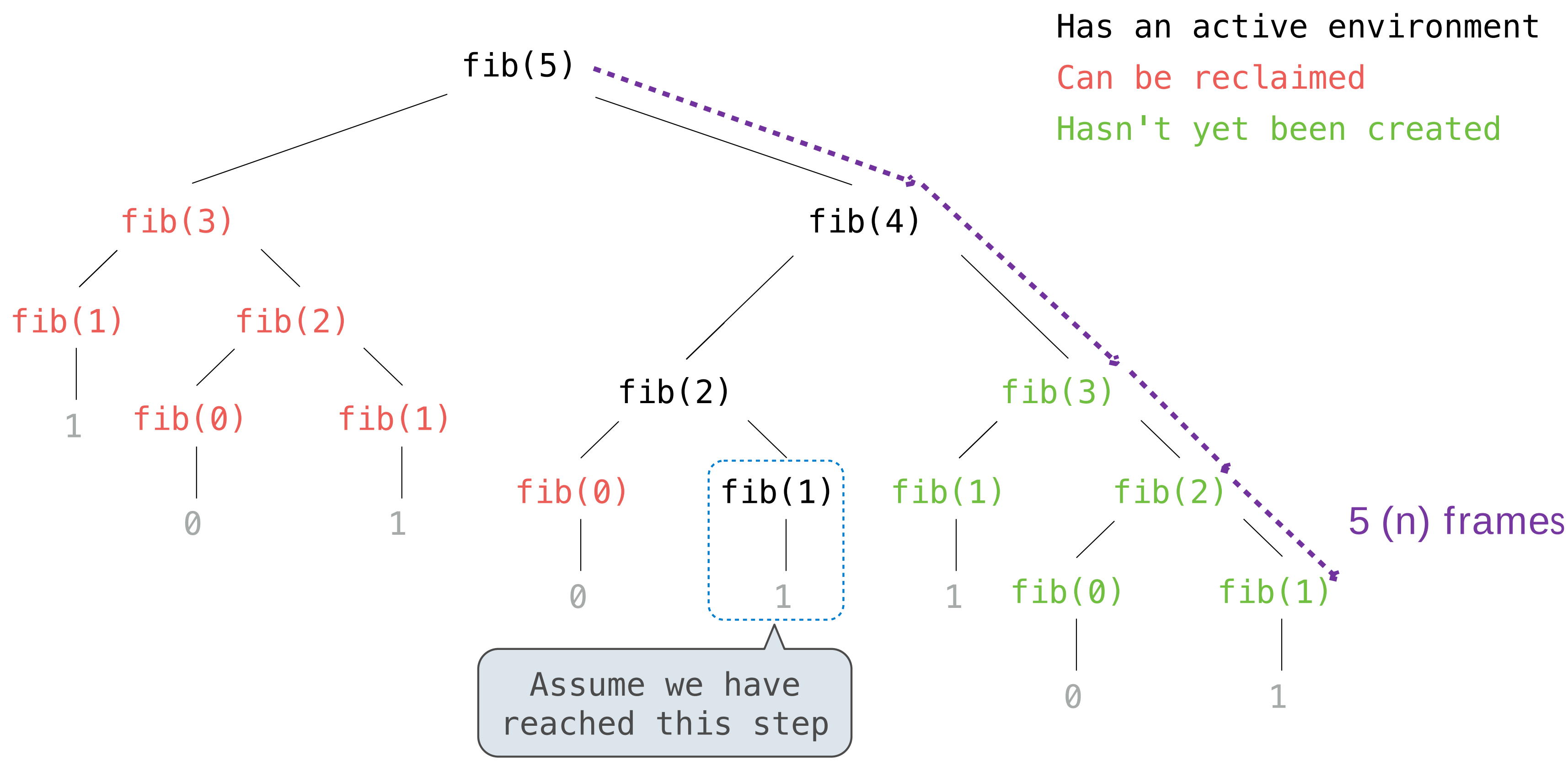
see next slide for why is 20

```python
def count_frames(f):
    def counted(n):
        counted.open_count += 1
        if counted.open_count > counted.max_count:
            counted.max_count = counted.open_count
        result = f(n)
        counted.open_count -= 1
        return result
    counted.open_count = 0
    counted.max_count = 0
    return counted

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



Fibonacci Space Consumption



Time

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):
```

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)

```
~/lec$ python3 -i ex.py
>>> factors(6)
4
>>> factors(24)
8
>>> factors(576)
21
>>> divides = count(divides)
>>> factors(576)
21
>>> divides.call_count
576
>>>
```

```python
"""Lecture 21 examples"""

def count(f):
    def counted(*args):
        counted.call_count += 1
        return f(*args)
    counted.call_count = 0
    return counted

def divides(k, n):
    return n % k == 0

def factors(n):
    total = 0
    for k in range(1, n+1):
        if divides(k, n):
            total += 1
    return total
```

```
>>> ^D
~/lec$ python3 -i ex.py
>>> factors_fast(576)
21
>>> divides.call_count
23
>>>
```

=sqrt(576) - 1

```python
@count
def divides(k, n):
    return n % k == 0

def factors(n):
    total = 0
    for k in range(1, n+1):
        if divides(k, n):
            total += 1
    return total

from math import sqrt

def factors_fast(n):
    total = 0
    sqrt_n = sqrt(n)
    k = 1
    while k < sqrt_n:
        if divides(k, n):
            total += 2
        k += 1
    if k*k == n:
        total += 1
    return total
```

Orders of Growth

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 $\mathbf{k_1}$ and $\mathbf{k_2}$ such that

$$\boxed{k_1 \cdot f(n) \leq \boxed{R(n)} \leq k_2 \cdot f(n)}$$

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):
```

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

Assumption: integers occupy a fixed amount of space

(Demo)

Exponentiation

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 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)
```

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

(Demo)

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

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

@trace
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)
```

```
>>> exp(2, 10)
exp(2, 10):
    exp(2, 9):
        exp(2, 8):
            exp(2, 7):
                exp(2, 6):
                    exp(2, 5):
                        exp(2, 4):
                            exp(2, 3):
                                exp(2, 2):
                                    exp(2, 1):
                                        exp(2, 0):
                                        exp(2, 0) -> 1
                                    exp(2, 1) -> 2
                                exp(2, 2) -> 4
                            exp(2, 3) -> 8
                        exp(2, 4) -> 16
                    exp(2, 5) -> 32
                exp(2, 6) -> 64
            exp(2, 7) -> 128
        exp(2, 8) -> 256
    exp(2, 9) -> 512
exp(2, 10) -> 1024
1024
```

```
>>> fast_exp(2, 10)
fast_exp(2, 10):
    fast_exp(2, 5):
        fast_exp(2, 4):
            fast_exp(2, 2):
                fast_exp(2, 1):
                    fast_exp(2, 0):
                    fast_exp(2, 0) -> 1
                fast_exp(2, 1) -> 2
            fast_exp(2, 2) -> 4
        fast_exp(2, 4) -> 16
    fast_exp(2, 5) -> 32
fast_exp(2, 10) -> 1024
1024
```

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
$\Theta(n)$	$\Theta(n)$
$\Theta(\log n)$	$\Theta(\log n)$

2^5 -> 2^10 (n: 5 -> 10)
just add one step
n -> k * n
time -> time * log k

Comparing Orders of Growth

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