Efficiency

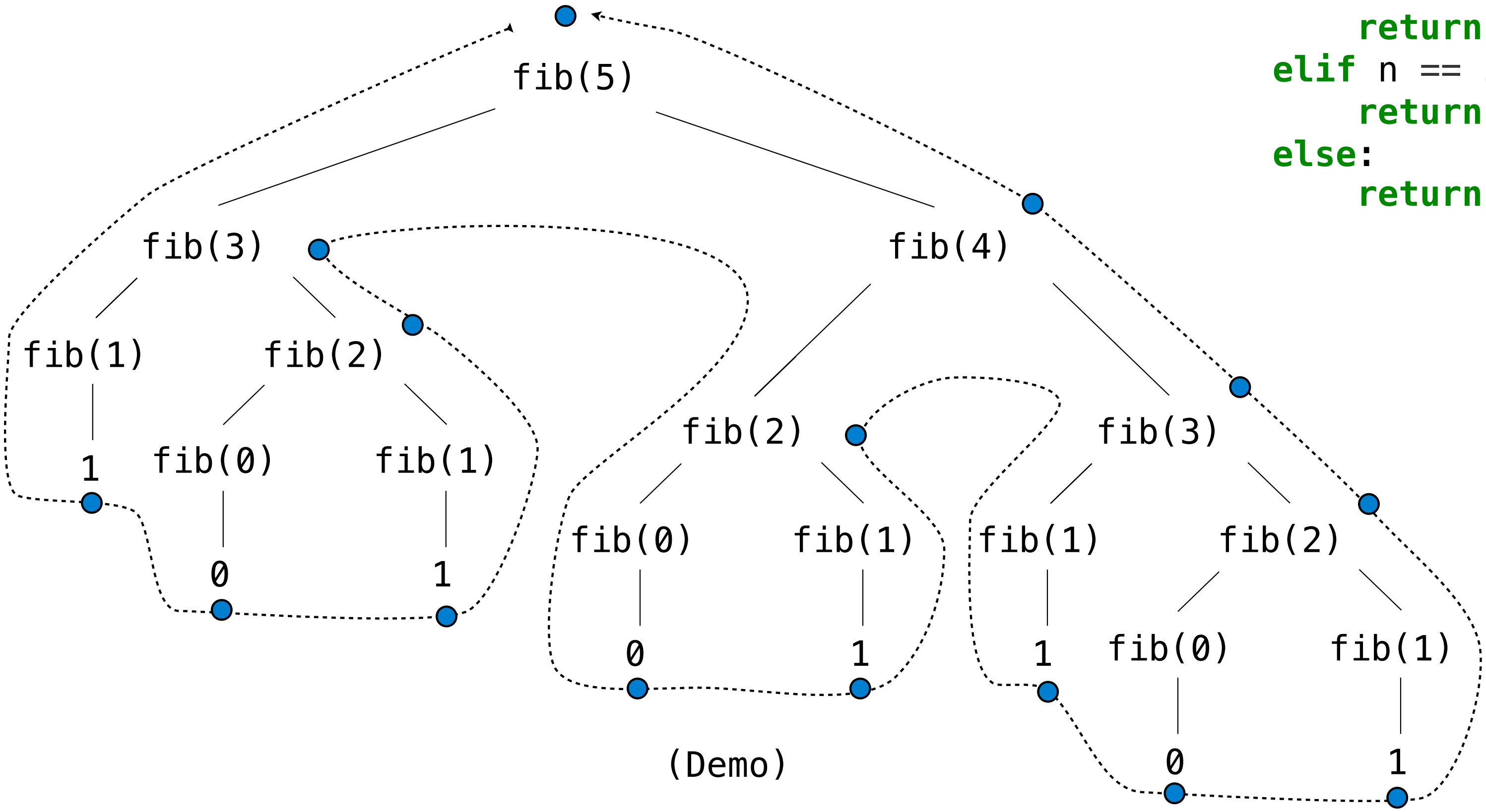
Announcements

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)
```



(Demo)

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

Memoization

Memoization

Idea: Remember the results that have been computed before

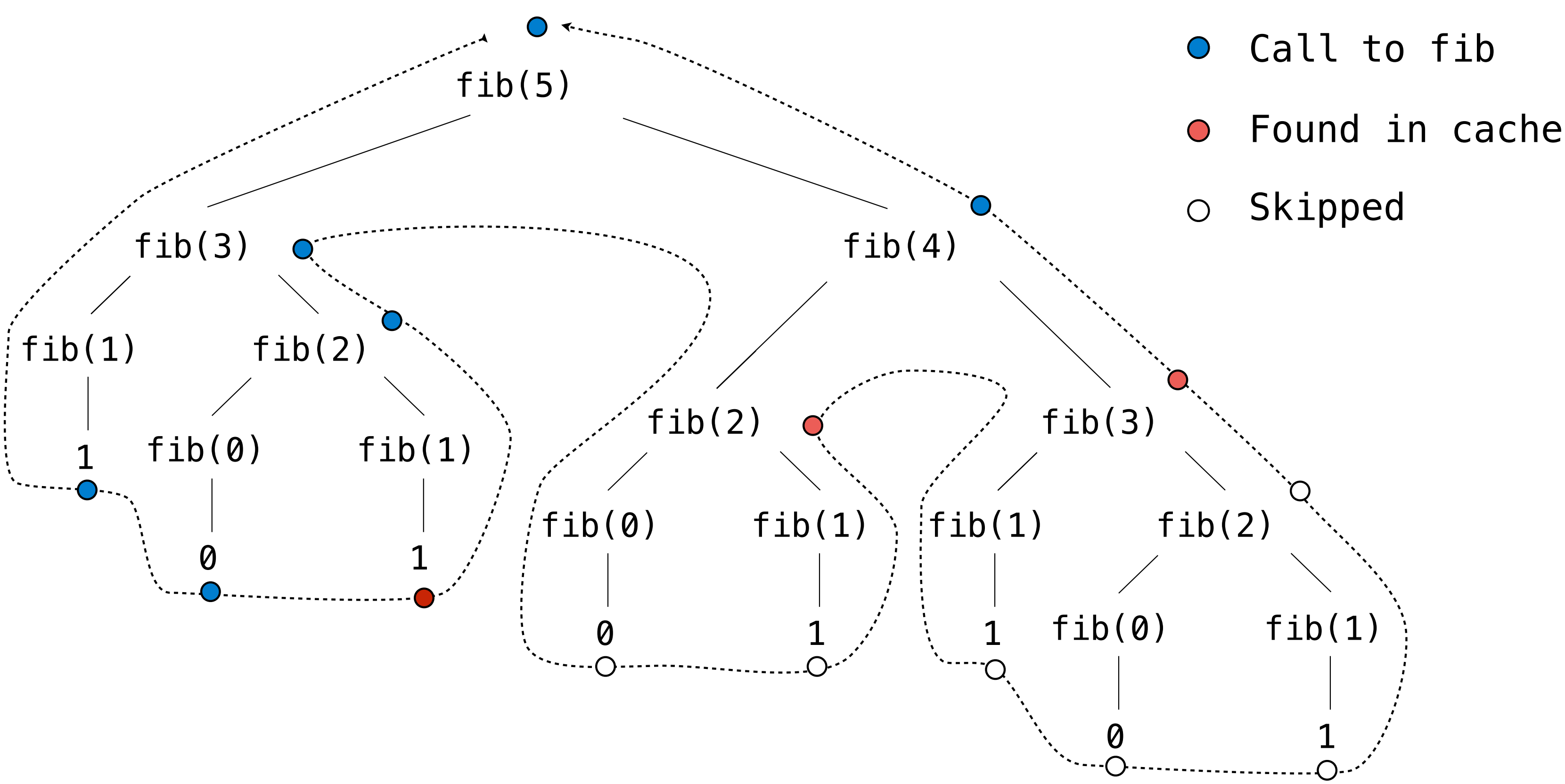
```
def memo(f):
    cache = {}
    def memoized(n):
        if n not in cache:
            cache[n] = f(n)
        return cache[n]
    return memoized
```

Keys are arguments that map to return values

Same behavior as f, if f is a pure function

(Demo)

Memoized Tree Recursion



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 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
```

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

(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)
```

Linear time:

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

```
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
```

Logarithmic time:

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

Orders of Growth

Quadratic Time

Functions that process all pairs of values in a sequence of length n take quadratic time

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

overlap([3, 5, 7, 6], [4, 5, 6, 5])
```

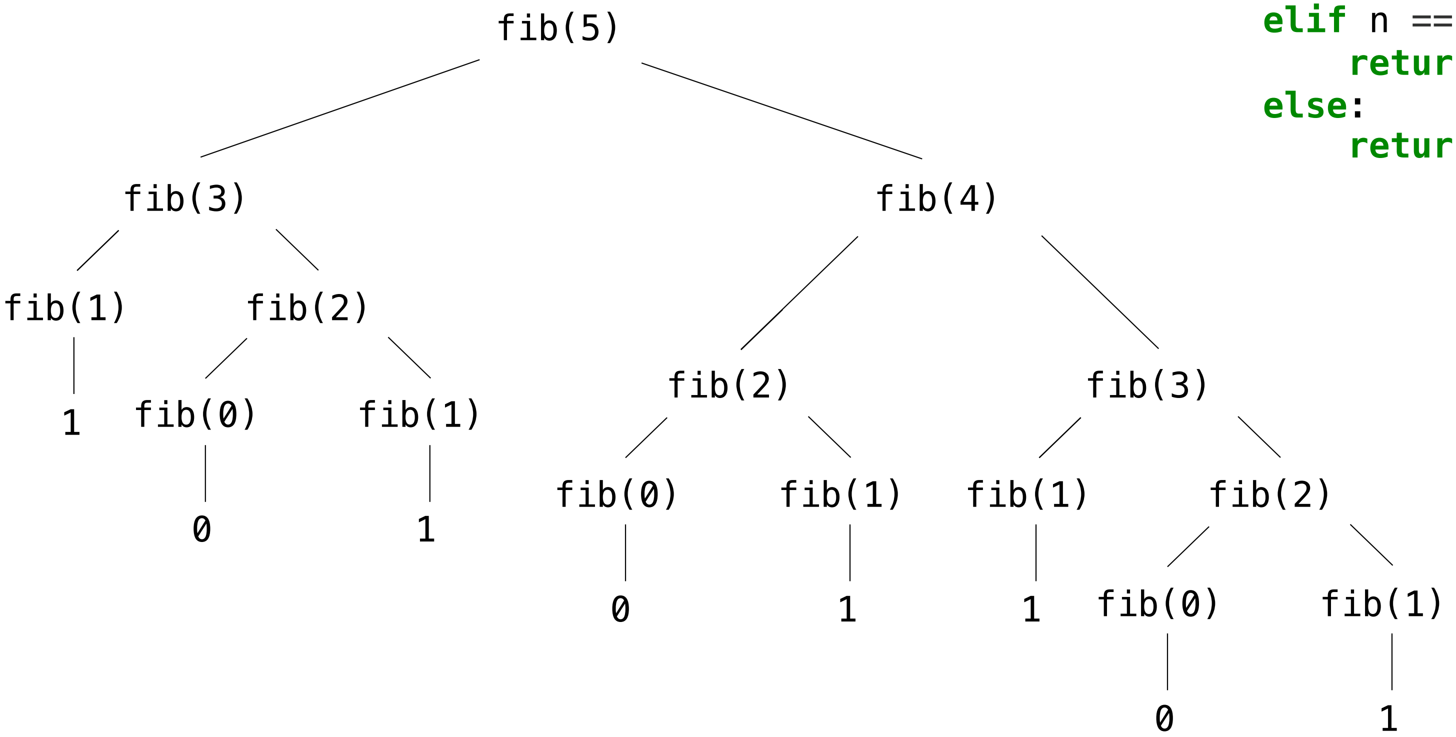
	3	5	7	6
4	0	0	0	0
5	0	1	0	0
6	0	0	0	1
5	0	1	0	0

(Demo)

Exponential Time

Tree-recursive functions can take exponential time

```
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

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

Common Orders of Growth

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

Incrementing *n* multiplies *time* by a constant

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

Quadratic growth. E.g., `overlap`

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

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

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

Incrementing *n* increases *time* by a constant

$$a \cdot (n+1) = (a \cdot n) + a$$

Logarithmic growth. E.g., `exp_fast`

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

$$a \cdot \ln(2 \cdot n) = (a \cdot \ln n) + a \cdot \ln 2$$

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

Order of Growth Notation

Big Theta and Big O Notation for Orders of Growth

Exponential growth. E.g., recursive `fib` Incrementing *n* multiplies *time* by a constant	$\Theta(b^n)$	$O(b^n)$
Quadratic growth. E.g., `overlap` Incrementing *n* increases *time* by *n* times a constant	$\Theta(n^2)$	$O(n^2)$
Linear growth. E.g., slow `exp` Incrementing *n* increases *time* by a constant	$\Theta(n)$	$O(n)$
Logarithmic growth. E.g., `exp_fast` Doubling *n* only increments *time* by a constant	$\Theta(\log n)$	$O(\log n)$
Constant growth. Increasing *n* doesn't affect time	$\Theta(1)$	$O(1)$

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

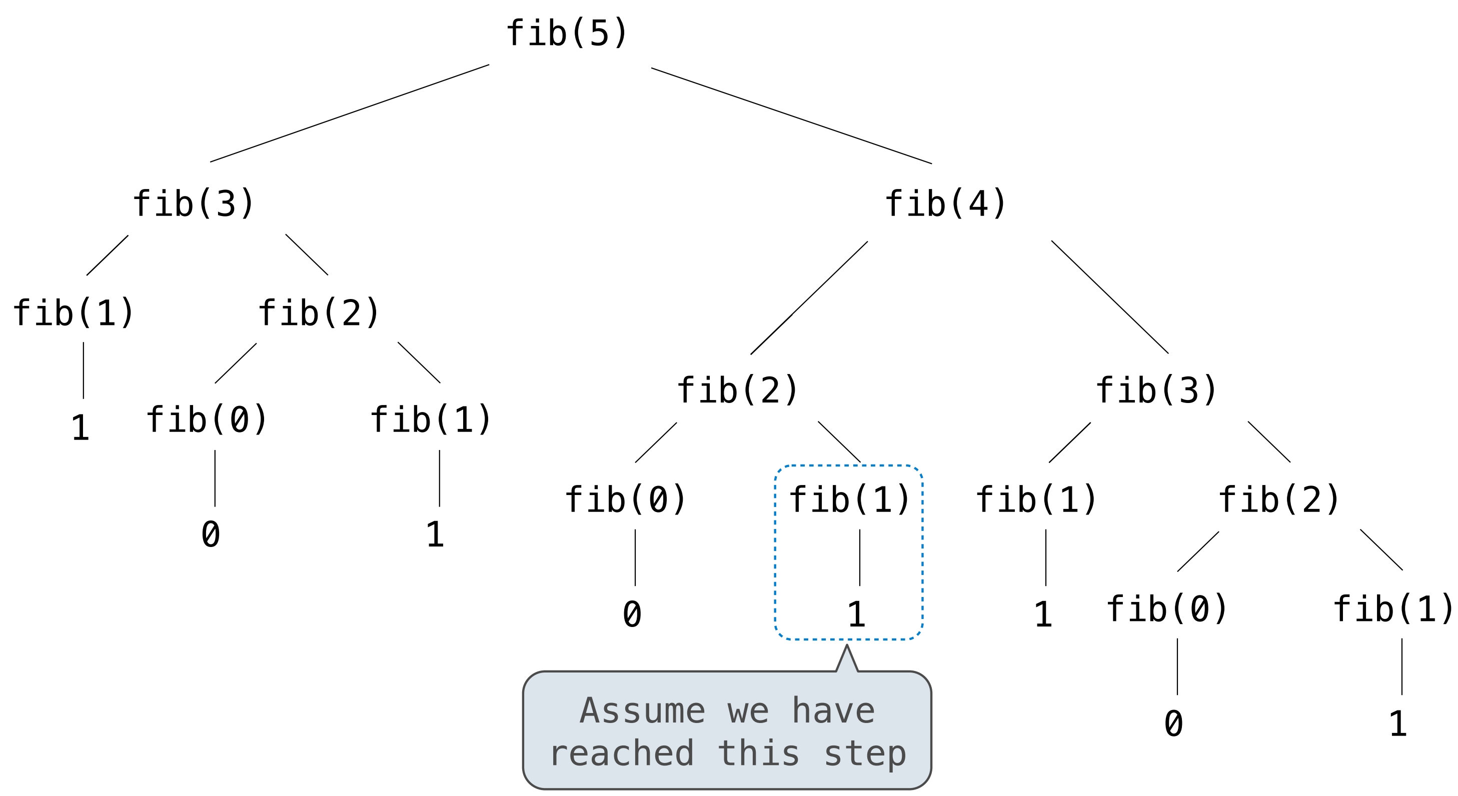
Active environments:

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

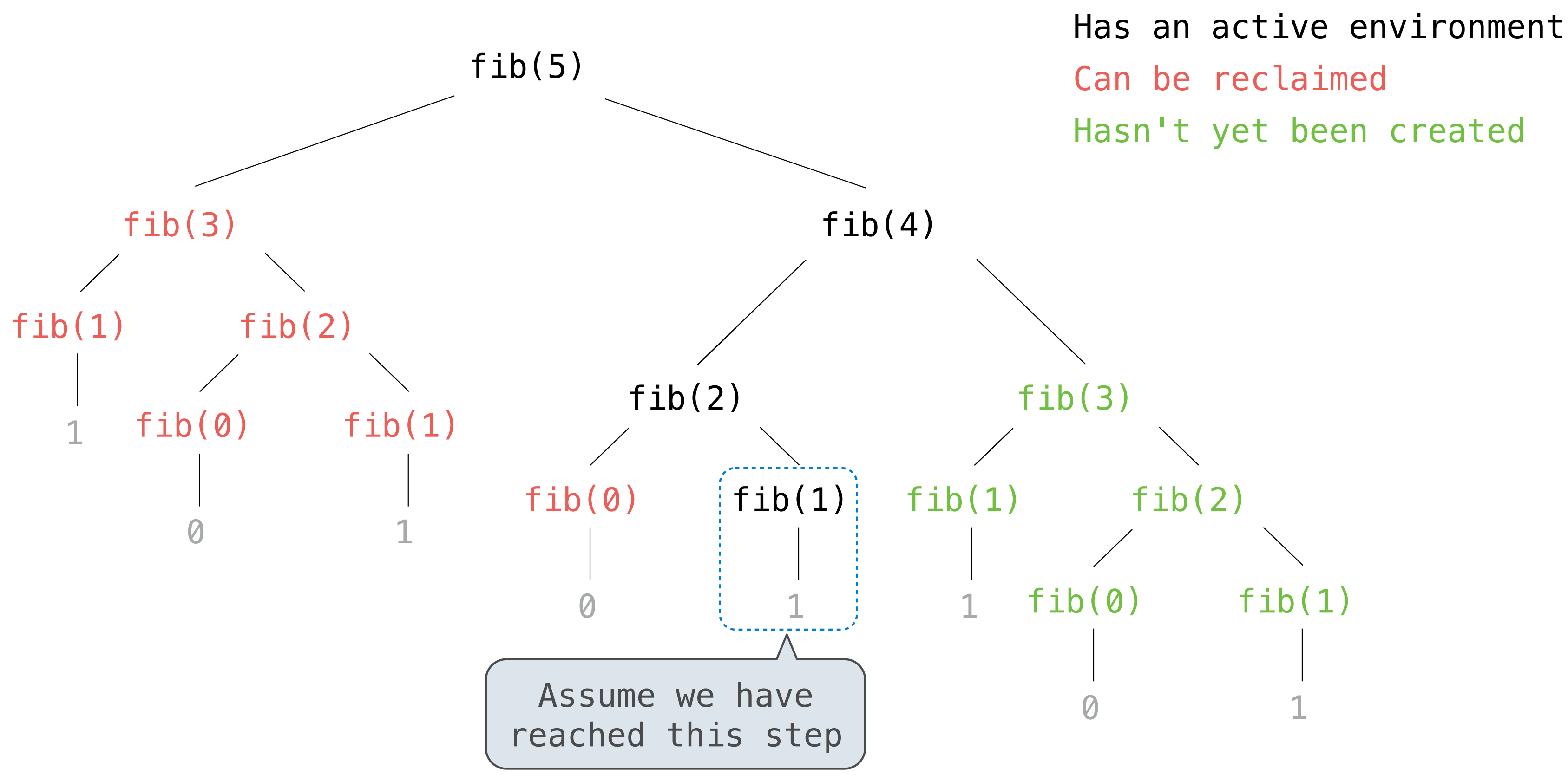
(Demo)

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



22

December 9, 2021

[ ]:
```
%%time
exp(2, 400)
```

```
CPU times: user 273 µs, sys: 44 µs, total: 317 µs
Wall time: 322 µs
```

[ ]:
```
2582249878086908589655919172003011874329705792829223512830659356540647622016841194629645353280137831435903171972747493376
```

[ ]:
```
%matplotlib inline
import matplotlib.pyplot as plt
plt.style.use('ggplot')
plt.rc('font', size=16)

from timeit import repeat
from numpy import median, percentile

def plot_times(name, xs, n=15):
    f = lambda x: name + '(' + str(x) + ')'
    g = globals()

    samples = []
    for _ in range(n):
        times = lambda x: repeat(f(x), globals=g, number=1, repeat=n)
        samples.append([median(times(x)) for x in xs])
    ys = [10e3 * median(sample) for sample in zip(*samples)]

    plt.figure(figsize=(8, 8))
    plt.plot(xs, ys)
    plt.xlabel('n')
    plt.ylabel('milliseconds')
```

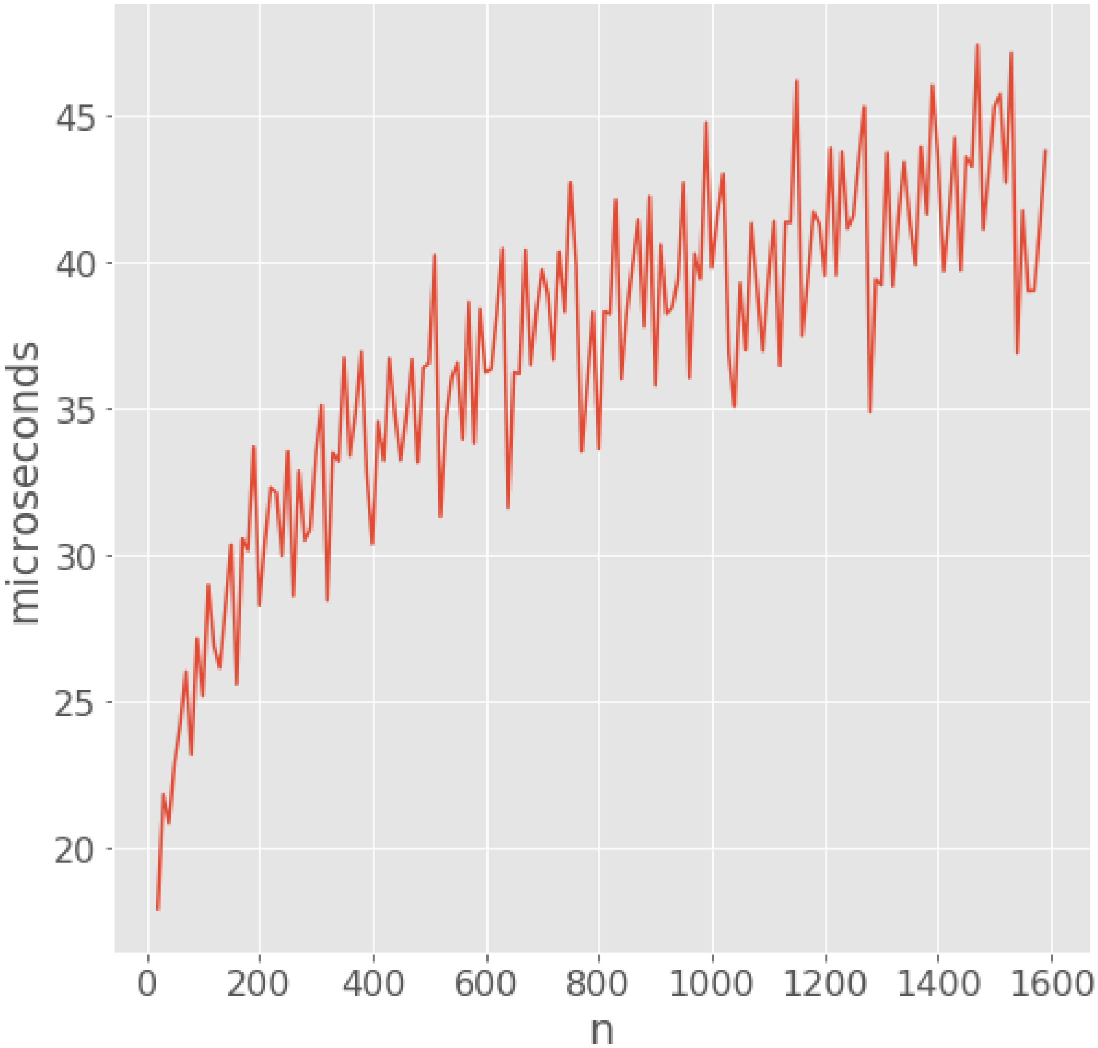
[ ]:
```
def exp(b, n):
    if n == 0:
        return 1
    else:
        return b * exp(b, n-1)
```

[ ]:
```
exp_2 = lambda n: exp(2, n)
plot_times('exp_2', range(20, 1600, 10))
```

[ ]:
```
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
```

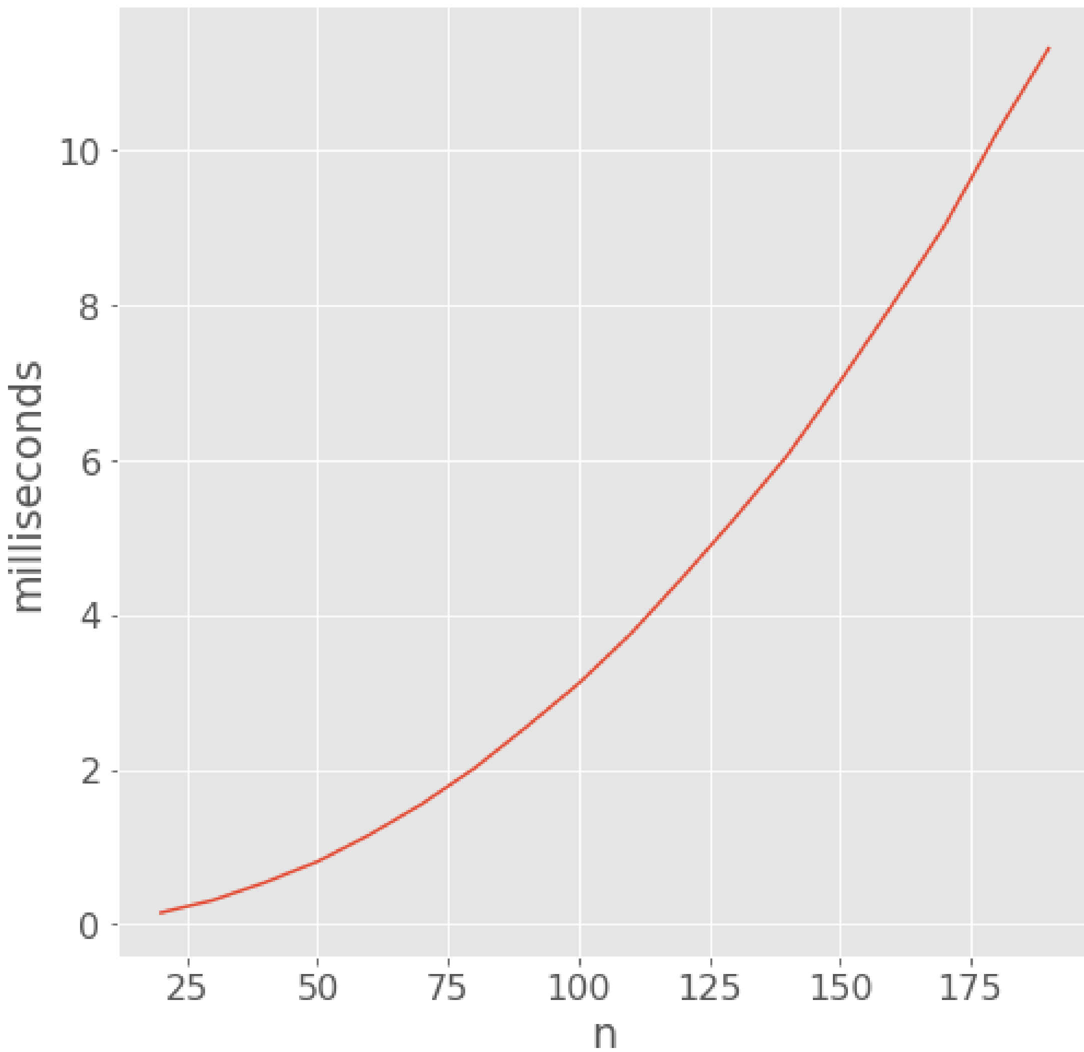
[ ]:
```
exp_2_fast = lambda n: exp_fast(2.0, n)
plot_times('exp_2_fast', range(20, 1600, 10))
```



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

overlap_ranges = lambda n: overlap(list(range(n)), list(range(n)))

plot_times('overlap_ranges', range(20, 200, 10))
```



milliseconds
n
0
2
4
6
8
10
25
50
75
100
125
150
175