

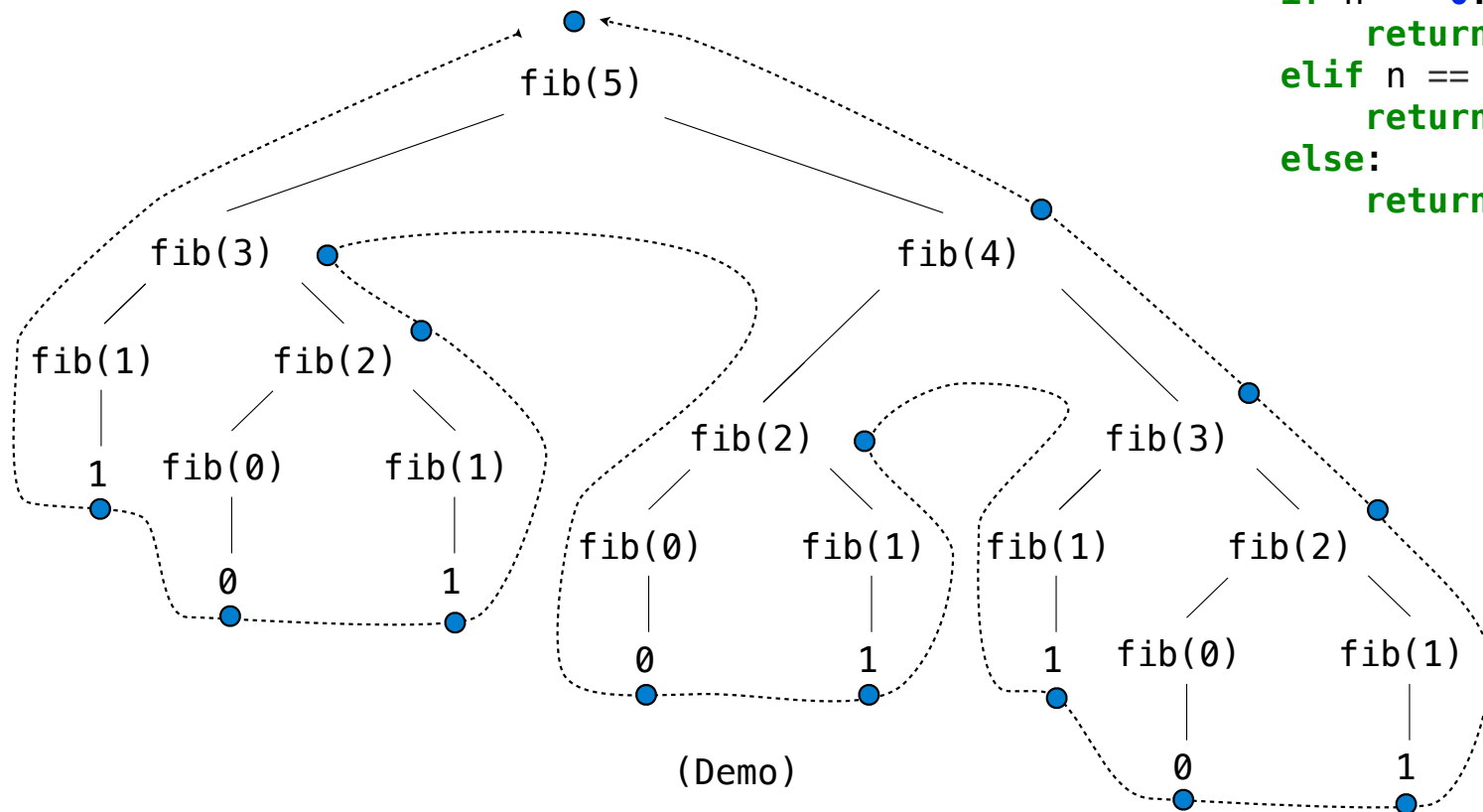
Efficiency

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)
```



Memoization

Memoization

Idea: Remember the results that have been computed before

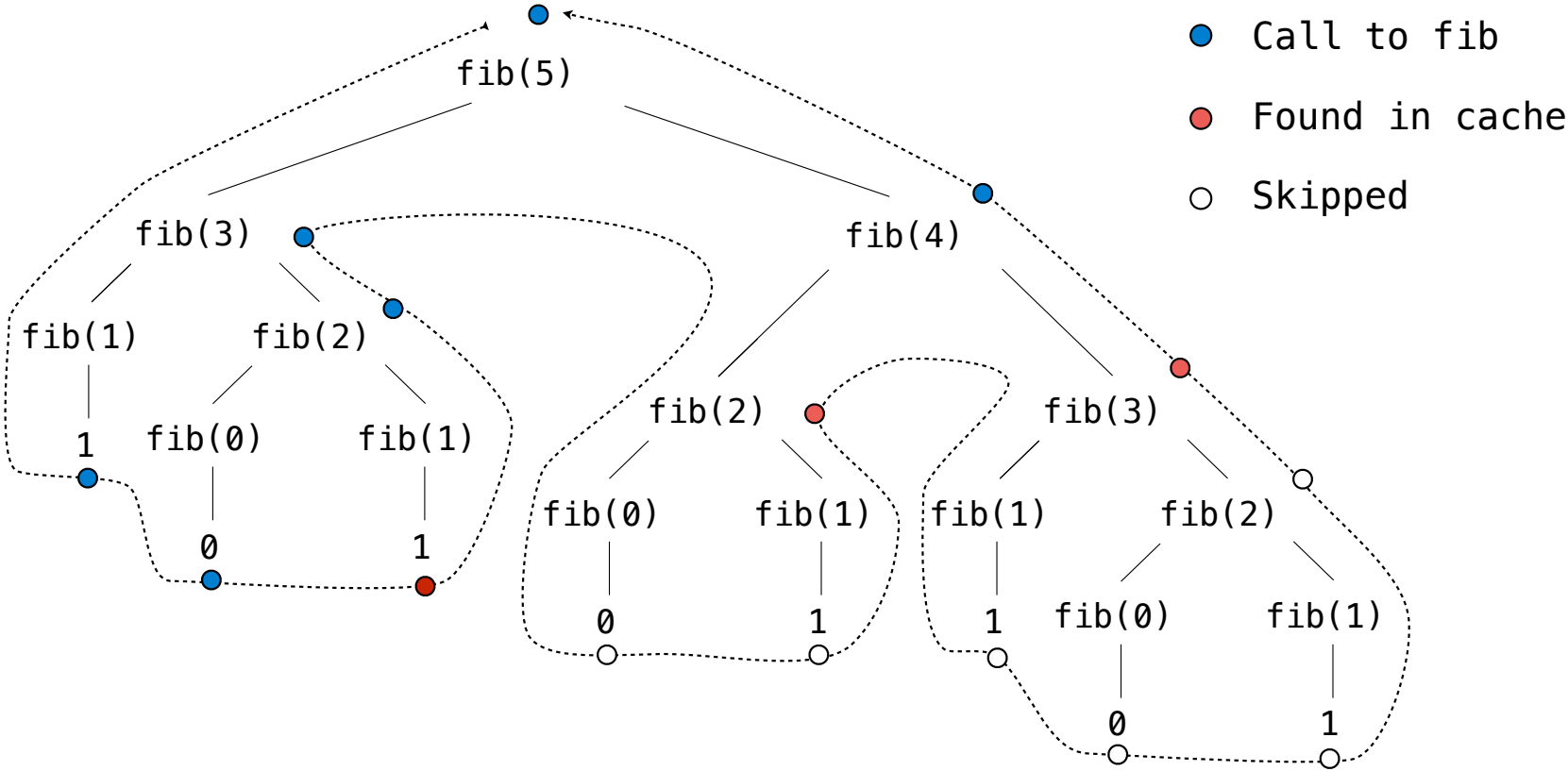
```
def memo(f):  
    cache = {}  
    def memoized(n):  
        if n not in cache:  
            cache[n] = f(n)  
        return cache[n]  
    return memoized
```

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)
```

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

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

(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 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 one step
- 1024x the input increases the time by only 10 steps

Orders of Growth

Quadratic Time

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

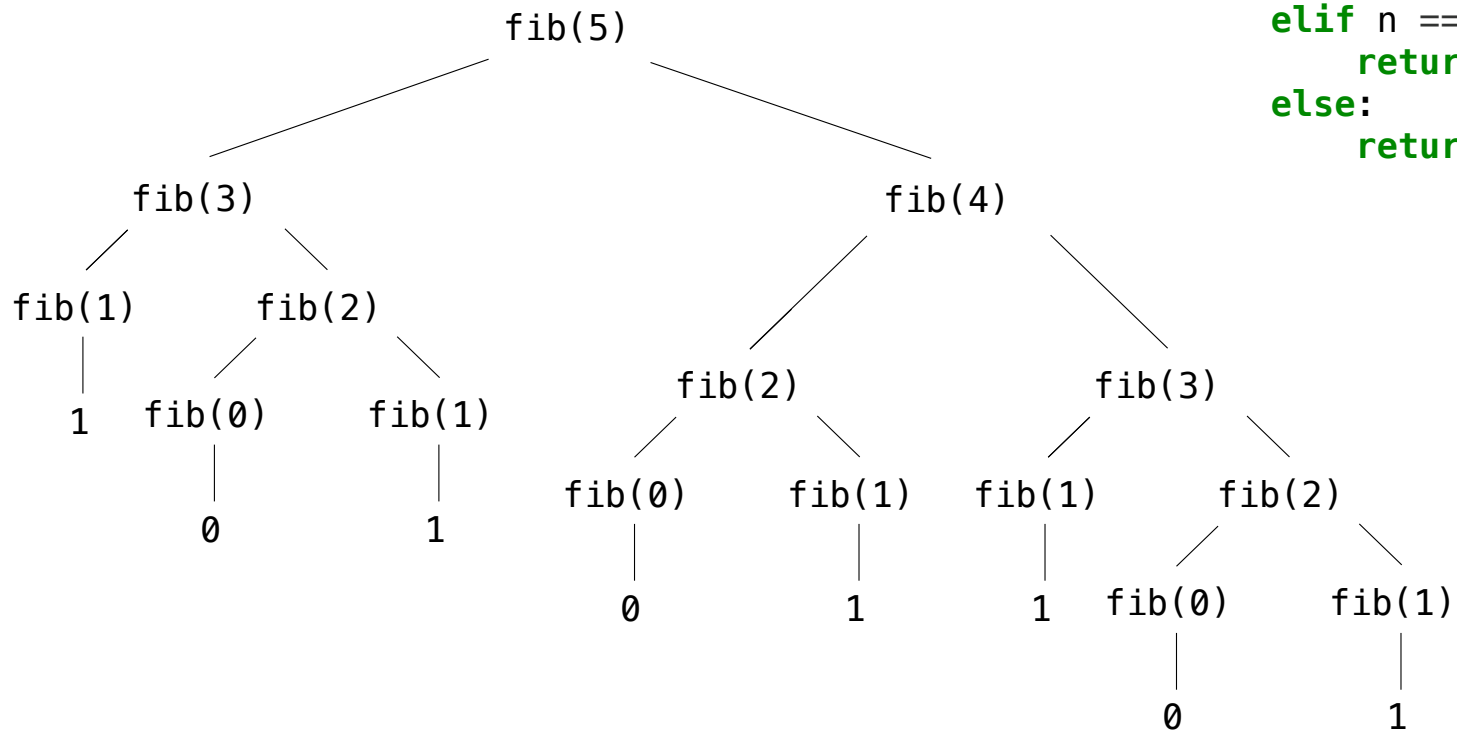
```
def overlap(a, b):  
    count = 0  
    for item in a:  
        for other in b:  
            if item == other:  
                count += 1  
    return count  
  
overlap([3, 5, 7, 6], [4, 5, 6, 5])
```

	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)
```



Common Orders 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

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

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

Fibonacci Space Consumption

