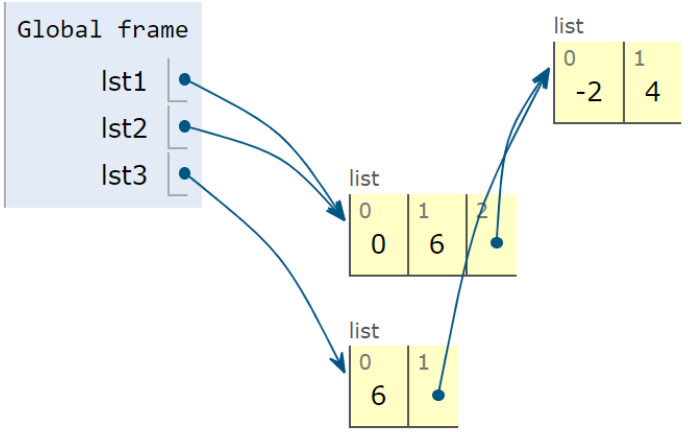
Lecture 12 - Mutable Functions & Growth

Review: Mutable Values

Identity vs. Equality in Environment Diagrams

- Review: For assignment statements, evaluate the right side, then assign to the left.
- Copying: When creating a copy, copy exactly what's "in the box".

```
>>> lst1 = [0, 6, [-2, 4]]
>>> lst2 = lst1
>>> lst3 = lst1[1:]
```



Mutating Within Functions

```
>>> def mystery(lst): # mutative function
...     lst.pop()
...     lst.pop()
>>> four = [4, 4, 4, 4]
>>> mystery(four)
>>> four
[4, 4]
```

Mutable Functions

Functions with behavior that changes over time

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

>>> square(5)
25
>>> square(5)
25
>>> square(5)
25
```

Returns the same value when called with the same input

```
def f(x):
    ...

>>> f(5)
25
>>> f(5)
26
>>> f(5)
27
```

Return values are different when called with the same input

Example - Withdraw

Let's model a bank account that has a balance of $100

```
>>> withdraw(25)
75
```

Argument: amount to withdraw

Return value: remaining balance

```
>>> withdraw(25)
50
```

Second withdrawal of the same amount

Different return value!

```
>>> withdraw(60)
'Insufficient funds'
```

Where's this balance stored?

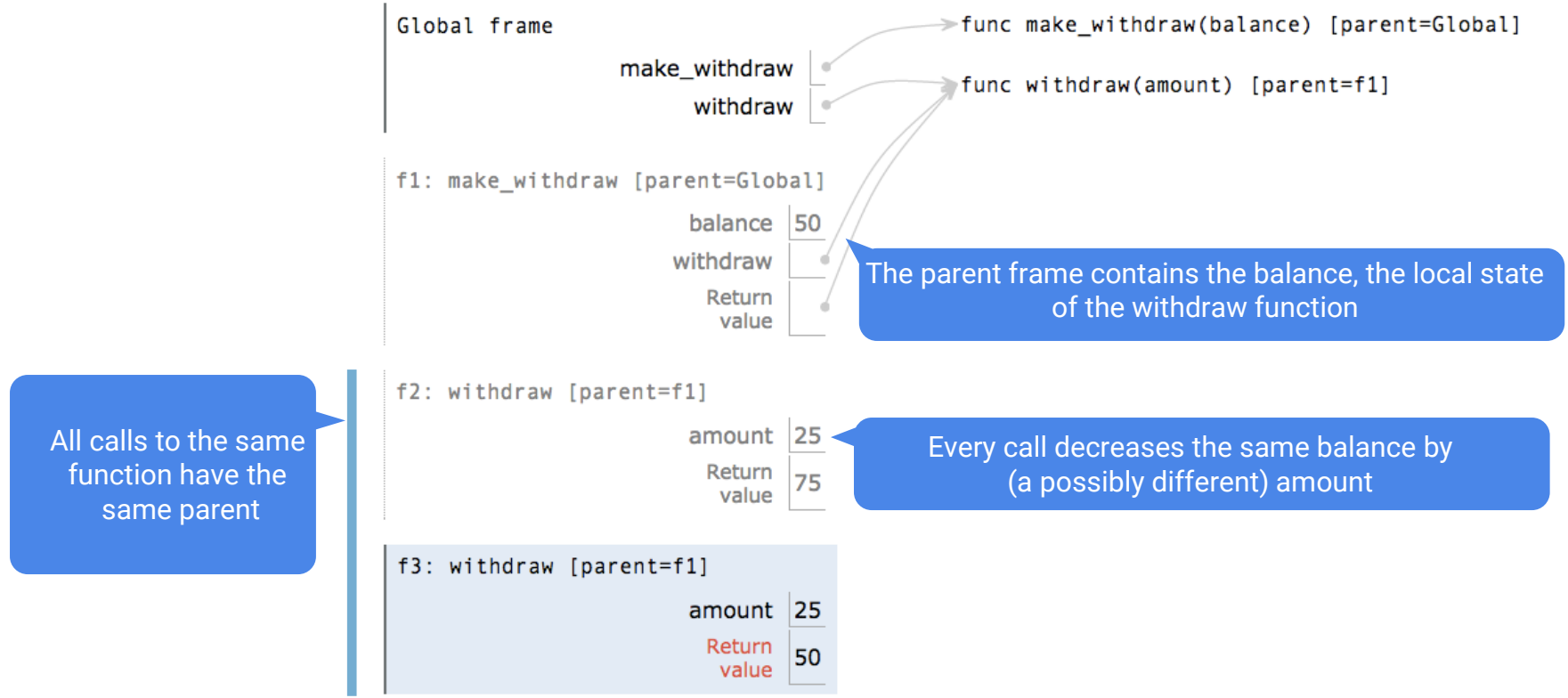
Within the parent frame of the function!

```
>>> withdraw(15)
35
```

A function has a body and a parent environment

```
>>> withdraw =
```

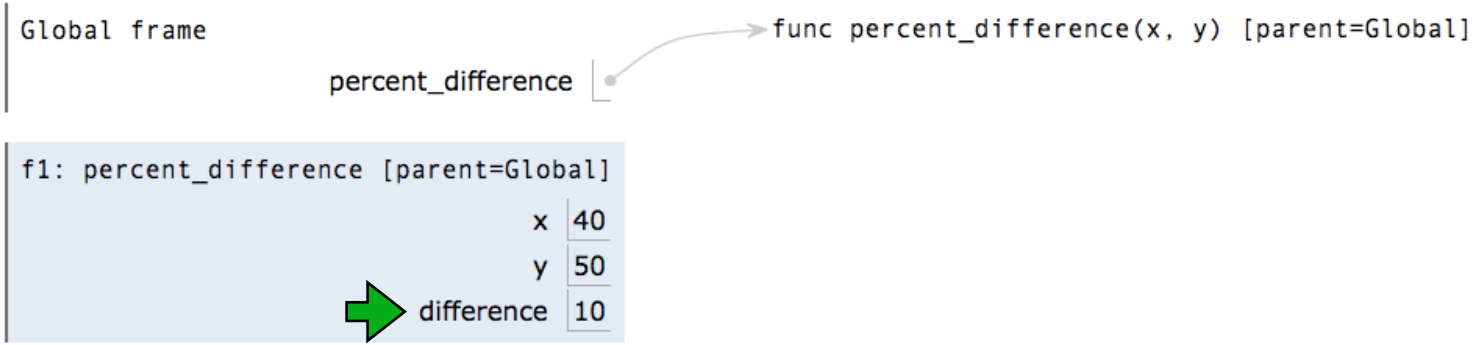
Persistent Local State Using Environments



Reminder: Local Assignment

```
def percent_difference(x, y):
    difference = abs(x-y)
    return 100 * difference / x
diff = percent_difference(40, 50)
```

Assignment binds name(s) to value(s) in the first frame of the current environment



Execution rule for assignment statements:

1. Evaluate all expressions right of =, from left to right

2. Bind the names on the left to the resulting values in the **current frame**

Non-Local Assignment & Persistent Local State

Demo

```
def make_withdraw(balance):
    """Return a withdraw function with a starting balance."""
    def withdraw(amount):
        nonlocal balance
        if amount > balance:
            return 'Insufficient funds'
        balance = balance - amount
        return balance
    return withdraw
```

Declare the name "balance" nonlocal at the top of the body of the function in which it is re-assigned

Re-bind balance in the first non-local frame in which it was bound previously

Mutable Functions

The Effect of Nonlocal Statements

nonlocal <name> , <name>, ...

Effect: Future assignments to that name change its pre-existing binding in the **first non-local frame** of the current environment in which that name is bound.

Python Docs: an "enclosing scope"

From the Python 3 language reference:

Names listed in a nonlocal statement must refer to pre-existing bindings in an enclosing scope.

Names listed in a nonlocal statement must not collide with pre-existing bindings in the local scope.

Current frame

http://docs.python.org/release/3.1.3/reference/simple_stmts.html#the-nonlocal-statement

http://www.python.org/dev/peps/pep-3104/

The Many Meanings of Assignment Statements

`x = 2`

Status	Effect
• No nonlocal statement • "x" **is not** bound locally	Create a new binding from name "x" to object 2 in the first frame of the current environment
• No nonlocal statement • "x" **is** bound locally	Re-bind name "x" to object 2 in the first frame of the current environment
• nonlocal x • "x" **is** bound in a non-local frame	Re-bind "x" to 2 in the first non-local frame of the current environment in which "x" is bound
• nonlocal x • "x" **is not** bound in a non-local frame	SyntaxError: no binding for nonlocal 'x' found
• nonlocal x • "x" **is** bound in a non-local frame • "x" also bound locally	SyntaxError: name 'x' is parameter and nonlocal

Python Particulars

Demo

Python pre-computes which frame contains each name before executing the body of a function.

Within the body of a function, all instances of a name must refer to the same frame.

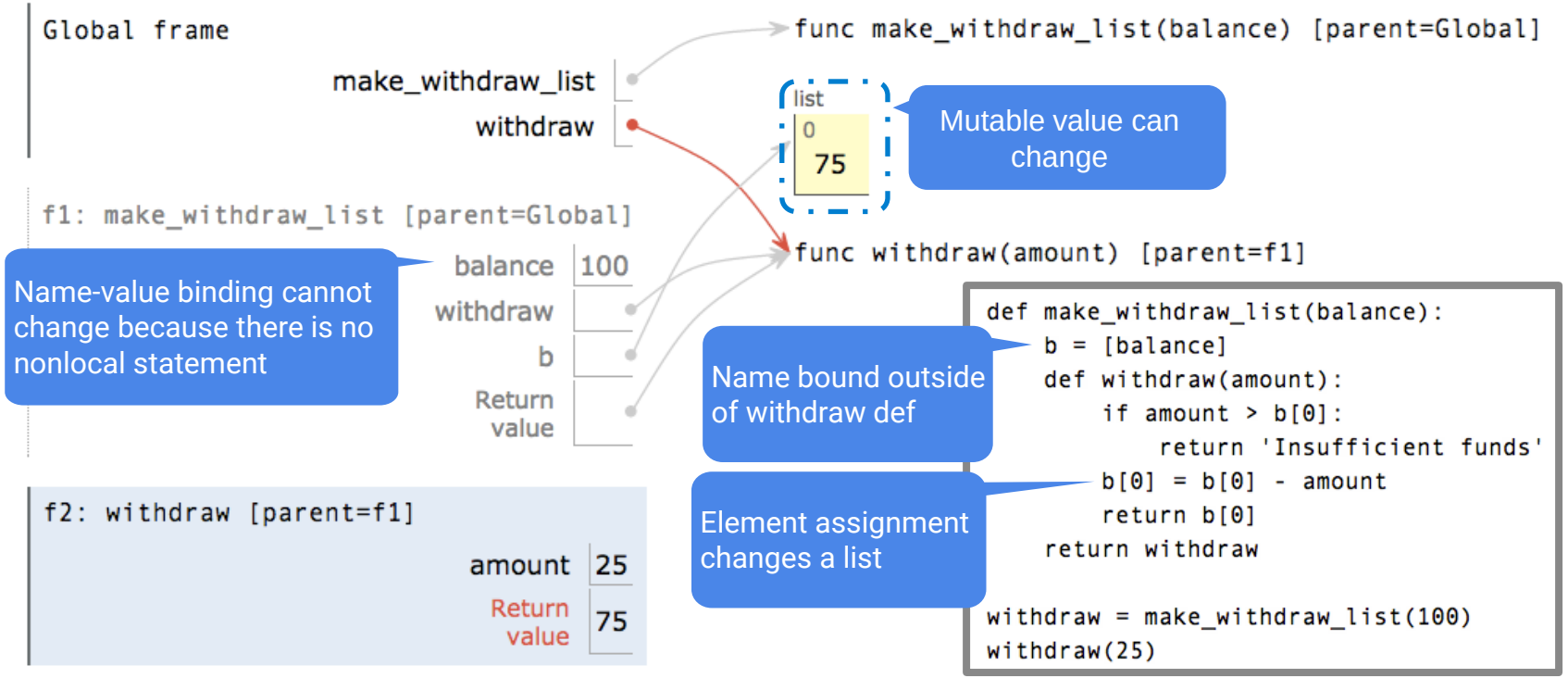
```
def make_withdraw(balance):
    def withdraw(amount):
        if amount > balance:
            return 'Insufficient funds'
        balance = balance - amount
        return balance
    return withdraw

wd = make_withdraw(20)
wd(5)
```

Local assignment

UnboundLocalError: local variable 'balance' referenced before assignment

Mutable Values & Persistent Local State

Mutable values can be changed *without* a nonlocal statement.



Multiple Mutable Functions

Demo

Referential Transparency, Lost

- Expressions are **referentially transparent** if substituting an expression with its value does not change the meaning of a program.

mul(add(2, mul(4, 6)), add(3, 5))

mul(add(2, 24), add(3, 5))

mul(26 , add(3, 5))

Mutation operations violate the condition of referential transparency because they do more than just return a value; **they change the environment**.

```
def f(x):
    x = 4
    def g(y):
        def h(z):
            nonlocal x
            x = x + 1
            return x + y + z
        return h
    return g
a = f(1)
b = a(2)
total = b(3) + b(4)
```

Summary (AKA what do I need to know for the MT)

- Nonlocal allows you to modify a binding in a parent frame, instead of just looking it up
- Don’t need a nonlocal statement to mutate a value
- A variable declared nonlocal must:
  - Exist in a parent frame (other than the global frame)
  - Not exist in the current frame

Program Performance

Measuring Performance

Demo

- Different functions run in different amounts of time
  - Different implementations of the same program can also run in different amounts of time
- How do we measure this?
  - Amount of time taken to run once?
  - Average time taken to run?
  - Average across a bunch of different computers?
  - Number of operations?

Demo

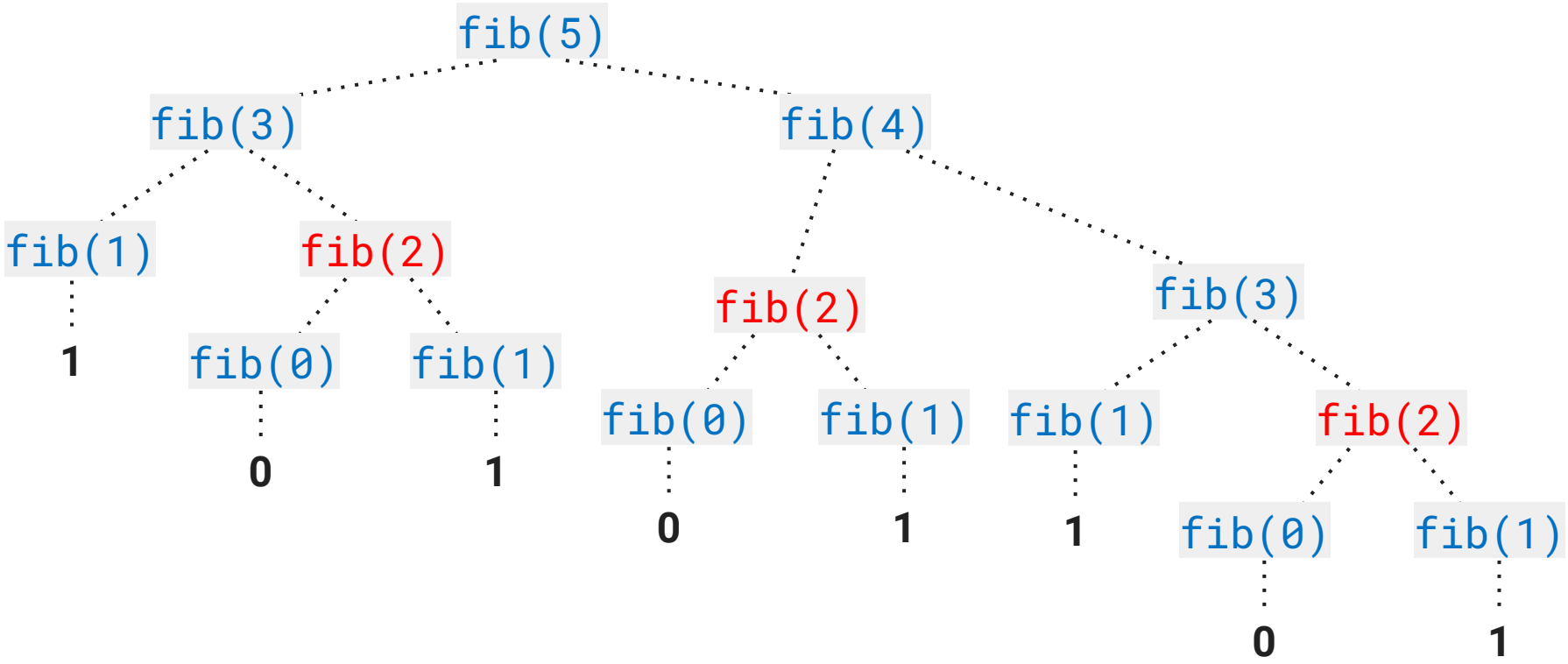
Improving Number of Operations

- Getting better performance usually requires clever thought
  - Some tools can be used to speed up programs (Ex: memoization, up next)
  - Other times, need to have a different approach or incorporate some insight

```
def fib(n):
    curr, next, i = 0, 1, 0
    while i < n:
        curr, next = next,
                     curr + next
                i += 1
    return curr
```

```
def fib(n):
    if n <= 1:
        return n
    else:
        return fib(n-1) +\
               fib(n-2)
```

Improving Fib with Memoization



Basic Idea to Improve:

```
def better_fib(n):
        if n == 0:
                return 0
        elif n == 1:
                return 1
        elif already called better_fib(n
                return stored value
        else:
                store & return better_fib(n - 2) + better_fib(n -
1)
```

Use a container to store these values!

Counting

How to count calls

Can’t always rely on having a program to count calls

- For an iterative function: step through the program, and identify how many times you need to go through the loop before exiting
- For a recursive function: draw out an environment diagram/call tree

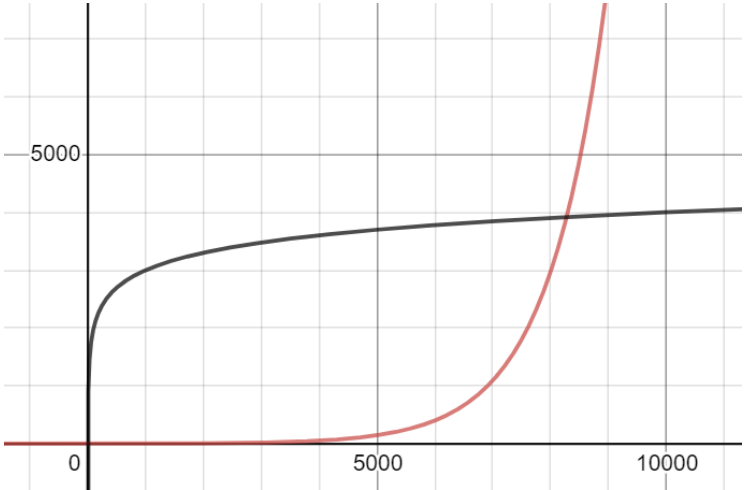
After you do this for a few example inputs, you can find patterns to estimate the number of steps for other inputs.

“Patterns” of Growth

There are common patterns for how functions grow

Because these patterns often aren’t linear, you often can’t use just one input to compare programs

Instead, you have to identify the overall pattern



Some quick notation

“a”, “b”, and “k” usually refer to constants (do not depend on any particular function call)

“x” and “n” refer to variables dependent on the input to the function (change with each function call)

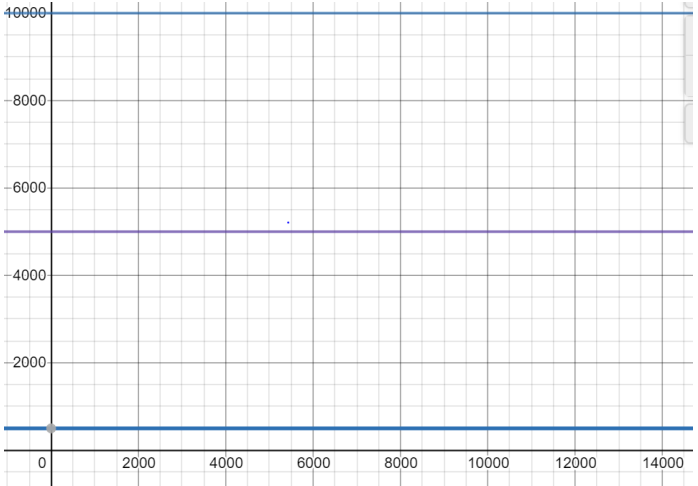
So x^b represents x^2, x^4, etc.

Common Patterns (I)

Constant Growth

operations the same regardless of input size

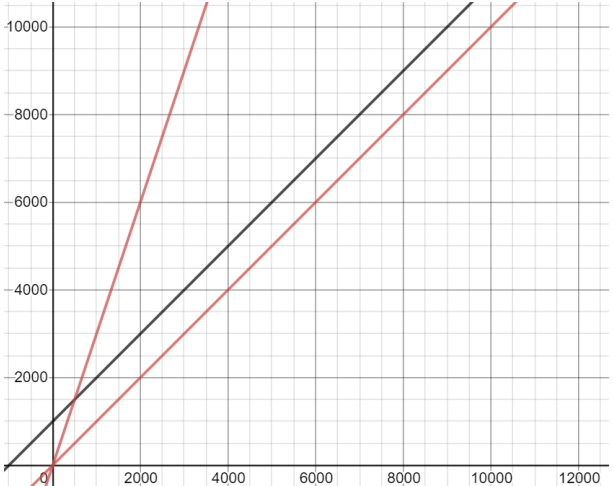
Grows like: k



Linear Growth

Increasing input by 1 adds a constant amount to # of operations

Grows like: ax + b



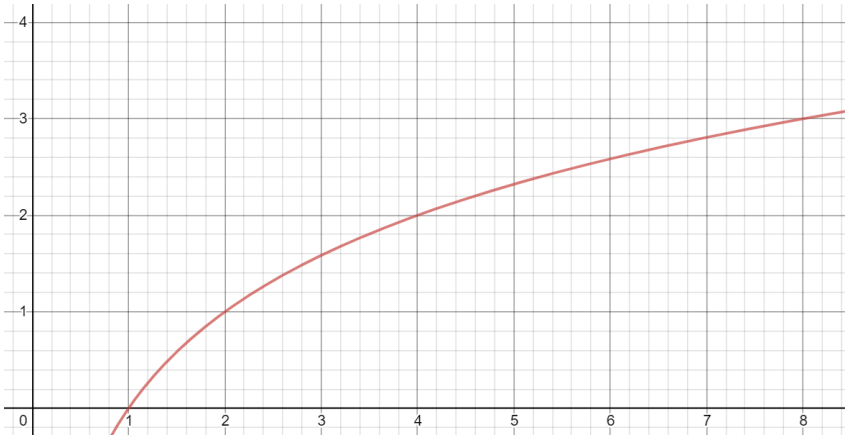
Common Patterns (II)

Demo

Logarithmic Growth

operations grows only when input is multiplied (doubled, tripled, etc.)

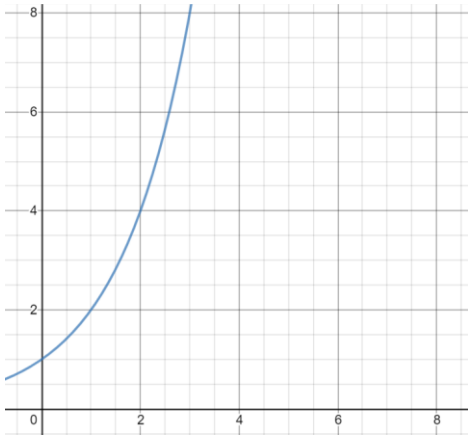
Grows like: $\log_b(x)$



Exponential Growth

Increasing input by 1 doubles (or triples, quadruples, etc.) # operations

Grows like: k^x (2^x, 3^x, etc.)



Exponentiation

Goal: Calculate b to the power n (≥ 0).

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

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

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

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

Goal: Calculate `b` to the power `n` (≥ 0).

Demo

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

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

1) How many total recursive calls do we have to make for:
    - exp_slow(2)
    - exp_slow(4)
    - exp_slow(8)
    - exp_slow(9)
2) How many total recursive calls do we have to make for the same inputs for exp_fast?
3) What pattern of growth do each of these follow?

Summary (AKA what do I need to know)

- There are different methods of programmatically or manually counting the number of operations for a function
  - You should be able to do this for small inputs
- There are 4 main “patterns” of growth which we can apply to sets of data
  - Given the number of operations for a set of inputs, should be able to identify these

What don’t you need to know?

- Formal notation for describing growth
  - Ex. $\Theta(2^n)$
- Identifying growth classes just by looking at code