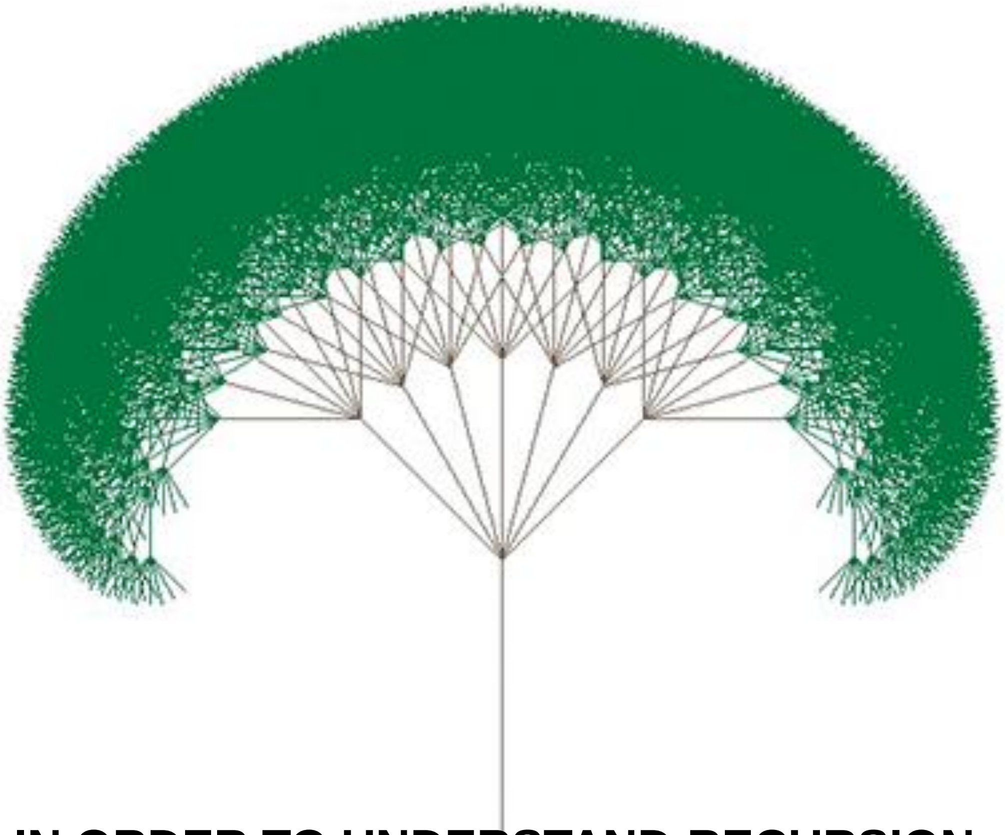
UNIT 5A
Recursion: Introduction



IN ORDER TO UNDERSTAND RECURSION,
ONE SHOULD FIRST UNDERSTAND RECURSION.

Announcements

- First written exam next week Wednesday

- All material from beginning is fair game
  - There are sample exams on the resources page

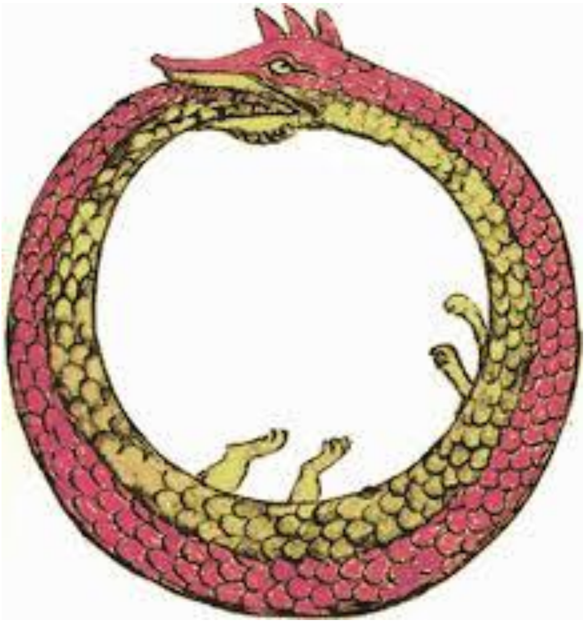
Last time

- Iteration: repetition with variation
- Linear search
- Insertion sort
- A first look at time complexity (measure of efficiency)

This time

- Introduction to recursion
- What it is
- Recursion and the stack
- Recursion and iteration
- Examples of simple recursive functions
- Geometric recursion: fractals

Recursion



The Loopless Loop

Recursion

- A **recursive function** is one that calls itself.

```
def i_am_recursive(x) :
    maybe do some work
    if there is more work to do :
        i_am_recursive(next(x))
    return the desired result
```

- Infinite loop? Not necessarily, not if next(x) needs less work than x.

Recursive Definitions

Every recursive function definition includes two parts:

- Base case(s) (non-recursive)
  One or more simple cases that can be done right away
- Recursive case(s)
  One or more cases that require solving "simpler" version(s) of the original problem.
  - By "simpler", we mean "smaller" or "shorter" or "closer to the base case".

Example: Factorial

- n! = n × (n-1) × (n-2) × ⋯ × 1

  2! = 2 × 1

  3! = 3 × 2 × 1

  4! = 4 × 3 × 2 × 1

- *alternatively:*

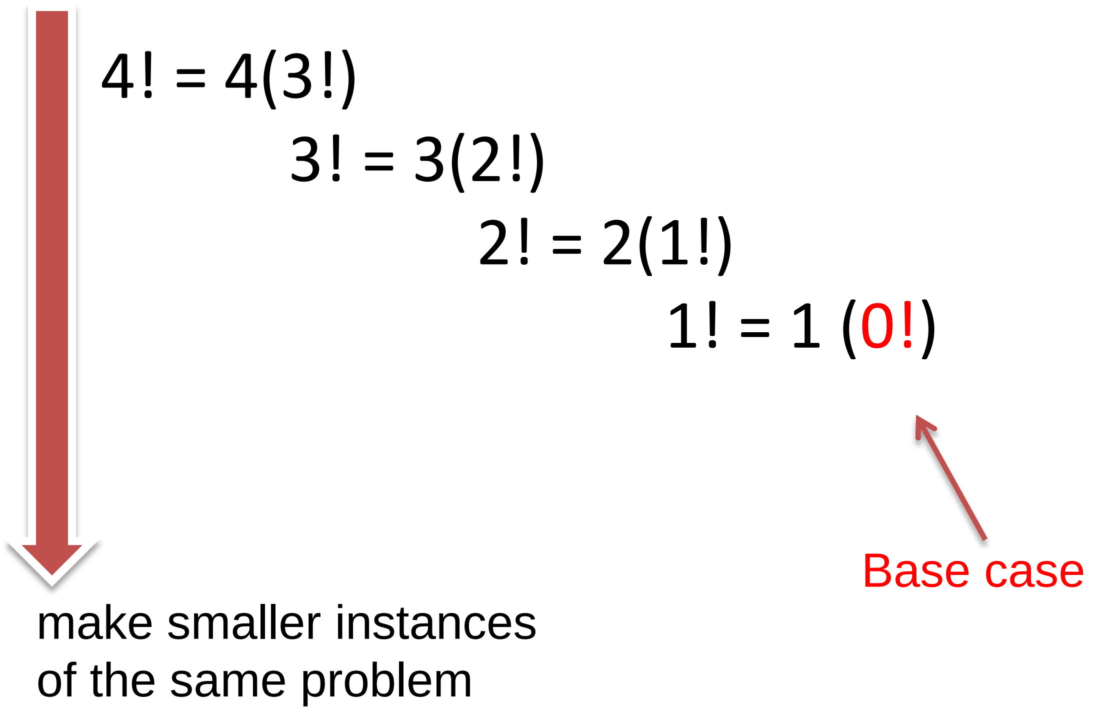
  0! = 1 **(Base case)**

  n! = n × (n-1)! **(Recursive case)**

  So 4! = 4 × 3!

  → 3! = 3 × 2! → 2! = 2 × 1! → 1! = 1 × 0!

Recursion conceptually



$4! = 4(3!)$

$3! = 3(2!)$

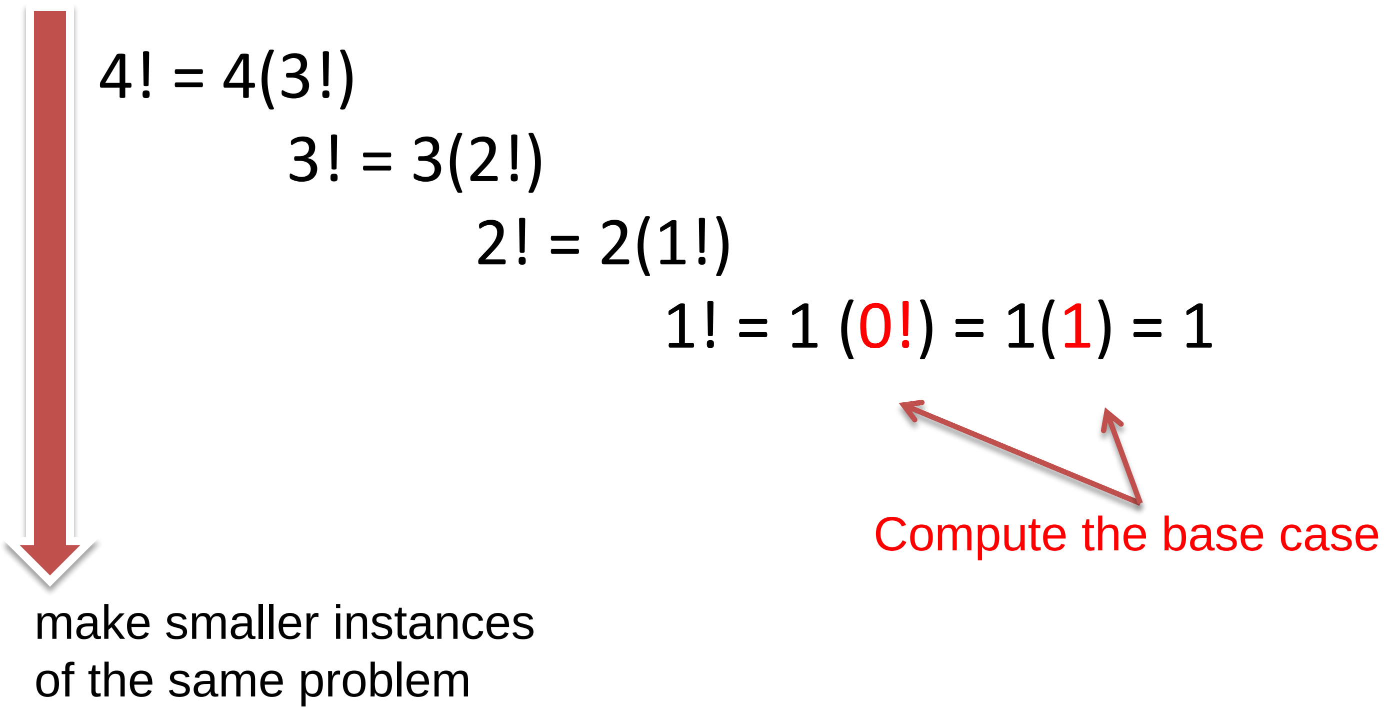
$2! = 2(1!)$

$1! = 1\ (0!)$

Base case

make smaller instances
of the same problem

Recursion conceptually



$4! = 4(3!)$

$3! = 3(2!)$

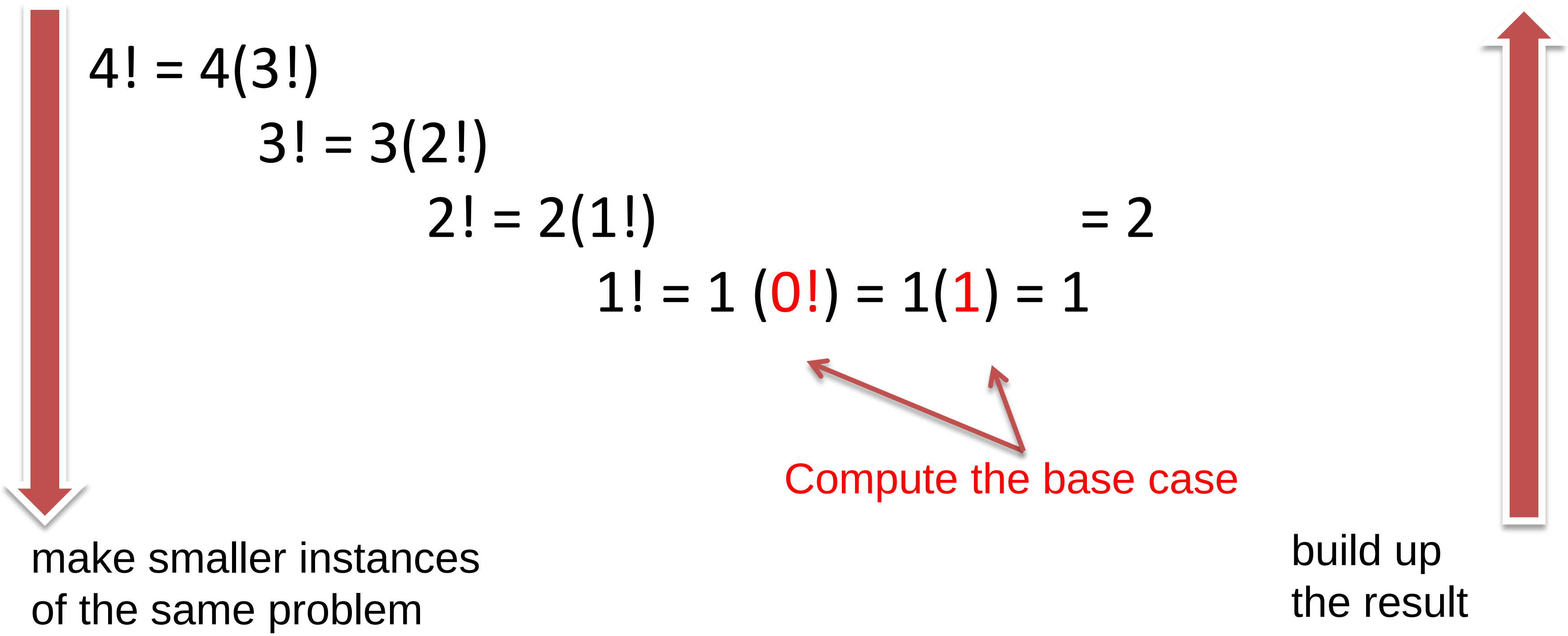
$2! = 2(1!)$

$1! = 1\,(0!) = 1(1) = 1$

Compute the base case

make smaller instances
of the same problem

Recursion conceptually



$4! = 4(3!)$

$3! = 3(2!)$

$2! = 2(1!) \qquad = 2$

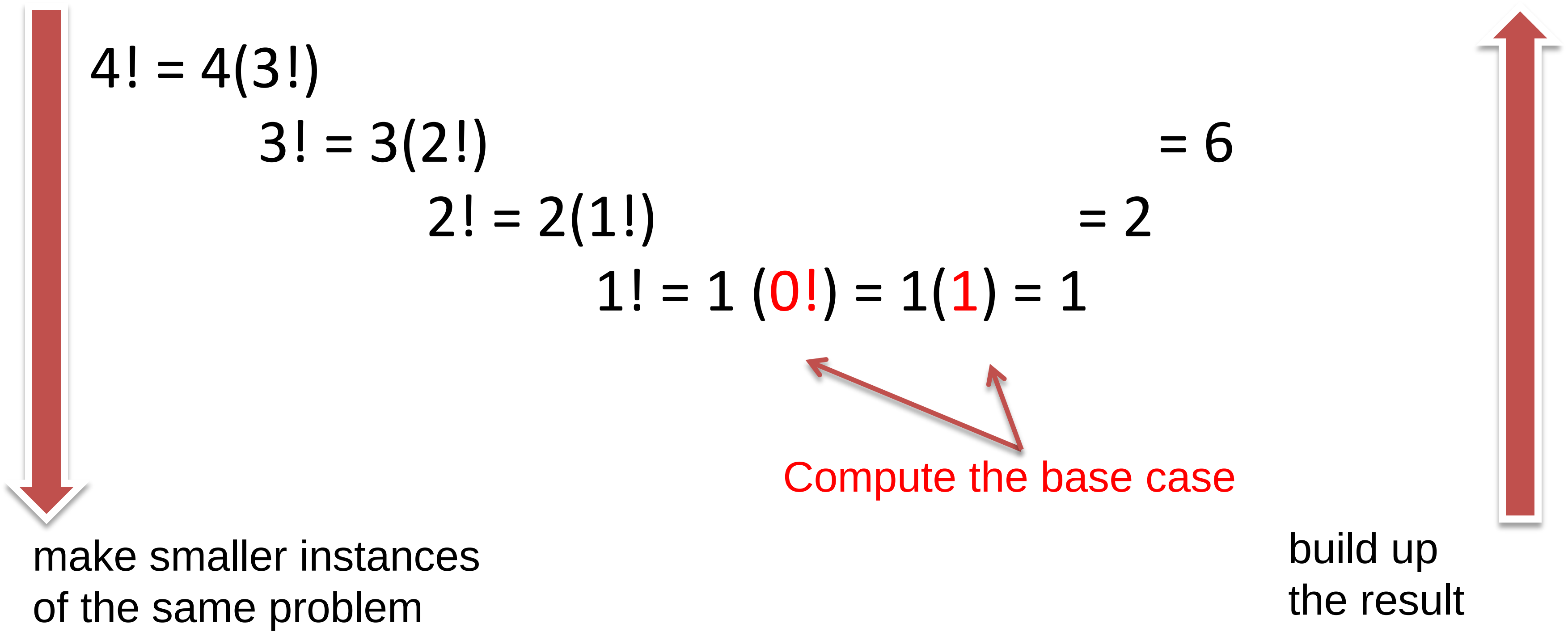
$1! = 1\ (0!) = 1(1) = 1$

Compute the base case

make smaller instances
of the same problem

build up
the result

Recursion conceptually



$4! = 4(3!)$

$3! = 3(2!) \qquad = 6$

$2! = 2(1!) \qquad = 2$

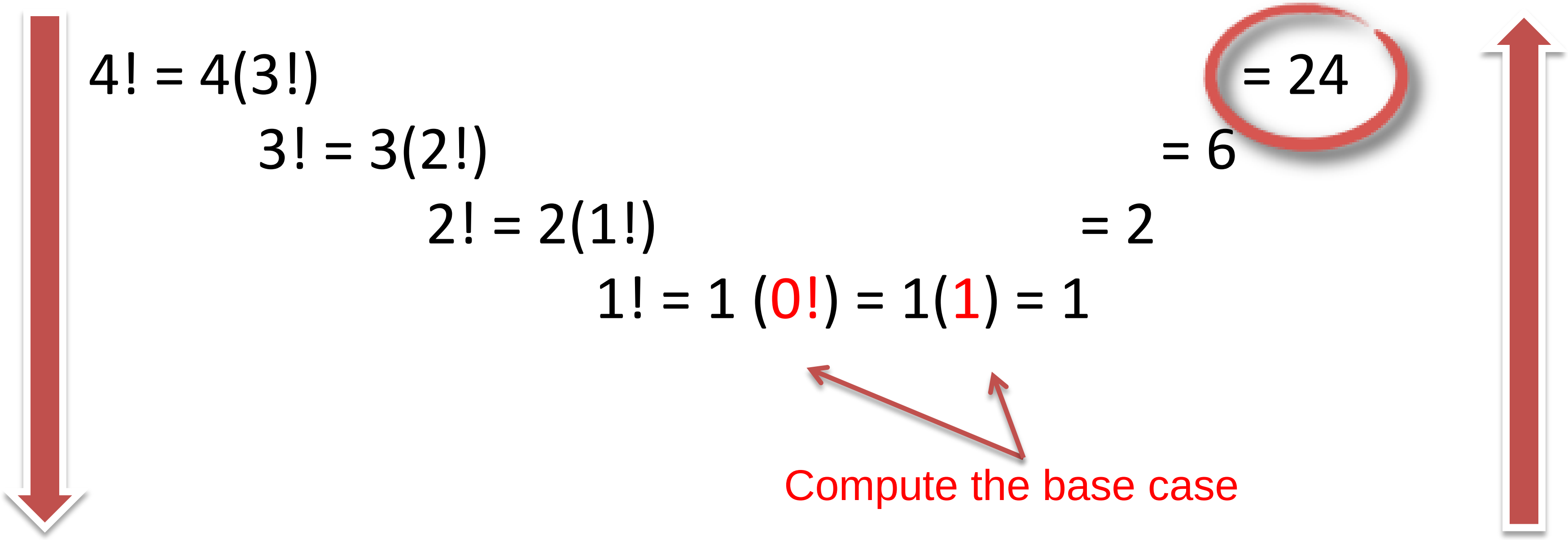
$1! = 1\,(0!) = 1(1) = 1$

Compute the base case

make smaller instances
of the same problem

build up
the result

Recursion conceptually



4! = 4(3!) = 24

3! = 3(2!) = 6

2! = 2(1!) = 2

1! = 1 (0!) = 1(1) = 1

Compute the base case

make smaller instances
of the same problem

build up
the result

Recursive Factorial in Python

```
# 0! = 1                      (Base case)
# n! = n × (n-1)!             (Recursive case)

def factorial(n):
    if n == 0:   # base case
        return 1
    else:        # recursive case
        return n * factorial(n-1)
```

Inside Python Recursion

S

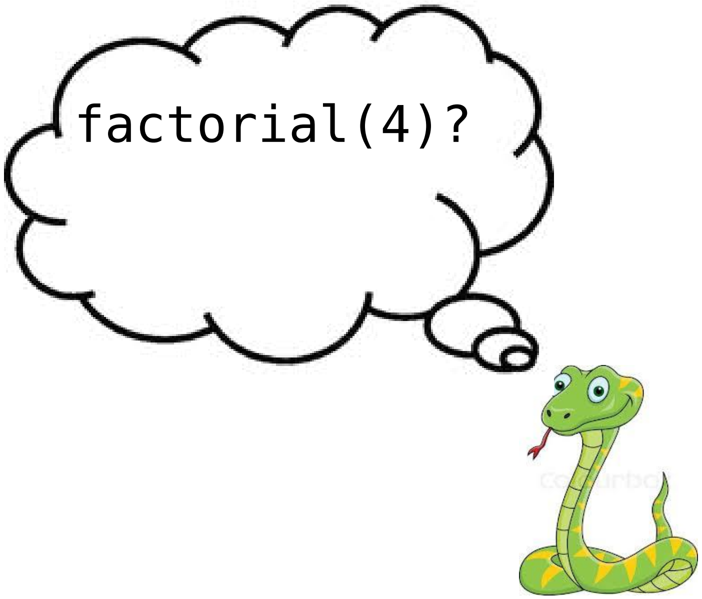
T

A

C

K

n=4



factorial(4)?

Inside Python Recursion

S

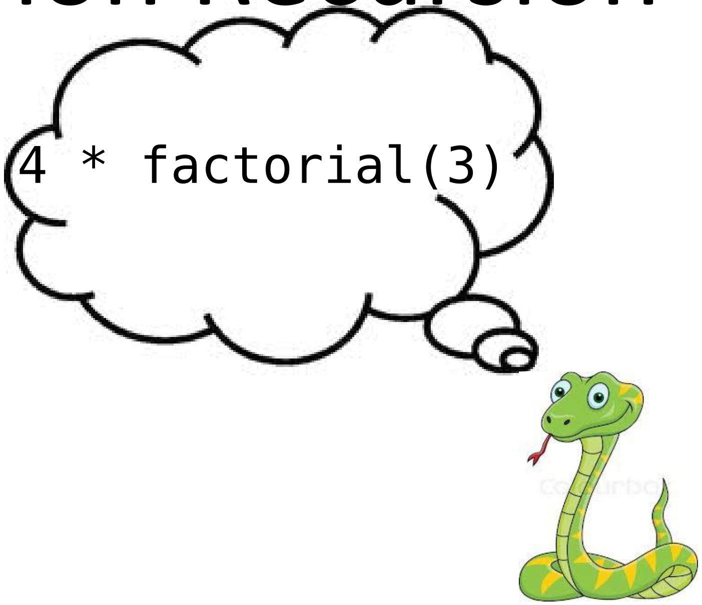
T

A

C

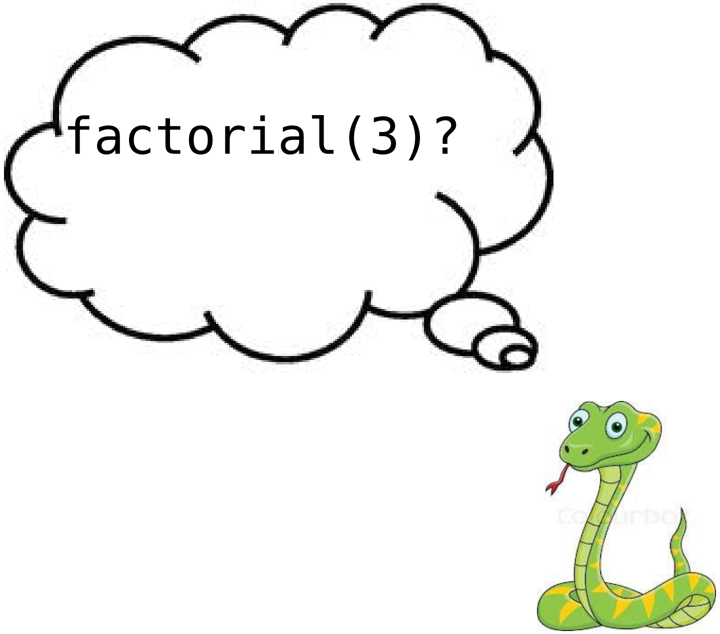
K

```
n=4    factorial(4)? = 4 * factorial(3)
```



Inside Python Recursion

S n=4 factorial(4)? = 4 * factorial(3)



T n=3 factorial(3)?

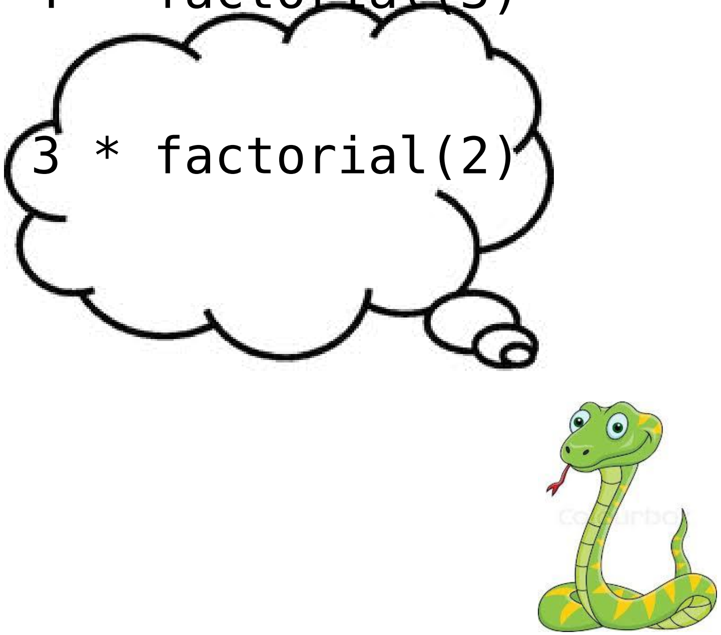
A

C

K

Inside Python Recursion

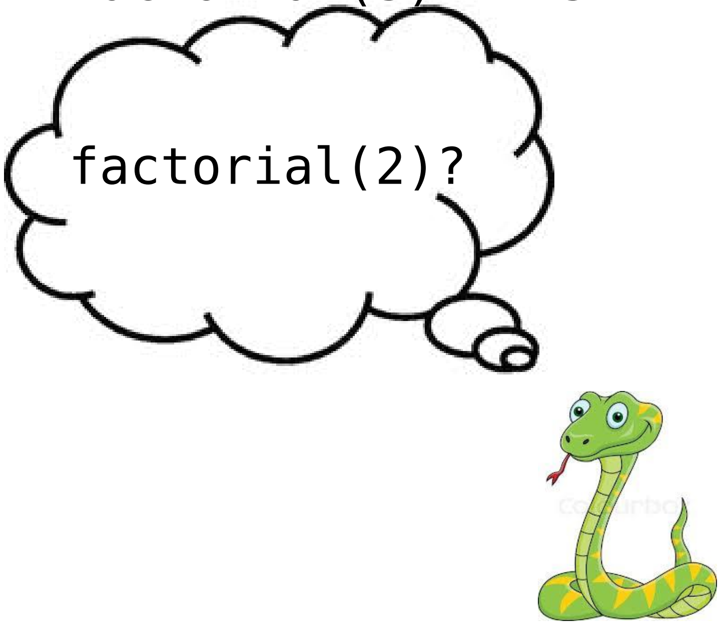
```
S  n=4    factorial(4)? = 4 * factorial(3)

T  n=3    factorial(3)? = 3 * factorial(2)

A

C

K
```



Inside Python Recursion

S n=4 `factorial(4)? = 4 * factorial(3)`

T n=3 `factorial(3)? = 3 * factorial(2)`



A n=2 `factorial(2)?`

C

K

Inside Python Recursion

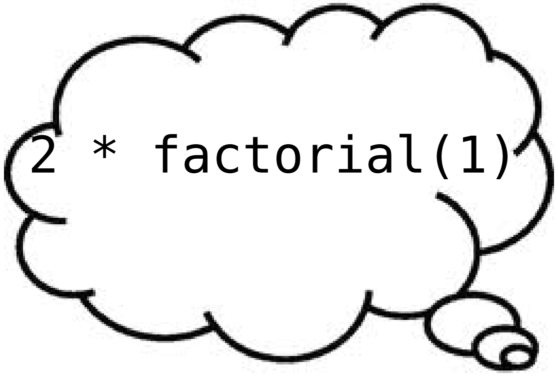
S n=4 `factorial(4)? = 4 * factorial(3)`

T n=3 `factorial(3)? = 3 * factorial(2)`

A n=2 `factorial(2)? = 2 * factorial(1)`

C

K



Inside Python Recursion

S n=4 `factorial(4)? = 4 * factorial(3)`

T n=3 `factorial(3)? = 3 * factorial(2)`

A n=2 `factorial(2)? = 2 * factorial(1)`

C n=1 `factorial(1)?`

K

Inside Python Recursion

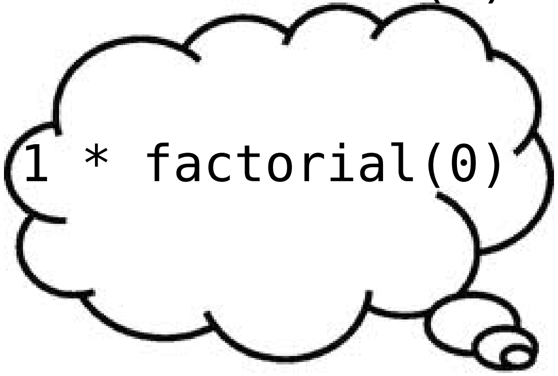
S n=4 factorial(4)? = 4 * factorial(3)

T n=3 factorial(3)? = 3 * factorial(2)

A n=2 factorial(2)? = 2 * factorial(1)

C n=1 factorial(1)? = 1 * factorial(0)

K



Inside Python Recursion

S	n=4	`factorial(4)? = 4 * factorial(3)`
T	n=3	`factorial(3)? = 3 * factorial(2)`
A	n=2	`factorial(2)? = 2 * factorial(1)`
C	n=1	`factorial(1)? = 1 * factorial(0)`
K	n=0	`factorial(0) = 1`

Inside Python Recursion

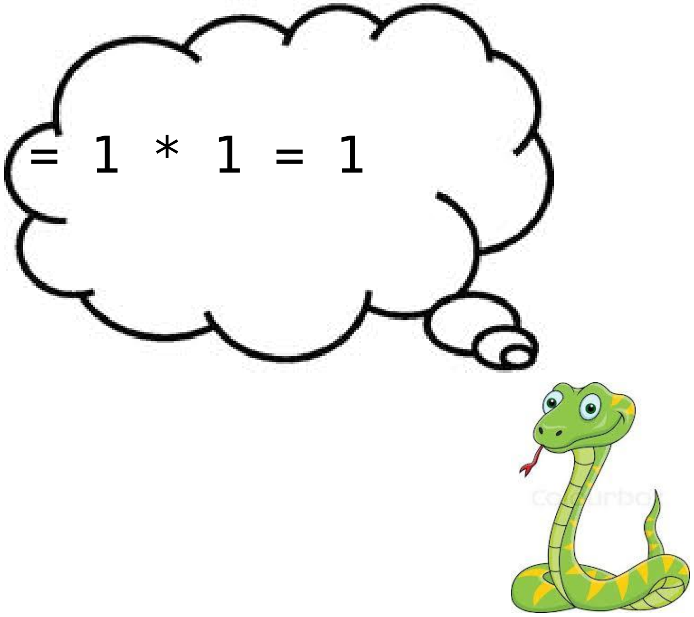
S n=4 `factorial(4)? = 4 * factorial(3)`

T n=3 `factorial(3)? = 3 * factorial(2)`

A n=2 `factorial(2)? = 2 * factorial(1)`

C n=1 `factorial(1) = 1 * 1 = 1`

K



Inside Python Recursion

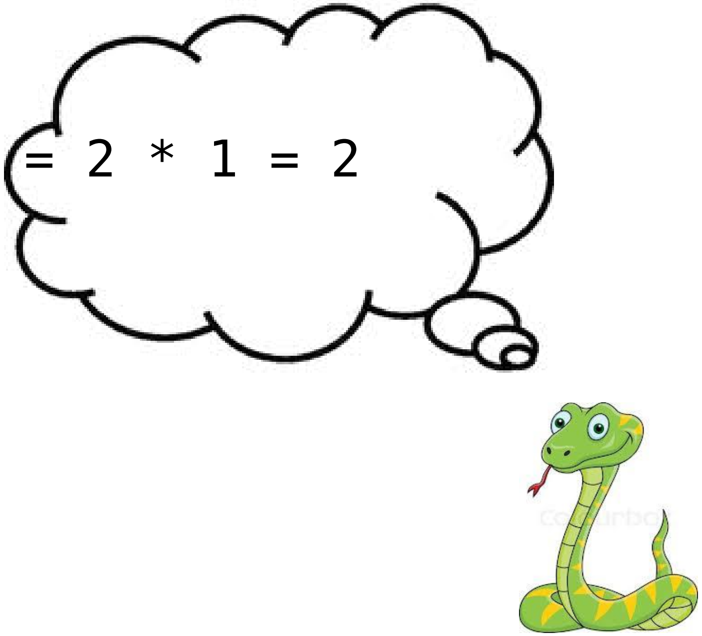
S n=4 `factorial(4)? = 4 * factorial(3)`

T n=3 `factorial(3)? = 3 * factorial(2)`

A n=2 `factorial(2) = 2 * 1 = 2`

C

K



Inside Python Recursion

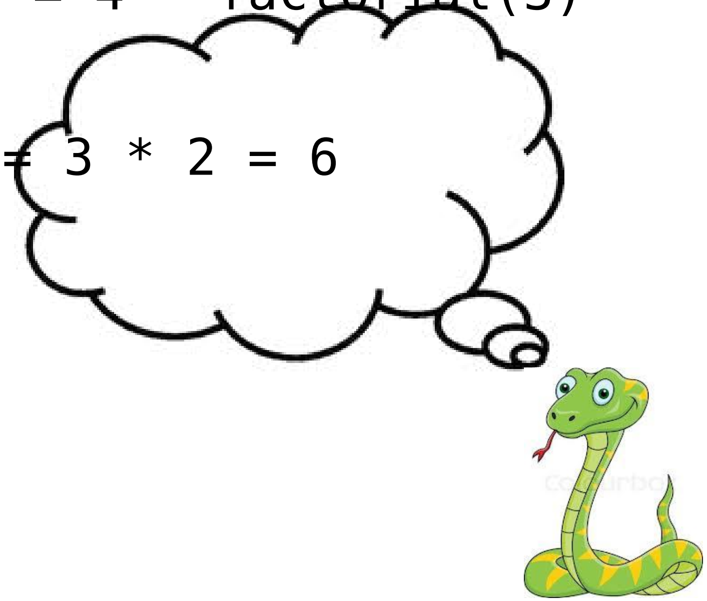
S n=4 factorial(4)? = 4 * factorial(3)

T n=3 factorial(3) = 3 * 2 = 6

A

C

K



Inside Python Recursion

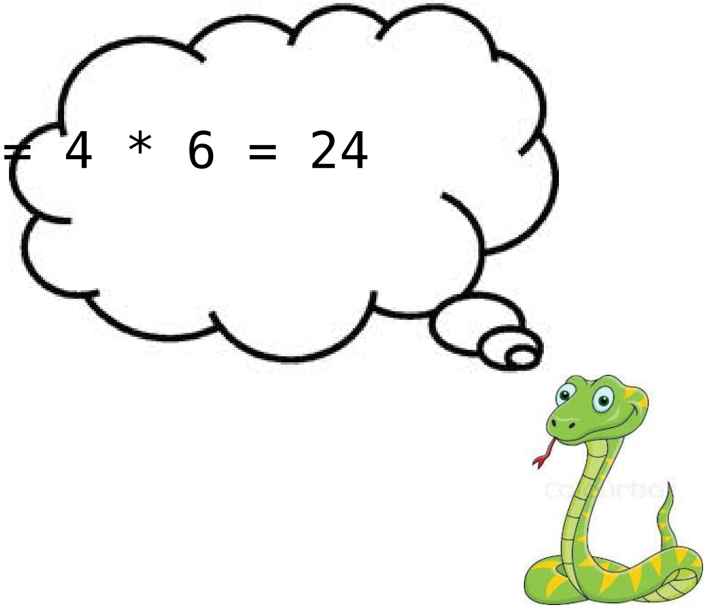
S n=4 `factorial(4) = 4 * 6 = 24`

T

A

C

K



Recursive vs. Iterative Solutions

- **For every recursive function**,
  there is an equivalent iterative solution.
- **For every iterative function**,
  there is an equivalent recursive solution.
- But **some problems** are easier to solve one way than the other way.
- And be aware that **most recursive programs** need space for the stack, behind the scenes

Factorial Function (Iterative)

```
def factorial(n):
    result = 1   # initialize accumulator var
    for i in range(1, n+1):
        result = result * i
    return result
```

Versus (Recursive):

```
def factorial(n):
    if n == 0:     # base case
        return 1
    else:          # recursive case
        return n * factorial(n-1)
```

A Strategy for Recursive Problem Solving (hat tip to Dave Evans)

- Think of the smallest size of the problem and write down the solution (base case)
- **Now assume you magically have a working function to solve any size.** How could you use it on a smaller size and **use the answer** to solve a bigger size? (recursive case)
- Combine the base case and the recursive case

Iteration to Recursion: exercise

- Mathematicians have proved
  $\pi^2/6 = 1 + 1/4 + 1/9 + 1/16 + \ldots$

We can use this to approximate π

Compute the sum, multiply by 6, take the square root

```
def pi_series_iter(n) :
    result = 0
    for i in range(1, n+1) :
        result = result + 1/(i**2)
    return result

def pi_approx_iter(n) :
    x = pi_series_iter(n)
    return (6*x)**(.5)
```

Let's convert this to a recursive function (see file pi_approx.py for a sample solution.)

Recursion on Lists

- First we need a way of getting a smaller input from a larger one:
  - Forming a sub-list of a list:

```
>>> a = [1, 11, 111, 1111, 11111, 111111]
>>> a[1:]
[11, 111, 1111, 11111, 111111]
>>> a[2:]
[111, 1111, 11111, 111111]
>>> a[3:]
[1111, 11111, 111111]
>>> a[3:5]
[1111, 11111]
>>>
```

the "tail" of list a

Recursive sum of a list

```
def sumlist(items):
    if items == []:
```

The smallest size list is the empty list.

Recursive sum of a list

```
def sumlist(items):
    if items == []:
        return 0
```

Base case:
The sum of an empty list is 0.

Recursive sum of a list

```
def sumlist(items):
    if items == []:
        return 0
    else:
```

Recursive case:
the list is not empty

Recursive sum of a list

```
def sumlist(items):
    if items == []:
        return 0
    else:
        ...sumlist(items[1:])...
```

What if **we already know** the sum of the list's tail?

Recursive sum of a list

```
def sumlist(items):
    if items == []:
        return 0
    else:
        return items[0] + sumlist(items[1:])
```

What if **we already know** the sum of the list's tail? We can just add the list's first element!

Tracing sumlist

```
def sumlist(items):
    if items== []:
        return 0
    else:
        return items[0] + sumlist(items[1:])
```

```
>>> sumlist([2,5,7])
sumlist([2,5,7]) = 2 + sumlist([5,7])
                       5 + sumlist([7])
                           7 + sumlist([])
                                   0
```

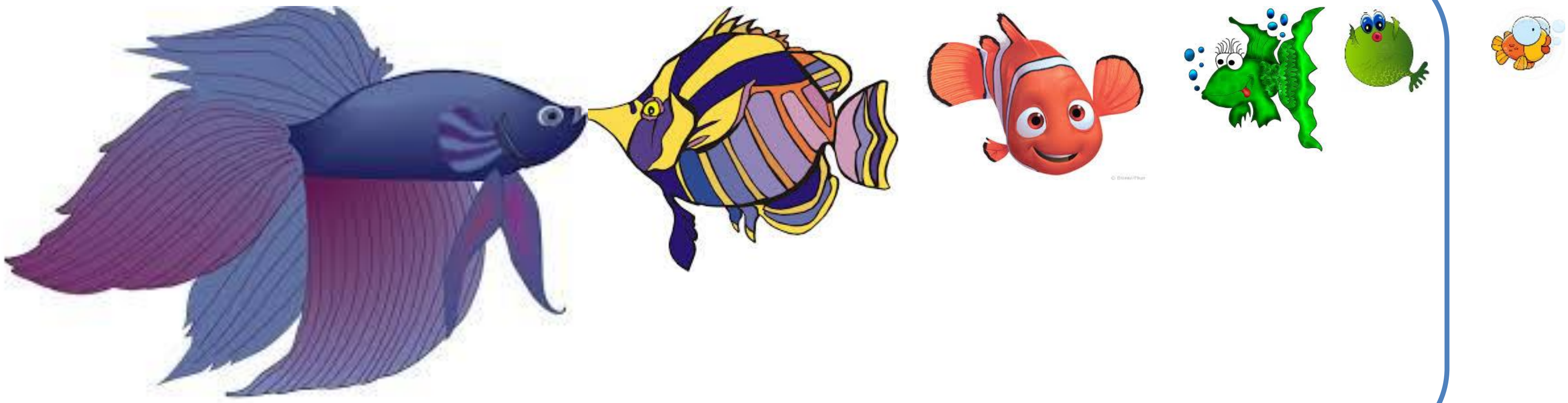
After reaching the base case, the final result is built up by the computer by adding 0+7+5+2.

List Recursion: exercise

- Let's create a recursive function rev(items)
- **Input:** a list of items
- **Output:** another list, with all the same items, but in reverse order
- **Remember:** it's usually sensible to break the list down into its *head* (first element) and its *tail* (all the rest). The tail is a smaller list, and so "closer" to the base case.
- Soooo… (picture on next slide)

Reversing a list: recursive case

See file rev_list.py

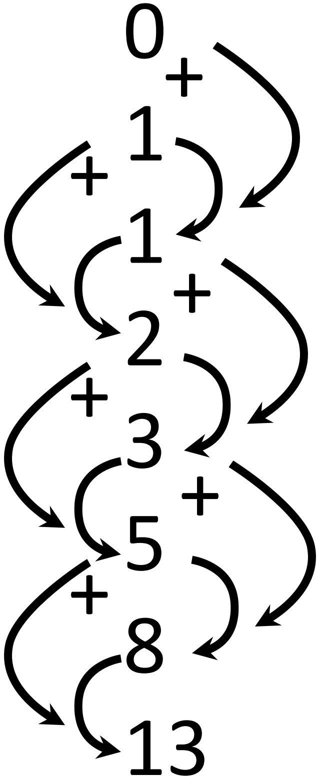


Multiple Recursive Calls

- So far we've used just one recursive call to build up our answer
- The real **conceptual** power of recursion happens when we need more than one!
- Example: Fibonacci numbers

Fibonacci Numbers

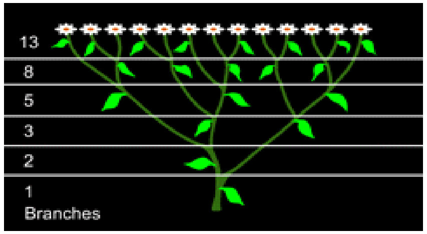
- A sequence of numbers:

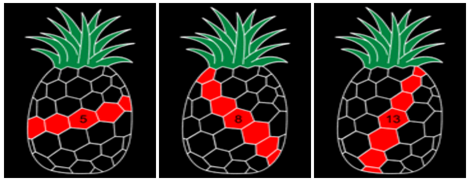


0
1
1
2
3
5
8
13
...

Fibonacci Numbers in Nature

- 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, etc.
- Number of branches on a tree, petals on a flower, spirals on a pineapple.
- Vi Hart's video on Fibonacci numbers (http://www.youtube.com/watch?v=ahXIMUkSXX0)





Recursive Definition

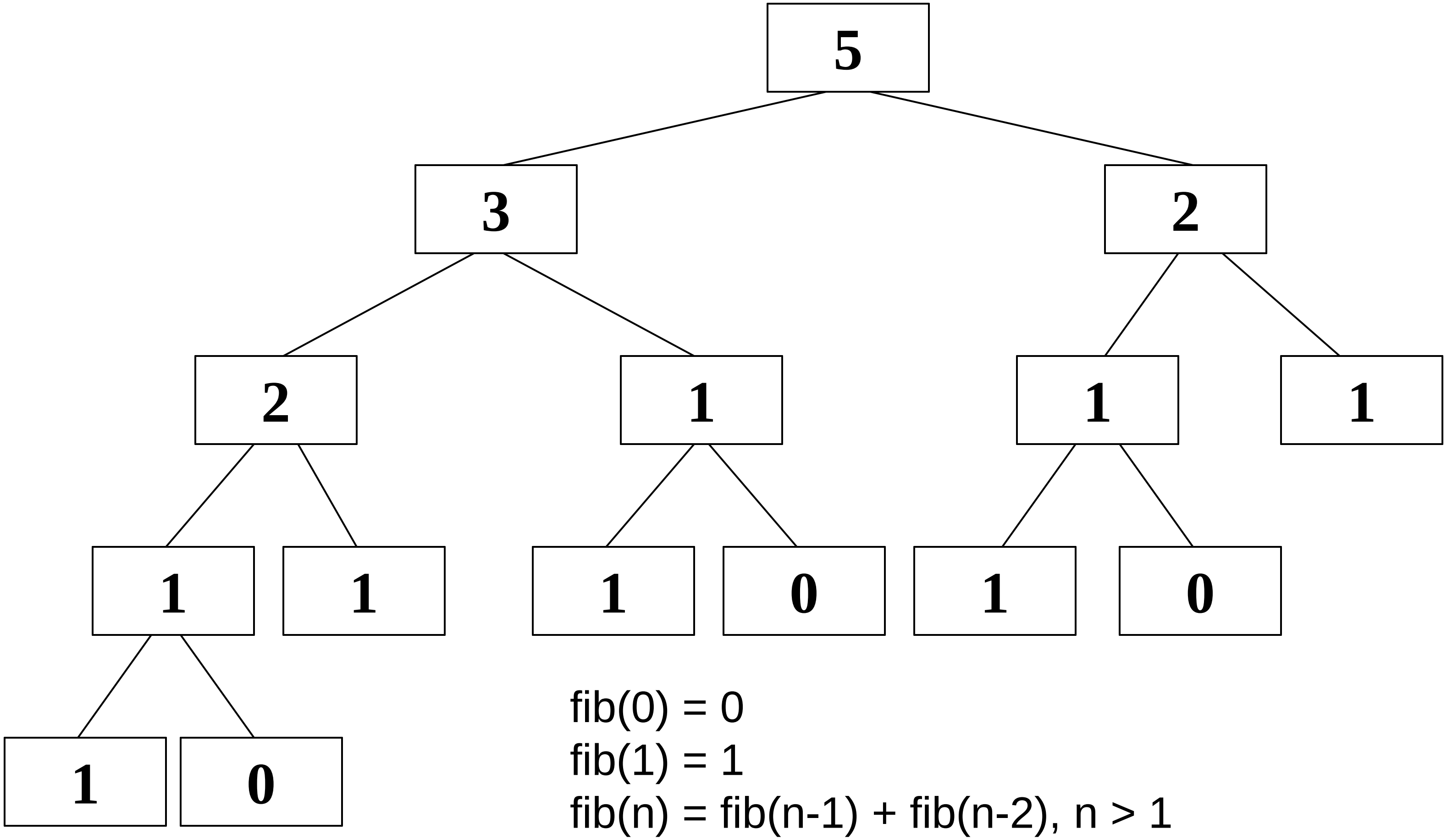
Let fib(n) = the nth Fibonacci number, n ≥ 0

- fib(0) = 0 (base case)
- fib(1) = 1 (base case)
- fib(n) = fib(n-1) + fib(n-2), n > 1

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

Two recursive calls!

Recursive Call Tree



fib(0) = 0
fib(1) = 1
fib(n) = fib(n-1) + fib(n-2), n > 1

Iterative Fibonacci

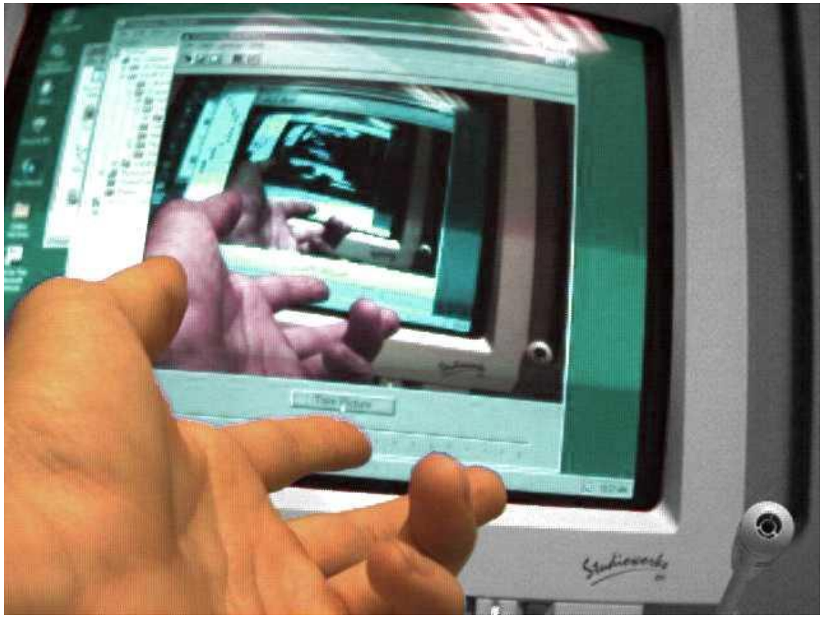
```
def fib(n):
    x = 0
    next_x = 1
    for i in range(1,n+1):
        x, next_x = next_x, x + next_x
    return x
```

simultaneous assignment

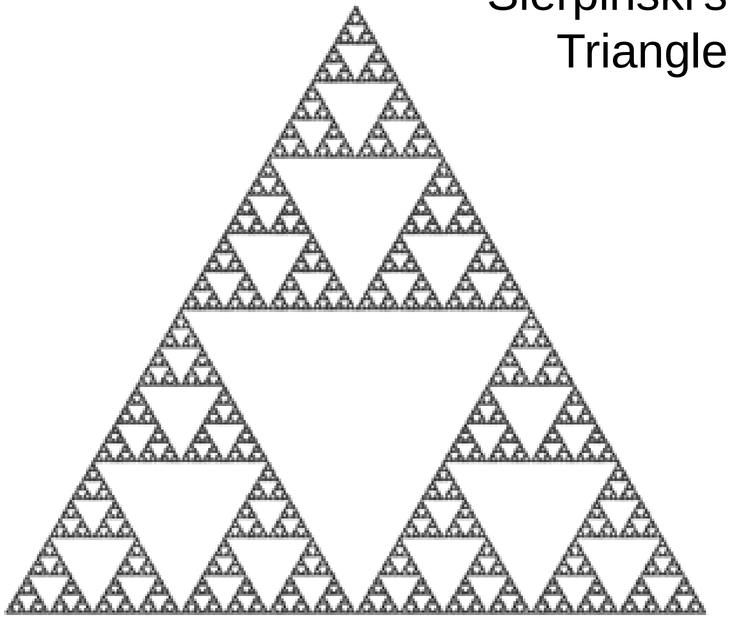
Faster than the recursive version. Why?

Geometric Recursion (Fractals)

- A recursive operation performed on successively smaller regions.

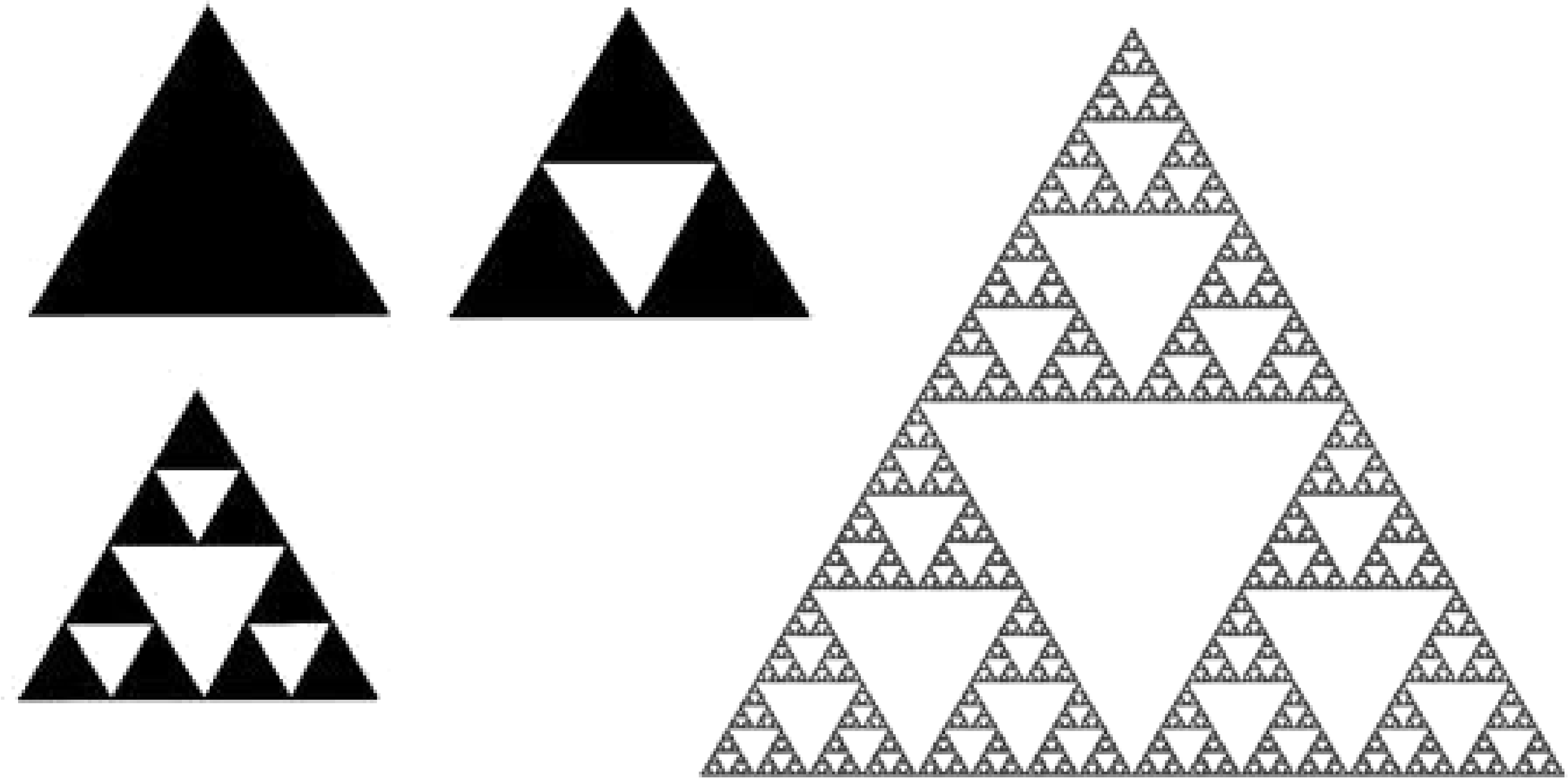


http://fusionanomaly.net/recursion.jpg

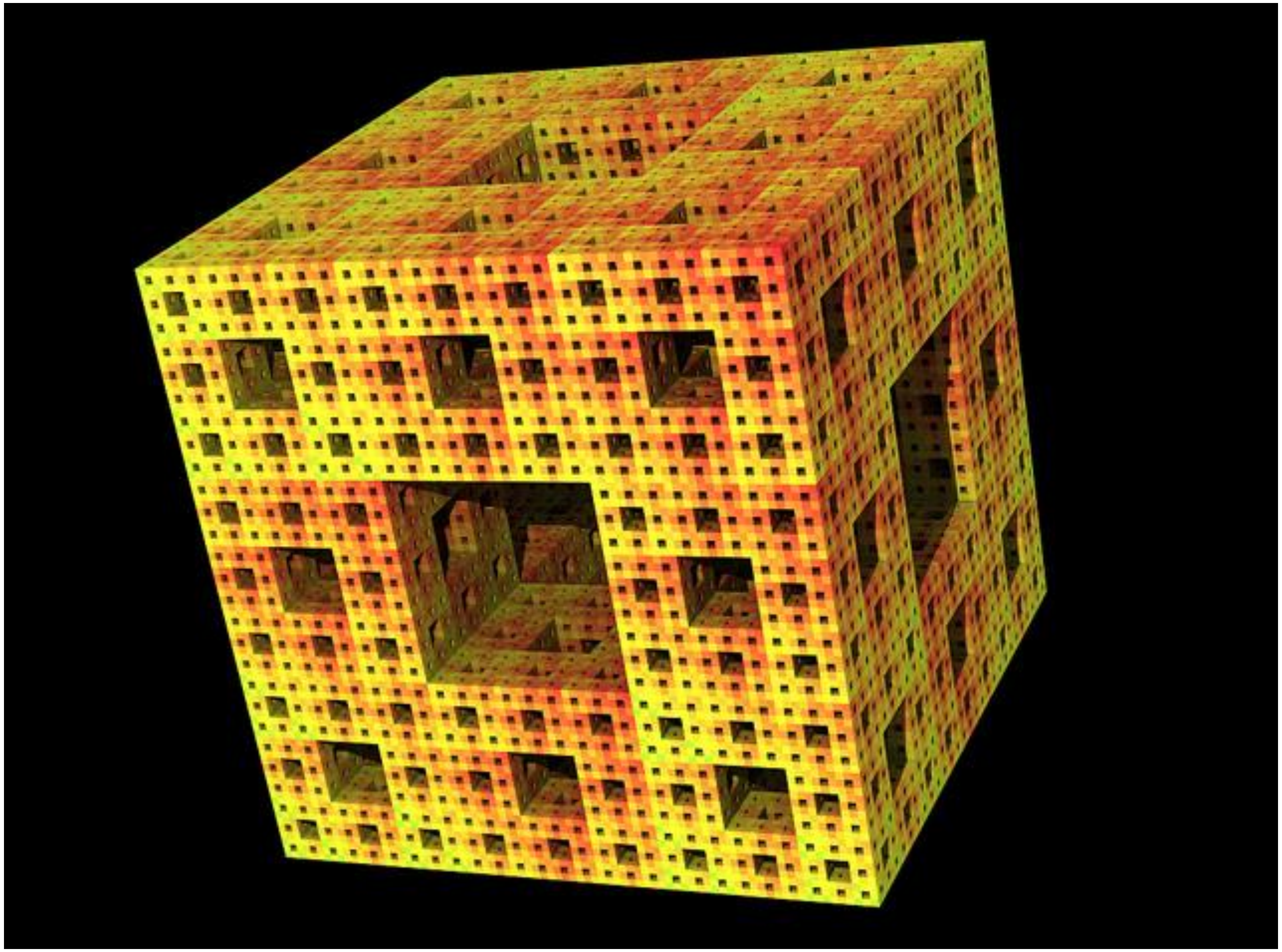


Sierpinski's Triangle

Sierpinski’s Triangle

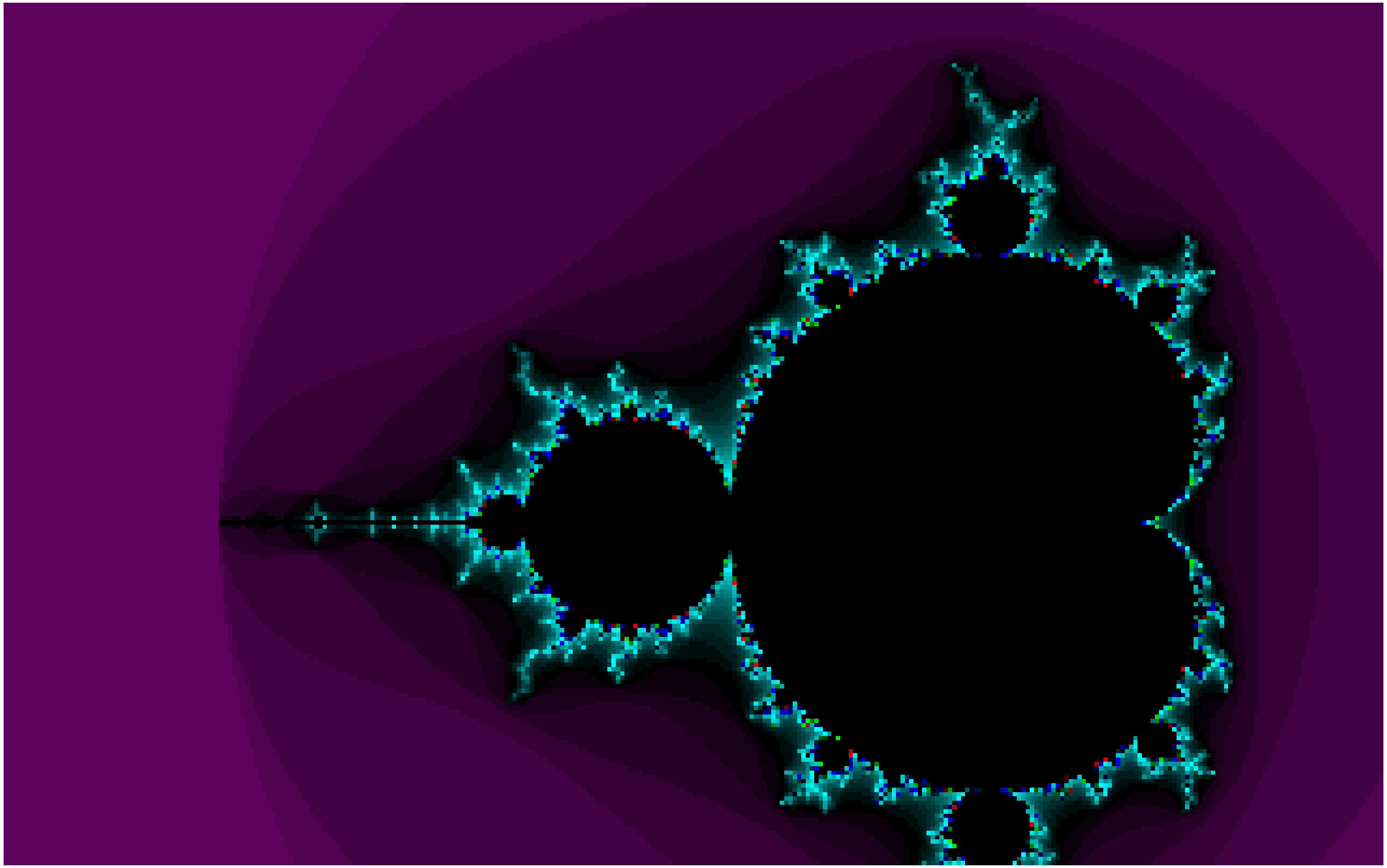


Sierpinski’s Carpet



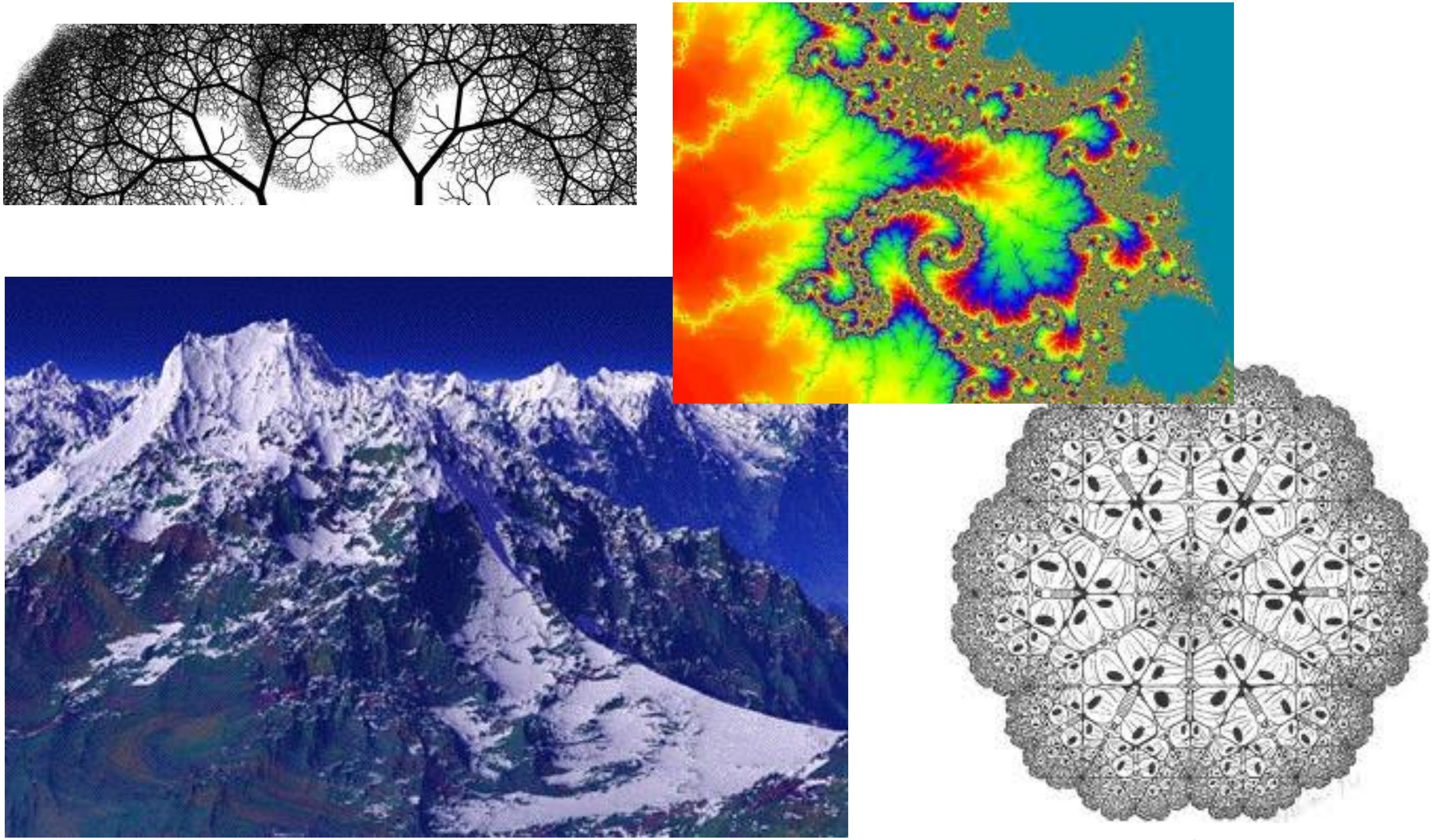
(the next slide shows an animation that could give some people headaches)

Mandelbrot set

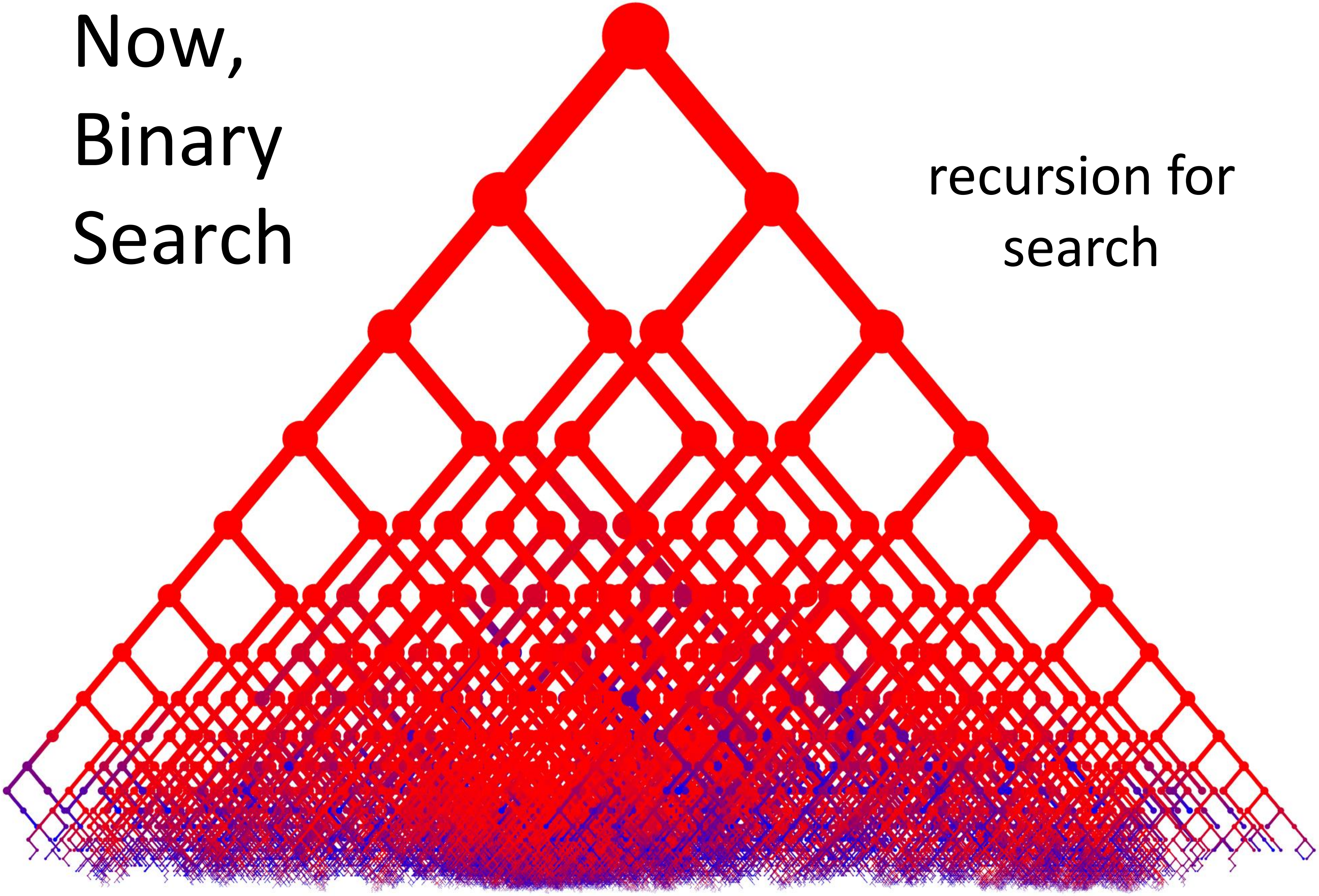


Source: Clint Sprott, http://sprott.physics.wisc.edu/fractals/animated/

Fancier fractals



Now, Binary Search



recursion for search

image: Matt Roberts, http://people.bath.ac.uk/mir20/blogposts/bst_close_up.php