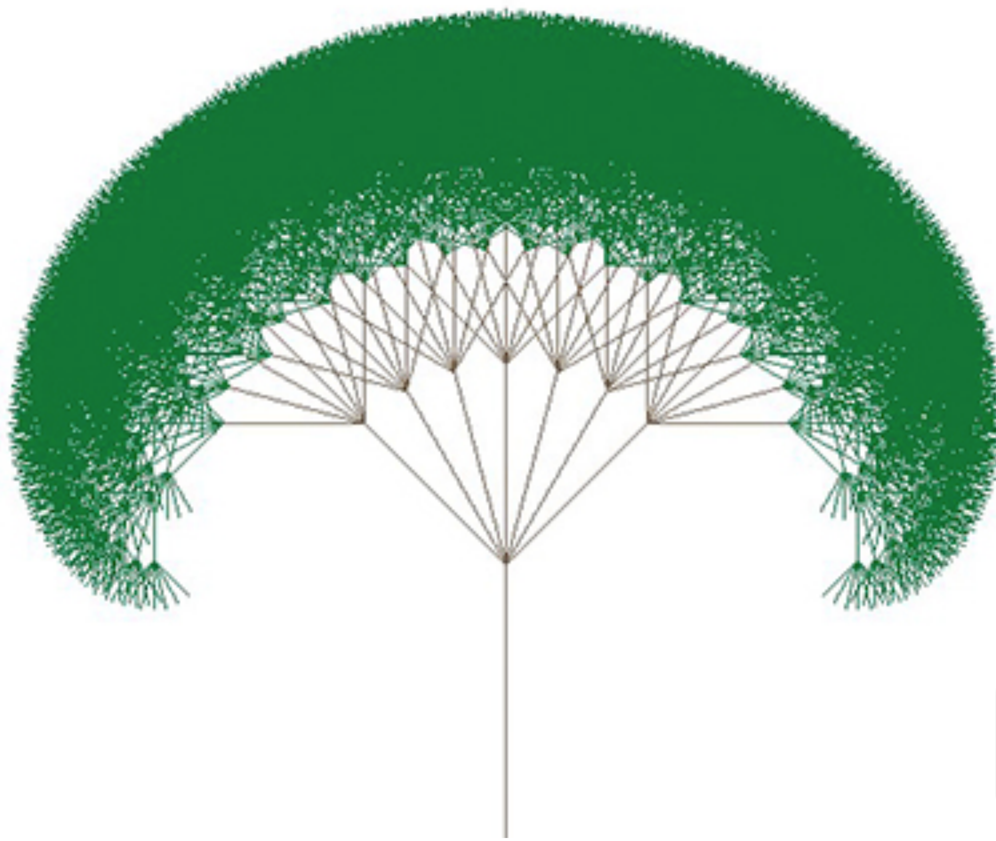
Recursion: Introduction



Announcements

- Deadlines
- Exam on Thursday: Units 1 – 5 (inclusive)
- PA 4 due tonight
- OLI Recursion over the weekend
- Monday: PA5 is due.

Today

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

Big-O Review

Asymptotic Analysis

- Beyond number of operations
- Goal: understanding behavior of program over the long run, with increasingly large inputs
- We are not concerned with constants factors:
  - How many iterations?
  - Not operations in each iteration
- Gives a useful approximation, suppresses details
- Worst-case

Order of Complexity

- We express this as the (time) order of complexity
- Normally expressed using Big-O notation.
- Big-0 is ignores constants, focuses on **highest power of n**

Number of iterations	**Order of Complexity**
n	O(n)
3n+3	O(n)
2n+8	O(n)

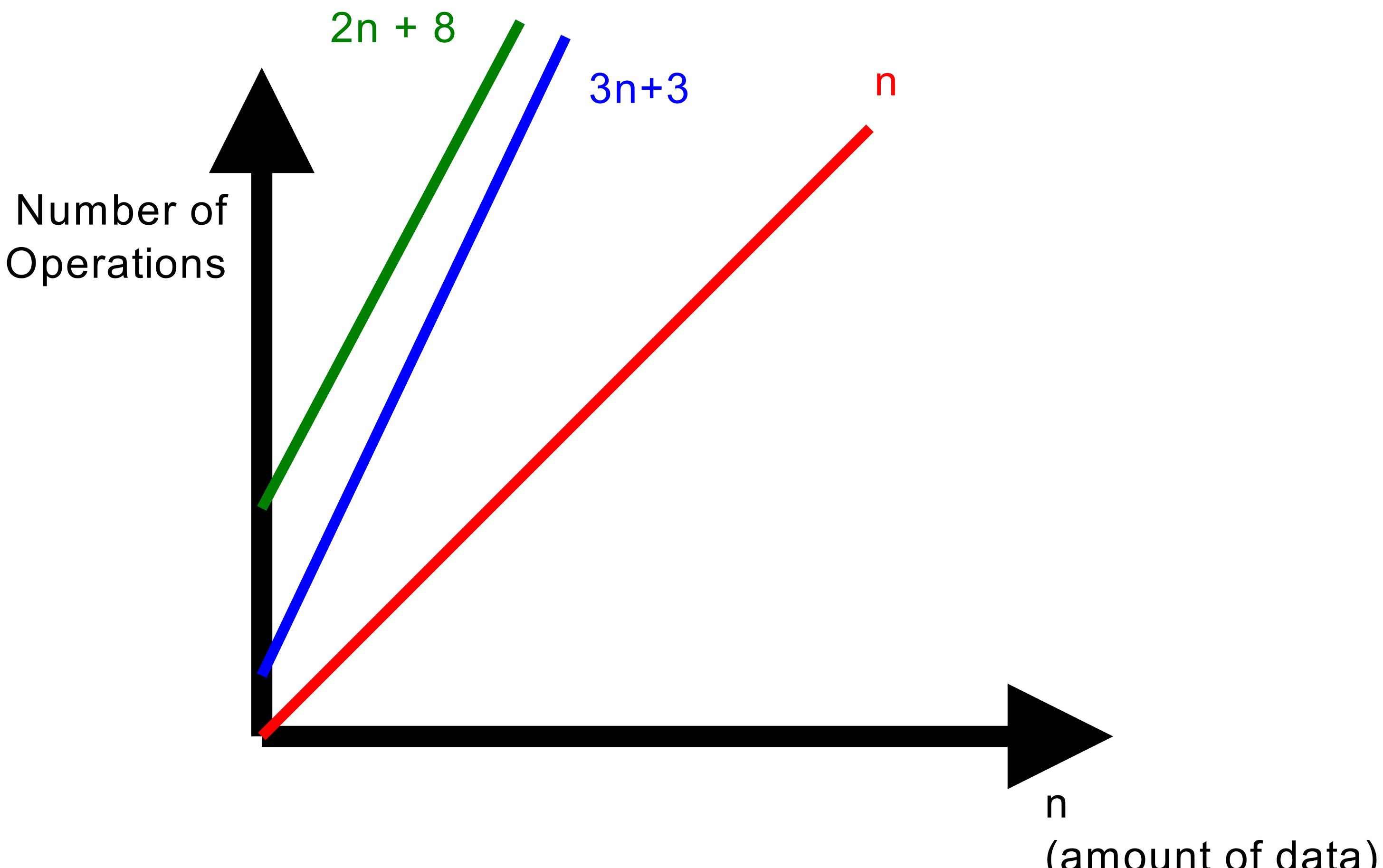
Linear Search: Worst Case

```
# let n = the length of list.
def search(list, key):
  index = 0                                      1
  while index < len(list):                       n+1
     if list[index] == key:                      n
          return index
     index = index + 1                           n
  return None                                    1
                                      Total:     3n+3
```

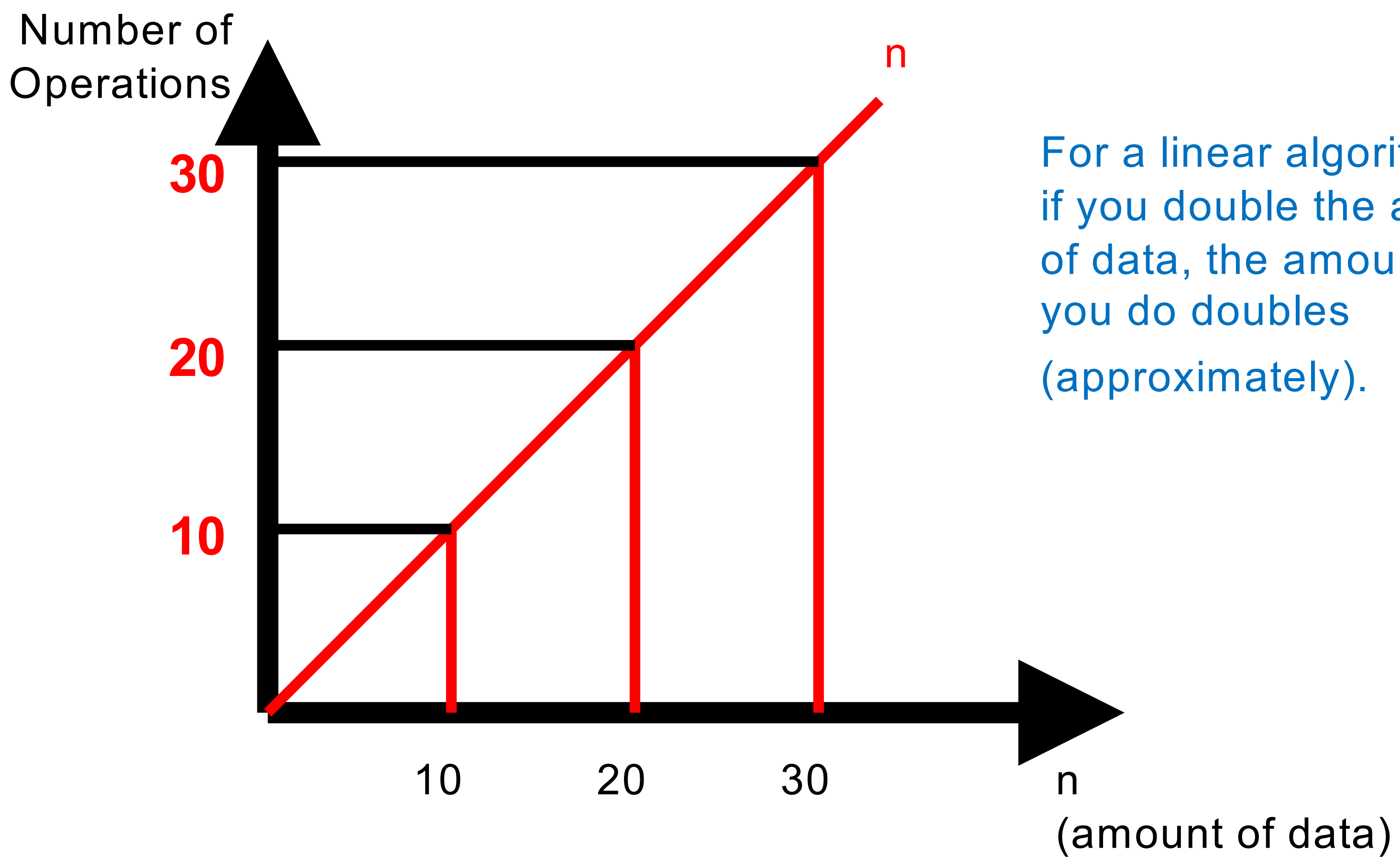
Linear Search: Worst Case Simplified

```
# let n = the length of list.
def search(list, key):
  index = 0
  while index < len(list):     n iterations
     if list[index] == key:
          return index
     index = index + 1
  return None
```

O(n) ("Linear")

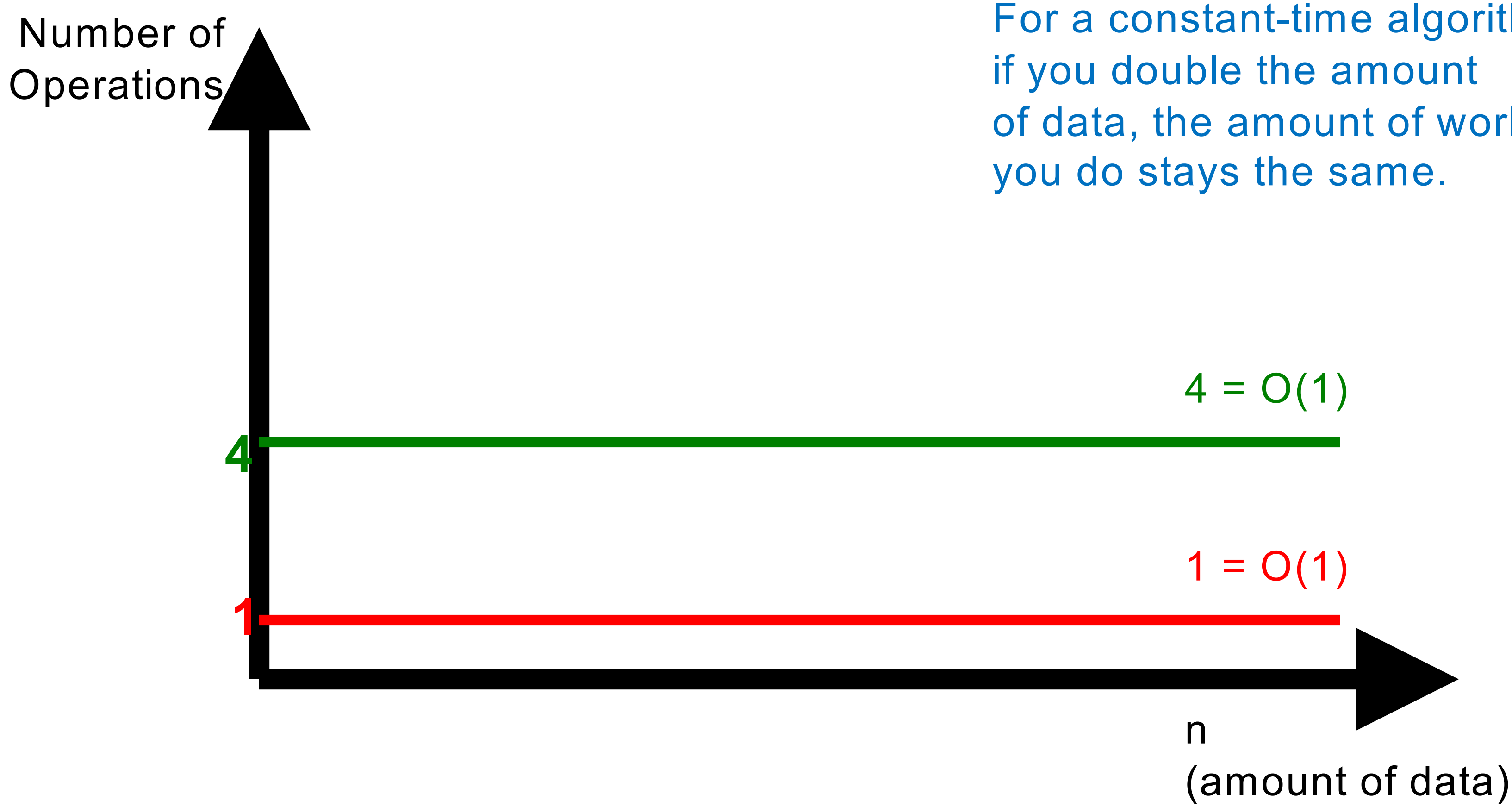


O(n)



For a linear algorithm, if you double the amount of data, the amount of work you do doubles (approximately).

O(1) (“Constant-Time”)



For a constant-time algorithm, if you double the amount of data, the amount of work you do stays the same.

Insertion Sort: worst case

```
# let n = the length of list.
def isort(list):
        i = 1
        while i != len(list):     #n-1 iterations
                move_left(list,i)
                i = i + 1
        return list
```

What is the cost of move_left?

Insertion Sort: cost of move left

```
# let n = the length of list.
def move_left(a, i):
      x = a.pop(i)                     n iterations
      j = i - 1
      while j >= 0 and a[j] > x: i iterations
            j = j - 1
      a.insert(j + 1, x)              n iterations
```

Total cost (at most): **n + i + n**

But what is **i**? To find out, look at isort, which calls move_left, supplying a value for **i**

Insertion Sort: worst case

```
# let n = the length of list.
def isort(list):
      i = 1
      while i != len(list):      #n-1 iterations
            move_left(list,i)   #i goes from 1 to n-1
            i = i + 1
      return list
```

Total cost: *cost of move_left as i goes from 1 to n-1*

Examining the cost

```
def isort(list):
    i = 1
    while i != len(list):
        move_left(list,i)
        i = i + 1
    return list
```

Examining the cost

```
def isort(list):
    i = 1
    while i != len(list):     n-1
        move_left(list,i)
        i = i + 1
    return list
```

Examining the cost

```
def isort(list):
    i = 1
    while i != len(list):        n-1
        move_left(list,i)
            pop
            while loop
            insert
        i = i + 1
    return list
```

Examining the cost

```
def isort(list):
    i = 1
    while i != len(list):        n-1
        move_left(list,i)
            pop.........................
            while loop                    n
            insert
        i = i + 1
    return list
```

Examining the cost

```
def isort(list):
    i = 1
    while i != len(list):        n-1
        move_left(list,i)
            pop..........................   n
            while loop
            insert ......................   n
        i = i + 1
    return list
```

Examining the cost

```
def isort(list):
    i = 1
    while i != len(list):        n-1
        move_left(list,i)
            pop & insert ........
            while loop           n + n

        i = i + 1
    return list
```

Examining the cost

```
def isort(list):
    i = 1
    while i != len(list):        n-1
        move_left(list,i)
            pop & insert ........        } 2n
            while loop
        i = i + 1
    return list
```

Examining the cost

```
def isort(list):
    i = 1
    while i != len(list):      n-1
        move_left(list,i)
            pop & insert .........   2n
            while loop               1+
        i = i + 1
    return list
```

Examining the cost

```
def isort(list):
    i = 1
    while i != len(list):        n-1
        move_left(list,i)
            pop & insert .........   2n
            while loop               1+2+
        i = i + 1
    return list
```

Examining the cost

```
def isort(list):
    i = 1
    while i != len(list):        n-1
        move_left(list,i)
            pop & insert .........   2n
            while loop               1+2+3...

        i = i + 1
    return list
```

How can we express this?

```
def isort(list):
    i = 1
    while i != len(list):        n-1
        move_left(list,i)
            pop & insert .........   2n
            while loop               1+2+3…n-1

        i = i + 1
    return list
```

How can we express this?

```
def isort(list):
    i = 1
    while i != len(list):        n-1
        move_left(list,i)
```

pop & insert 2n

while loop 1+2+3…n-1

```
        i = i + 1
    return list
```

1+2+3…n-1

Test for n = 7

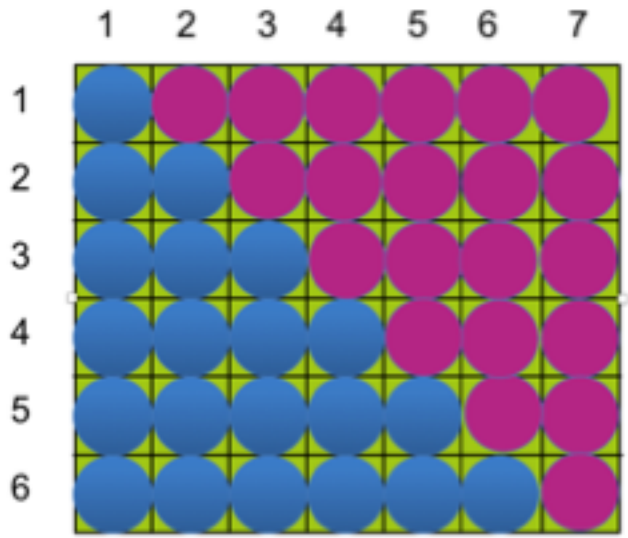
`1+2+3…n-1`

Test for n = 7.

1+2+3+4+5+6

1+2+3…n-1

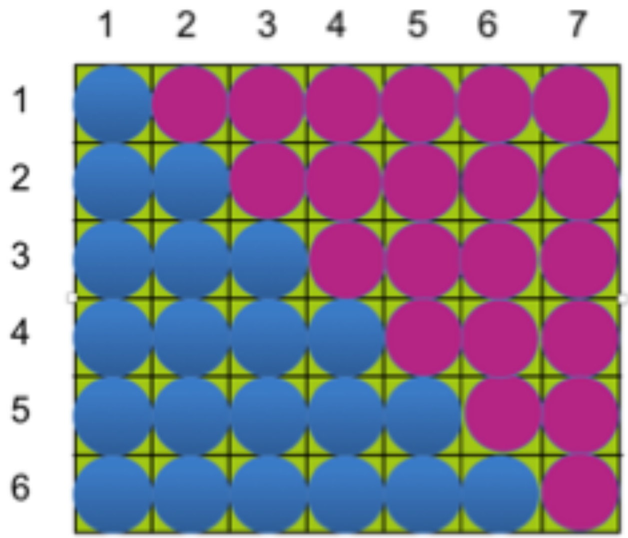
Test for n = 7.



$$1+2+3+4+5+6$$

$$1+2+3 \ldots n-1$$

Test for n = 7.

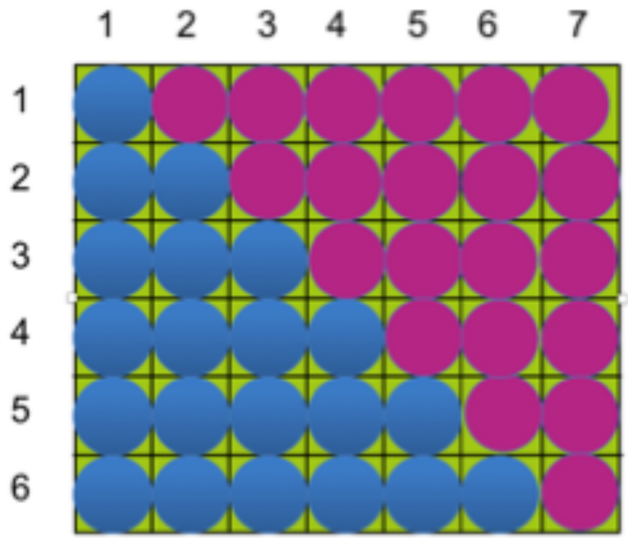


(6) * (7) / 2 blue circles

1+2+3+4+5+6

1+2+3…n-1

Test for n = 7.



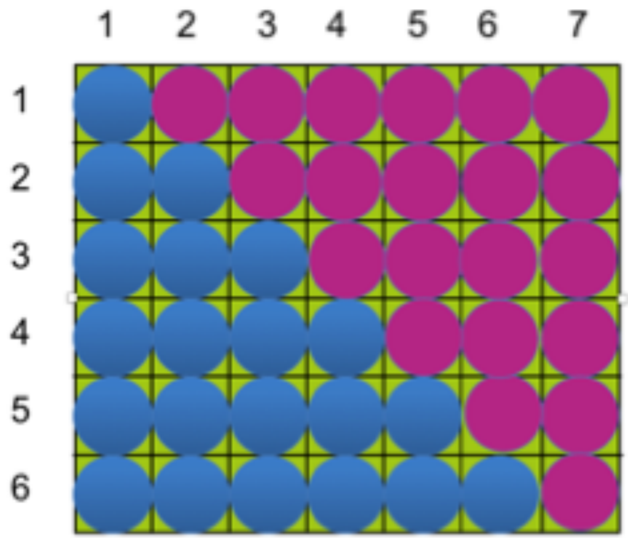
(6) * (7) / 2 blue circles

(n-1) * (n) / 2 blue circles

1+2+3+4+5+6

1+2+3…n-1

Our equation ...



(6) * (7) / 2 blue circles

(n-1) * (n) / 2 blue circles

(n-1)*n/2

1+2+3…n-1

Our equation …

```
(n-1)*n/2
1+2+3…n-1
```

How can we express this?

```
def isort(list):
        i = 1
        while i != len(list):        n-1
                move_left(list,i)
                        pop & insert .........   2n
                        while loop              1+2+3…n-1
                i = i + 1
        return list
```

$$(n-1)*n/2$$

$$1+2+3\ldots n-1$$

How can we express this?

```
def isort(list):
    i = 1
    while i != len(list):      n-1
        move_left(list,i)
            pop & insert .........   2n
            while loop               1+2+3…n-1

        i = i + 1
    return list
```

$$(n-1)*n/2$$

Combine to calculate

```
def isort(list):
    i = 1
    while i != len(list):        n-1
        move_left(list,i)
            pop & insert .........   2n +
            while loop               (n-1)*n/2

        i = i + 1
    return list
```

How can we express this?

```
def isort(list):
    i = 1
    while i != len(list):          n-1
        move_left(list,i)
            pop & insert & while   } 2n + (n-1)*n/2

        i = i + 1
    return list
```

How can we express this?

```
def isort(list):
    i = 1
    while i != len(list):        n-1
        move_left(list,i)        {2n + (n-1)*n/2



        i = i + 1
    return list
```

How can we express this?

```
def isort(list):
    i = 1
    while i != len(list):        n-1
        move_left(list,i)          { (2n + (n-1)*n/2)



        i = i + 1
    return list
```

Total number of operations

```
def isort(list):
    i = 1
    while i != len(list):
        move_left(list,i)



        i = i + 1
    return list
```

(n-1) * (2n + (n-1)*n/2)

Generalizing…

$$(n-1) * (2n + (n-1)*n/2)$$

- $= 2n^2 - 2n + (n^2 - n) / 2$
- $= (5n^2 - 5n) / 2$
- $= (5/2)n^2 - (5/2)n$

Highest order term? …

$(5/2)n^2 - (5/2)n$

$$n^2$$

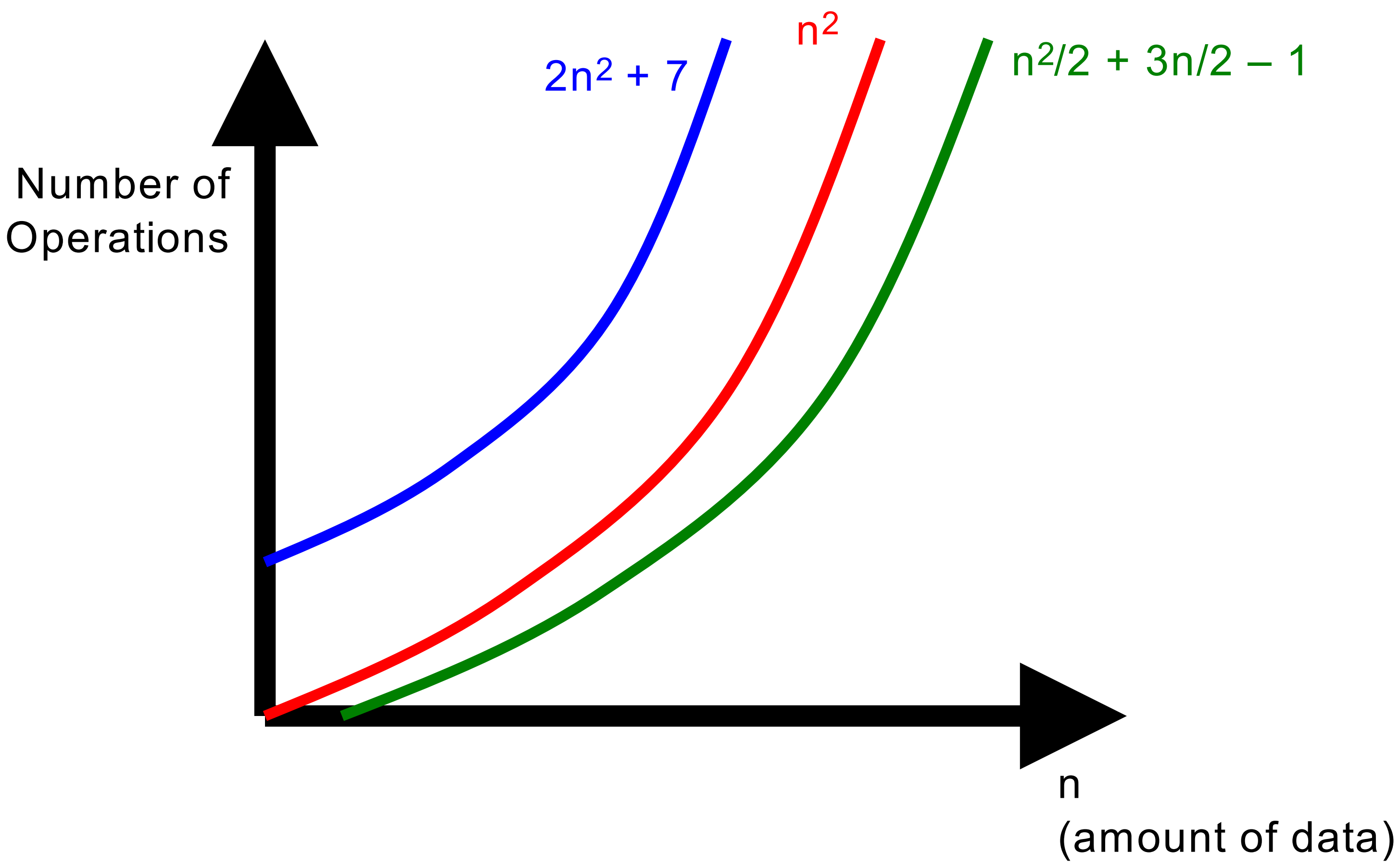
Order of Complexity

Number of operations	Order of Complexity
n^2	$O(n^2)$
$(5/2)n^2 - (1/2)n$	$O(n^2)$
$2n^2 + 7$	$O(n^2)$

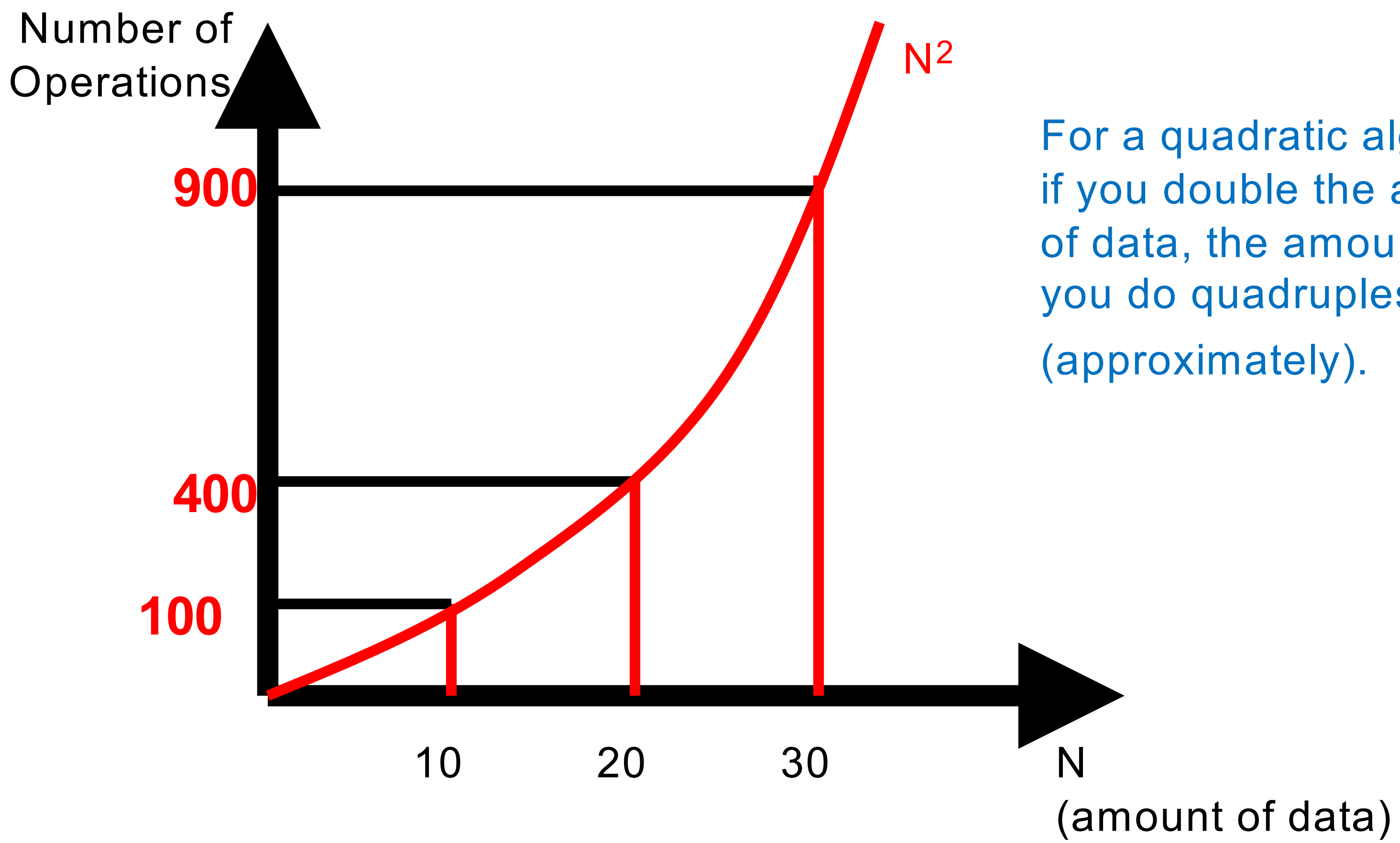
Usually doesn't matter what the constants are…
we are only concerned about the highest power of n.

$f(n)$ is $O(g(n))$ means $f(n) < g(n) \cdot k$ for some positive k

$O(n^2)$ ("Quadratic")



$O(n^2)$



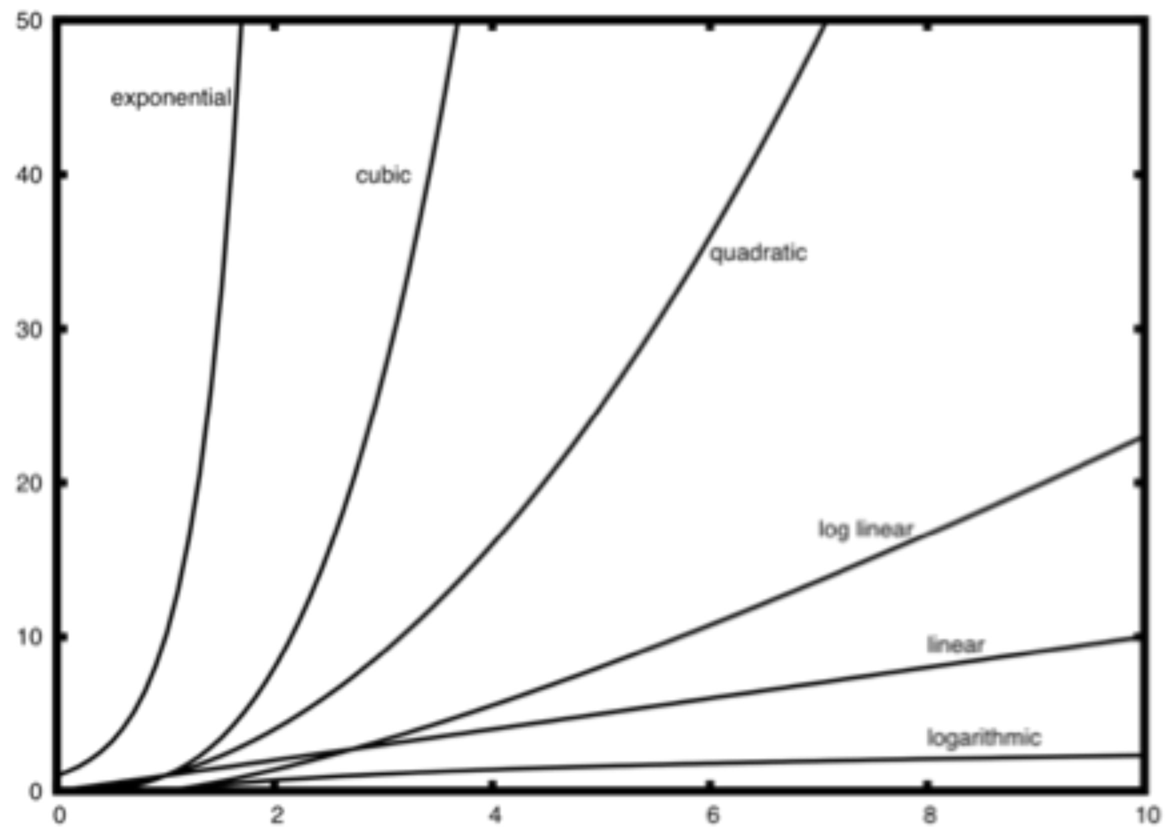
For a quadratic algorithm, if you double the amount of data, the amount of work you do quadruples (approximately).

Two Examples

Linear Sort	$O(n)$	linear
Insertion Sort	$O(n^2)$	quadratic

Big O

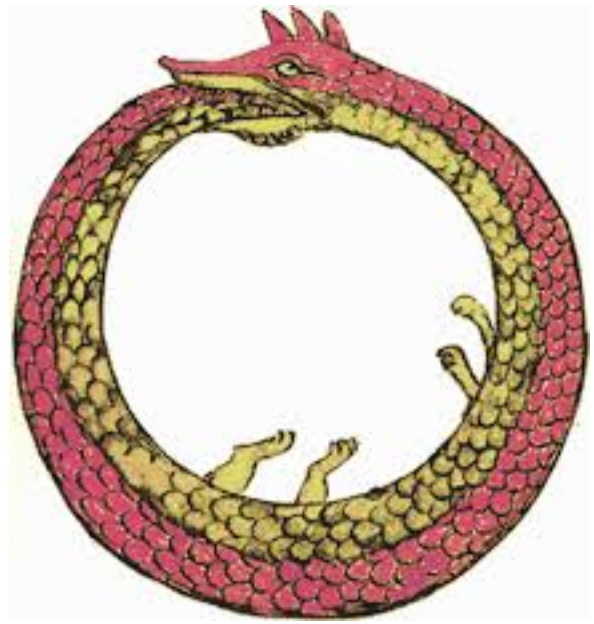
- $O(1)$ constant
- $O(\log n)$ logarithmic
- $O(n)$ linear
- $O(n \log n)$ log linear
- $O(n^2)$ quadratic
- $O(n^3)$ cubic
- $O(2^n)$ exponential



How work increases

Input Size	O(n)	$O(n^2)$	$O(n^3)$	$O(2^n)$
2	2	4	8	4
4	4	16	64	16
8	8	64	512	256
16	16	256	4096	65536
32	32	1024	32768	4294967296

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 directly or immediately
  - 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

- $n! = n \times (n-1) \times (n-2) \times \cdots \times 1$

  $2! = 2 \times 1$

  $3! = 3 \times 2 \times 1$

  $4! = 4 \times 3 \times 2 \times 1$

$9! = 362,880$

$10! = 3,628,800$

$10! = 10 \times 9!$

- *alternatively:*

  $0! = 1$ (Base case)

  $n! = n \times (n-1)!$ (Recursive case)

  So $4! = 4 \times 3!$ ➔ $3! = 3 \times 2!$ ➔ $2! = 2 \times 1!$ ➔ $1! = 1 \times 0!$ ➔ $0! = 1$

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!) = 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!) = 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

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

Recipe for Writing Recursive Functions (by Dave Feinberg)

1. Write `if`. (Why?)

   There must be at least 2 cases: base and recursive

2. Handle simplest case(s).

   No recursive call needed (base case).

3. Write recursive calls(s).

   Input is slightly simpler to get closer to base case.

4. Assume the recursive call works!

   Ask yourself: What does it do?

   Ask yourself: How does it help?

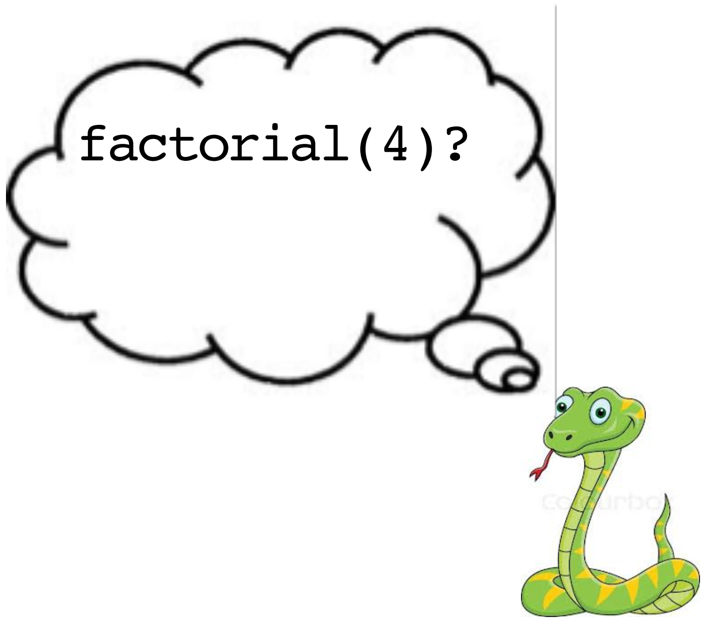
Recursive Factorial in Python

$0! = 1$	(Base case)
$n! = n \times (n-1)!$	(Recursive case)

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

S
T
A
C
K

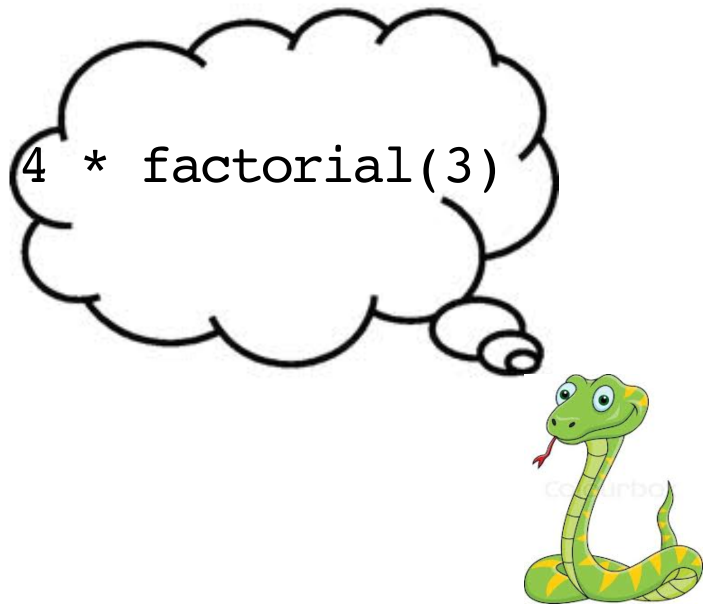
n=4



S
T
A
C
K

n=4

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



S
T
A
C
K

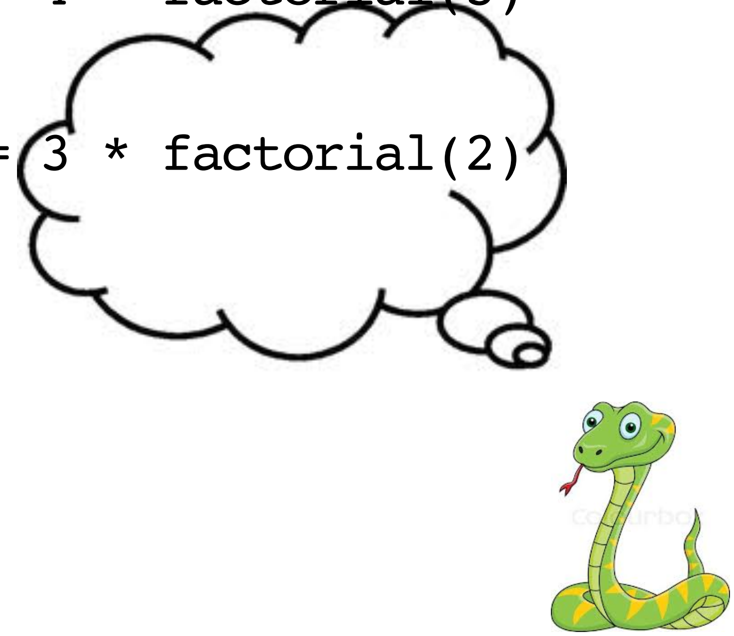
n=4 `factorial(4)? = 4 * factorial(3)`

n=3 `factorial(3)?`

S
T
A
C
K

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

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



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

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

A n=2 factorial(2)?

C

K

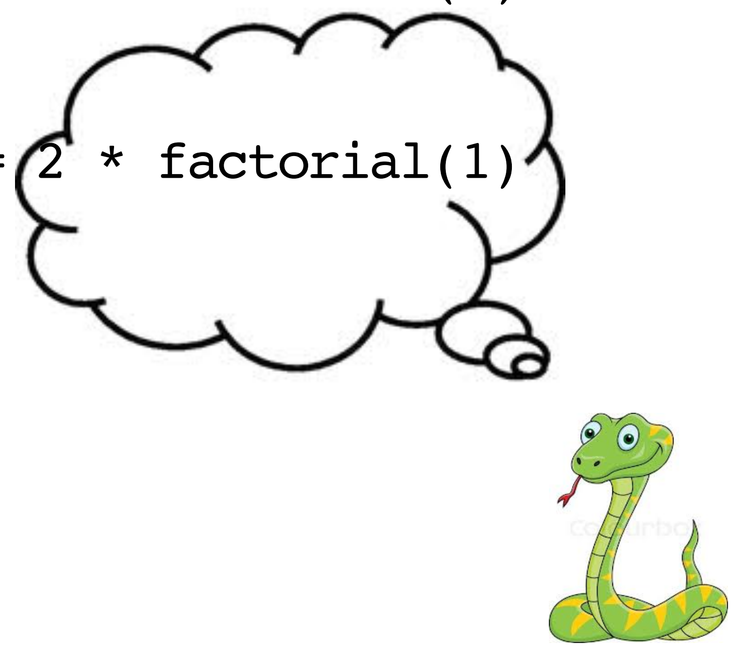
S n=4 factorial(4)? = 4 * factorial(3)

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

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

C

K

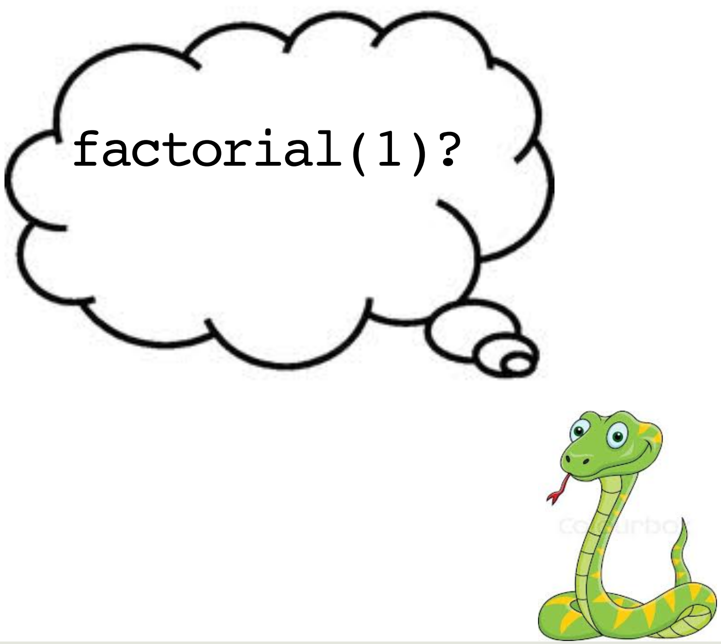


S n=4 `factorial(4)? = 4 * factorial(3)`

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

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

C n=1



K

S
T
A
C
K

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

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

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

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

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

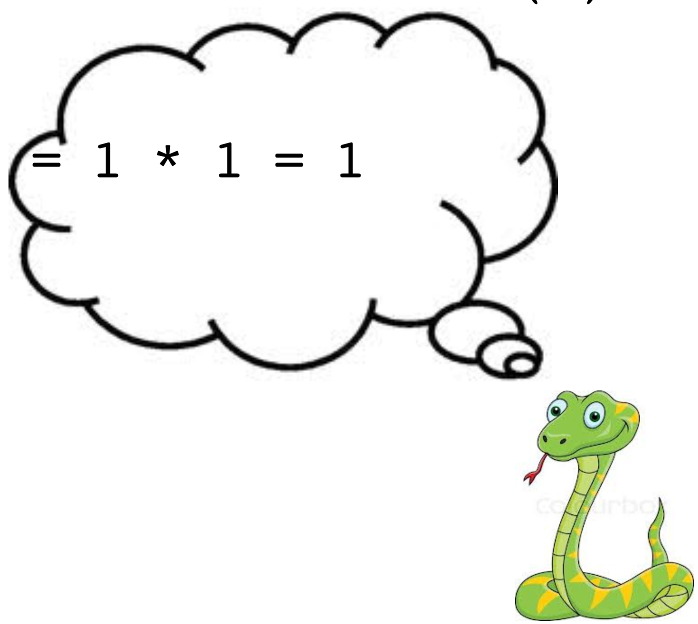
S n=4 factorial(4)? = 4 * factorial(3)

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



S

T

A

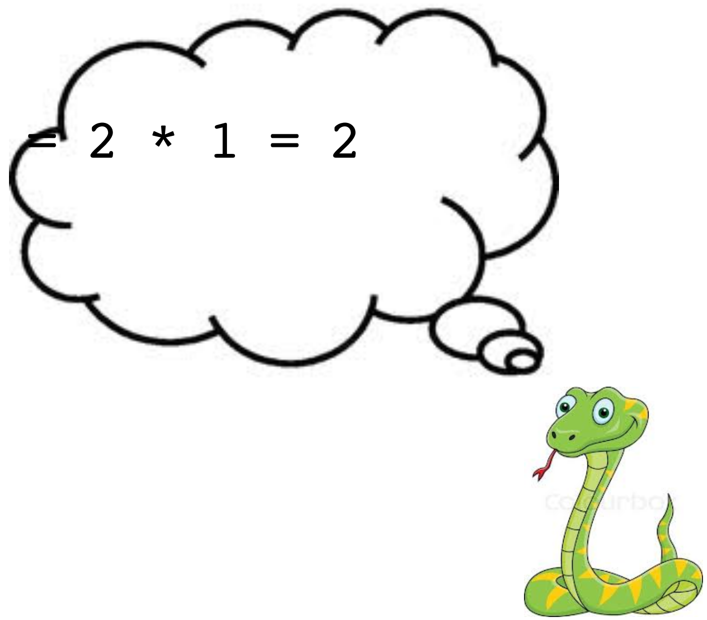
C

K

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

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

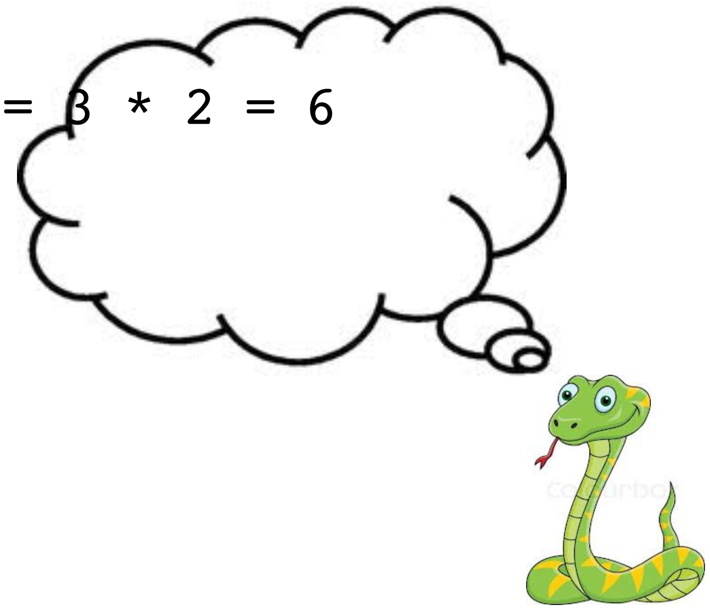
n=2 `factorial(2) = 2 * 1 = 2`



S
T
A
C
K

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

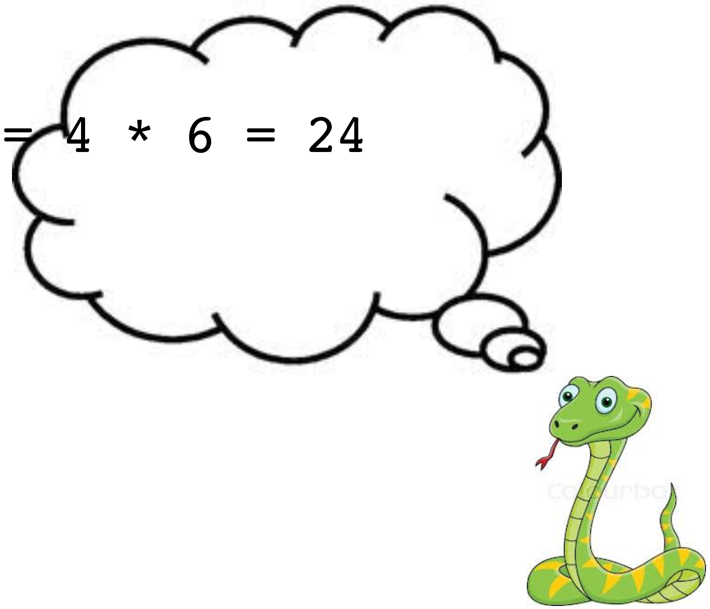
n=3 `factorial(3) = 3 * 2 = 6`



S
T
A
C
K

n=4

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



Recursive vs. Iterative Solutions

- For **every recursive function**, ← calls itself

  there is an equivalent iterative solution.

- For **every iterative function**, ← for loop, while loop

  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)
- Be optimistic. **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

Recursion on Lists

Do we know how to use iteration to sum the elements in a list?

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             :
```

What is the smallest size list?

Recursive sum of a list

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

The smallest size list is the empty list.

What is the sum of an 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(              ) ...
```

What is a simpler/smaller case?

Recursive sum of a list

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

"tail" of list

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 in the list's first element!

Tracing `sumlist`

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

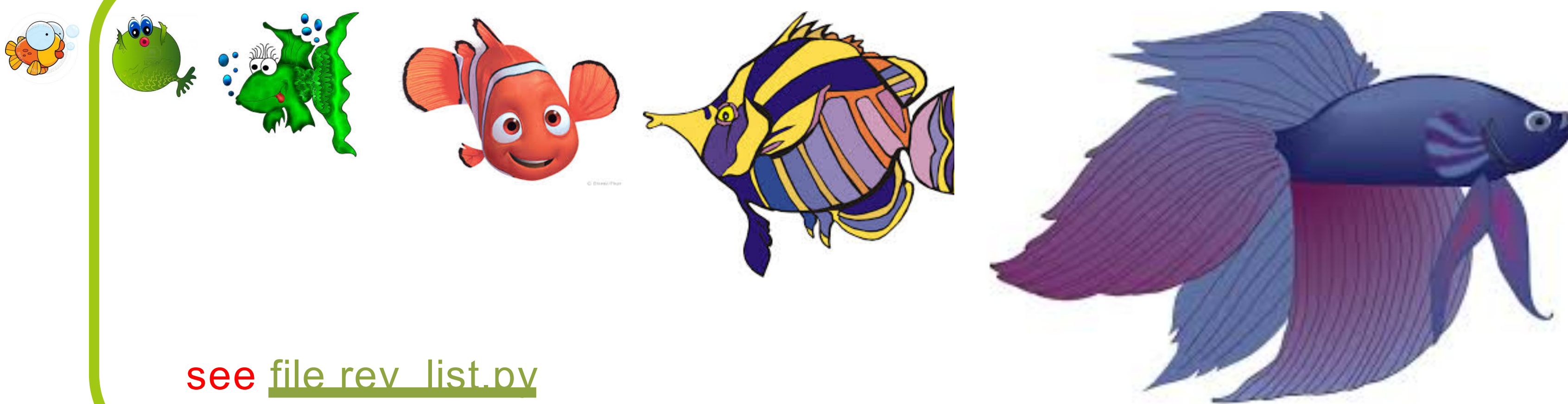
```
>>> sumlist([2,5,7])

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.

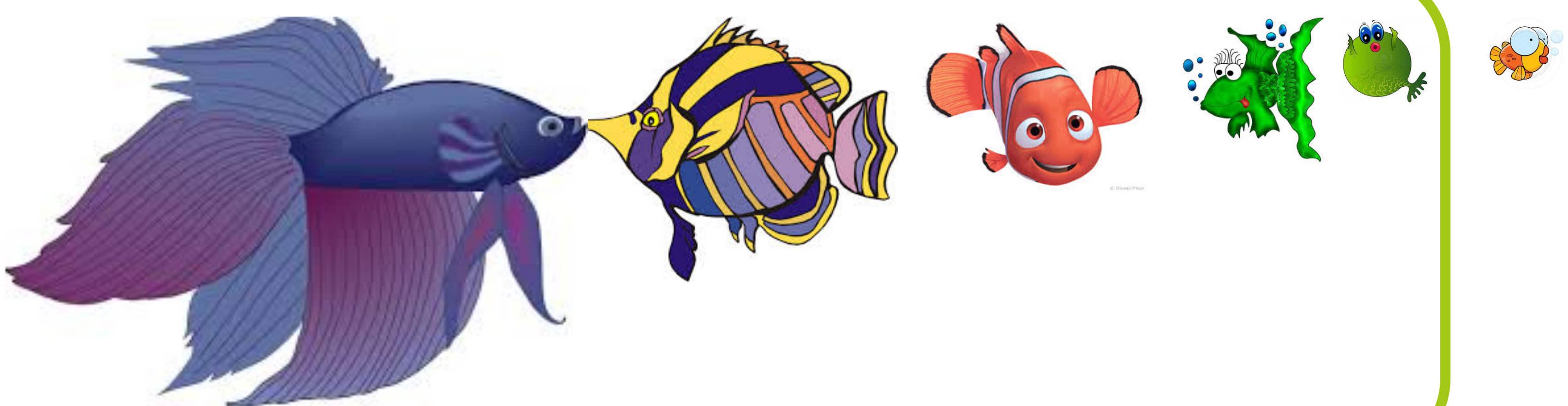
List Recursion: exercise

- 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

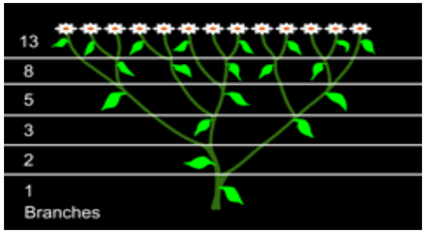
Fibonacci Numbers

- 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)







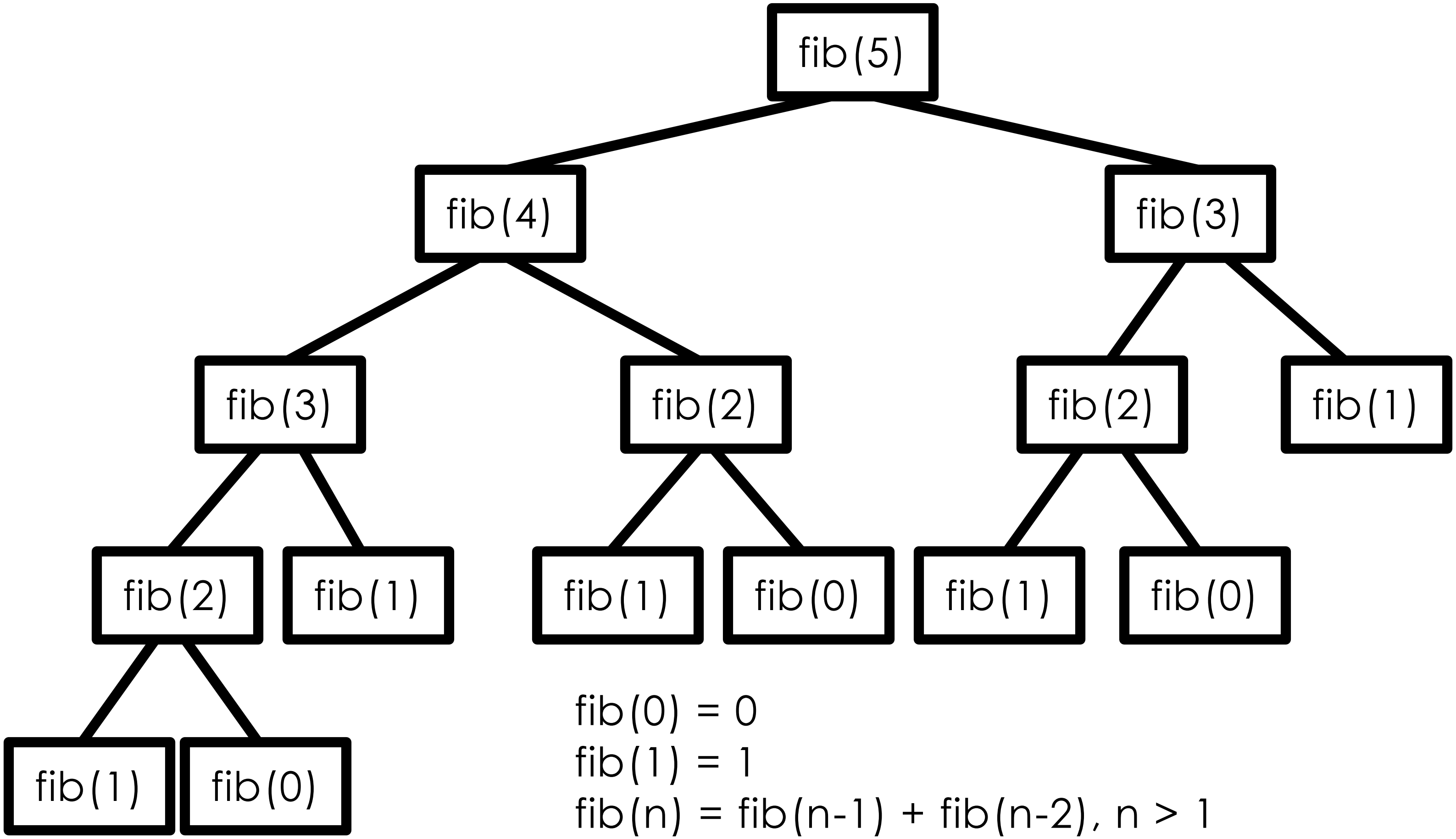
Recursive Fibonacci

- 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)
```

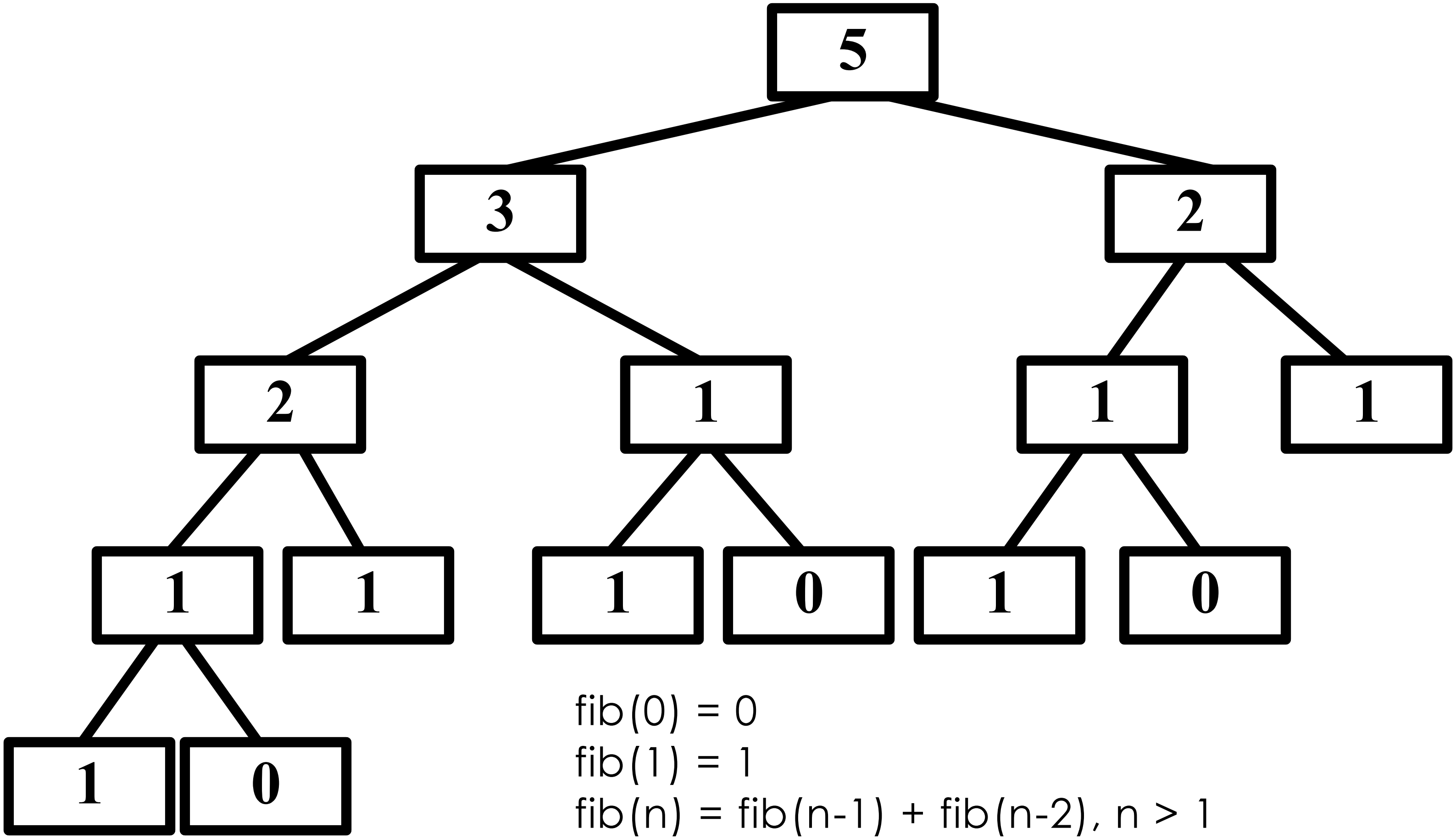
Two recursive calls!

Recursive Call Tree



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

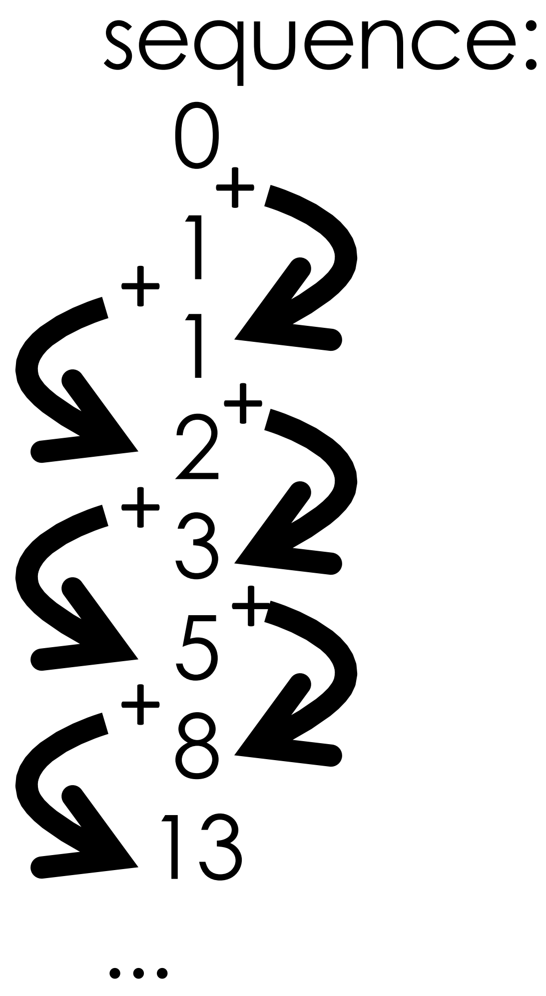
Recursive Call Tree



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):
        old_x = x
        x = next_x
        next_x = old_x + x
    return x
```



Simultaneous Assignment

Assign values to multiple variables in a single statement:

```
sum, diff = x + y, x - y
x, y = y, x
```

Iterative Fibonacci

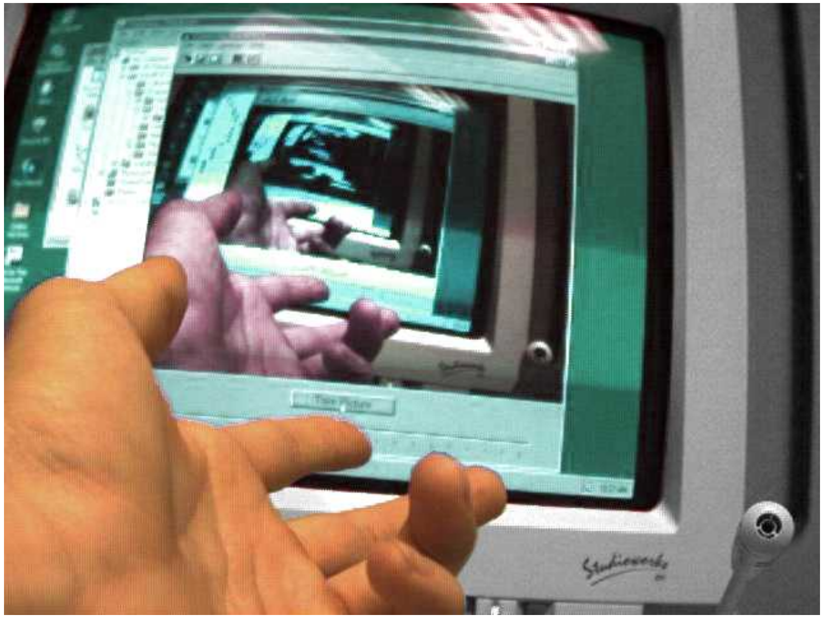
```
def fib(n):
    x = 0
    next_x = 1
    for i in range(1,n+1):
        x, next_x = next_x, x + next_x
    return x
```

SIMULTANEOUS ASSIGNMENT

Faster than the recursive version. Why?

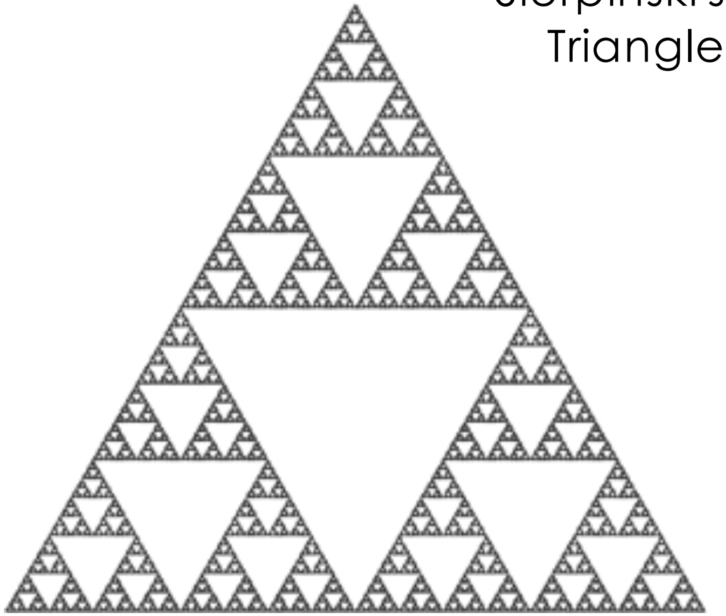
Geometric Recursion (Fractals)

- A recursive operation performed on successively smaller regions.

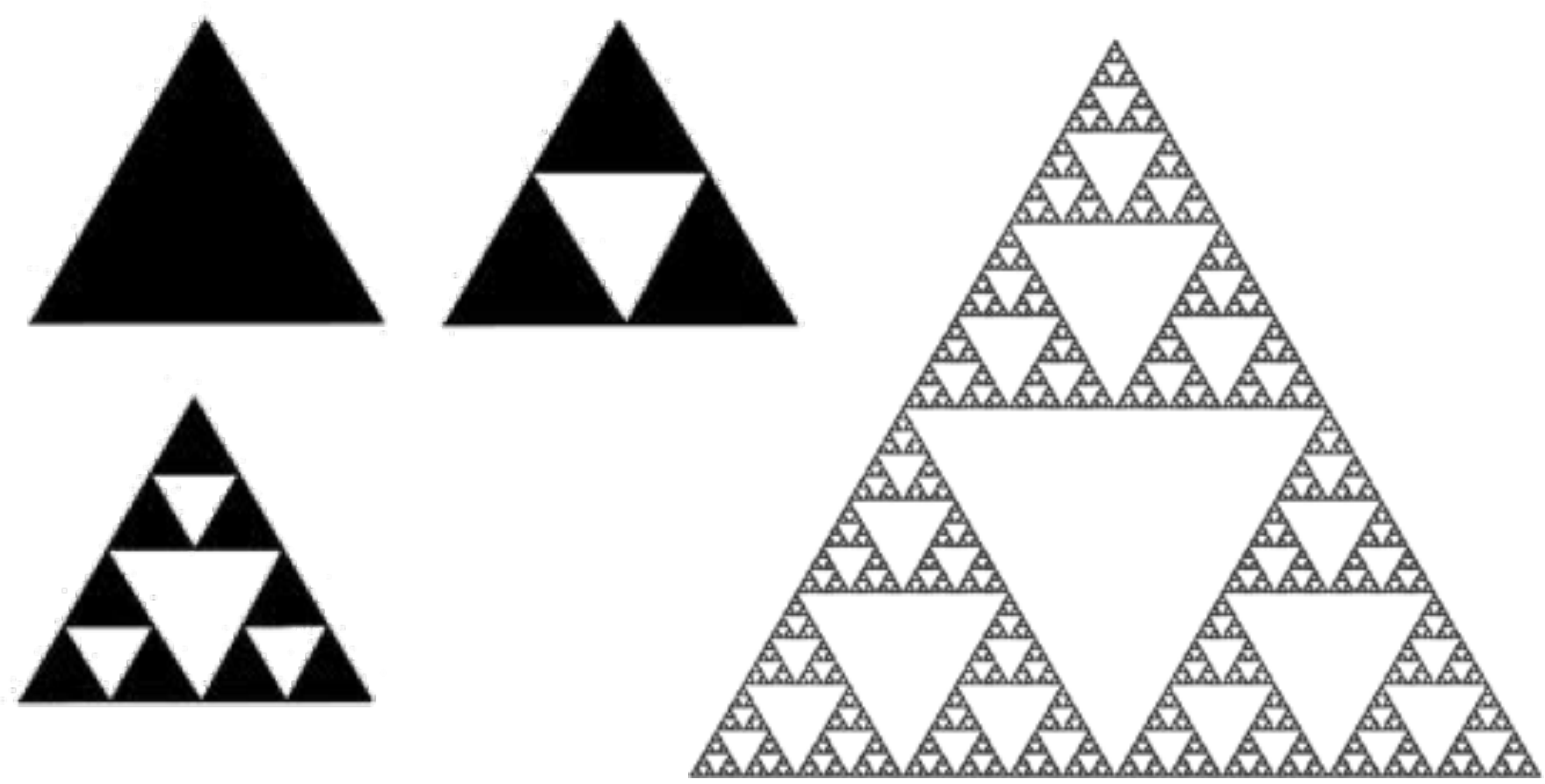


http://fusionanomaly.net/recursion.jpg

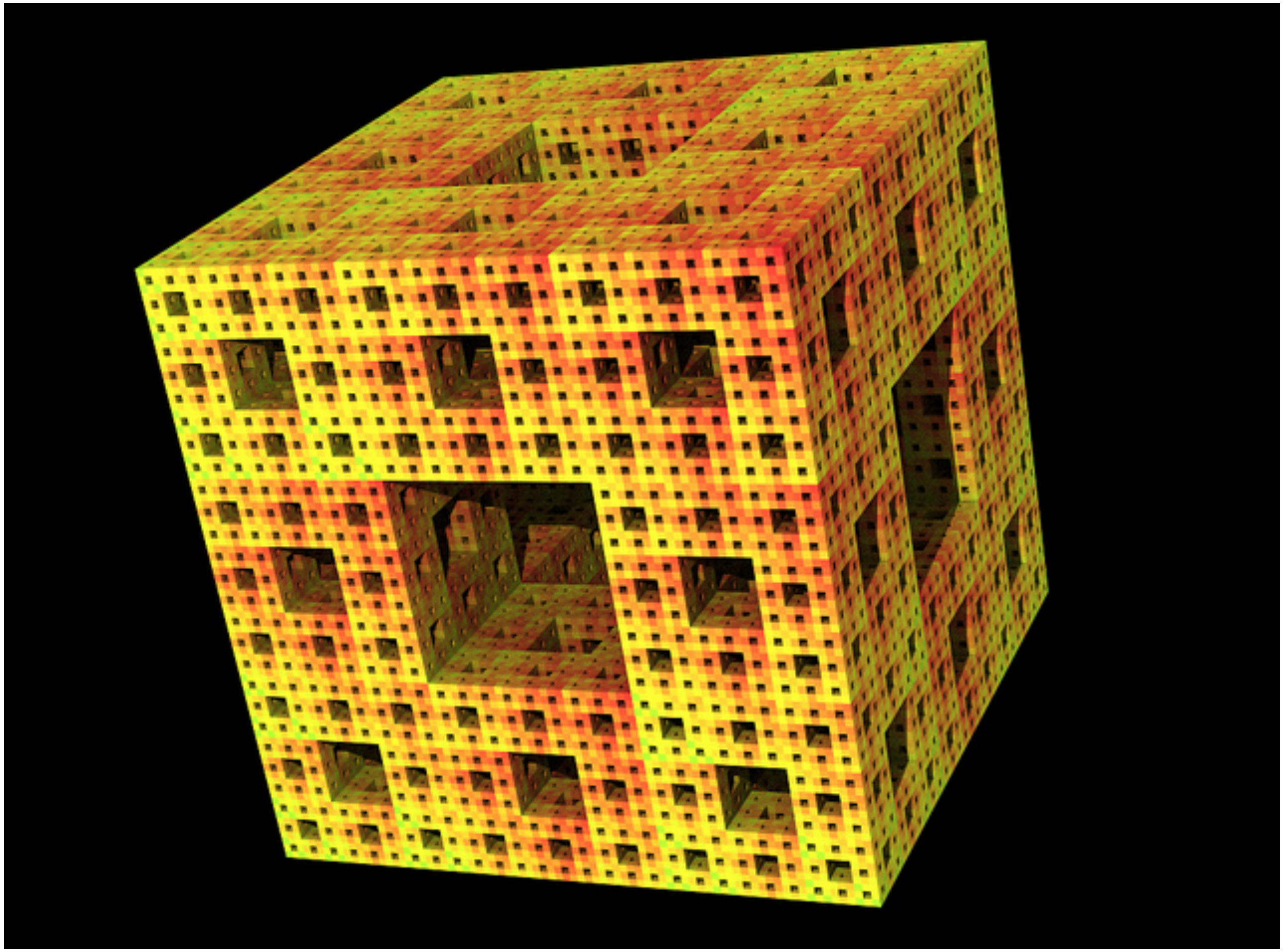
Sierpinski's Triangle



Sierpinski's Triangle

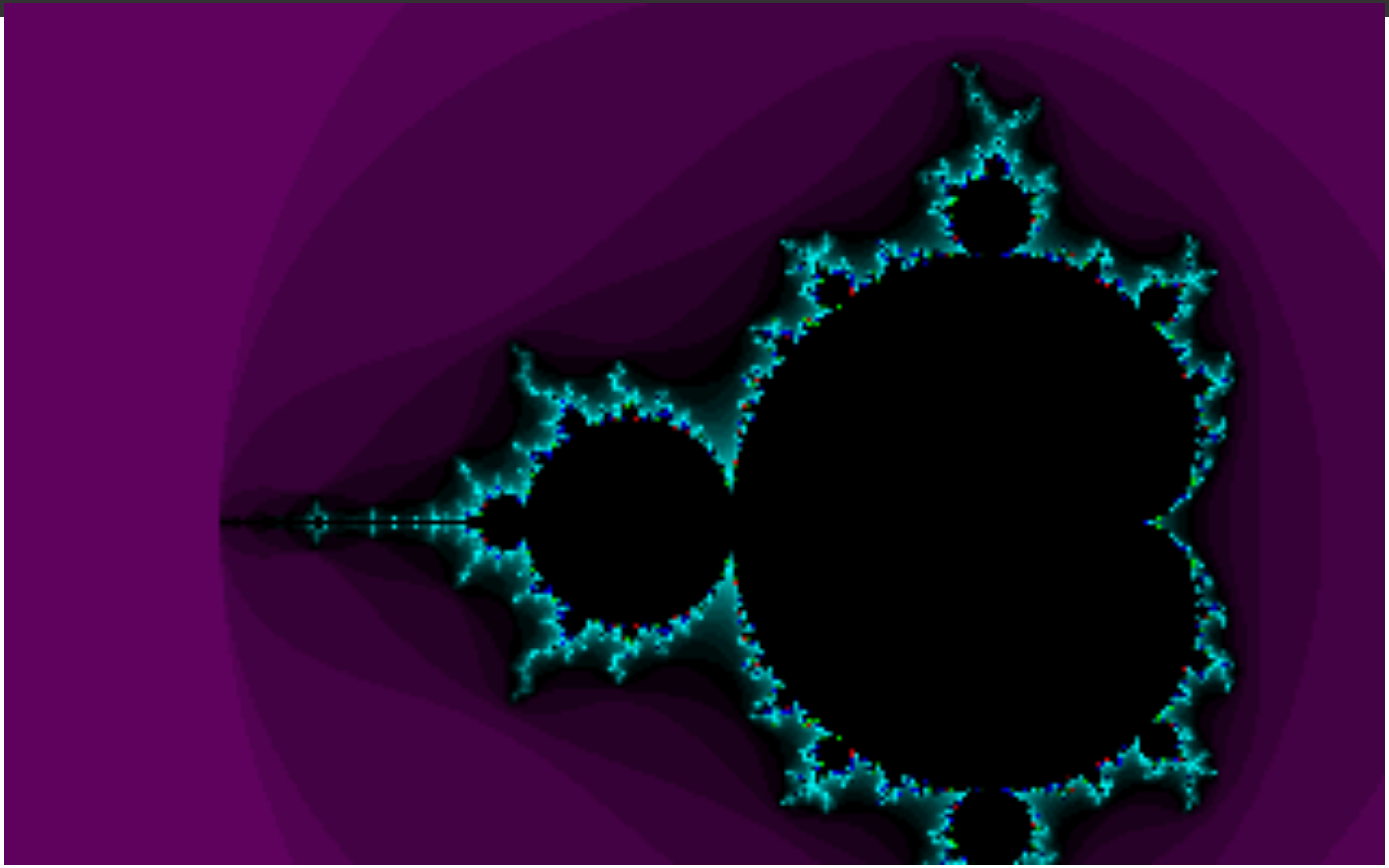


Sierpinski's Carpet



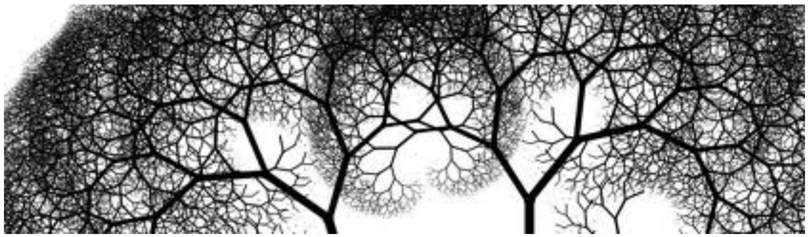
(the next slide shows an animation that could give some people headaches)

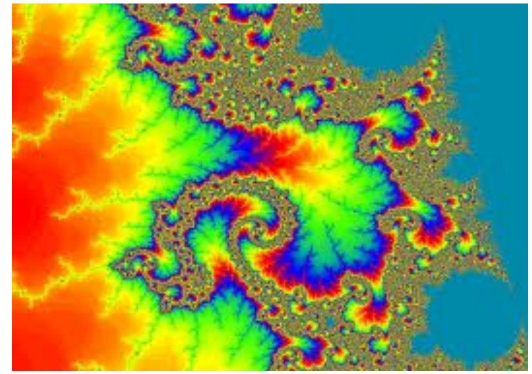
Mandelbrot set

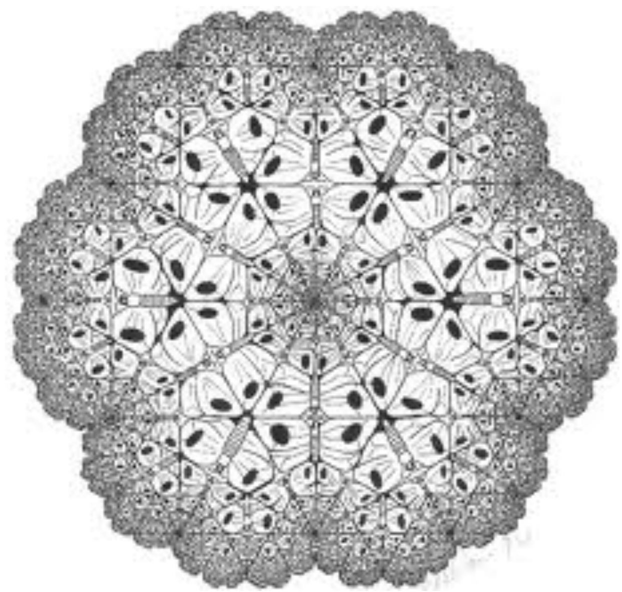


Source: Clint Sprott, http://sprott.physics.wisc.edu/fractals/animated/

Fancier fractals







Next Lecture

recursion for search

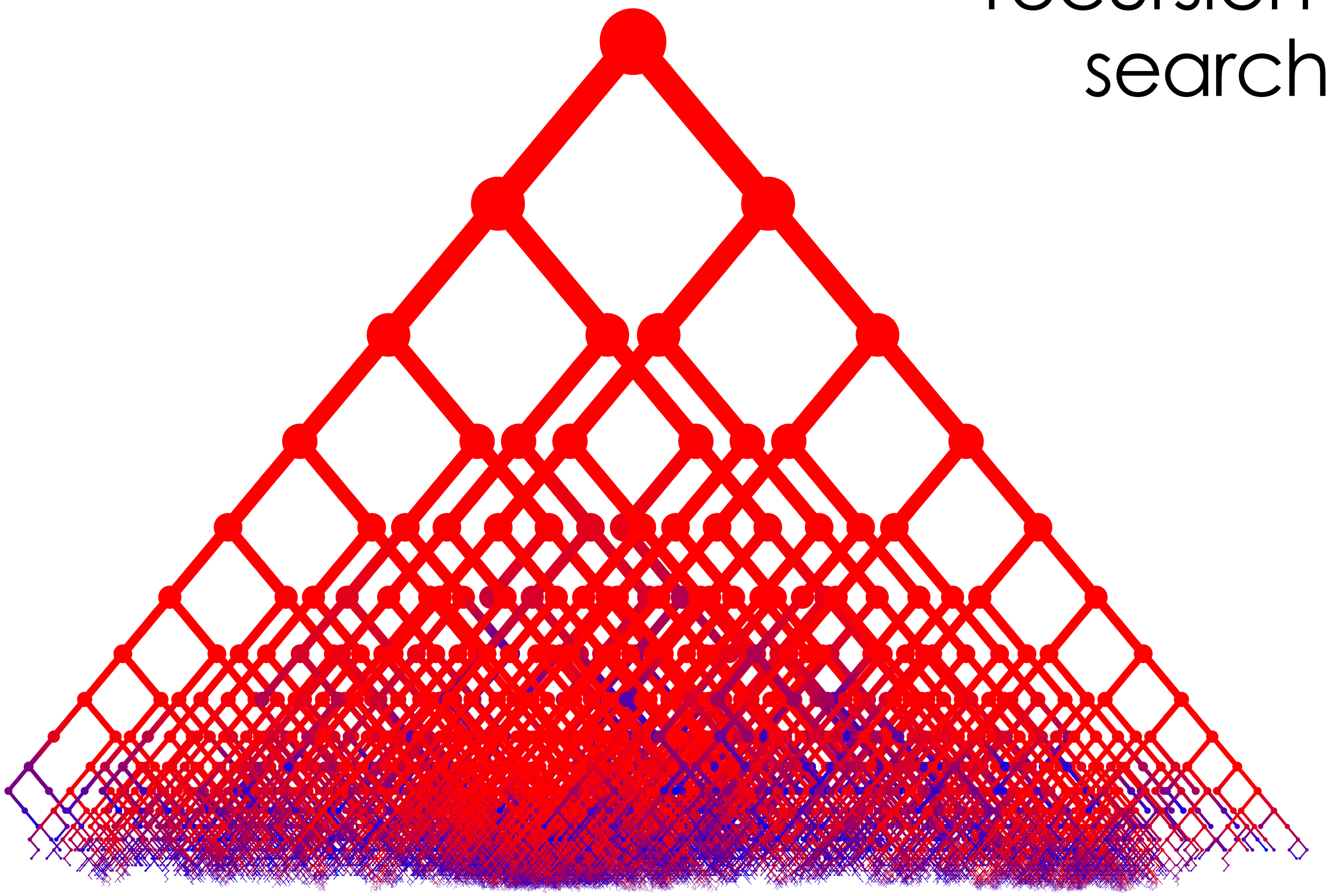


image: Matt Roberts, http://people.bath.ac.uk/mir20/blogposts/bst_close_up.php