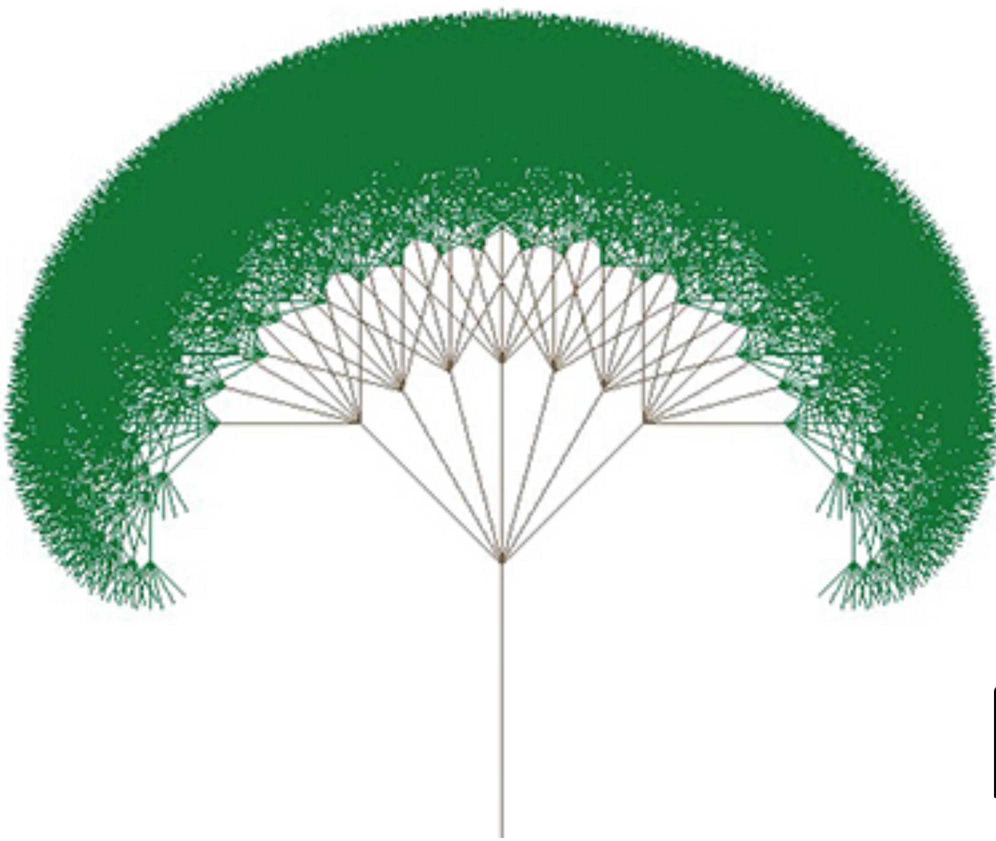
Recursion: Introduction



Announcements

- Deadlines
  - Exam on Thursday: Units 1 – 5 (inclusive)
  - PA 4 due tonight
  - PS 4 due now!
- Monday:
  - PA5 is due Mon night
  - OLI Recursion over the weekend
  - Lab 6

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

- **Goal: understanding behavior of program over the long run, with increasingly large inputs**

Asymptotic Analysis

- **Goal: understanding behavior of program over the long run, with increasingly large inputs**
- **Assumptions**:
  - Input (also known as **n**) changes
  - All the other factors/operations are constant
- **As a result**: We are not concerned with constants factors:
  - How many iterations?
  - Not operations in each iteration

Asymptotic Analysis

- **Goal: understanding behavior of program over the long run, with increasingly large inputs**
- **Assumptions**:
  - Input (also known as **n**) changes
  - All the other factors/operations are constant
- **As a result**: 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-O 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)

Why don’t constants matter?

For n = 10000

n^2

n^3

$45*n^2$

Why don't constants matter?

For n = 10000

n^2

10000*10000 =
100000000 = **10^8**

$45*n^2$

45*10000*10000 =
4500000000 = **$45*10^8$**

n^3

10000*10000*10000 =
1000000000000 = **10^{12}**

Why don't constants matter?

For n = 10000

n^2

$10000*10000 =$
$100000000 = \mathbf{10^8}$

$\approx$

$45*n^2$

$45*10000*10000 =$
$4500000000 = \mathbf{45*10^8}$

$\ll$

n^3

$10000*10000*10000 =$
$1000000000000 = \mathbf{10^{12}}$

Order of Complexity

- Big-O is ignores constants, focuses on **highest power of n**

Number of iterations	**Order of Complexity**
$\mathbf{n}$	$O(n)$
$5\mathbf{n}$	$O(n)$
$4\mathbf{n}+2$	$O(n)$
$\mathbf{n^2}$	$O(n^2)$
$4\mathbf{n^2}$	$O(n^2)$
$3+\mathbf{n^2}$	$O(n^2)$
$5\mathbf{n^2} + 3n + 1$	$O(n^2)$
$\mathbf{n^3} + n^2 + n + 1$	$O(n^3)$

Quick Examples

```
def complex_1(n):
    i = 0
    while  i < n:
        # do something
```

Quick Examples

```
def complex_1(n):
    i = 0
    while  i < n:
        # do something
```

n times = O(n)

Quick Examples

```
def complex_1(n):
    i = 0
    while  i < n:
        # do something
```

n times = O(n)

```
def complex_2(n):
    for i in range(n):
        for j in range(n):
            # do something
```

Quick Examples

```
def complex_1(n):
    i = 0
    while  i < n:
        # do something
```

n times = O(n)

```
def complex_2(n):
    for i in range(n):
        for j in range(n):
            # do something
```

n times * n times = $O(n^2)$

Quick Examples

```
def complex_1(n):
    i = 0
    while  i < n:
        # do something
```

n times = O(n)

```
def complex_2(n):
   for i in range(n):
     for j in range(n):
        # do something
```

n times * n times = $O(n^2)$

```
def complex_3(n):
   i = 0
   while  i < n:
       # do something
       complex_2(n)
```

Quick Examples

```
def complex_1(n):
    i = 0
    while  i < n:
        # do something
```

n times = O(n)

```
def complex_2(n):
  for i in range(n):
    for j in range(n):
      # do something
```

n times * n times = O(n^2)

```
def complex_3(n):
  i = 0
  while  i < n:
      # do something
      complex_2(n)
```

n times * n^2 times = O(n^3)

Quick Examples

```
def complex_1(n):
    i = 0
    while  i < n:
        # do something
```

n times = O(n)

```
def complex_2(n):
    for i in range(n):
        for j in range(n):
            # do something
```

n times * n times = O(n^2)

```
def complex_3(n):
    i = 0
    while  i < n:
        # do something
        complex_2(n)
```

n times * n^2 times = O(n^3)

```
def complex_2(n):
    for i in range(n):
        for j in range(n):
            for k in range(n):
                # do something
```

Quick Examples

```
def complex_1(n):
    i = 0
    while  i < n:
        # do something
```

n times = O(n)

```
def complex_2(n):
    for i in range(n):
        for j in range(n):
            # do something
```

n times * n times = $O(n^2)$

```
def complex_3(n):
    i = 0
    while  i < n:
        # do something
        complex_2(n)
```

n times * n^2 times = $O(n^3)$

```
def complex_2(n):
    for i in range(n):
        for j in range(n):
            for k in range(n):
                # do something
```

n times * n times * n times = $O(n^3)$

Linear Search O(n)

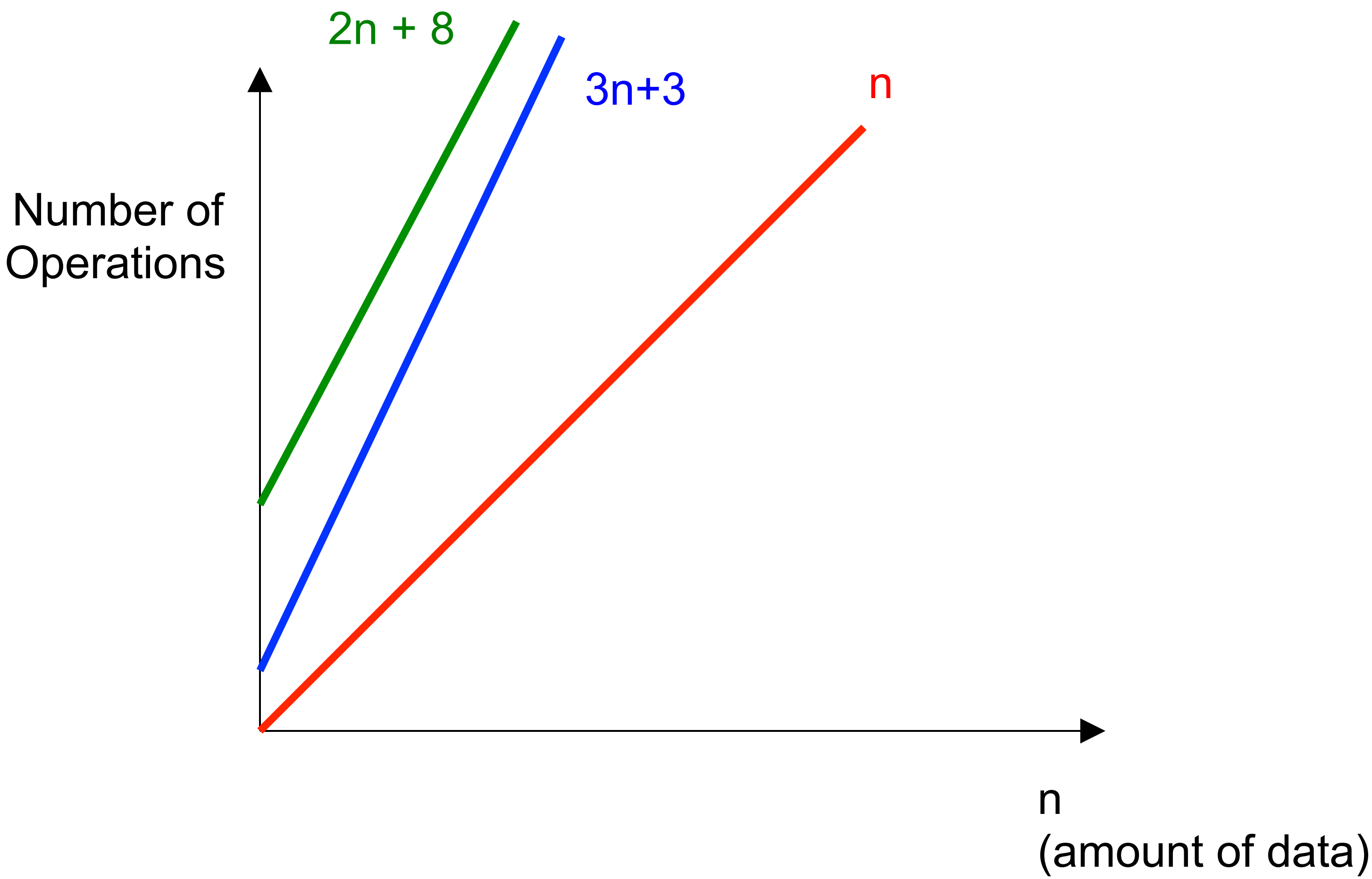
Linear Search: **Worst Case**

```
# 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
```

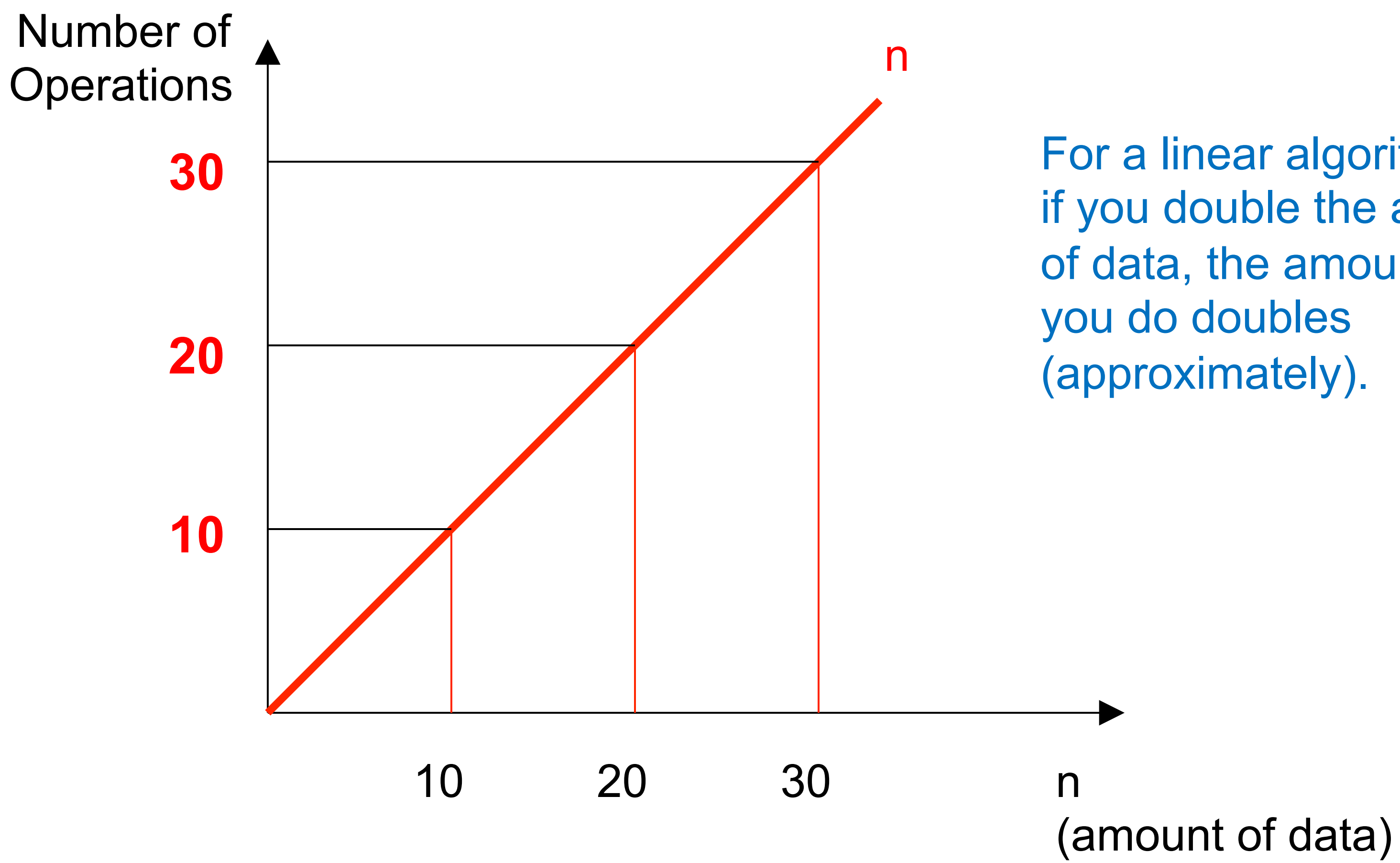
Linear Search: **Worst Case** Simplified

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

O(n) ("Linear")



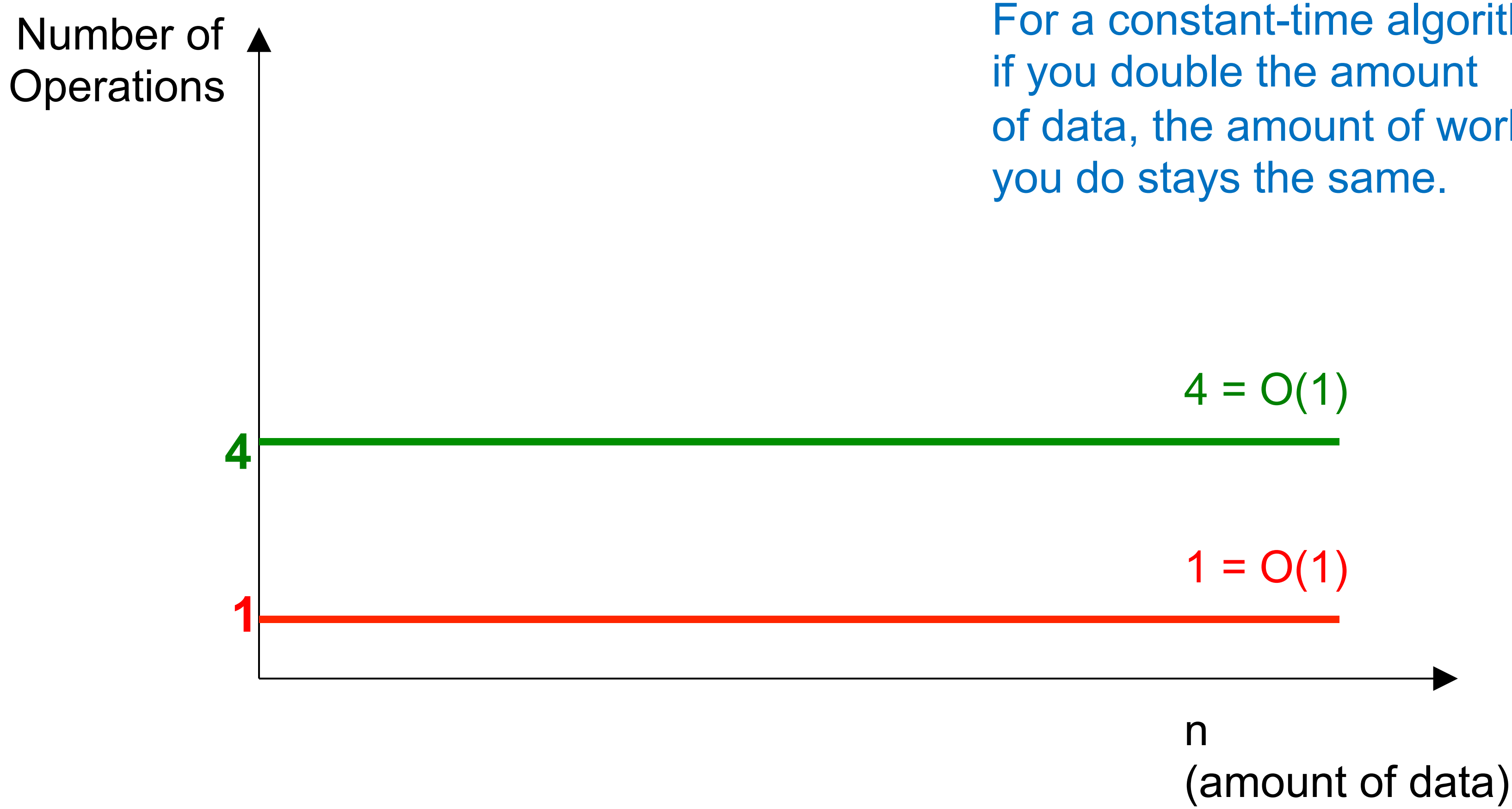
O(n)



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 $O(n^2)$

Insertion Sort: **worst case**

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

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
```

Insertion Sort: worst case

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

Insertion Sort: worst case

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

    return list
```

n-1 iterations

Insertion Sort: worst case

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

    return list
```

Insertion Sort: worst case

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

      return list
```

n-1 iterations

n iterations

n iterations

Total cost (at most):

(n-1)*(2n)

Insertion Sort: worst case

```
# let n = the length of list.
def isort(list):
    i = 1
    while i != len(list):                     n-1 iterations
        x = list.pop(i)                       n iterations
        j = i - 1
        while j >= 0 and a[j] > x:            1,2,3..n-1 iter
            j = j - 1
        list.insert(j + 1, x)                 n iterations
        i = i + 1

    return list
```

Total cost (at most):

(n-1)*(2n)+

Insertion Sort: worst case

```
# let n = the length of list.
def isort(list):
    i = 1
    while i != len(list):              n-1 iterations
        x = list.pop(i)                 n iterations
        j = i - 1
        while j >= 0 and a[j] > x:     1,2,3..n-1 iter
            j = j - 1
        list.insert(j + 1, x)           n iterations
        i = i + 1

    return list
```

Total cost (at most):

(n-1)*(2n)+(1+2+3+..+n-1)

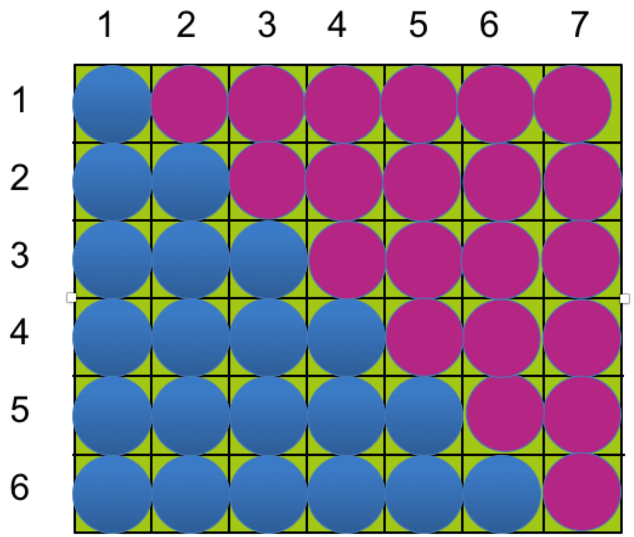
Generalizing…

Total cost (at most):

`(n-1)*(2n)+(1+2+3+..+n-1)`

- How to find `(1+2+3+..+n-1)` ?

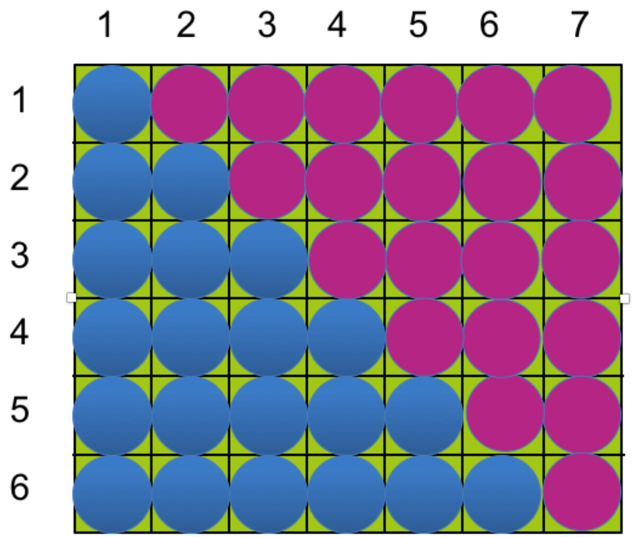
Test for n = 7.



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

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

Our equation …



(6) * (7) / 2 blue circles

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

(n-1)*n/2

1+2+3…n-1

Generalizing…

Total cost (at most):

`(n-1)*(2n)+(1+2+3+..+n-1)`

- `(1+2+3+..+n-1)` → n*(n-1)/2

Generalizing…

Total cost (at most):

`(n-1)*(2n)+(1+2+3+..+n-1)`

- `(1+2+3+..+n-1)` → n*(n-1)/2
- `(n-1)*(2n)+(1+2+3+..+n-1)`
- $=2n^2 - 2n + (n^2 - n) / 2$
- $= (5n^2 - 5n) / 2$
- $= (5/2)n^2 - (5/2)n$

Generalizing...

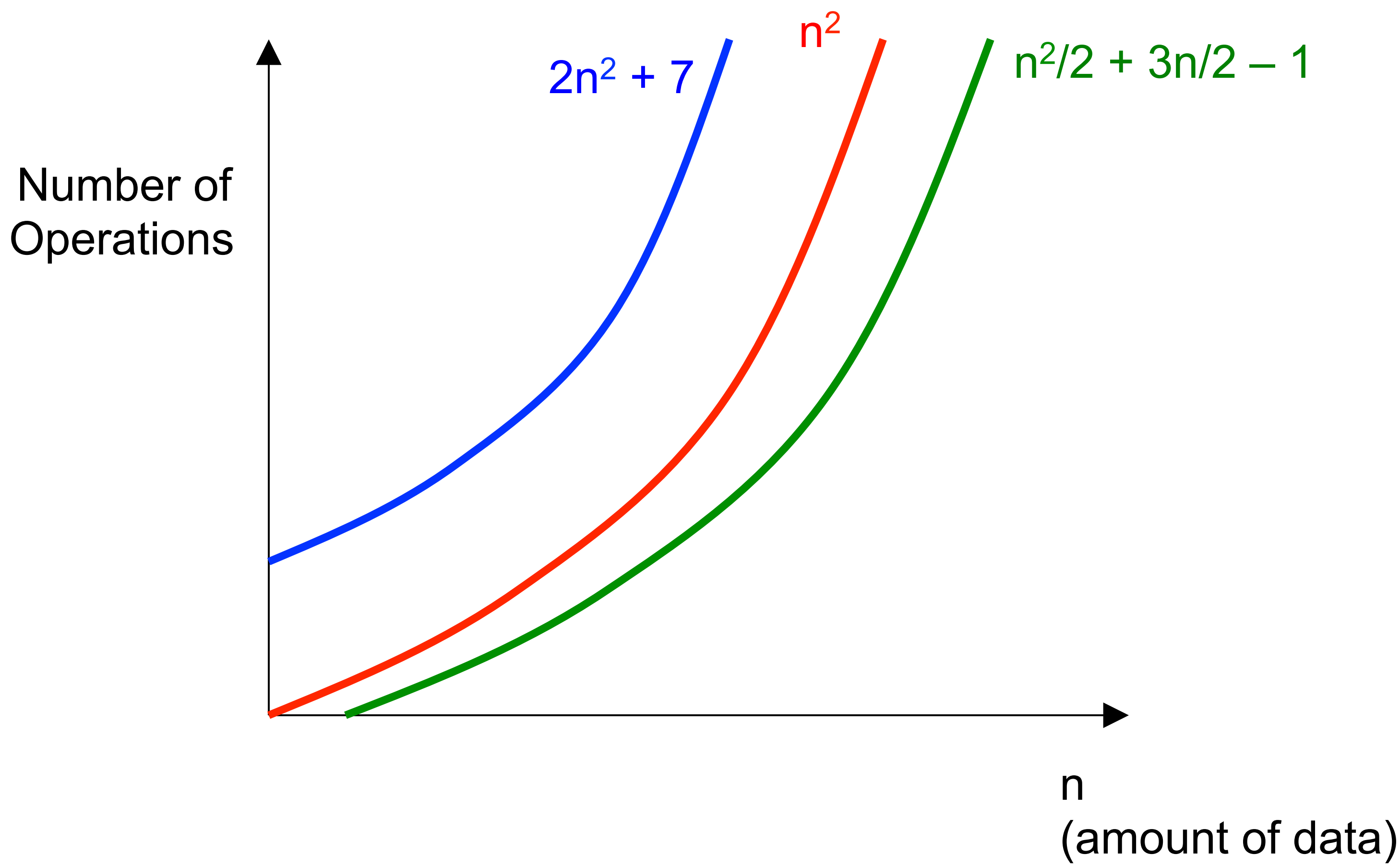
Total cost (at most):

`(n-1)*(2n)+(1+2+3+..+n-1)`

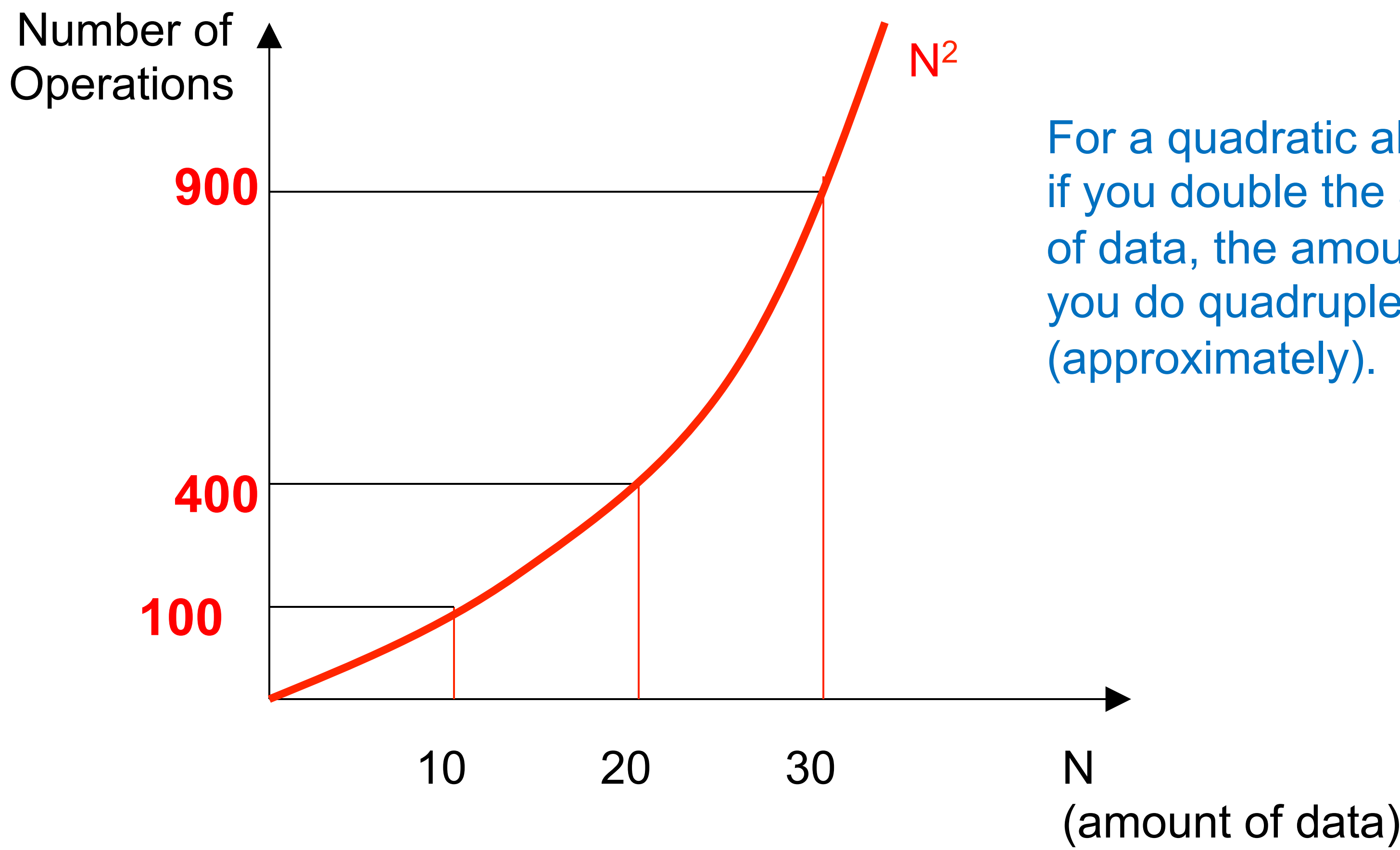
- `(1+2+3+..+n-1)` → n*(n-1)/2
- `(n-1)*(2n)+(1+2+3+..+n-1)`
- $=2n^2 - 2n + (n^2 - n) / 2$
- $= (5n^2 - 5n) / 2$
- $= (5/2)n^2 - (5/2)n$

$$n^2$$

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



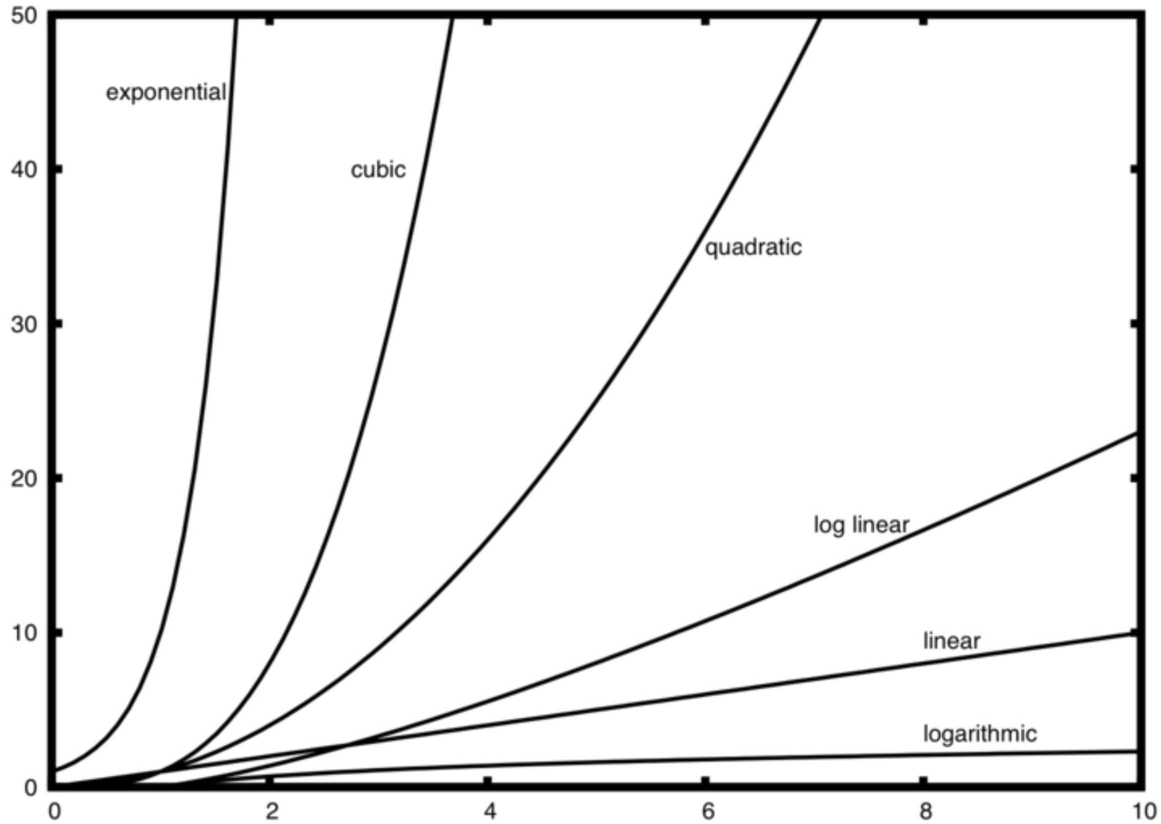
$O(n^2)$



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

Big O

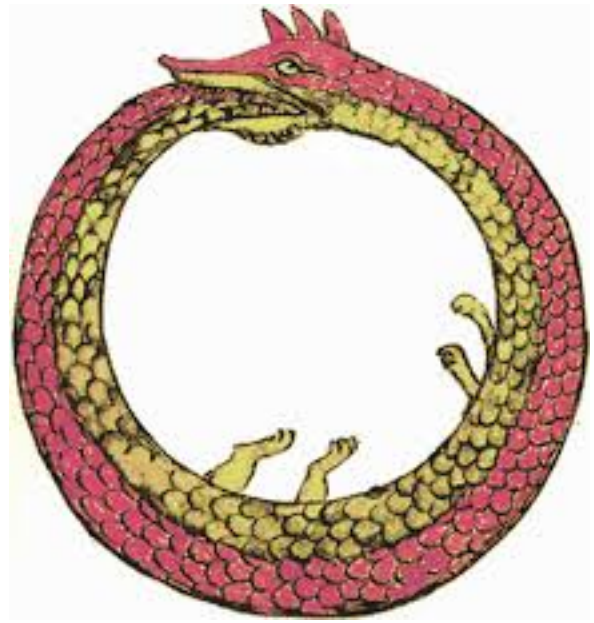
- $O(1)$ constant
- $O(\log n)$ logarithmic
- $O(n)$ linear
- $O(n \log n)$ log linear
- $O(n^2)$ quadratic
- $O(n^3)$ cubic
- $O(2^n)$ exponential



How work increases

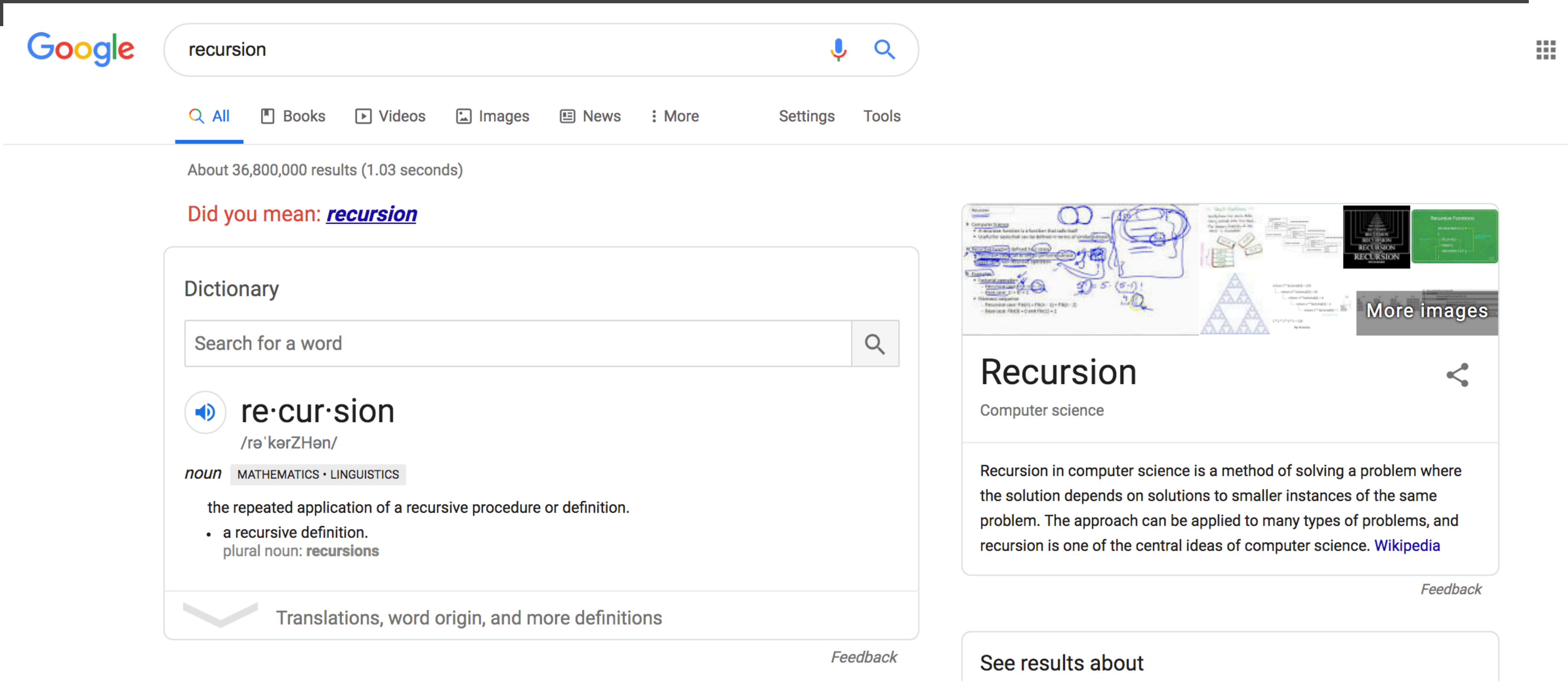
Input Size	$O(n)$	$O(n^2)$	$O(n^3)$	$O(2^n)$
2	2	4	8	4
4	4	16	64	16
8	8	64	512	256
16	16	256	4096	65536
32	32	1024	32768	4294967296
1000	1000	1000000	1000000000	10715086071862673209484250490600018105614048117055336074437503883703510511249361224931983788156958581275946729175531468251871452856923140435984577574698574803934567774824230985421074605062371141877954182153046474983581941267398767559165543946077062914571196477686542167660429831652624386837205668069376

Recursion



The Loopless Loop


PROBLEMS WITH RECURSION ?
↓Please take one↓
WHAT THE-
PROBLEMS WITH RECURSION ?
↓Please take one↓
smbc-comics.com


Google
recursion
All
Books
Videos
Images
News
More
Settings
Tools
About 36,800,000 results (1.03 seconds)
Did you mean: recursion
Dictionary
Search for a word
re·cur·sion
/rəˈkərZHən/
noun MATHEMATICS • LINGUISTICS
the repeated application of a recursive procedure or definition.
a recursive definition.
plural noun: recursions
Translations, word origin, and more definitions
Feedback
More images
Recursion
Computer science
Recursion in computer science is a method of solving a problem where the solution depends on solutions to smaller instances of the same problem. The approach can be applied to many types of problems, and recursion is one of the central ideas of computer science. Wikipedia
Feedback
See results about

Recursion

- Algorithmically:
  - Take a problem and solve it by reducing it to a simpler/smaller version of the same problem
- In programming:
  - A technique where a function calls itself
  - A **recursive function** is one that **calls itself**.

Recursion

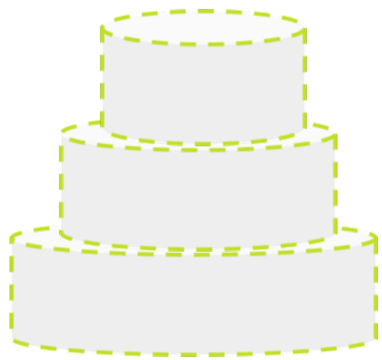
- Algorithmically:
  - Take a problem and reduce it to a simpler/smaller version of the same problem

- In programming:
  - A technique where a function calls itself
  - 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
```

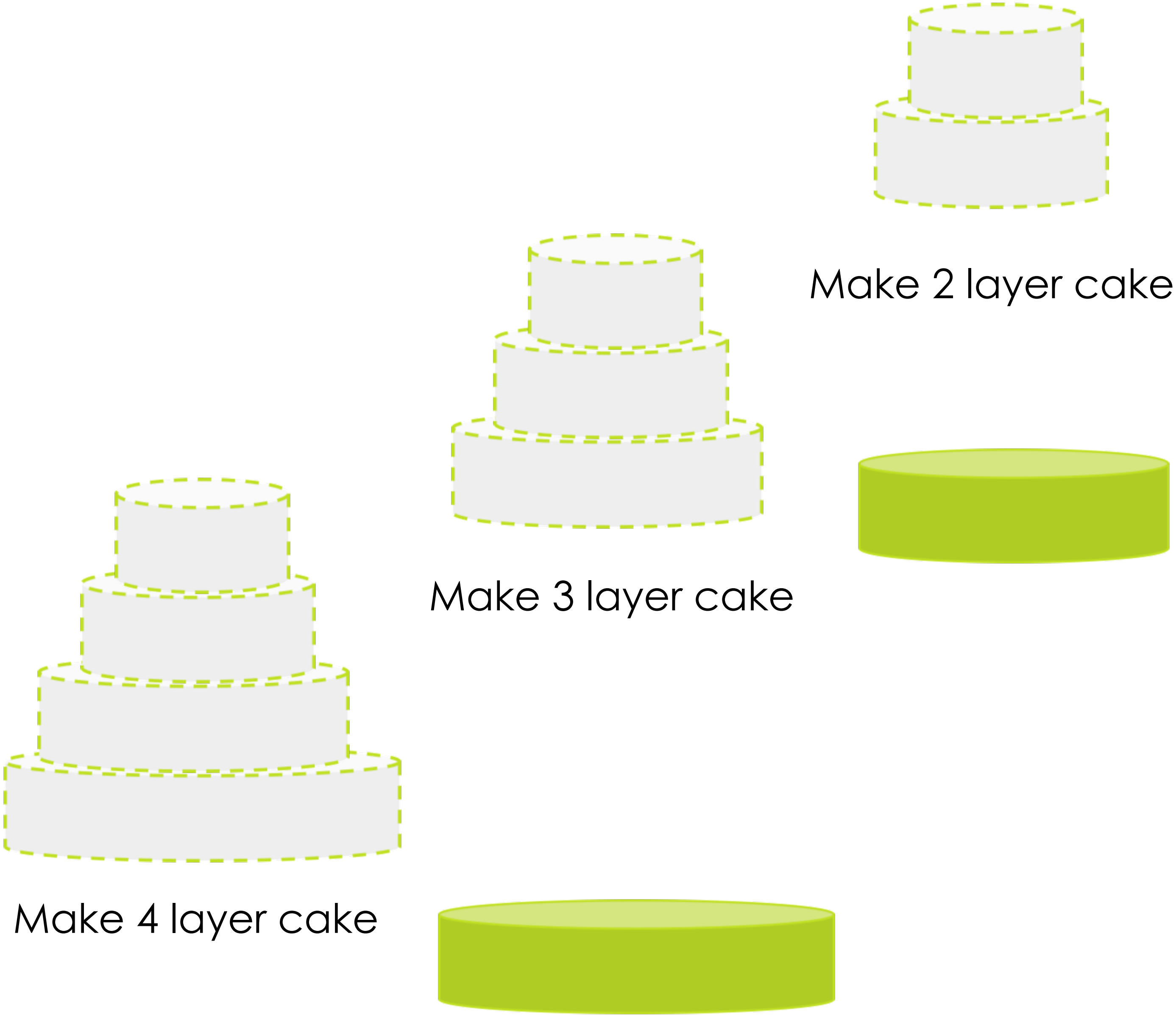
- Infinite loop? Not necessarily, not if `next(x)` needs less work than `x`.

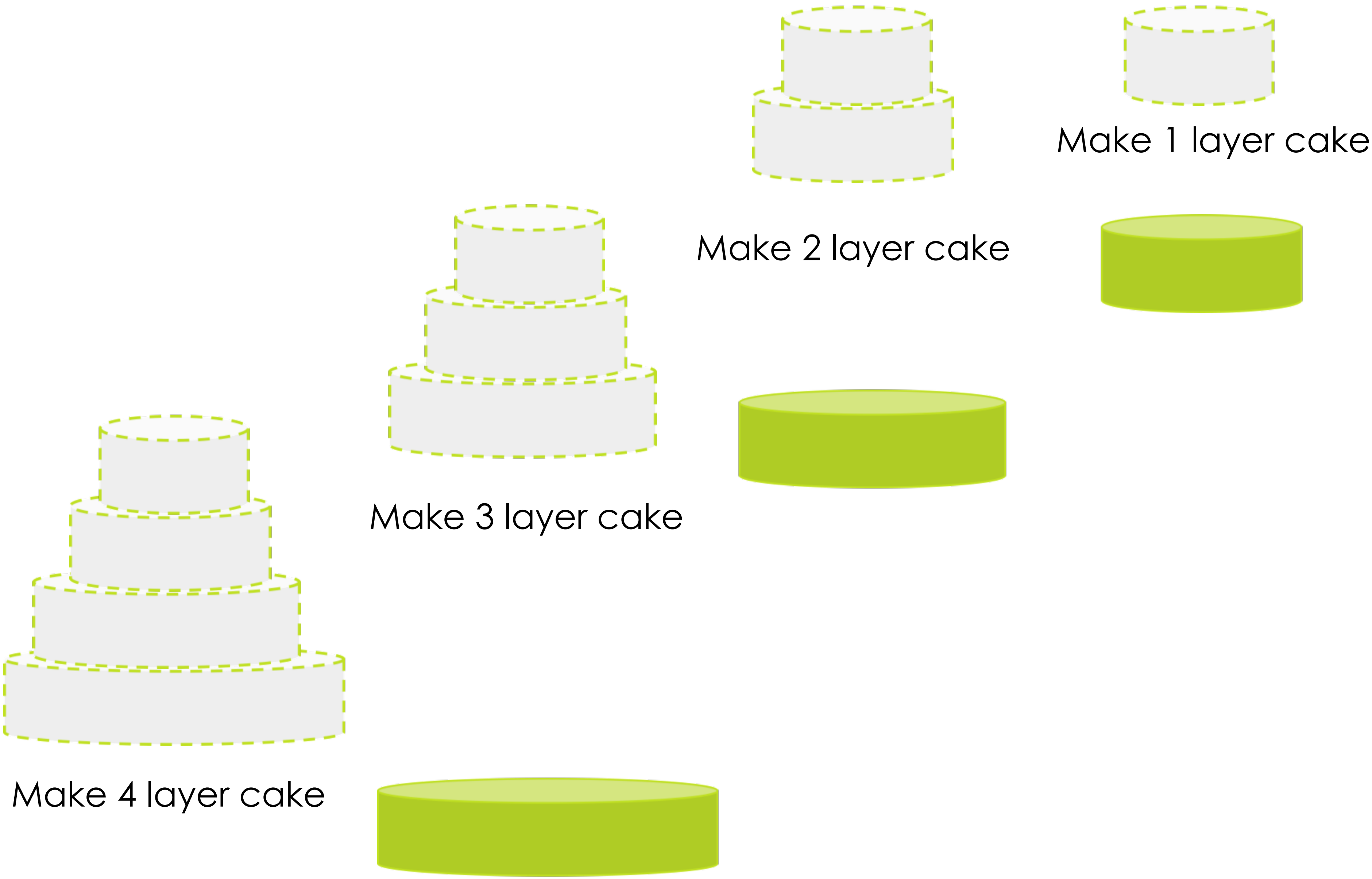
Make 4 layer cake

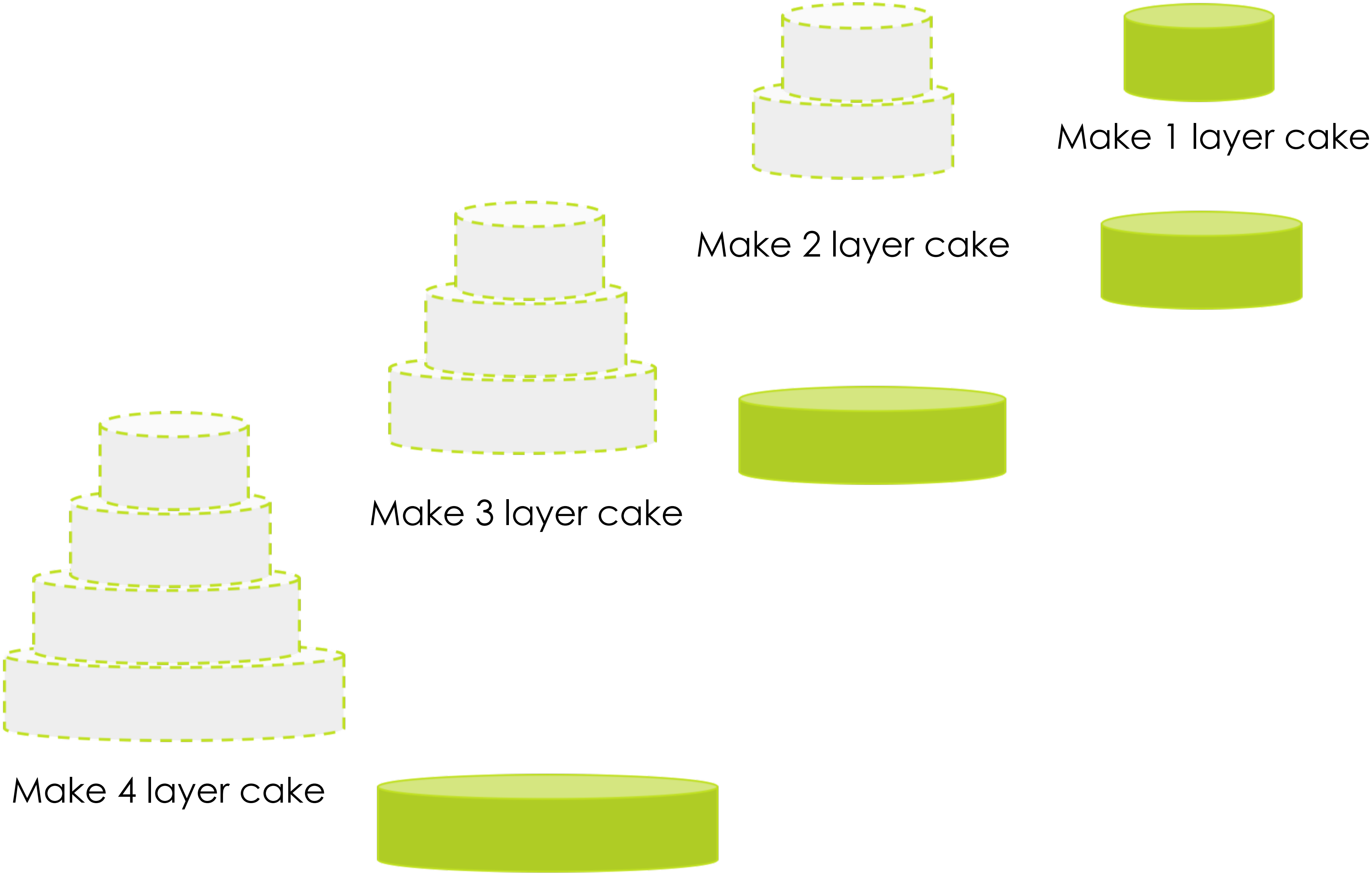


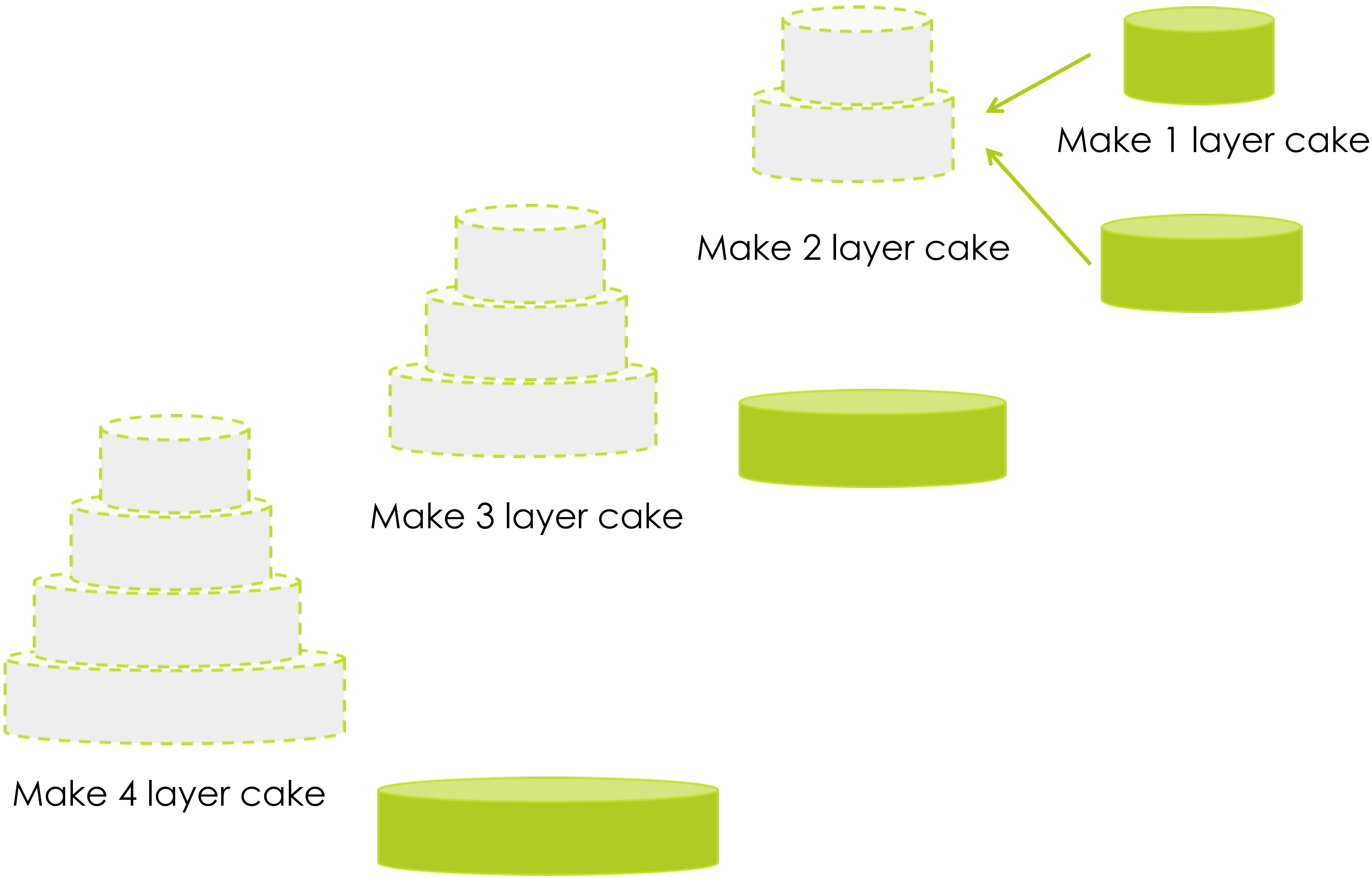
Make 3 layer cake

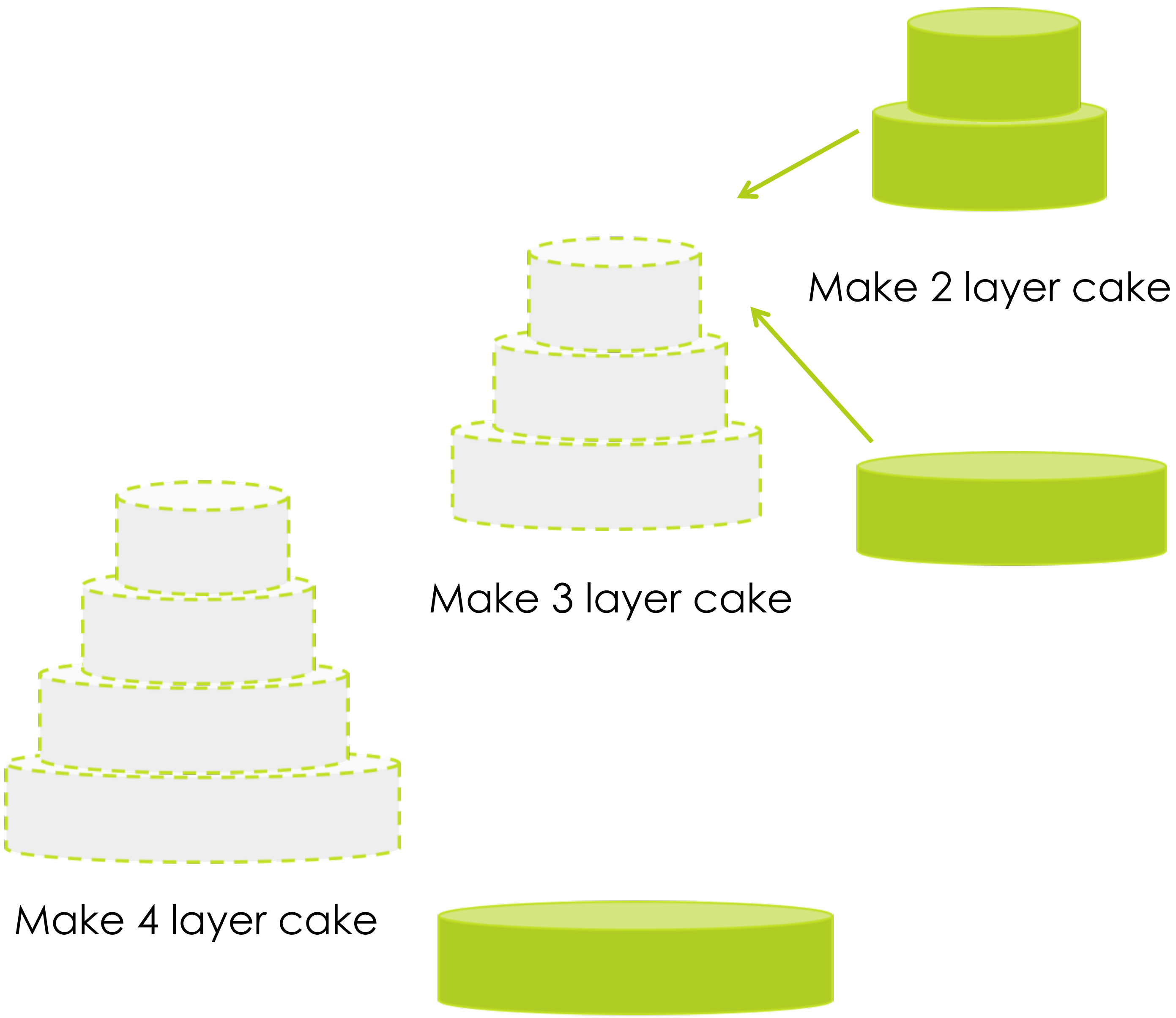
Make 4 layer cake

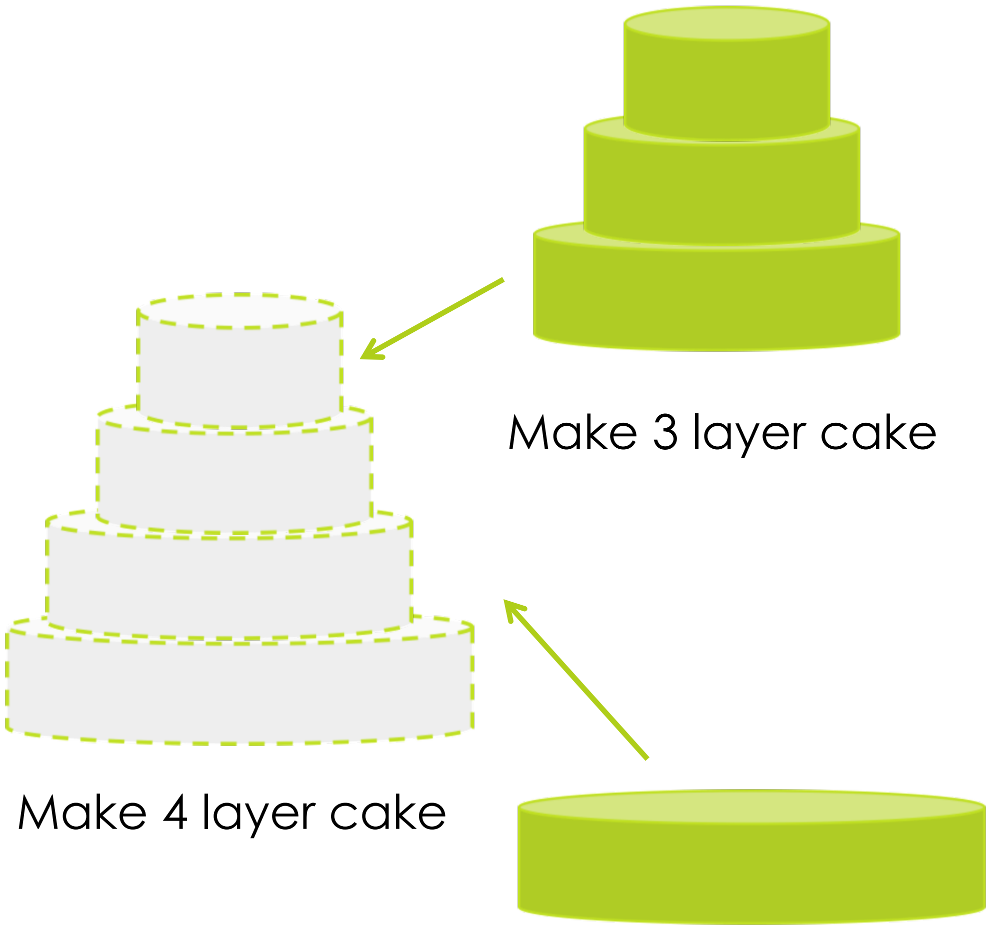

Make 2 layer cake
Make 3 layer cake
Make 4 layer cake


Make 1 layer cake
Make 2 layer cake
Make 3 layer cake
Make 4 layer cake


Make 1 layer cake
Make 2 layer cake
Make 3 layer cake
Make 4 layer cake


Make 1 layer cake
Make 2 layer cake
Make 3 layer cake
Make 4 layer cake

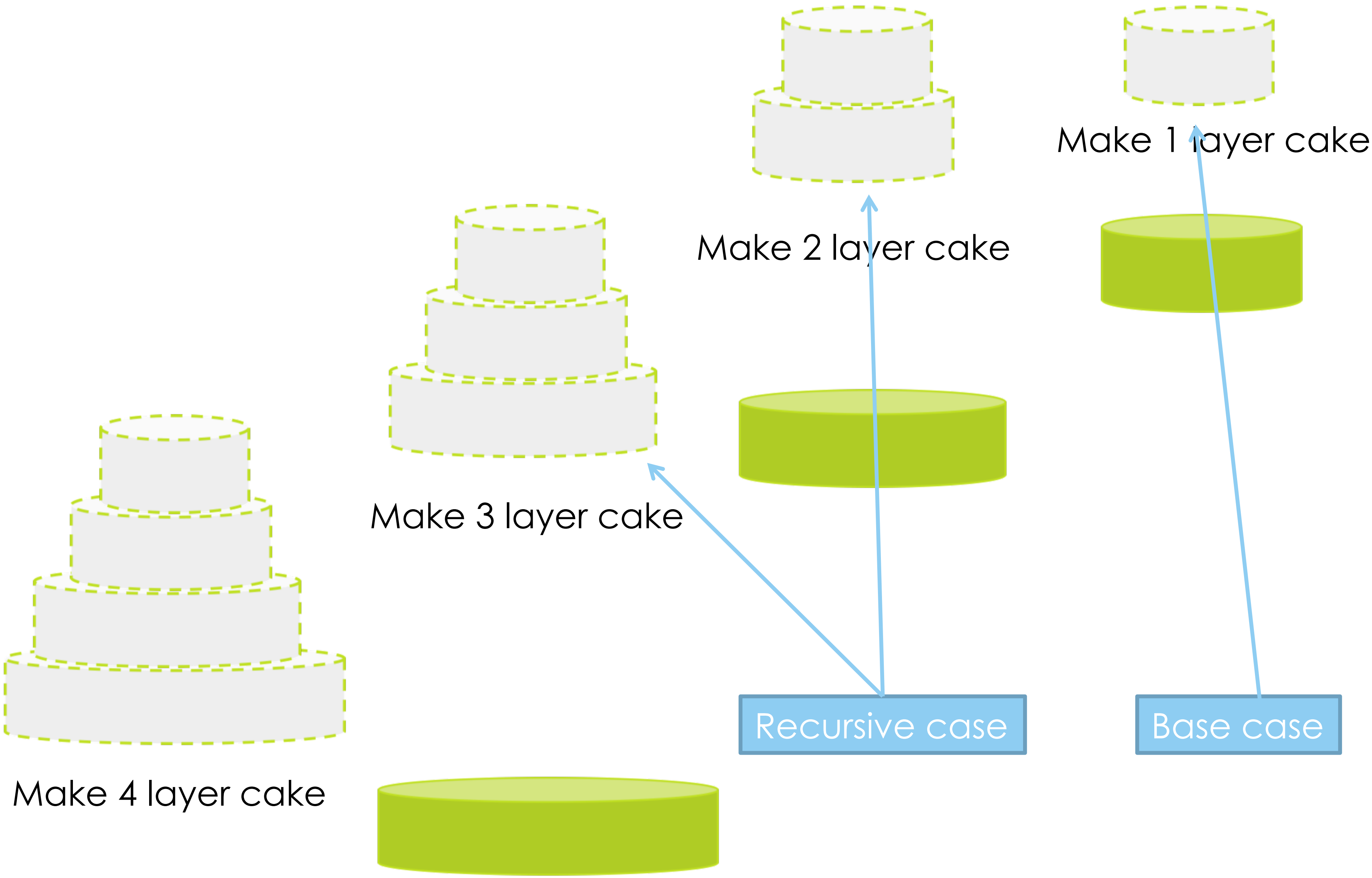

Make 2 layer cake
Make 3 layer cake
Make 4 layer cake


Make 3 layer cake
Make 4 layer cake

Make 4 layer cake

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".


Make 1 layer cake
Make 2 layer cake
Make 3 layer cake
Make 4 layer cake
Recursive case
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

$4! = 4(3!)$

$3! = 3(2!)$

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

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

Compute the base case

make smaller instances
of the same problem

build up
the result

Recursion conceptually

$4! = 4(3!)$

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

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

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

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?

Recursion in Python

Recursive Factorial in Python

- For what `n` do we know the factorial?

`n = 0` →

```
if n == 0:
    return 1
```

Recursive Factorial in Python

- For what `n` do we know the factorial?

```
n = 0        →    if n == 0:
                      return 1
```

- How do we reduce the problem? Rewrite in terms of something simpler each time

```
n*(n-1)!     →    else:
                      return n * factorial(n-1)
```

Recursive Factorial in Python

- For what `n` do we know the factorial?

```
n = 0        →    if n == 0:        # base case
                      return 1
```

- How do we reduce the problem? Rewrite in terms of something simpler each time

```
n*(n-1)!     →    else:             # recursive case
                      return n * factorial(n-1)
```

Recursive Factorial in Python

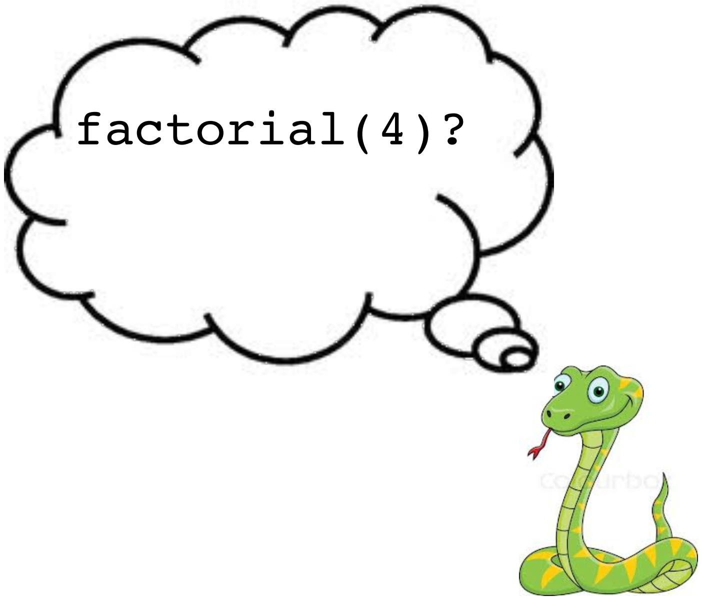
$0! = 1$	(Base case)
$n! = n \times (n-1)!$	(Recursive case)

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

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

S
T
A
C
K

n=4



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

S
T
A
C
K

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

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

S T A C K

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

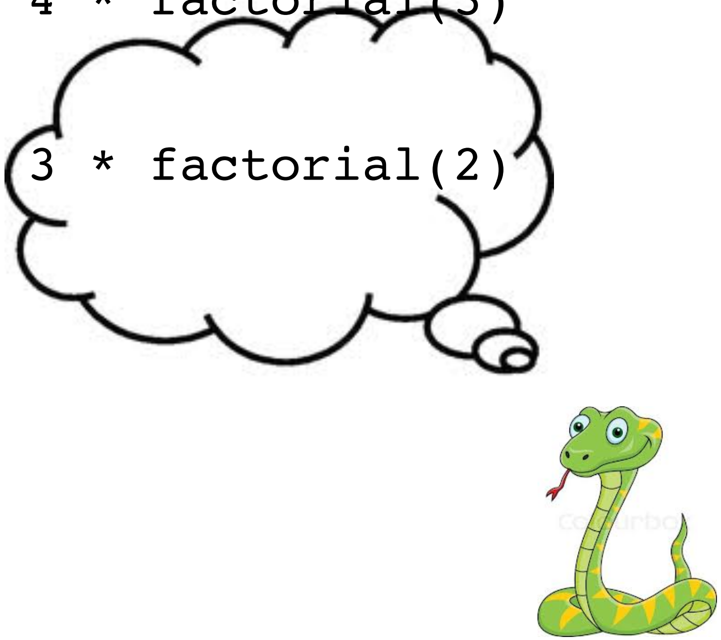
n=3   factorial(3)?

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

S
T
A
C
K

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

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



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

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

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

A n=2 factorial(2)?

C

K

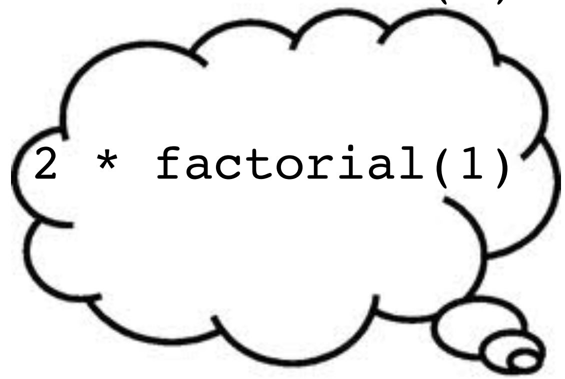
```python
def factorial(n):
    if n == 0:      # base case
        return 1
    else:           # recursive case
        return n * factorial(n-1)
```

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)`



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

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)?	

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

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)

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

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

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

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

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

S
T
A
C
K

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

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

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

S
T
A
C
K

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

Recursion vs Iteration

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 two ways

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

```
# Recursive version of factorial
  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

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]
```

the "tail" of list `a`

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]
```

the "tail" of list `a`

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]
```

the "tail" of list a

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

Tracing `sumlist`

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

Tracing `sumlist`

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

Tracing `sumlist`

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

Tracing `sumlist`

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

Tracing `sumlist`

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

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

Tracing `sumlist`

```
>>> 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.

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!

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

Fibonacci Numbers

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

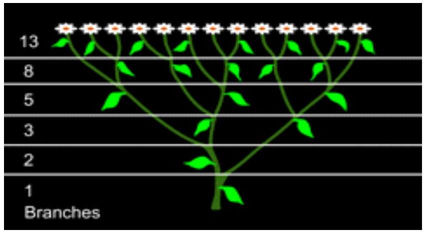
Fibonacci Numbers

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

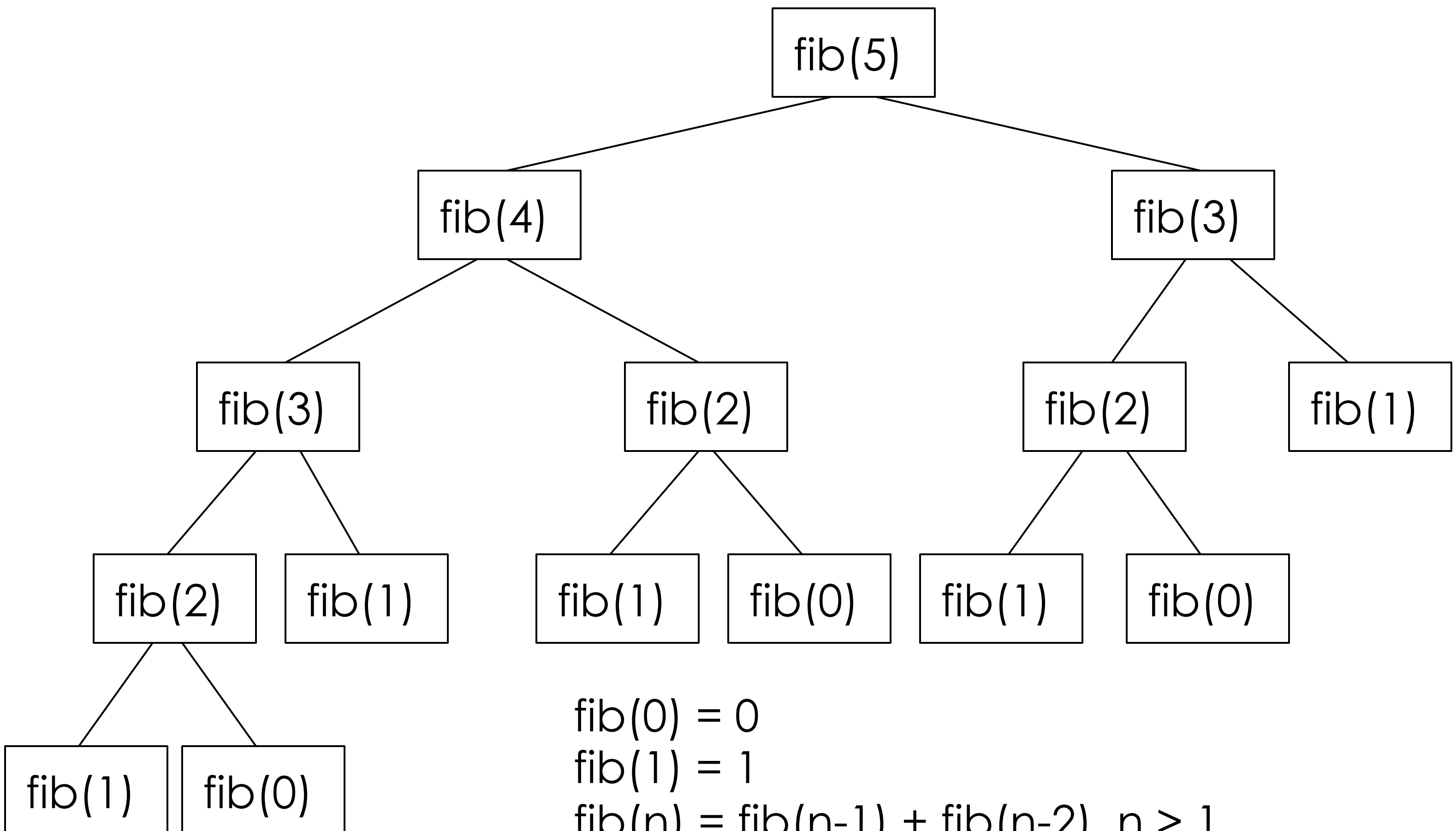
Recursive Fibonacci

- Let fib(n) = the nth Fibonacci number, $n \geq 0$
  - $fib(0) = 0$ (base case)
  - $fib(1) = 1$ (base case)
  - $fib(n) = fib(n-1) + fib(n-2)$, $n > 1$

Two recursive calls!

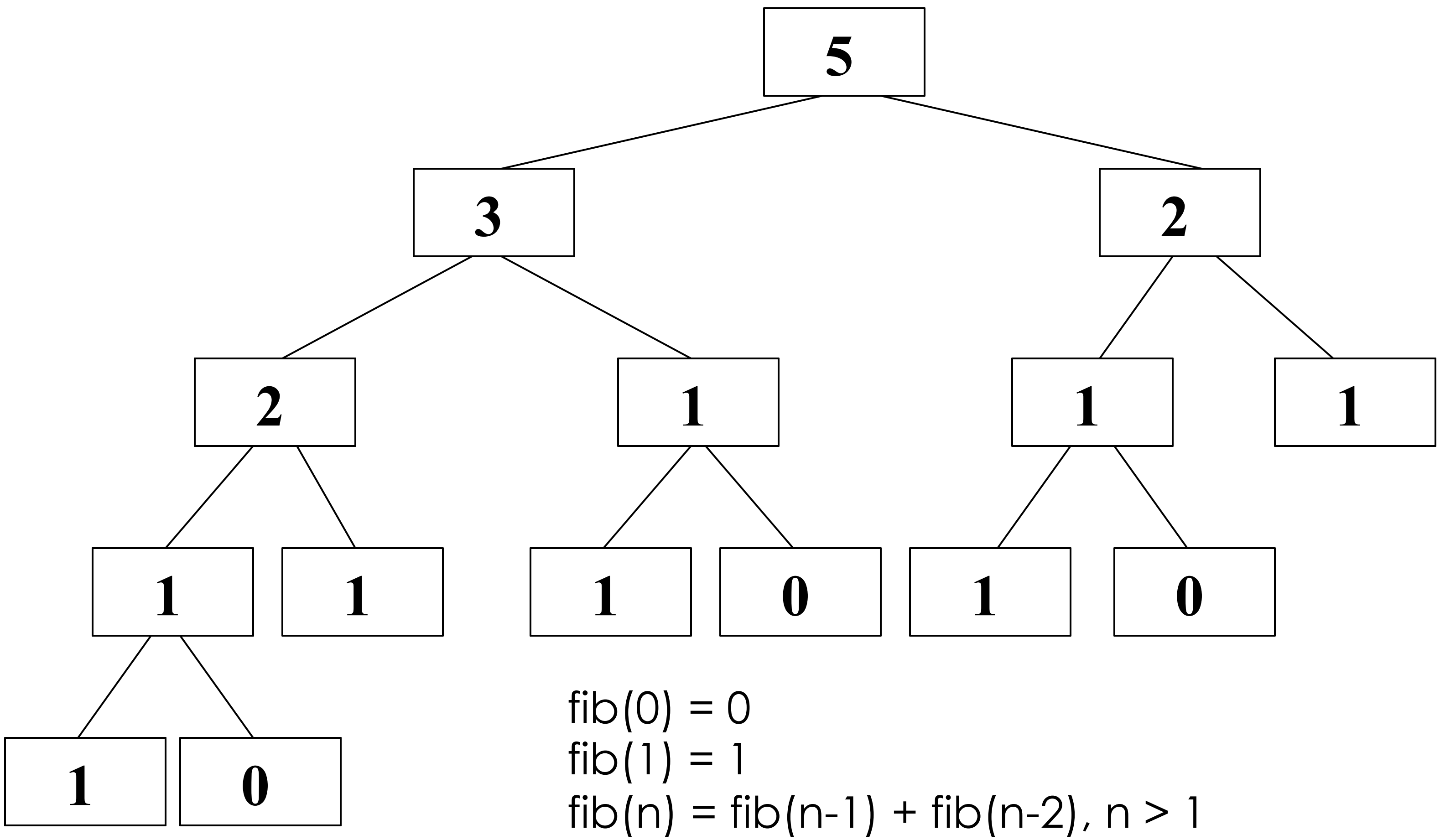
```
def fib(n):
    if n == 0 or n == 1:
        return n
    else:
        return fib(n-1) + fib(n-2)
```

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
```

sequence:

0
1
1
2
3
5
8
13
...

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
```

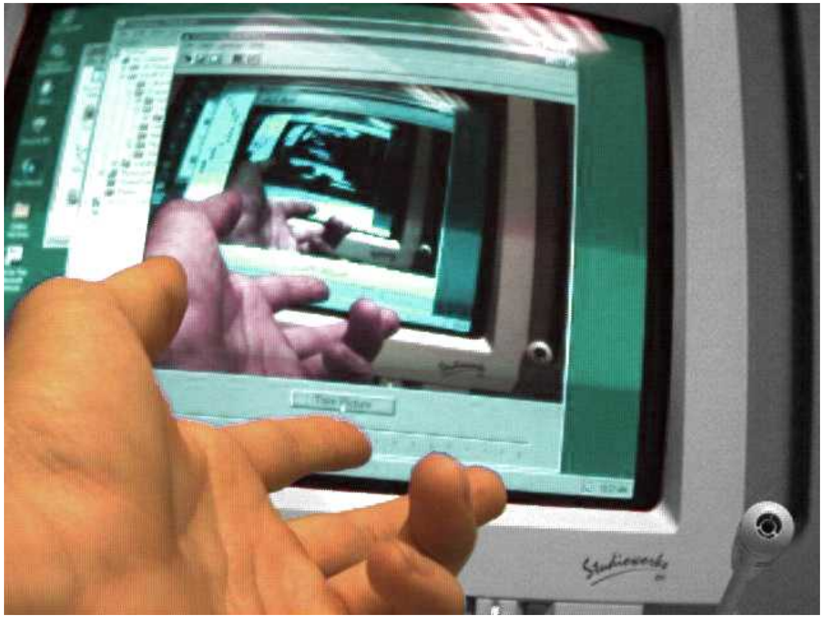
SIMULTANEOUS ASSIGNMENT

Faster than the recursive version. Why?

Fractals: More on Recursion

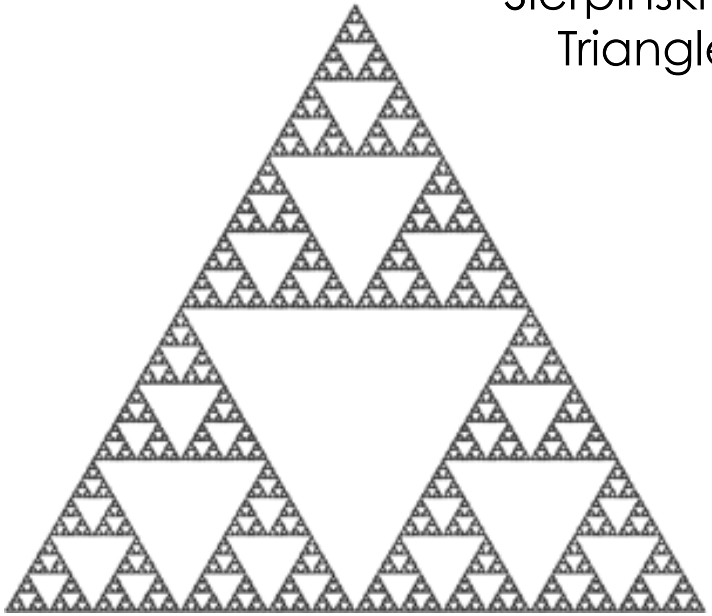
Geometric Recursion (Fractals)

- A recursive operation performed on successively smaller regions.

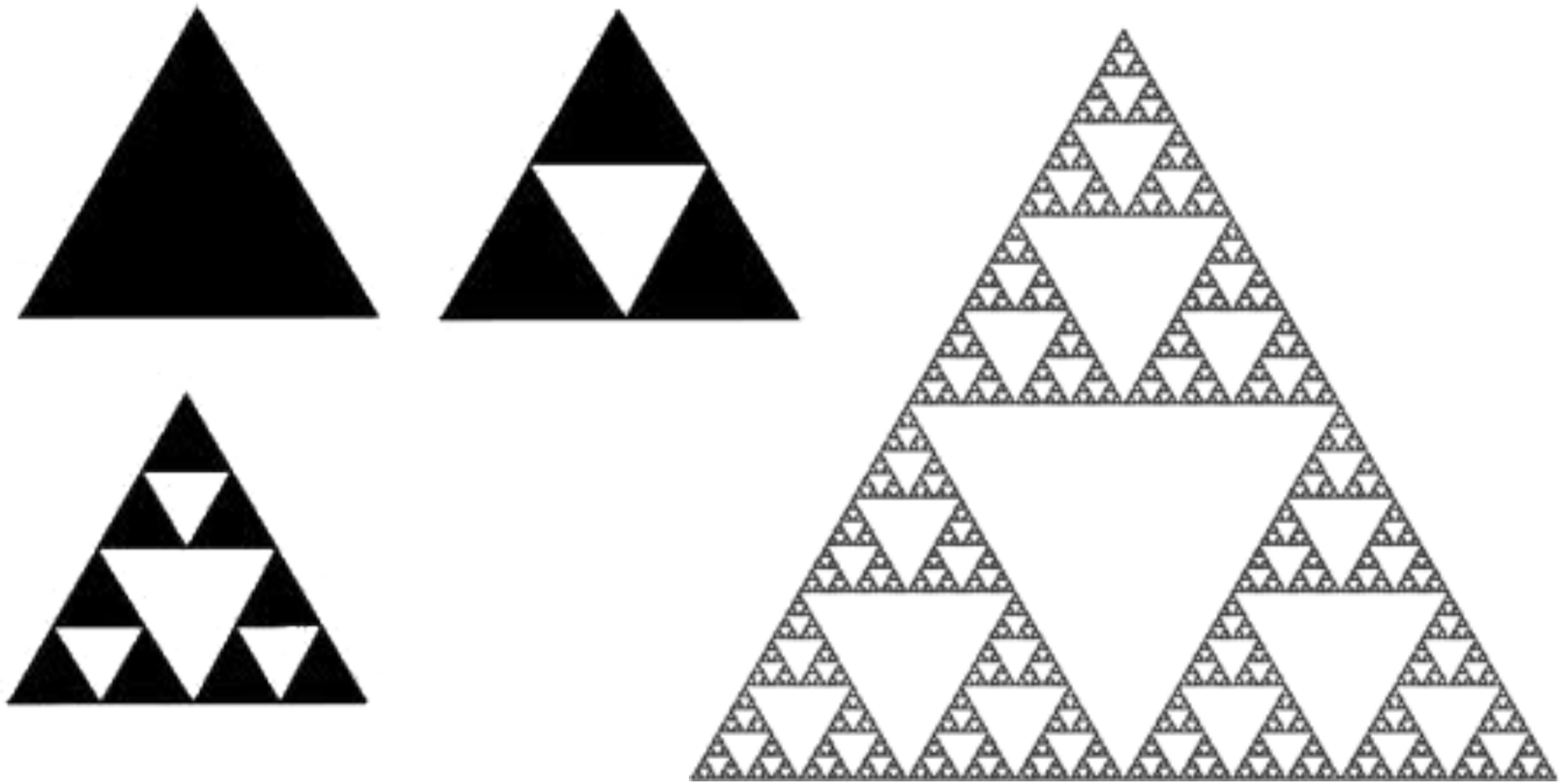


http://fusionanomaly.net/recursion.jpg

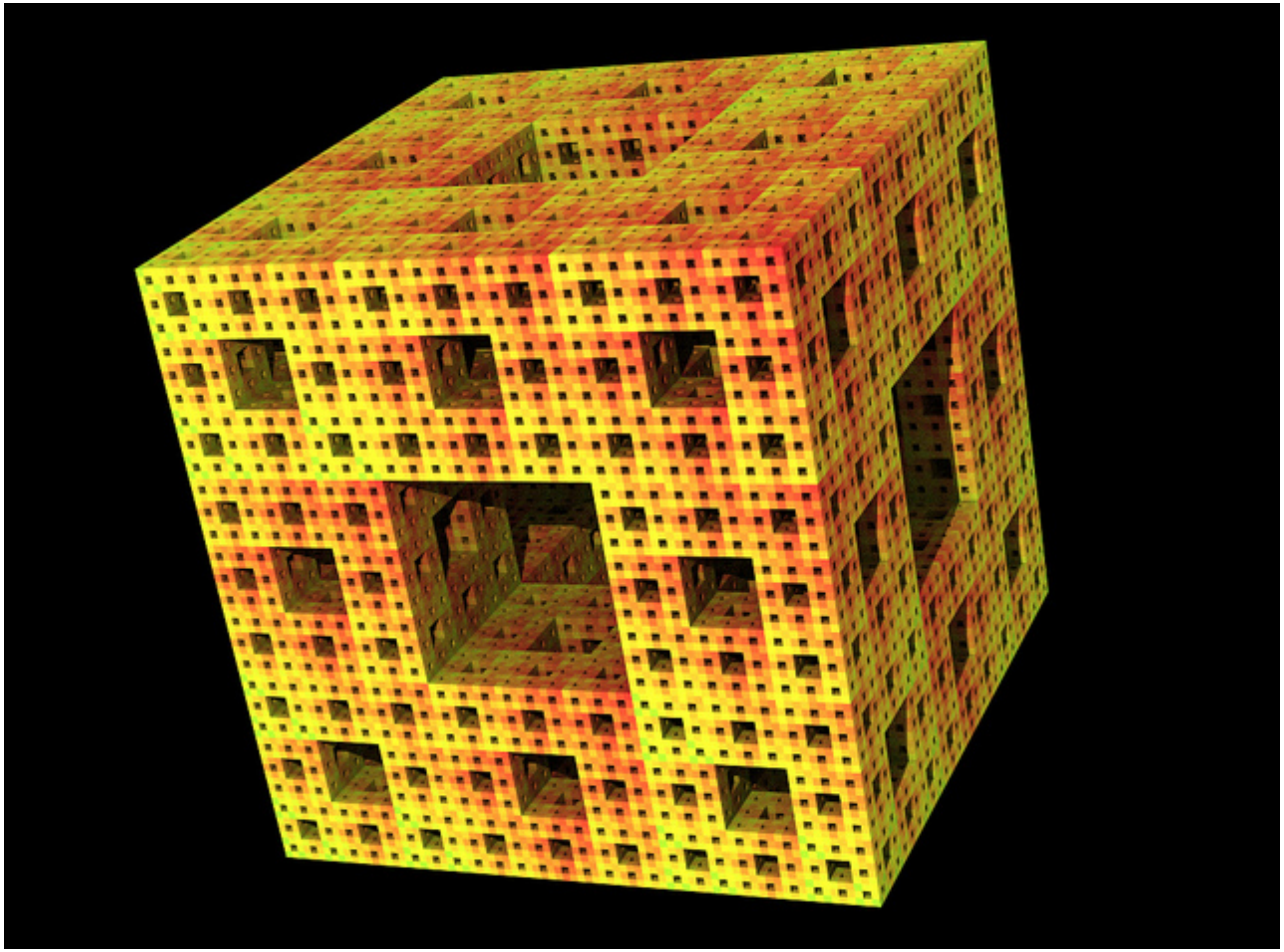
Sierpinski's Triangle



Sierpinski's Triangle

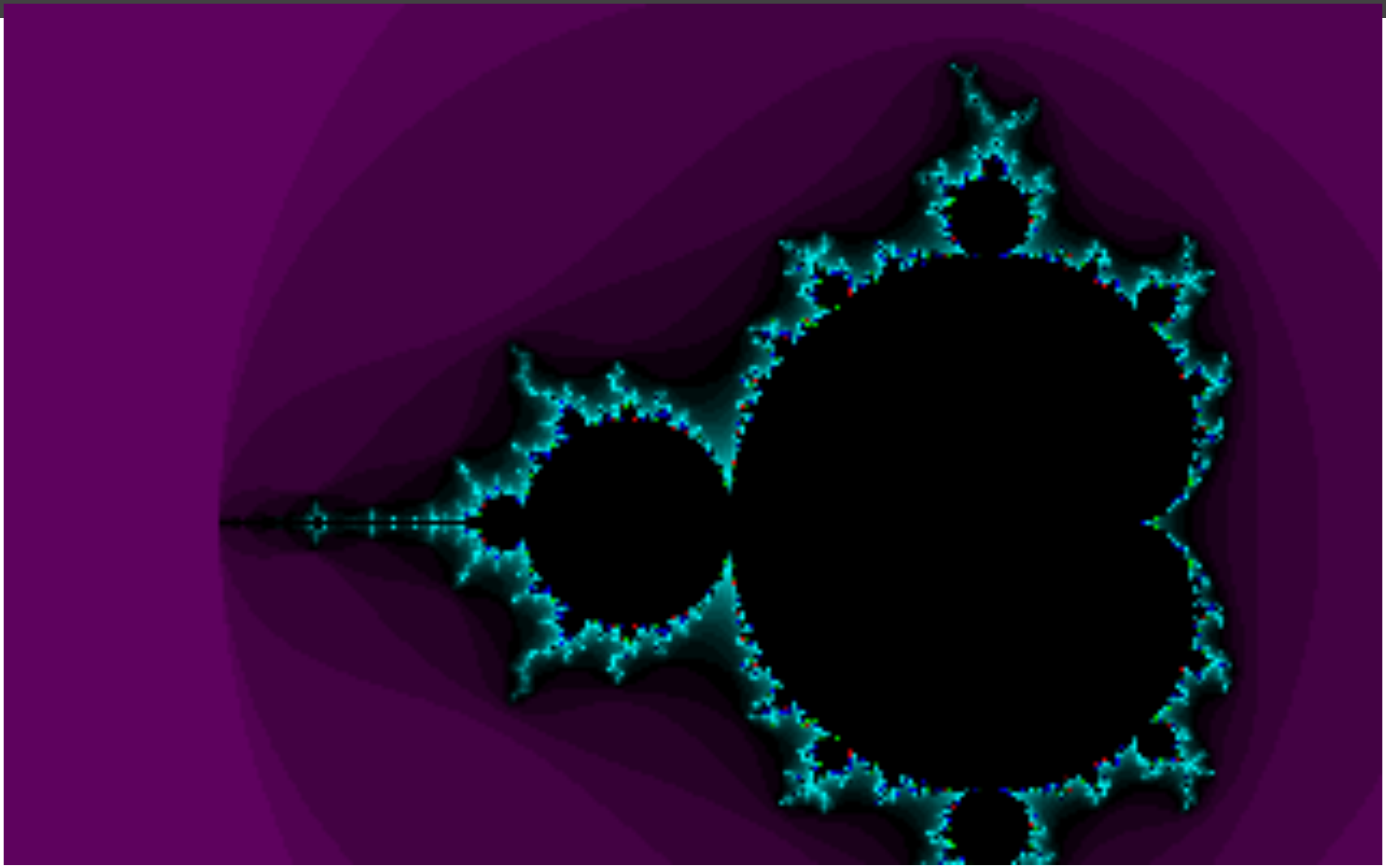


Sierpinski's Carpet



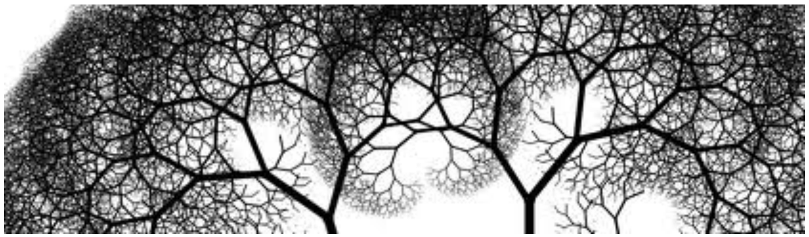
(the next slide shows an animation that could give some people headaches)

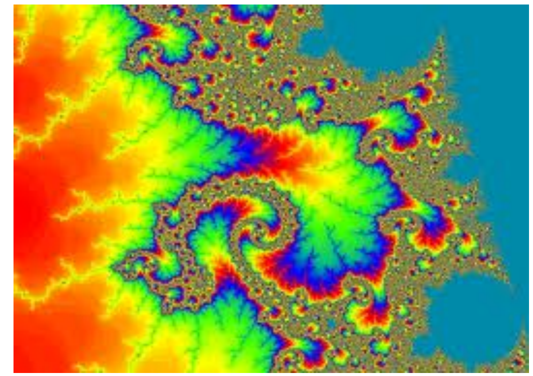
Mandelbrot set

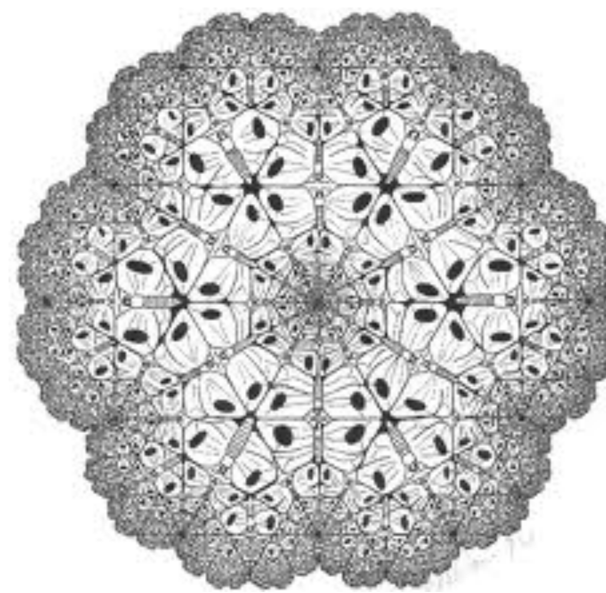


Source: Clint Sprott, http://sprott.physics.wisc.edu/fractals/animated/

Fancier fractals







Next Lecture

recursion for search

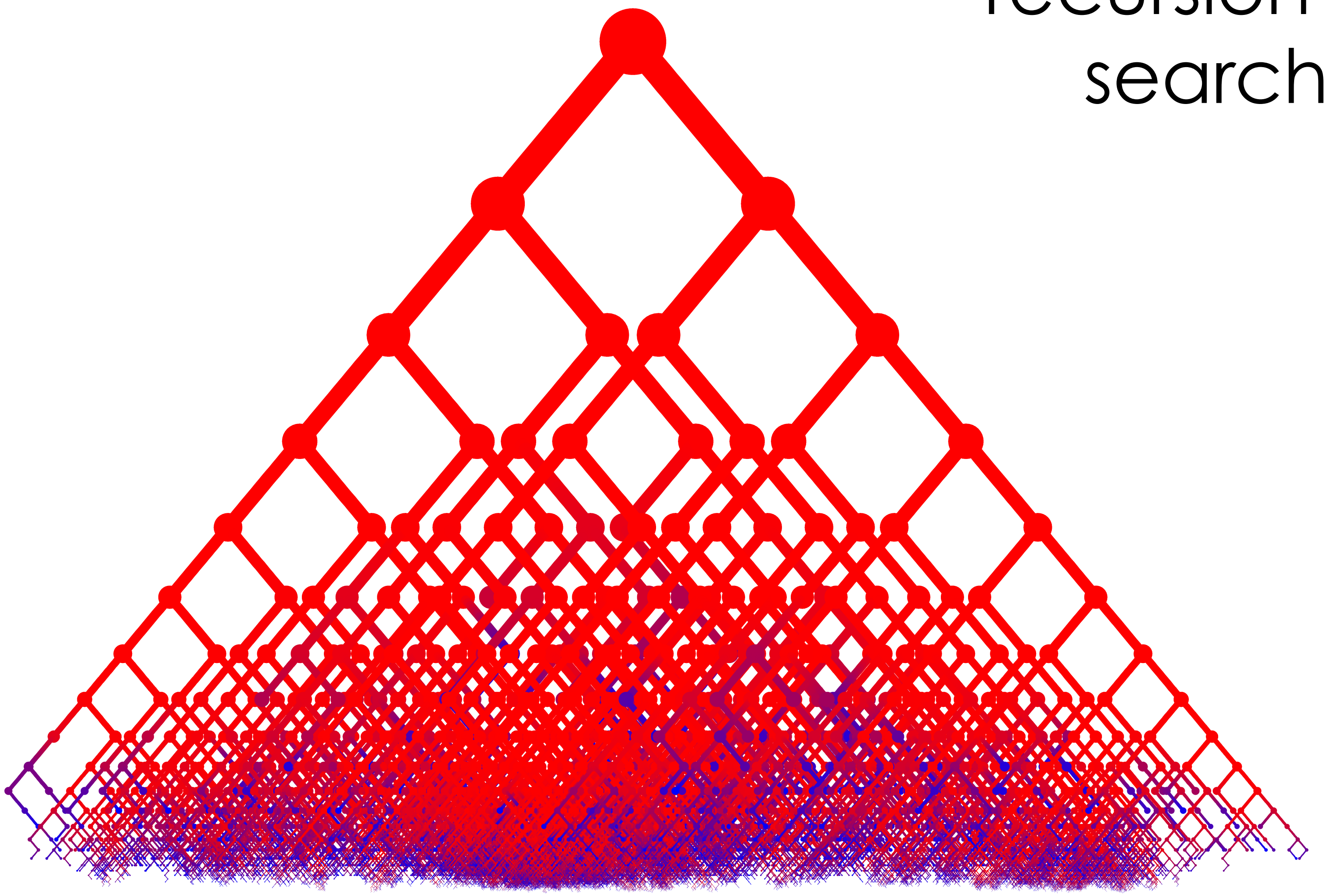


image: Matt Roberts, http://people.bath.ac.uk/mir20/blogposts/bst_close_up.php