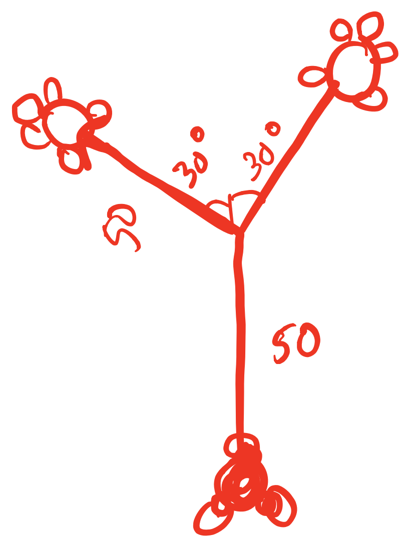
Recursion

a.k.a., CS's version of mathematical induction

As close as CS gets to magic

Reminder: Final Exam Tues (3/19) 4p - 7p.

in Chem 1171

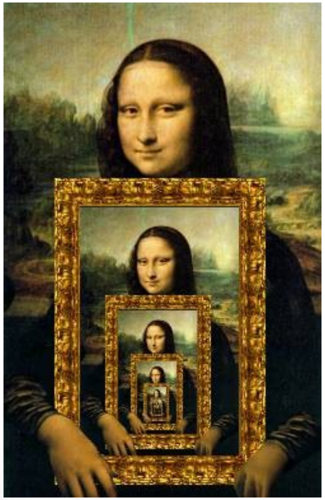


Example - drawing a tree using recursion

See code written in lecture

Let recursion draw you in….

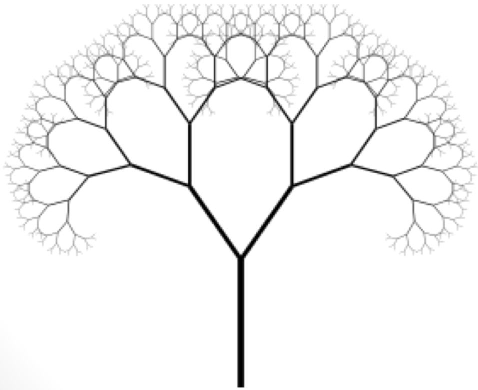
- Recursion occurs when something is described in terms of itself
- Describe these pictures



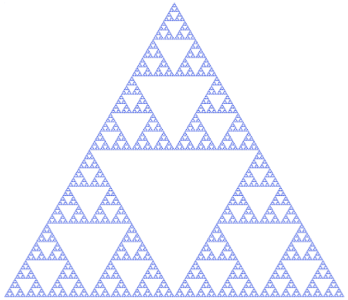
Recursion !

- General idea: Solve problems by describing it in terms of a smaller version of itself

- Applications:
  Fractals, advanced data structures, file systems



Tree



Sierpinski triangle

Koch's snowflake

Function ***design***

Thinking ***sequentially***

factorial

```
5!  =  120

5!  =  5 * 4 * 3 * 2 * 1

N!  =  N * (N-1) * (N-2) * … * 3 * 2 * 1
```

3! = 3 x 2 x 1 = 6

4! = 4 x 3 x 2 x 1

= 4 * 3!

Recursive
description of

Thinking *recursively*

your proble

```
N!  =  N * (N-1)! , if  N > 1
    =  1, if N <= 1
```

Recursion == ***self***-reference!

Designing Recursive Functions

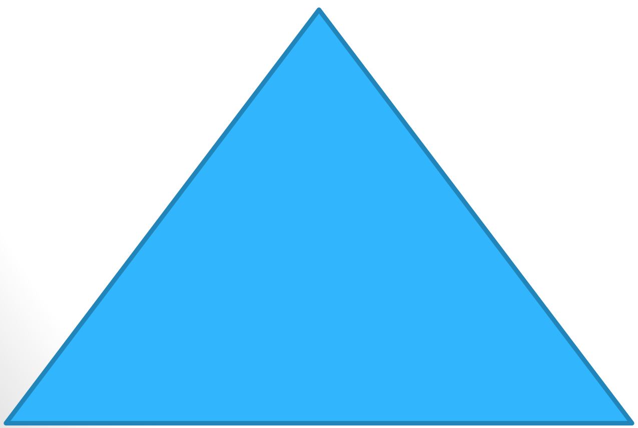
```python
def fac(N):
    if N <= 1:
        return 1
```

Base case:

Solution to inputs where the answer is trivial

(top of the pyramid)

Base case: N <= 1



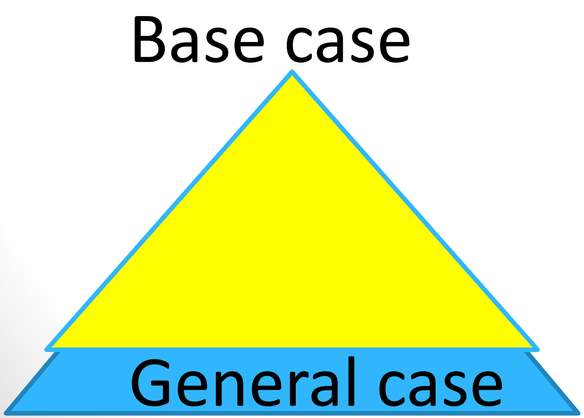
General case: N>1

Designing Recursive Functions

```
def fac(N):
    if N <= 1:
        return 1
```

Base case

```
    else: # solve for any N
        rest = fac(N-1)
```



Designing Recursive Functions

```
def fac(N):

    if N <= 1:
        return 1

    else:
        rest = fac(N-1)
        return rest * N
```

Base case

Recursive case

Human: Base case and 1 step

Computer: Everything else

Thinking recursively !

```
def fac(N):

    if N <= 1:
        return 1

    else:
        return N*fac(N-1)
```

Base case

Recursive case (shorter)

Human: Base case and 1 step

Computer: Everything else

Warning: *this is legal!*

```
def fac(N):
    return N * fac(N-1)
```

legal **!=** *recommended*

```
def fac(N):
    return N * fac(N-1)
```

No ***base case*** -- the calls to **`fac`** will never stop!

Make sure you have a **base case**, *then* worry about the recursion...

How functions ***work***…

I might have a guess…

Three functions:

What is **`demo(-4)`** ?

```
def demo(x):
    return x + f(x)

def f(x):
    return 11*g(x) + g(x/2)

def g(x):
    return -1 * x
```

How functions work…

```
def demo(x):
    return x + f(x)

def f(x):
    return 11*g(x) + g(x/2)

def g(x):
    return -1 * x

  >>> demo(-4) ?
```

```
demo
  x = -4
  return -4 + f(-4)
```

How functions work…

```
def demo(x):
    return x + f(x)

def f(x):
    return 11*g(x)+g(x/2)

def g(x):
    return -1 * x
```

```
demo
  x = -4
  return -4 + f(-4)
```

```
f
  x = -4
  return 11*g(x) + g(x/2)
```

```
>>> demo(-4)
```
?

How functions work…

```
def demo(x):
    return x + f(x)

def f(x):
    return 11*g(x)+g(x/2)

def g(x):
    return -1 * x
```

```
demo
  x = -4
  return -4 + f(-4)
```

```
f
  x = -4
  return 11*g(x) + g(x/2)
```

These are different **x**'s !

```
>>> demo(-4)  ?
```

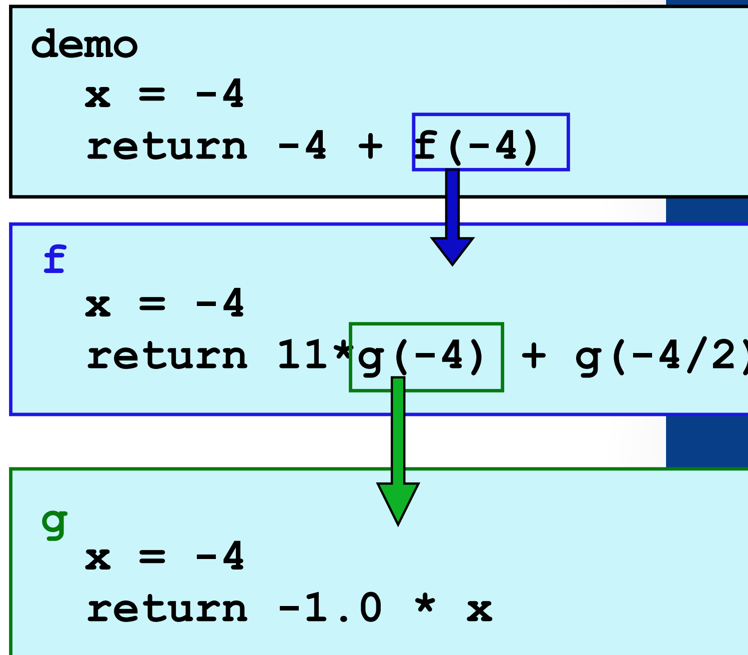
How functions work…

```
def demo(x):
    return x + f(x)

def f(x):
    return 11*g(x)+g(x/2)

def g(x):
    return -1 * x

>>> demo(-4) ?
```



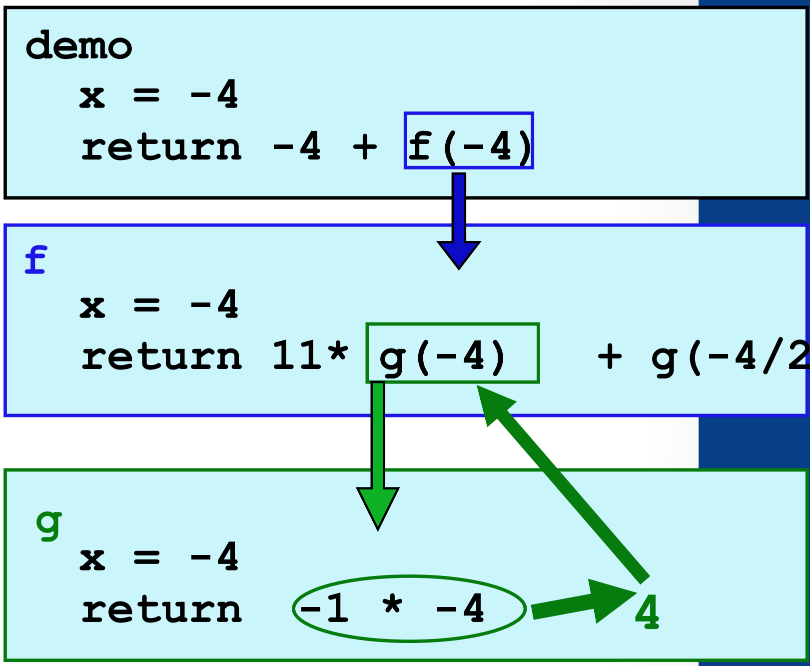
How functions work…

```
def demo(x):
    return x + f(x)

def f(x):
    return 11*g(x)+g(x/2)

def g(x):
    return -1 * x

 >>> demo(-4)  ?
```



How functions work…

```
def demo(x):
    return x + f(x)

def f(x):
    return 11*g(x)+g(x/2)

def g(x):
    return -1 * x
```

```
demo
  x = -4
  return -4 + f(-4)
```

```
f
  x = -4
  return 11* 4  + g(-4/2)
```

`>>> demo(-4)` ?

What happens next in program memory?

A. f() returns, its local variables are removed from memory

B. g() is called, new local variable (x) is created in memory

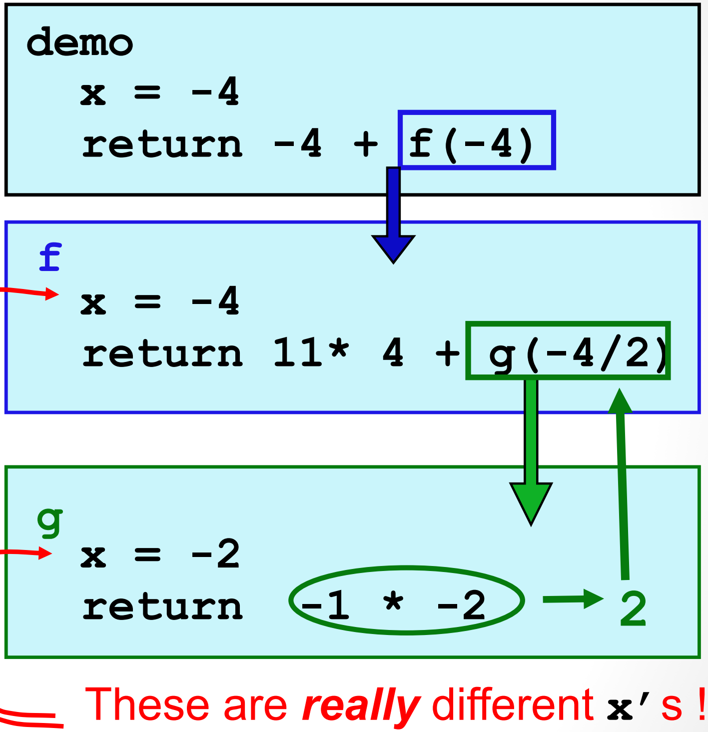
How functions work…

```
def demo(x):
    return x + f(x)


def f(x):
    return 11*g(x) + g(x/2)


def g(x):
    return -1 * x
```

```
>>> demo(-4)  ?
```



How functions work…

```
def demo(x):
    return x + f(x)

def f(x):
    return 11*g(x) + g(x/2)

def g(x):
    return -1 * x
```

```
demo
  x = -4
 return -4 + f(-4)
```

```
f
  x = -4
 return 11* 4  + 2
```

```
>>> demo(-4)
```
?

How functions work…

```
def demo(x):
    return x + f(x)

def f(x):
    return 11*g(x) + g(x/2)

def g(x):
    return -1 * x
```

```
demo
  x = -4
 return -4 + f(-4)
```

```
f
  x = -4
 return 11* 4  + 2
```

46

```
>>> demo(-4)
```

?

How functions work…

```
def demo(x):
    return x + f(x)

def f(x):
    return 11*g(x) + g(x/2)

def g(x):
    return -1 * x
```

```
demo
  x = -4
  return -4 +  46
```

```
>>> demo(-4)
42
```

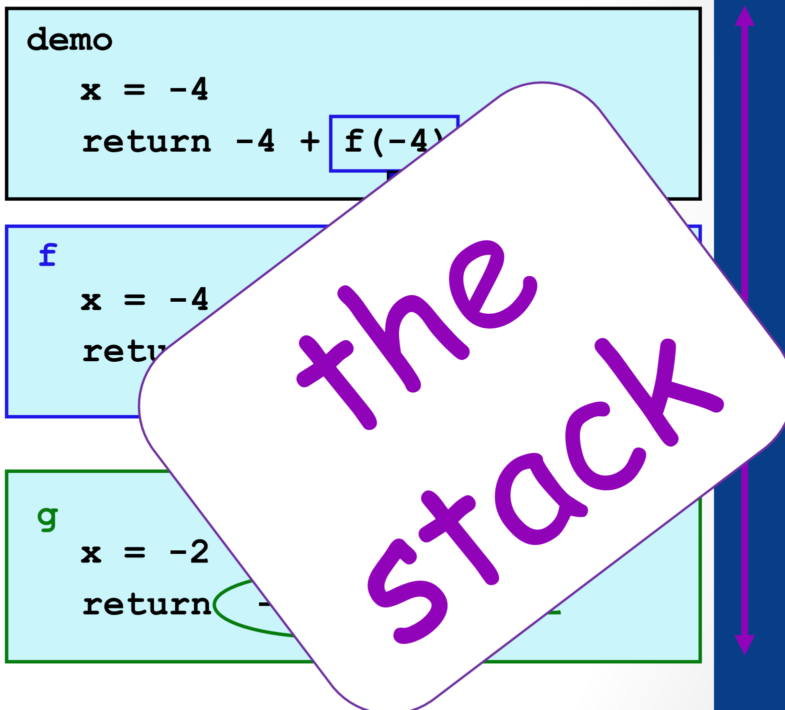
42

Function *stack*ing

```
def demo(x):
    return x + f(x)

def f(x):
    return 11*g(x) + g(x/2)

def g(x):
    return -1 * x
```



"The stack..."

(1) keeps separate variables for each function call...

(2) remembers where to send results back to...


```
def fac(N):
    if N <=1:
      return 1
    return fac(N)
```

Roadsigns and recursion

examples of self-fulfilling danger

Behind the curtain…

```
def fac(N):

    if N <= 1:
        return 1

    else:
        return N * fac(N-1)
```

```
>>> fac(1)
```

Result: 1

The base case is **No Problem**!

Behind the curtain…

```
def fac(N):

    if N <= 1:
        return 1

    else:
        return N * fac(N-1)
```

```
fac(5)
```

Behind the curtain…

```
def fac(N):

    if N <= 1:
        return 1

    else:
        return N * fac(N-1)
```

$$\underbrace{\texttt{fac(5)}}_{\texttt{5 * fac(4)}}$$

Behind the curtain…

```
def fac(N):

    if N <= 1:
        return 1

    else:
        return N * fac(N-1)
```

```
fac(5)

5 * fac(4)

    4 * fac(3)
```

Behind the curtain…

```
def fac(N):

    if N <= 1:
        return 1

    else:
        return N * fac(N-1)
```

```
fac(5)

5 * fac(4)

     4 * fac(3)

          3 * fac(2)
```

Behind the curtain…

```python
def fac(N):

    if N <= 1:
        return 1

    else:
        return N * fac(N-1)
```

```
fac(5)
5 * fac(4)
    4 * fac(3)
        3 * fac(2)
            2 * fac(1)
```

Behind the curtain…

```
def fac(N):

    if N <= 1:
        return 1

    else:
        return N * fac(N-1)
```

"The Stack"

Remembers all of the individual calls to `fac`

```
fac(5)
5 * fac(4)
    4 * fac(3)
        3 * fac(2)
            2 * fac(1)
                  1
```

Behind the curtain…

```
def fac(N):

    if N <= 1:
        return 1

    else:
        return N * fac(N-1)
```

```
fac(5)

5 * fac(4)

    4 * fac(3)

        3 * fac(2)

            2 *   1
```

Behind the curtain…

```python
def fac(N):

    if N <= 1:
        return 1

    else:
        return N * fac(N-1)
```

```
fac(5)

5 * fac(4)

    4 * fac(3)

        3 *   2
```

Behind the curtain…

```
def fac(N):

    if N <= 1:
        return 1

    else:
        return N * fac(N-1)
```

```
fac(5)

5 * fac(4)

      4 *   6
```

Behind the curtain…

```
def fac(N):

    if N <= 1:
        return 1

    else:
        return N * fac(N-1)
```

$$\overbrace{5\ *\ \ 24}^{\texttt{fac(5)}}$$

Behind the curtain…

```
def fac(N):

    if N <= 1:
        return 1

    else:
        return N * fac(N-1)
```

```
fac(5)
```

Result: 120

Let recursion do the work for you.

Exploit self-similarity

Produce short, elegant code

Less work !

Let recursion do the work for you.

Exploit self-similarity

Produce short, elegant code

Less work !

```
def fac(N):
    if N <= 1:
        return 1
    else:
        rest = fac(N-1)
        return rest * N
```

You handle the base case – the easiest case!

Recursion does <u>almost</u> all of the rest of the problem!

You specify one step progress towards the base case

But you ***do*** need to do one step yourself…

```
def fac(N):

    if N <= 1:
        return 1
    else:
        return fac(N)
```

This will not work !