Recursion

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

As close as CS gets to magic

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

```
>>> 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) ?
```

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

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

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)
```
?

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

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

```
g
  x = -4
  return -1.0 * 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)
```

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

```
g
  x = -4
  return  -1 * -4      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) ?
```

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

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

```
g
  x = -2
  return  -1 * -2  →  2
```

These are ***really*** different `x`'s !

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
```

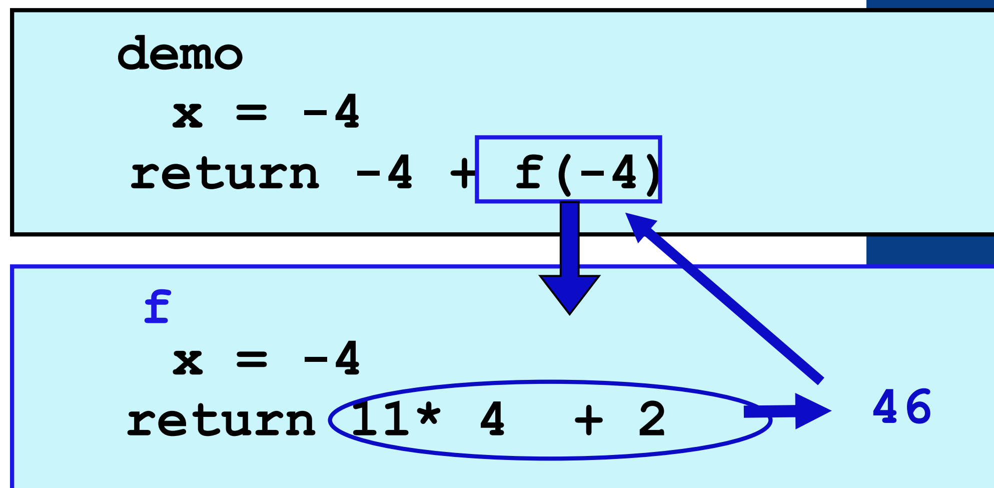
```
demo
  x = -4
  return -4 + f(-4)
```

```
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(-4)  ?
```

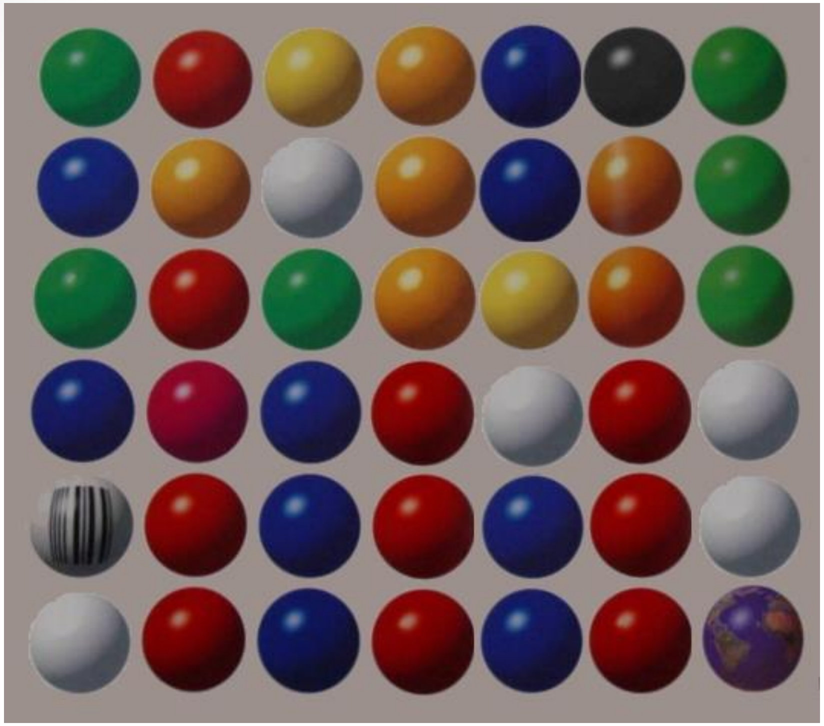
How functions work…

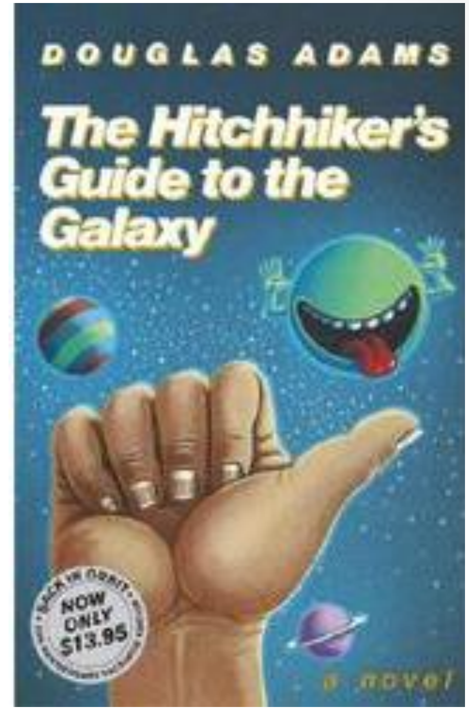
```
def demo(x):
    return x + f(x)

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





Douglas Adams's 42 puzzle

answer: 42 **question:** unknown

The Ultimate Answer

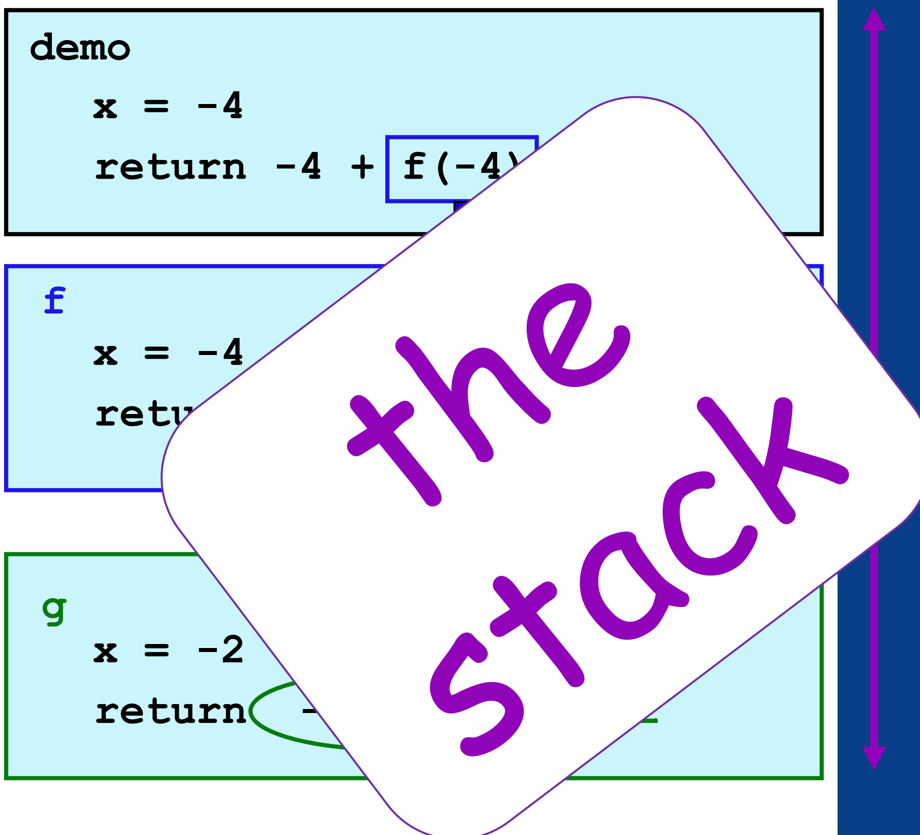
[edit]

According to *The Hitchhiker's Guide to the Galaxy*, researchers from a pan-dimensional, hyper-intelligent race of beings constructed the second greatest computer in all of time and space, Deep Thought, to calculate the Ultimate Answer to Life, the Universe, and Everything. After seven and a half million years of pondering the question, Deep Thought provides the answer: "forty-two." The reaction:

> *"Forty-two!" yelled Loonquawl. "Is that all you've got to show for seven and a half million years' work?"*
> *"I checked it very thoroughly," said the computer, "and that quite definitely is the answer. I think the problem, to be quite honest with you, is that you've never actually known what the question is."*

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...

Function ***design***

Thinking *sequentially*

factorial

```
5!  =  120

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



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

Thinking *recursively*

factorial

```
5!  =  120
5!  =  5 * 4 * 3 * 2 * 1
5!  =
```

Recursion ==
self-reference!

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

Warning: *this is legal!*

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

What happens when `fac(4)` is called?

A. It returns the correct result (24)
B. The execution never stops
C. It produces a return value that is incorrect

I wonder how this code will STACK up!?

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...

The base case!

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

For which N is the result of `fac(N)` trivial to calculate?

A. 1
B. 6
C. 20
D. 100


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

Roadsigns and recursion

examples of self-fulfilling danger

Thinking recursively !

```
def fac(N):

    if N <= 1:
        return 1
```

Base case

Thinking recursively !

```
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

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)
```

```
def fac(N):

    if N <= 1:
        return 1

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

Behind the curtain…

```
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…

```
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)
```

```
fac(5)

5 *   24
```

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.

$$\left.\begin{array}{c}\text{Exploit self-similarity}\\[1ex]\text{Produce short, elegant code}\end{array}\right\}\ \textbf{Less work !}$$

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 !

Which two key properties of recursive functions does this code satisfy?

```
def count(n):
  print(n)
  count(n-1)
```

A. Base case exists
B. Recursive case makes progress towards the ase case
C. Both
D. Neither

What is the output of count(3)?

```
def count(n):
    if n < 0:
        return
    count(n-1)
    print(n)
```

A. 3 2 1 0 (each number printed on a new line)
B. 0 1 2 3 (each number printed on a new line)
C. Error or no output because the result of recursive call is not returned
D. Execution never stops because the function does not satisfy one of the two properties of recursive functions