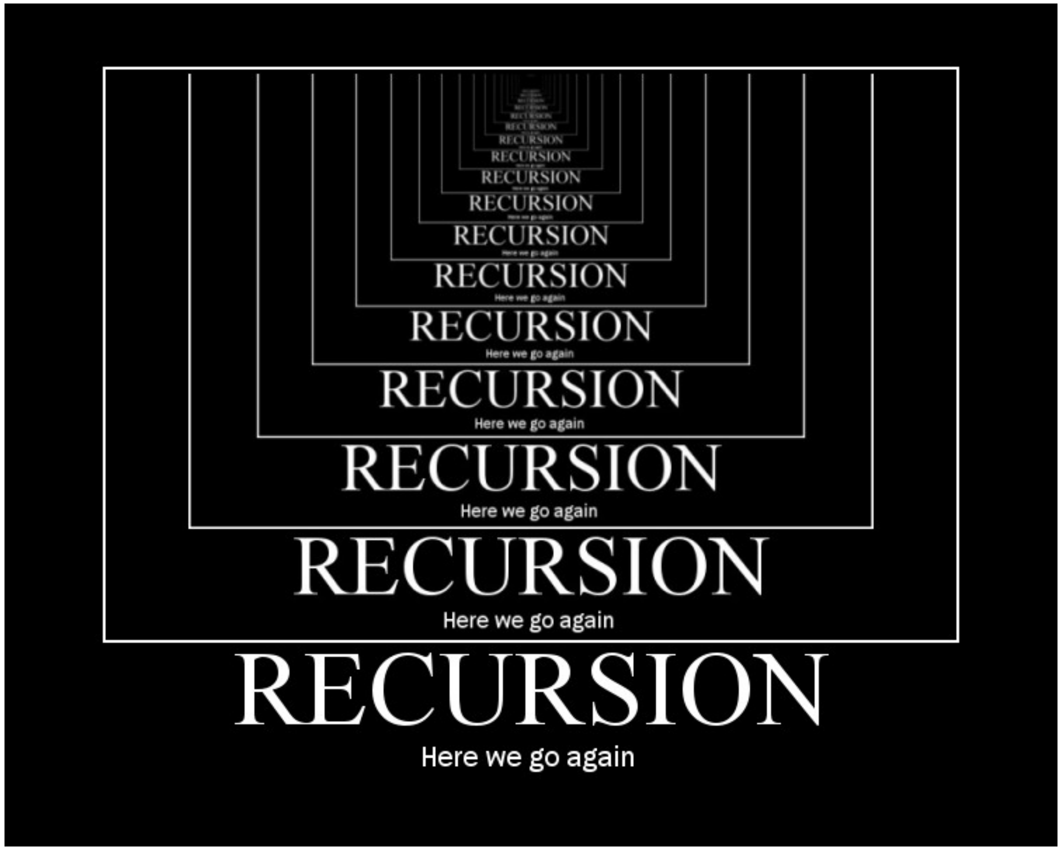
Recursion



ASLAM SAFLA
ASLAM@CS.UCT.AC.ZA

Concept: Recursion

It was a dark and stormy night, and the head of the brigands said to Antonio: "Antonio, tell us a tale". And so Antonio began:

"It was a dark and stormy night and the head of the brigands said to Antonio, "Antonio, tell us a tale". And so Antonio began:

"It was a dark and stormy night and the head of the brigands said to Antonio, "Antonio, tell us a tale". And so Antonio began:

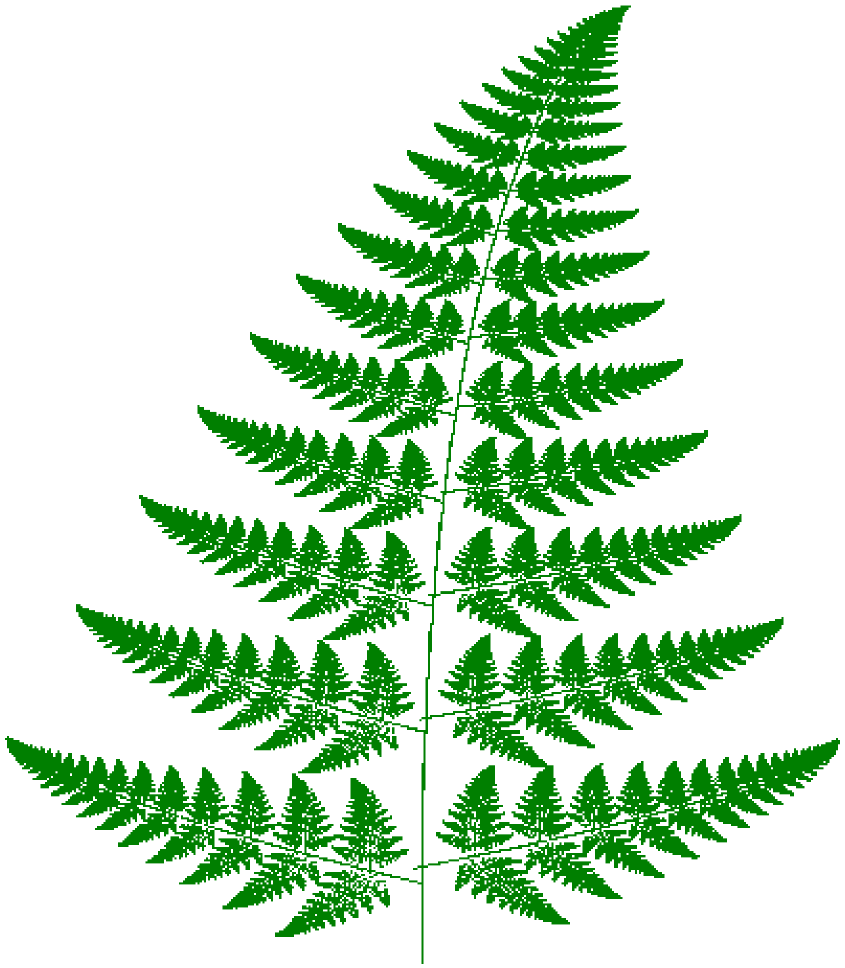
"It was a dark and stormy night and the head of the brigands said to Antonio, "Antonio, tell us a tale". And so Antonio began:

"It was a dark and stormy night and the head of the brigands said to Antonio, "Antonio, tell us a tale". And so Antonio began:

"It was a dark and stormy night and ….

Concept: Recursion

- the fern leaf



Concept: Recursion

- selfies with mirrors

Recursive definitions:

A description of something that refers to itself is called a **recursive definition**.

Examples?

Recursive definitions:

A description of something that refers to itself is called a **recursive definition**.

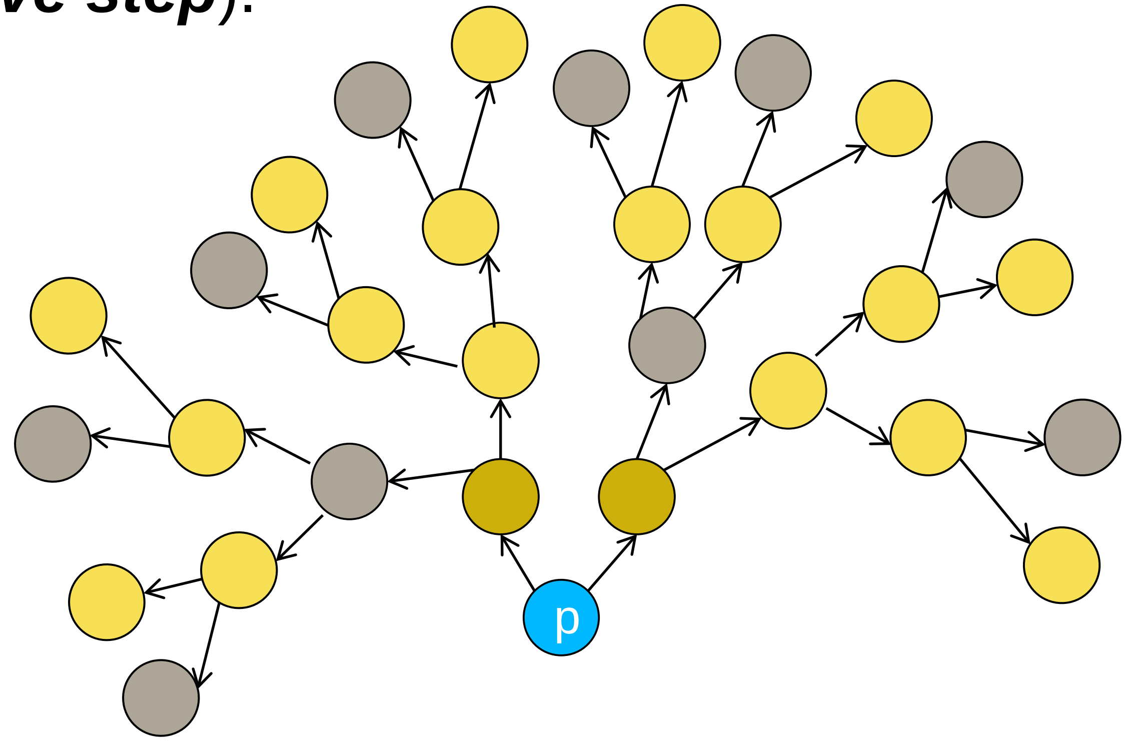
e.g

frabjuous: An adjective used to describe something that is **frabjuous**.

Recursive definitions

A recursive definition of the ancestors of person *p*:

- *p*'s parents are *p*'s ancestors (***base case***);
- The parents of any ancestors of *p* are also the ancestors of *p* (***recursive step***).



Recursive definitions:

A description of something that refers to itself is called a **recursive definition**.

Another example?

Recursive definitions:

A description of something that refers to itself is called a **recursive definition**.

e.g

The **set of prime numbers** can be defined as the unique set of positive integers satisfying:

- 1 is not a prime number
- any other positive integer is a prime number if and only if it is not divisible by any **prime number** smaller than itself

Cute, but…

…why is recursion useful to us?

Recursive functions

A function can ***call itself***.

A function that does this is a **recursive function.**

```
def brigand():
    print("It was a dark and stormy night, and the head of the brigands said to Antonio:“Antonio, tell us a tale”. And so Antonio began:")
    brigand()

brigand()
```

Recursive functions

A function can ***call itself***.

A function that does this is a **recursive function.**

It may not be obvious why that is a good thing, but it turns out to be one of the most magical and interesting things a program can do.

Recursive **Functions**

A *recursive* function is a function that includes a call to itself

- based on the general problem-solving technique of breaking down a task into subtasks
  - often called “divide-and-conquer”

Recursion can be used whenever **one subtask is a smaller version of the original task**.

Defining recursive funcions

A recursive function **calls itself**

Recursive functions have 2 key elements:

- one or more **recursive calls**
- **stopping condition**, or **base case**, where no recursion is required

Recursion is the equivalent of mathematical induction!

Recursive functions

Now, with a **base case**

```
def brigand(n):
    if n==0: print("No, said Antonio.")
    else:
        print("It was a dark and stormy night, and the head of the brigands said to Antonio:“Antonio, tell us a tale”. And so Antonio began:")
        n-=1
        brigand(n)
```

Recursion: Factorial

Classic introductory example - Factorial function:

$$n! = 1 \times \ldots \times (n-1) \times n$$

$$n! = n \times (n-1)!$$ # recursive call

$$0! = 1$$ # stopping condition, or base case

Recursive Factorial function

```
def factRec(n):
    if n==0:
        return 1  #base case – ends recursion
    else:
        return n*factRec(n-1)
    #recursive call – does a little work and uses the
   results from smaller version of same problem
```

function definition – must have parameter

UNIVERSITY OF CAPE TOWN

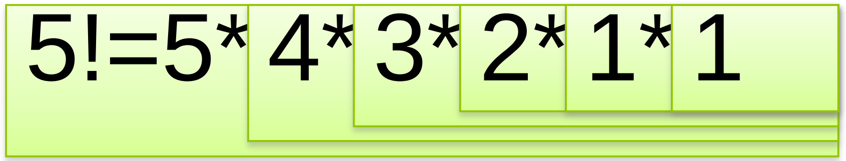
Further explanation-
Recursion: Factorial

Classic introductory example - Factorial function:

$$n! = 1 \times \ldots \times (n-1) \times n$$

$$n! = (n-1)! \times n$$ # recursive call

$$0! = 1$$ # stopping condition, or base case



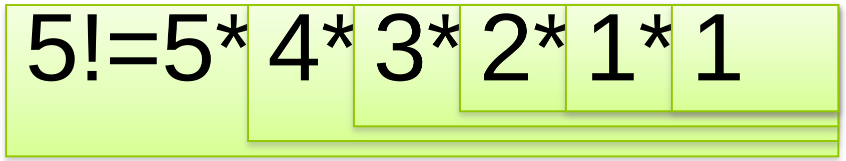
Further explanation- Recursion: Factorial

Classic introductory example - Factorial function:

$$n! = 1 \times \ldots \times (n-1) \times n$$

$$n! = (n-1)! \times n$$ # recursive call

$$0! = 1$$ # stopping condition, or base case



Further explanation-
Recursion: Factorial

Classic introductory example - Factorial function:

$$n! = 1 \times \ldots \times (n-1) \times n$$

$$n! = (n-1)! \times n$$ # recursive call

$$0! = 1$$ # stopping condition, or base case

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

Further explanation-
Recursion: Factorial

Classic introductory example - Factorial function:

$$n! = 1 \times \ldots \times (n-1) \times n$$

$$n! = (n-1)! \times n$$ # recursive call

$$0! = 1$$ # stopping condition, or base case

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

Further explanation- Recursion: Factorial

Classic introductory example - Factorial function:

$$n! = 1 \times \ldots \times (n-1) \times n$$

$$n! = (n-1)! \times n$$ # recursive call

$$0! = 1$$ # stopping condition, or base case

5!=5* 4* 3*2

Further explanation- Recursion: Factorial

Classic introductory example - Factorial function:

$$n! = 1 \times \ldots \times (n-1) \times n$$

$$n! = (n-1)! \times n$$ # recursive call

$$0! = 1$$ # stopping condition, or base case

5!=5* 4*6

Further explanation-
Recursion: Factorial

Classic introductory example - Factorial function:

$$n! = 1 \times \ldots \times (n-1) \times n$$

$$n! = (n-1)! \times n$$ # recursive call

$$0! = 1$$ # stopping condition, or base case

5!=5*24

Further explanation- Recursion: Factorial

Classic introductory example - Factorial function:

$$n! = 1 \times \ldots \times (n-1) \times n$$

$n! = (n-1)! \times n$ # recursive call

$0! = 1$ # stopping condition, or base case

```
120
```

UNIVERSITY OF CAPE TOWN

Iterative solution: Factorial function

```
def fact(n):
    f=1
    for i in range(1,n+1):
        f=f*i
    return f
```

Checkpoint: Write a recursive function to sum the first n positive integers

```
#fill in the code for this function
def sumRec(n):
    #code goes here
    #what is the base case?
    #what is the recursive step?
```

Iterative solution

```
def sum (max):
   sum = 0
   for a in range(max+1):
      sum += a
   return sum
```

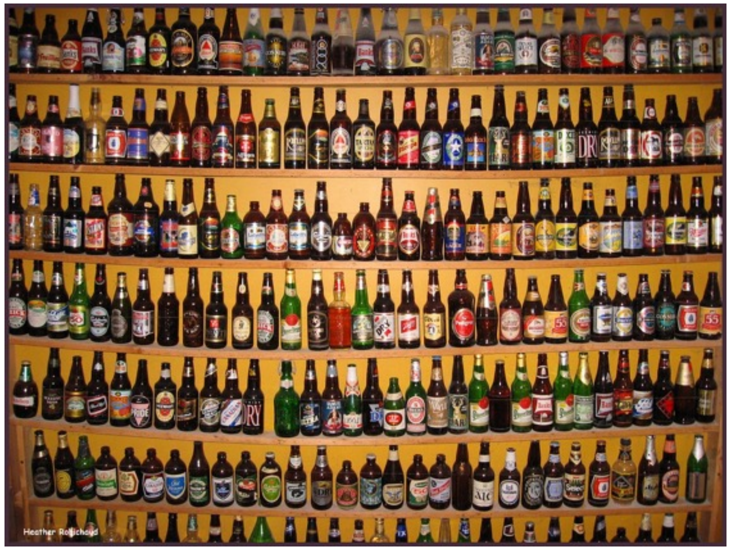
Fun with recursion

100 bottles of beer on the wall,
100 bottles of beer.
If one of those bottles should happen to fall,
99 bottles of beer on the wall.

99 bottles of beer on the wall,
99 bottles of beer.
If one of those bottles should happen to fall,
98 bottles of beer on the wall.

98 bottles of beer on the wall,
98 bottles of beer.
If one of those bottles should happen to fall,
97 bottles of beer on the wall.

97 bottles of beer on the wall,
97 bottles of beer.
If one of those bottles should happen to fall,
96 bottles of beer on the wall…



Checkpoint from arrays

Rewrite the code for the function Enigma(lst, item) so that it works as follows. This function should return True if every element in lst is greater than item.

For example (in the Python3 interpreter):

```
>>>Enigma(["buffalo","zebra","goldfish"],"aardvark")
>>>True
>>>Enigma([1,2,3,1,5,6],3)
>>>False
```

Checkpoint from arrays: do recursive version!

Rewrite the code for the function Enigma(lst, item) so that it works as follows. This function should return True if every element in lst is greater than item.

For example (in the Python3 interpreter):

```
>>>Enigma(["buffalo","zebra","goldfish"],"aardvark")

>>>True

>>>Enigma([1,2,3,1,5,6],3)

>>>False
```

```
def enigma(lst,item):
    if lst==[]: return True #stopping case 1
    if lst[0]<= item: return False #stopping case 2
    return enigma(lst[1:],item) #recursive case
```

More Recursion Problems

- Calculate x^n
- **Count the number of characters in a string.**
- Count the number of words in a list.
- **Search/replace characters in a string.**
- Calculate Greatest Common Denominator (GCD)
    - Look at Euclid's Algorithm
- Print out a list of values.
- Reverse a string.
- **Find the sum of integers from m to n.**

Pitfall: Infinite Recursion

In our examples, the series of recursive calls eventually reached a call of the method that did not involve recursion (a stopping case).

If instead, every recursive call had produced another recursive call, then a call to that method would, in theory, run forever.

- This is called ***infinite recursion.***
- In practice, such a method runs until the computer runs out of resources, and the program terminates abnormally

Infinite recursion example

```
def sumRec(n):
    if n==0:   #logic error here
        return 0  #builtins.RuntimeError: maximum recursion depth exceeded in comparison
    else:
        return sumRec(n-1)+n
```

A Closer Look at Recursion

When the computer encounters a recursive call, it must temporarily suspend its execution of a function

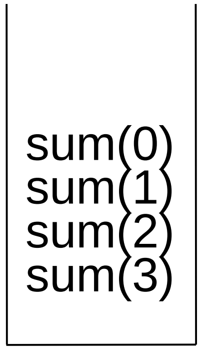
- It does this because *it must know the result of the recursive call before it can proceed*
- It saves all the information it needs to continue the computation later on, when it returns from the recursive call

Ultimately, this entire process terminates when one of the recursive calls does not depend upon recursion to return.

What's Happening Inside Python

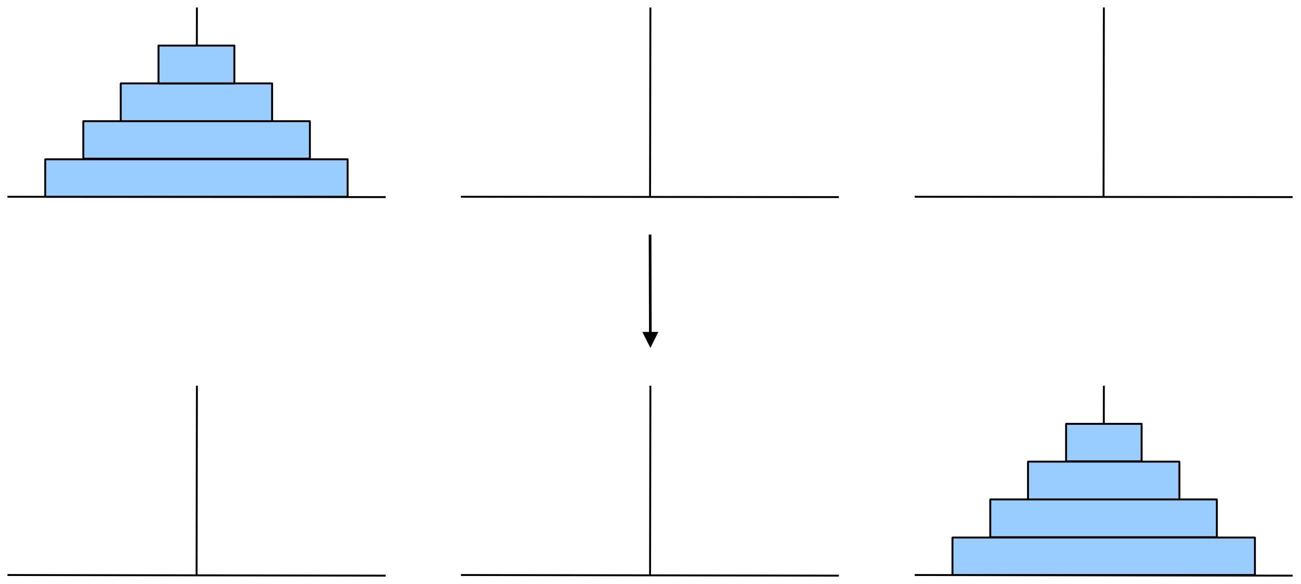
Python keeps track of every function that has been called on a store in memory called a **stack**.

- This allows Python to return to the next point after a function call.
- The same holds for recursive functions.
  - e.g., sum(3) calls sum(2) calls sum(1) calls sum(0)



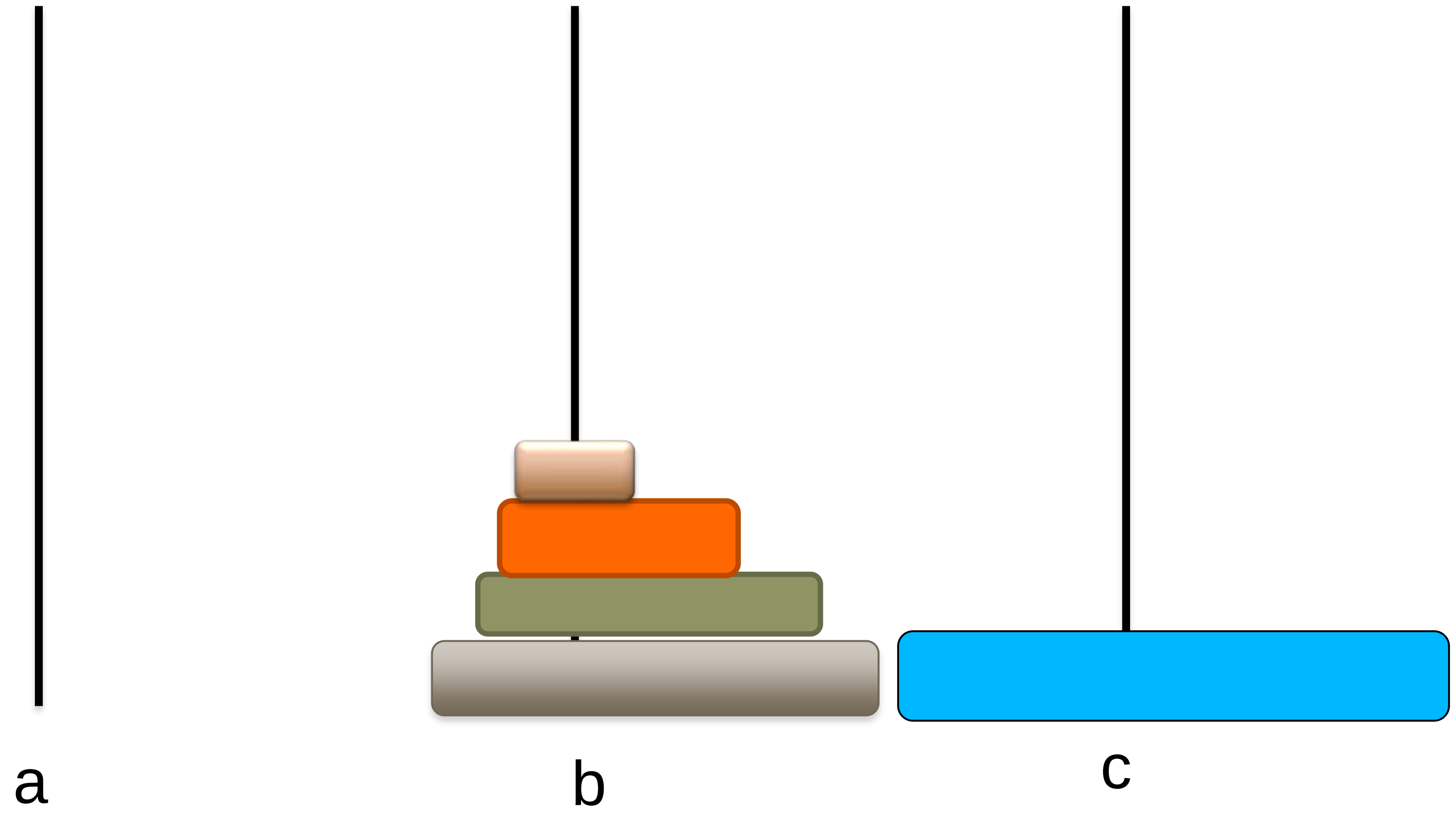
Towers of Hanoi

- Problem: Move a stack of discs one disc at a time from one tower to another, such that no disc may be placed on a larger disc.

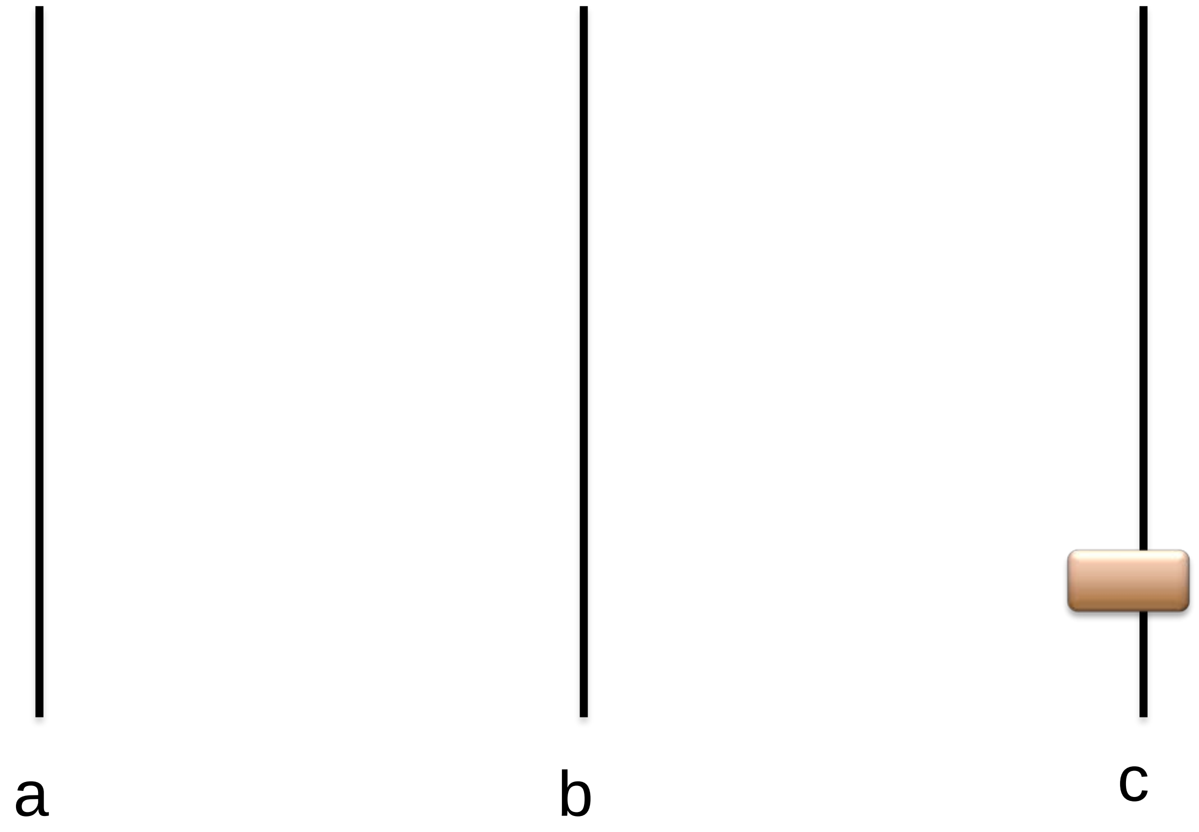


puzzle game invented in the late 1800s

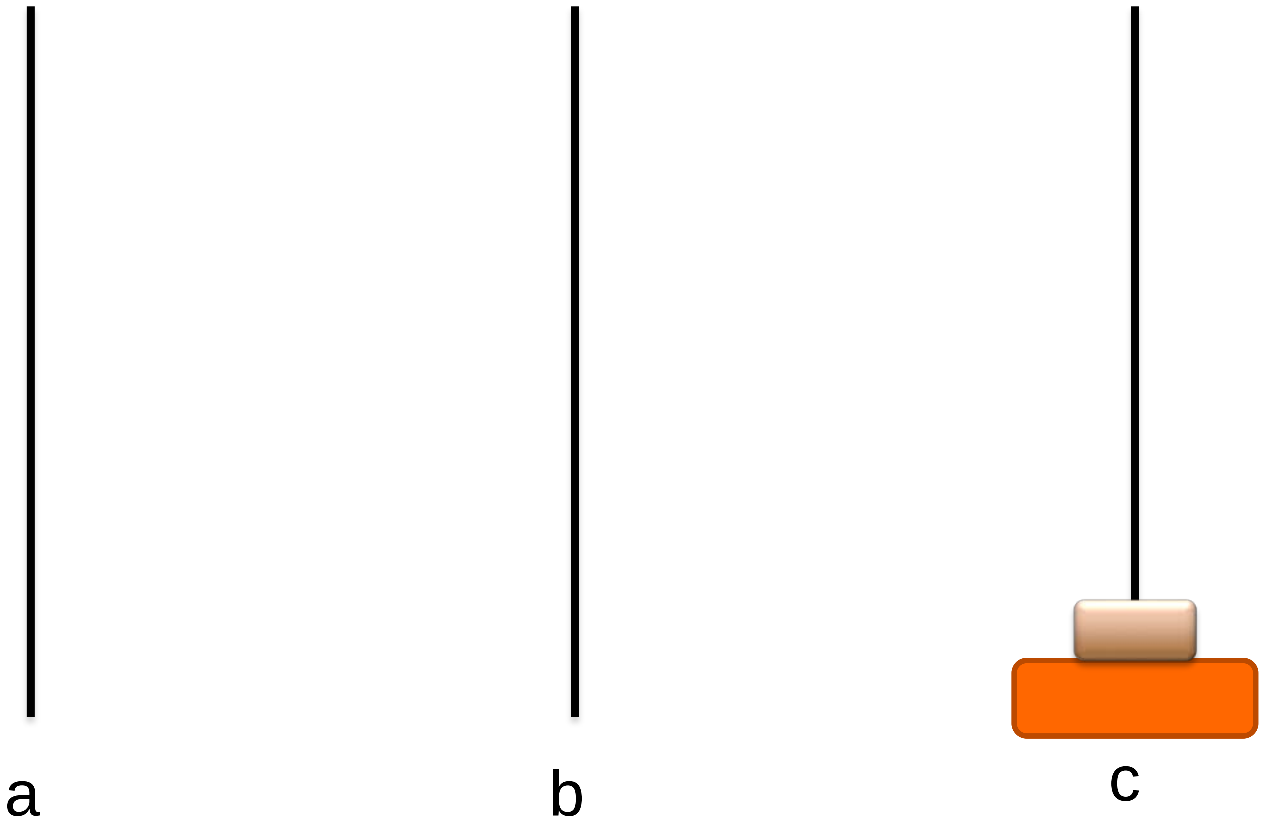
Hanoi



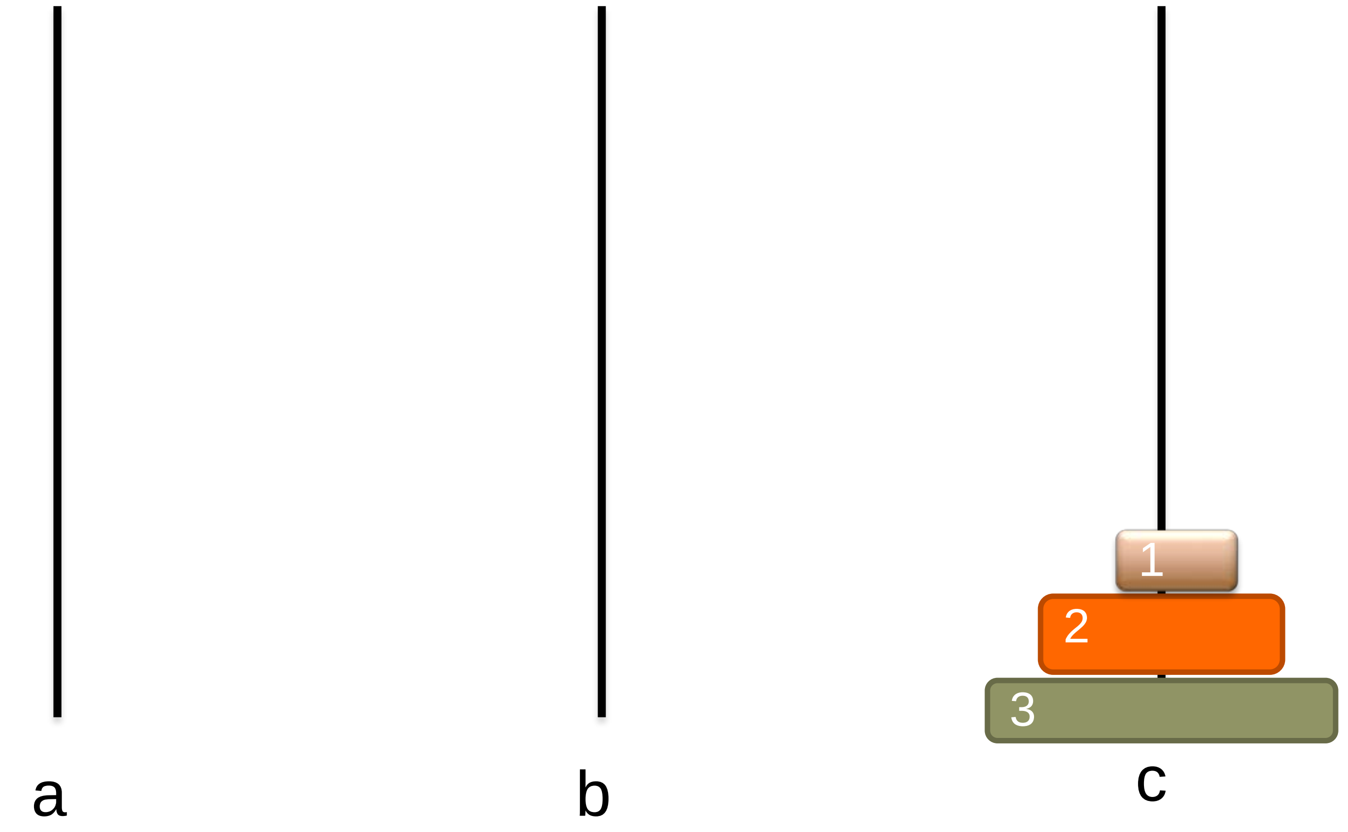
Hanoi: 1 disk



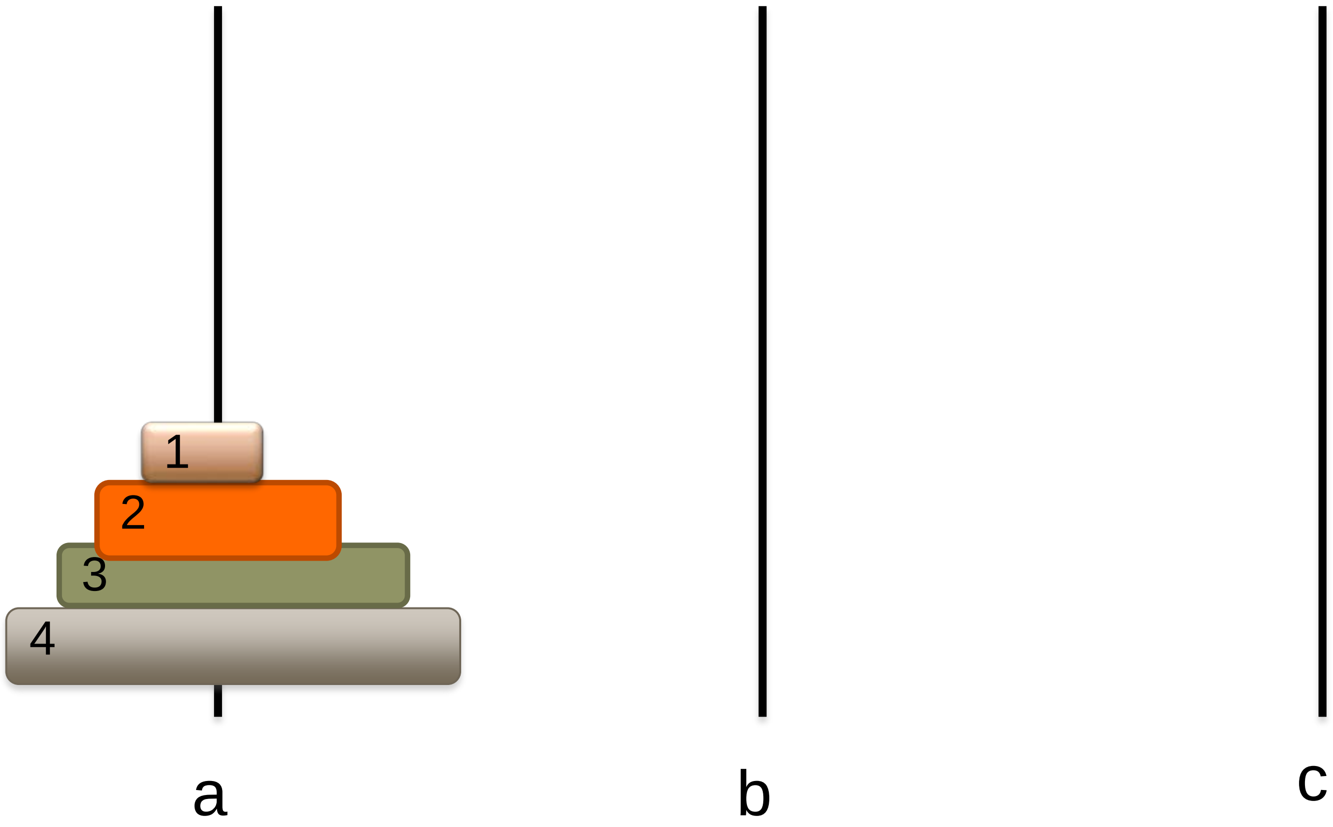
Hanoi: 2 disks



Hanoi: 3 disks



Hanoi: 4 disks

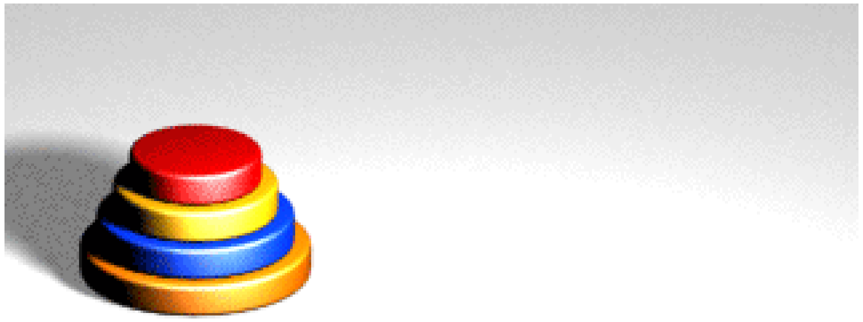


Towers of Hanoi

- Algorithm:
  - Move n-1 discs from source to spare tower
  - Move nth disc from source to destination tower
  - Move n-1 discs from spare to destination tower

  - Stop when no more discs … or one disc

https://www.youtube.com/watch?v=rVPuzFYlfYE



Checkpoint

Write recursive code to convert a decimal number into octal (base 8).

Checkpoint: what does this recursive function compute?

```
def mystery(x):
    if x==1:
        return 2
    if x==0:
        return 1
  return 2*mystery(x-1)
```

Write a recursive function to

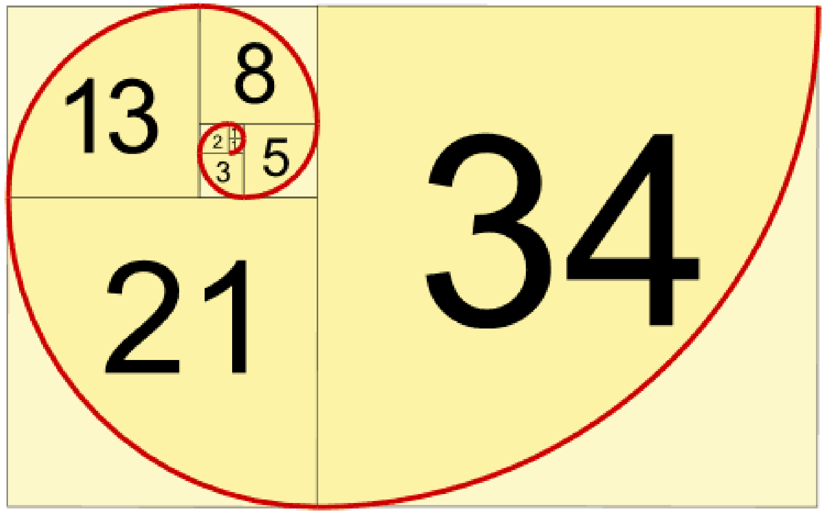
- Draw a picture like this->

Hourglass(“MARSUIPIAL”)

```
MARSUPIAL
MARSUPIA
MARSUPI
MARSUP
MARSU
MARS
MAR
MA
M
MA
MAR
MARS
MARSU
MARSUP
MARSUPI
MARSUPIA
MARSUPIAL
```

Iterative Fibonnaci numbers

```
def fib(n):
    curr=1
    prev=1
    for i in range(n-2):
        curr,prev=curr+prev,curr
    return curr
```



Recursive Fibonacci numbers

Fibonacci in Australia

In 1859, a farmer introduced 24 grey rabbits to remind him of home. At the time, the man wrote:

"The introduction of a few rabbits could do little harm and might provide a touch of home, in addition to a spot of hunting."

For one pair, by 1900….(480 months)…

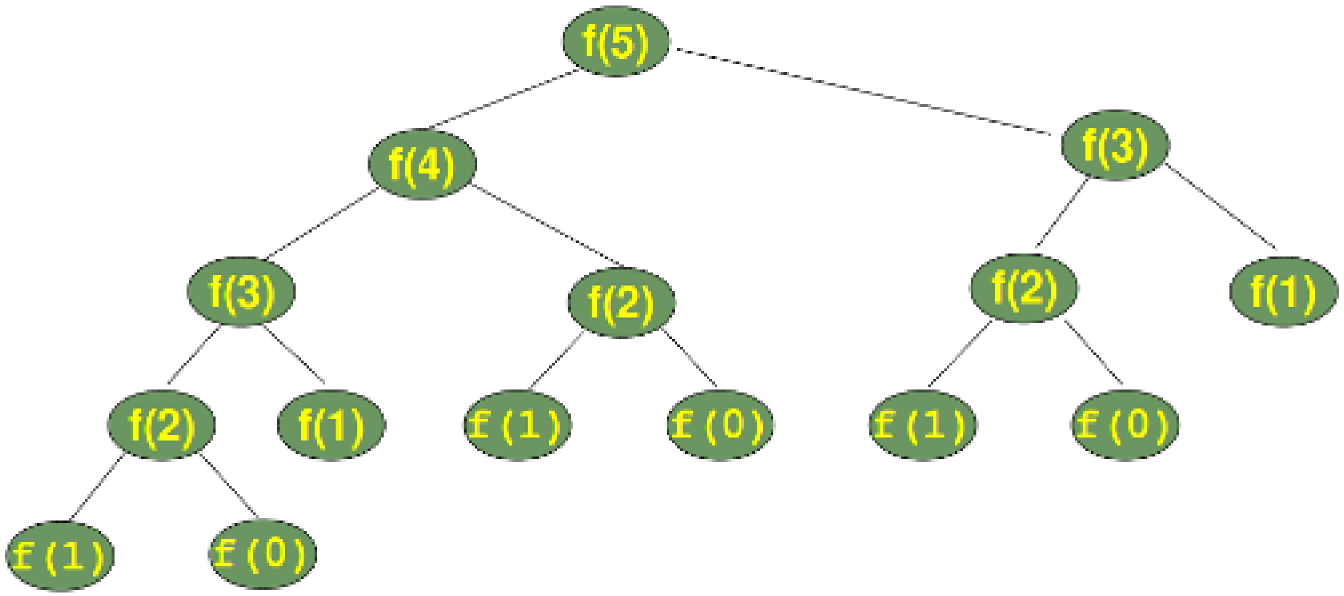
Fib(480)

Compare with timer

Recursive Fibonacci numbers

Elegant solution.

Not very efficient, because of many duplicate function calls



Recursion Versus Iteration

- Recursion is not absolutely necessary
  - Any task that can be done using recursion can also be done in a nonrecursive manner
  - A nonrecursive version of a method is called an *iterative version*
- An iteratively written method will typically use loops of some sort in place of recursion
- A recursively written method can be simpler, but will usually run slower and use more storage than an equivalent iterative version

Checkpoint: what does this recursive function compute?

```
def mystery2(a,b):
    if b==0:
        return 0
    if b%2==0:
        return mystery2(a+a,b//2)
    return mystery2(a+a,b//2)+a
```

More Recursion Problems

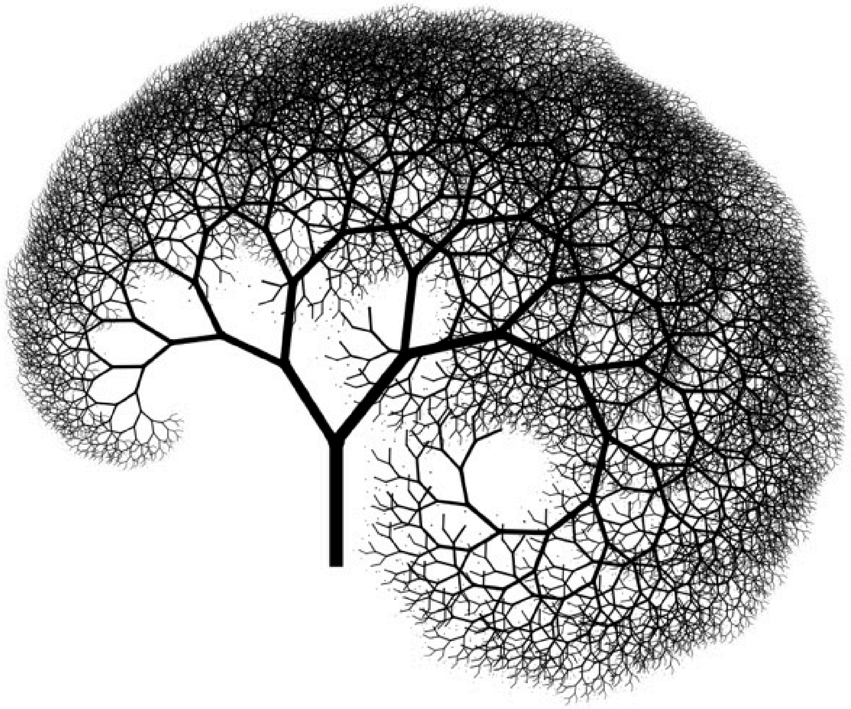
- Calculate x^n
- **Count the number of characters in a string.**
- Count the number of words in a list.
- **Search/replace characters in a string.**
- Calculate Greatest Common Denominator (GCD)
    - Look at Euclid’s Algorithm
- Print out a list of values.
- Reverse a string.
- **Find the sum of integers from m to n.**

Iterative version of prefix sum

```
#iterative definition
def  prefixSum(arr):
    tmp=[]
    for i in range(len(arr)):
        if i==0:
            tmp.append(arr[0]) #the first one is just a copy
        else:
            tmp.append(tmp[i-1]+arr[i]) #cummulative sum
    return tmp
```

Checkpoint: write a recursive version of prefix sum

```
#recursive definition
def  prefixSumRec(arr):
    #fill in the rest of the function
```

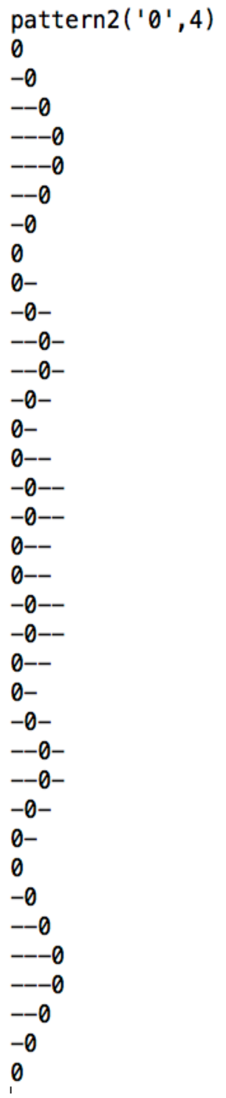


Checkpoint: what does this recursive function display?

```
def pattern(s,n):
    if n==0:
        return
    print(s)
    pattern('-'+s,n-1)
    print(s)
```

Challenge 1:

Now, with the pattern function, how do I write a function to display this type of pattern?



```
pattern2('0',4)
0
-0
--0
---0
---0
--0
-0
0
0-
-0-
--0-
--0-
-0-
0-
0--
-0--
-0--
0--
0--
-0--
-0--
0--
0-
-0-
--0-
--0-
-0-
0-
0
-0
--0
---0
---0
--0
-0
0
```

Challenge 2:

A nested number list is a list whose elements are either:

- numbers
- nested number lists

(nice recursive definition)

e.g. [1, 2, [11, 13], [8,[2,3]]]

Write a recursive function to sum all the numbers in a nested number list

e.g. `r_sum([1, 2, [11, 13], [8,[2,3]]])` returns 40

Nice recursive example

*Which is the **more efficient algorithm?***

```
def recPow( a,n):
    """raises a to int power n"""
    if n==0: return 1
    else:
        return a*recPow(a,n-1)
```

```
def recPowAlt ( a,n):
    """raises a to int power n"""
    if n==0: return 1
    else:
        factor=recPowAlt(a,n//2)
        if n%2==0:
            return factor*factor
        else: return factor*factor*a
```

Recursion - Justification

Recursion is one of the most important ideas in computer science, but it's usually viewed as one of the harder parts of programming to grasp.

We can work out very concise and elegant solutions to problems by **thinking recursively**.

Basic approach traversing for non-linear data structures… such as trees

Also, there are problems whose solutions are **inherently recursive**, because they need to keep track of prior state. e.g.:

- divide-and-conquer algorithms such as Quicksort (coming soon….)

All of these algorithms can be implemented iteratively with the help of a stack, but the need for the stack arguably nullifies the advantages of the iterative solution.

History of Recursion

Also, key concept for functional programming languages, such as Haskell

Lisp is the second-oldest high-level programming language in widespread use today; only Fortran is older.

- recursion a key component of language
- Driven by AI

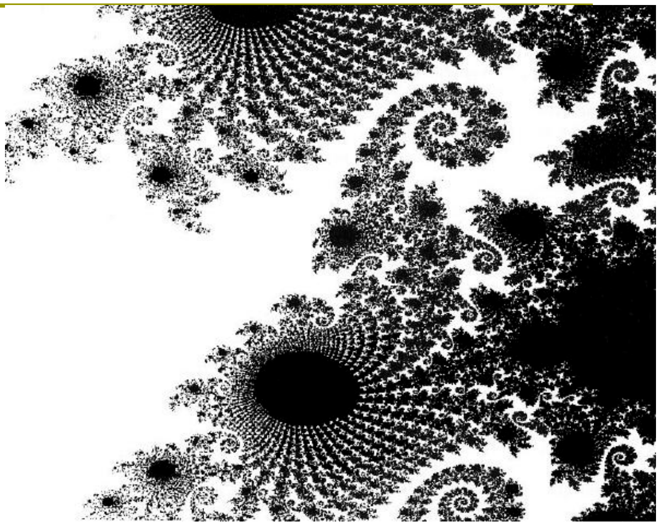
Resurgence today, especially for parallel programming (Haskell)

Thinking Recursively

1. There is no infinite recursion
   - Every chain of recursive calls must reach a stopping case
2. Each stopping case returns the correct value for that case
3. For the cases that involve recursion: *if* all recursive calls return the correct value, *then* the final value returned by the method is the correct value

These properties follow a technique also known as

mathematical induction



2D arrays

treasure_map.py