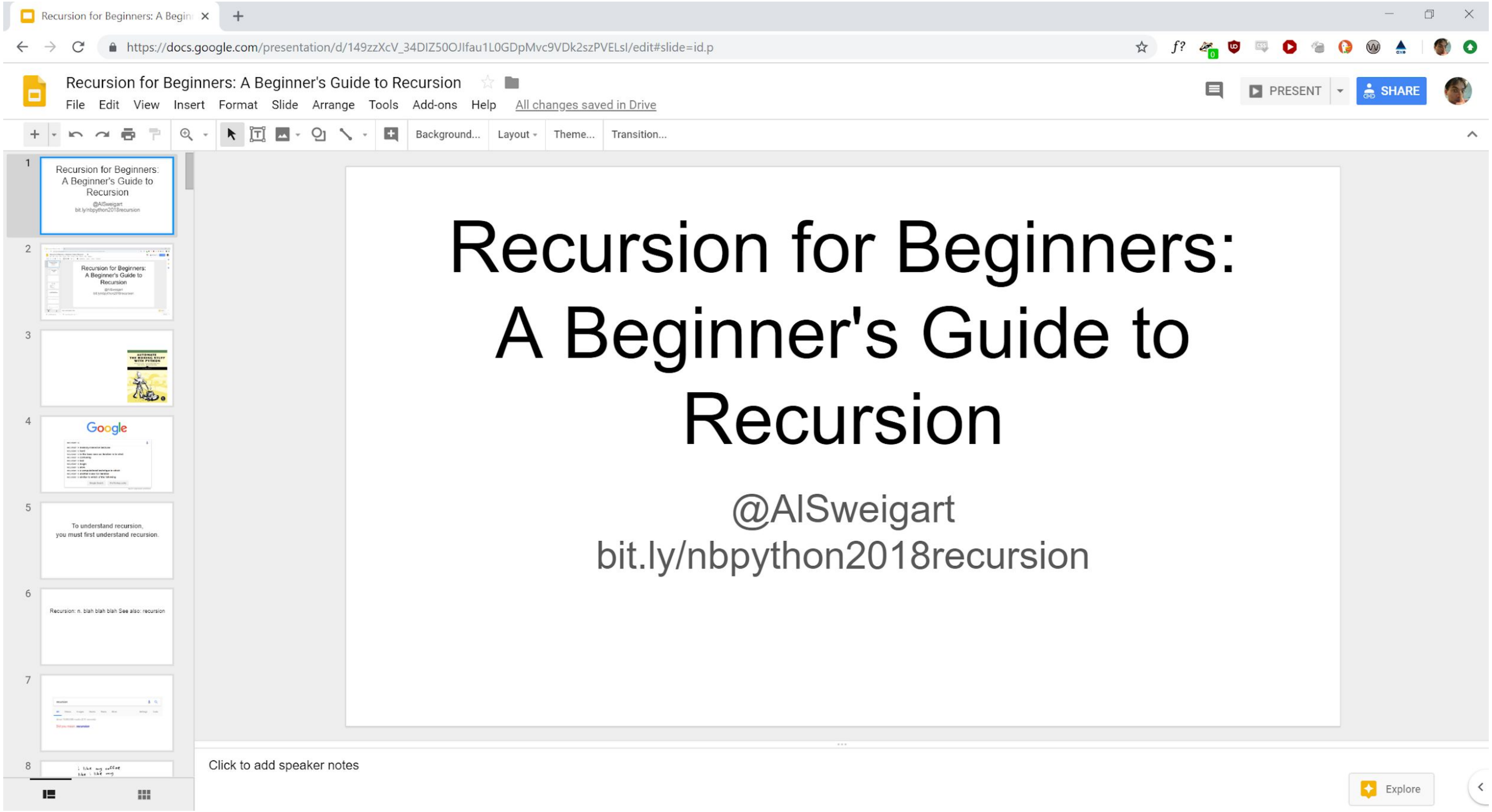
Recursion for Beginners: A Beginner's Guide to Recursion

@AlSweigart al@inventwithpython.com

bit.ly/nbpython2018recursion



Recursion for Beginners: A Beginner's Guide to Recursion

File Edit View Insert Format Slide Arrange Tools Add-ons Help All changes saved in Drive

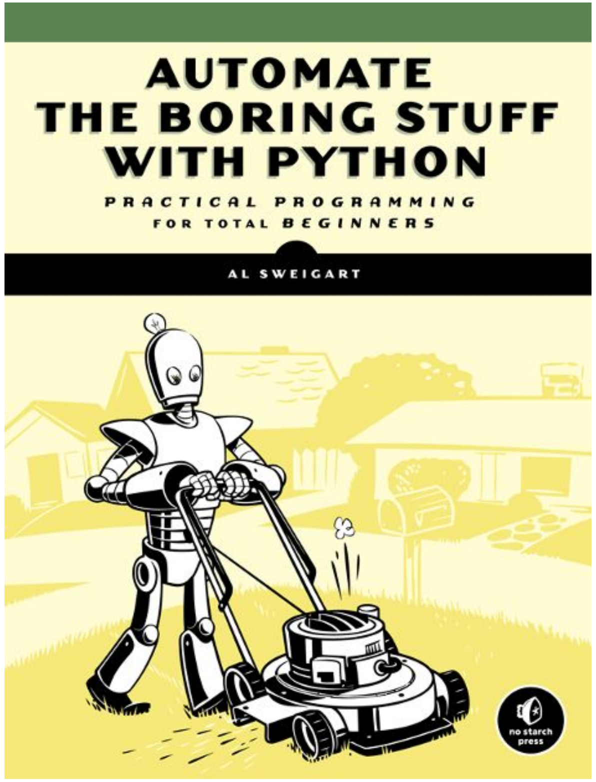
Recursion for Beginners: A Beginner's Guide to Recursion

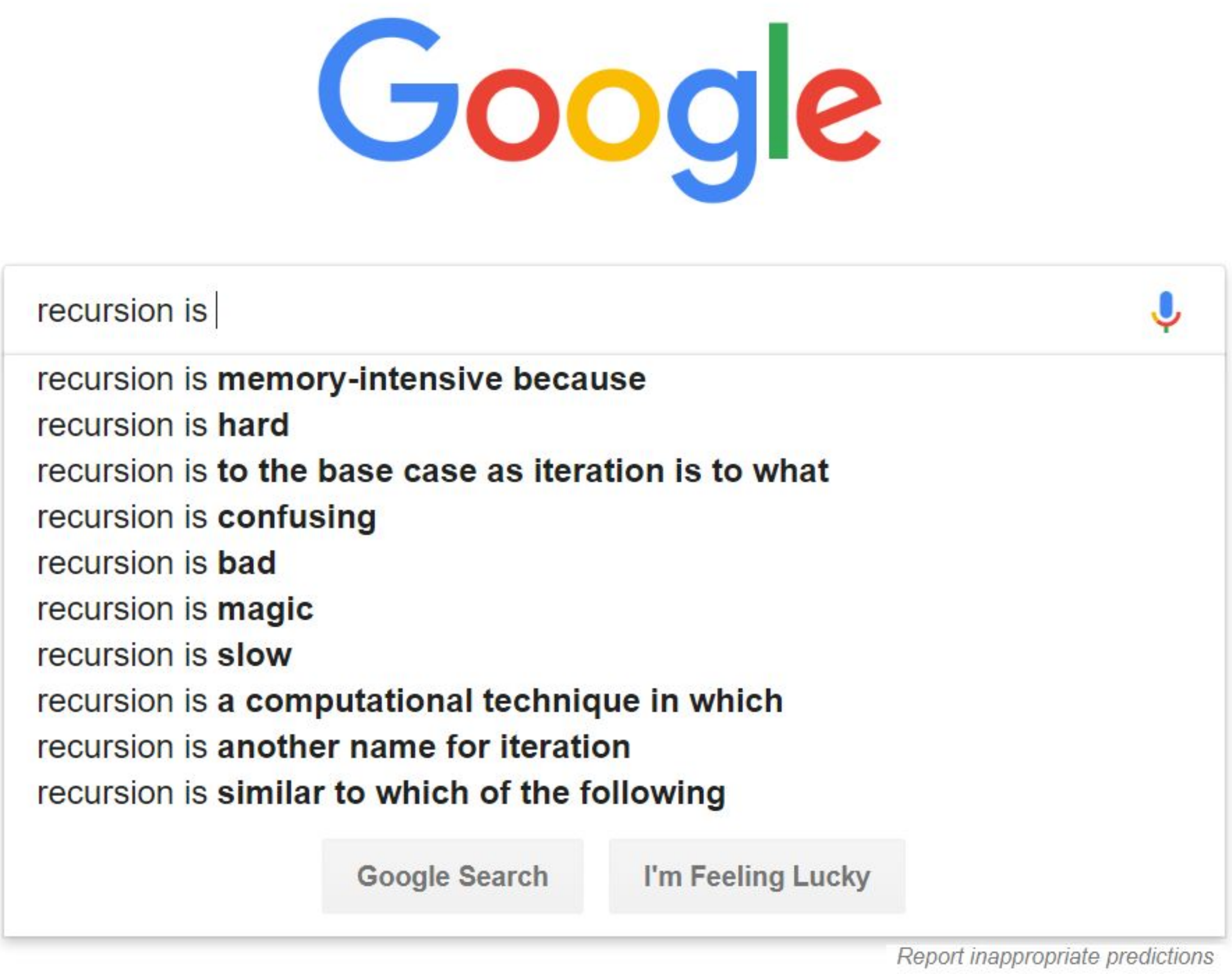
@AlSweigart
bit.ly/nbpython2018recursion

To understand recursion, you must first understand recursion.

Recursion: n. blah blah blah See also: recursion

Click to add speaker notes


AUTOMATE THE BORING STUFF WITH PYTHON
PRACTICAL PROGRAMMING FOR TOTAL BEGINNERS
AL SWEIGART
no starch press

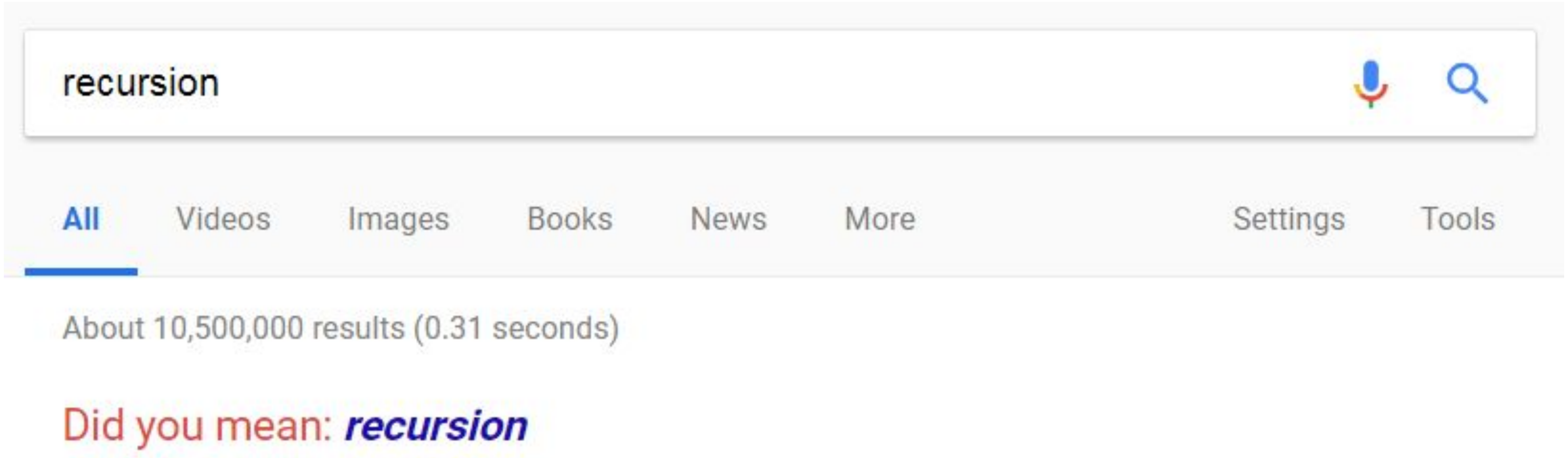


recursion is

recursion is **memory-intensive because**
recursion is **hard**
recursion is **to the base case as iteration is to what**
recursion is **confusing**
recursion is **bad**
recursion is **magic**
recursion is **slow**
recursion is **a computational technique in which**
recursion is **another name for iteration**
recursion is **similar to which of the following**

To understand recursion,
you must first understand recursion.

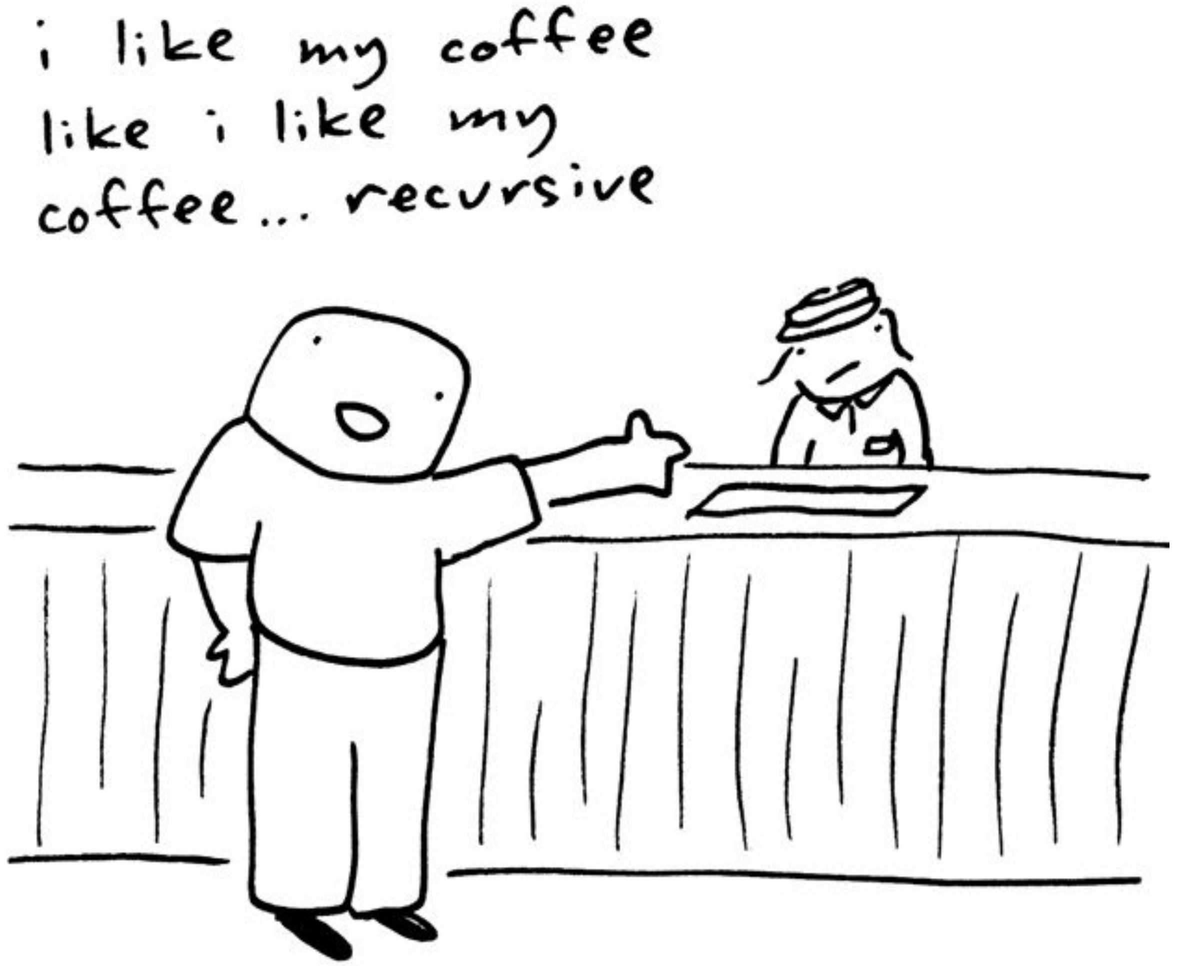
Recursion: n. blah blah blah See also: recursion


recursion

All Videos Images Books News More Settings Tools

About 10,500,000 results (0.31 seconds)

Did you mean: ***recursion***



Toothpaste For Dinner.com

I'M SO META, EVEN THIS ACRONYM

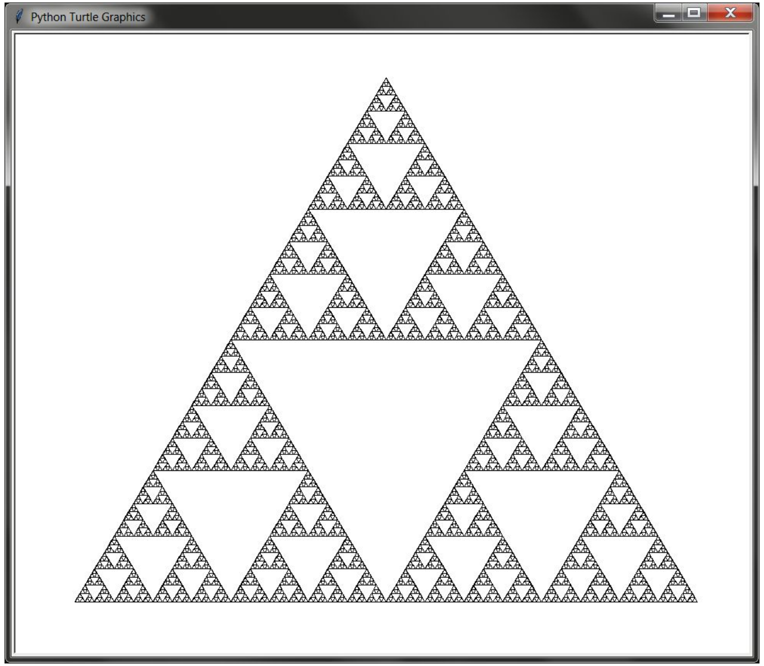
I'M SO META, EVEN THIS ACRONYM

I.S.M.E.T.A.

A recursive thing is something whose definition includes itself.

Sierpinski Triangle

https://github.com/asweigart/recursion_examples



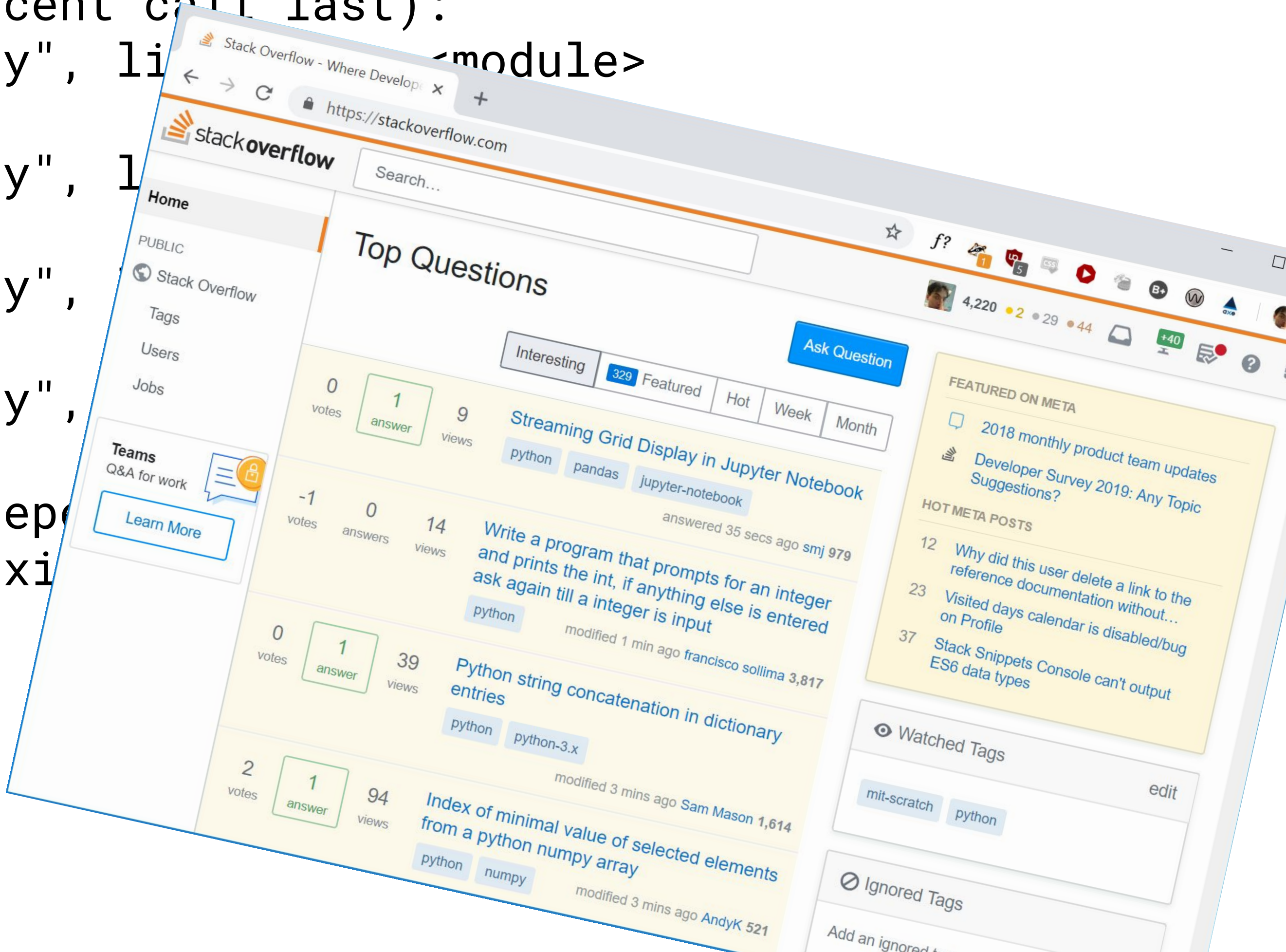
In programming, recursion is when a function calls itself.

In programming, recursion is when a function calls itself.

```
def shortest():
    shortest()
```

```
Traceback (most recent call last):
  File "shortest.py", line 4, in <module>
    shortest()
  File "shortest.py", line 2, in shortest
    shortest()
  File "shortest.py", line 2, in shortest
    shortest()
  File "shortest.py", line 2, in shortest
    shortest()
  [Previous line repeated 996 more times]
RecursionError: maximum recursion depth exceeded
```

```
Traceback (most recent call last):
  File "shortest.py", li                    <module>
    shortest()
  File "shortest.py", l
    shortest()
  File "shortest.py",
    shortest()
  File "shortest.py",
    shortest()
  [Previous line repe
RecursionError: maxi
```



Top Questions

0 votes | 1 answer | 9 views — Streaming Grid Display in Jupyter Notebook — python, pandas, jupyter-notebook — answered 35 secs ago smj 979

-1 votes | 0 answers | 14 views — Write a program that prompts for an integer and prints the int, if anything else is entered ask again till a integer is input — python — modified 1 min ago francisco sollima 3,817

0 votes | 1 answer | 39 views — Python string concatenation in dictionary entries — python, python-3.x — modified 3 mins ago Sam Mason 1,614

2 votes | 1 answer | 94 views — Index of minimal value of selected elements from a python numpy array — python, numpy — modified 3 mins ago AndyK 521

FEATURED ON META
2018 monthly product team updates
Developer Survey 2019: Any Topic Suggestions?

HOT META POSTS
12 Why did this user delete a link to the reference documentation without...
23 Visited days calendar is disabled/bug on Profile
37 Stack Snippets Console can't output ES6 data types

Watched Tags — edit
mit-scratch, python

Ignored Tags
Add an ignored

To understand recursion,
you must first understand recursion.

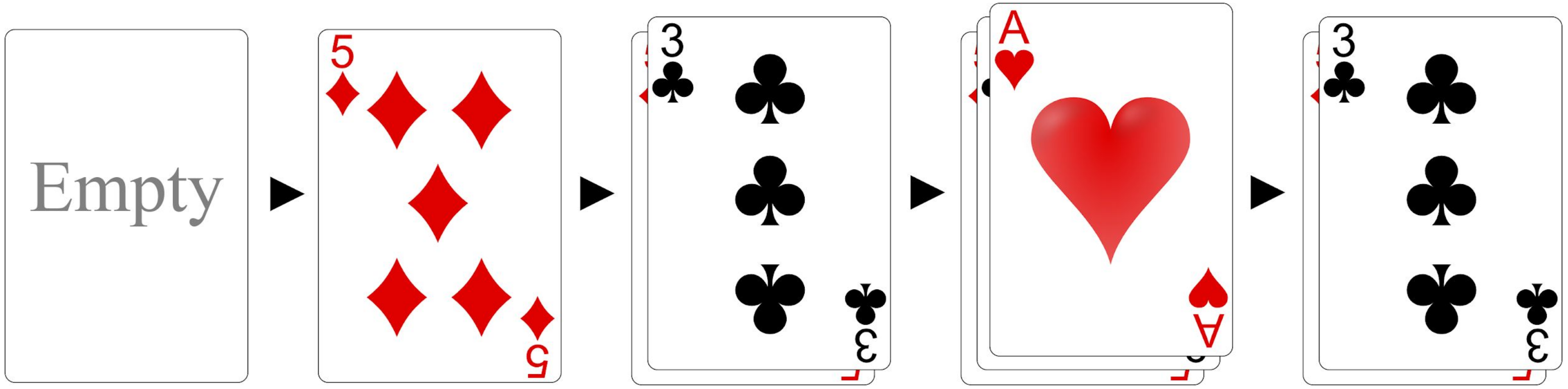
To understand recursion,

you must first understand ~~recursion~~.

STACKS

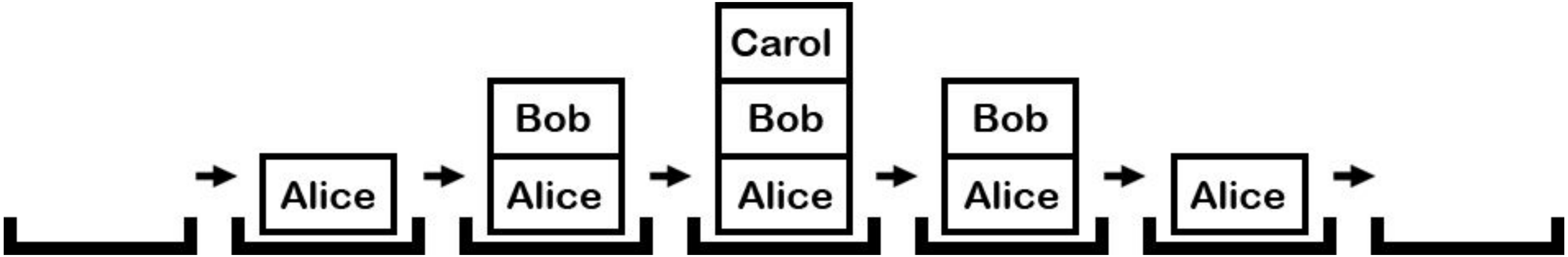
A stack is a data structure that holds a sequence of data and only lets you interact with the topmost item.

A stack is a data structure that holds a sequence of data and only lets you interact with the topmost item.



First-In, Last-Out (FILO)

A stack is a data structure that holds a sequence of data and only lets you interact with the topmost item.



Adding to the top of the stack is called **pushing**.
Removing from the top of the stack is called **popping**.

Adding to the top of the stack is called **pushing**.
Removing from the top of the stack is called **popping**.

Python’s lists are a stack if you only use **append()** and **pop()**.

```
>>> spam = []
>>> spam.append('Alice')
>>> spam.append('Bob')
>>> spam.append('Carol')
>>> spam.pop()
'Carol'
>>> spam
['Alice', 'Bob']
```

```python
def a():
    print('Start of a()')
    b()
    print('End of a()')

def b():
    print('Start of b()')
    c()
    print('End of b()')

def c():
    print('Start of c()')
    print('End of c()')

a()
```

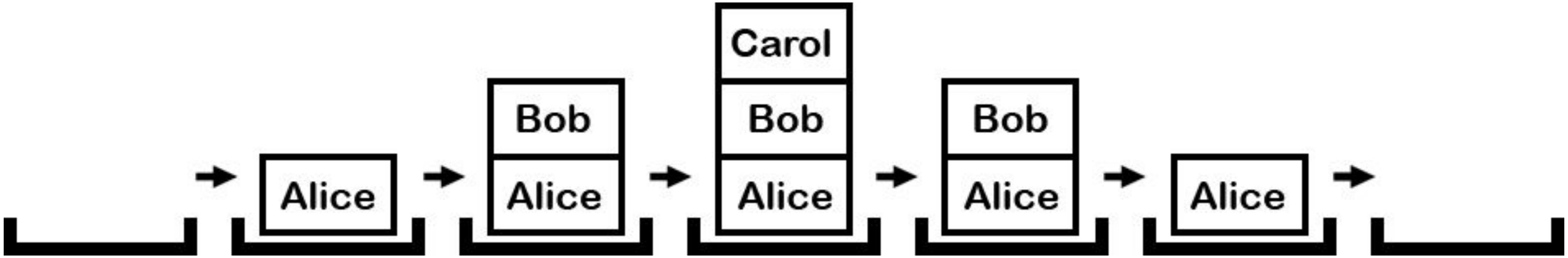
```python
def a():
    print('Start of a()')
    b()
    print('End of a()')

def b():
    print('Start of b()')
    c()
    print('End of b()')

def c():
    print('Start of c()')
    print('End of c()')

a()
```

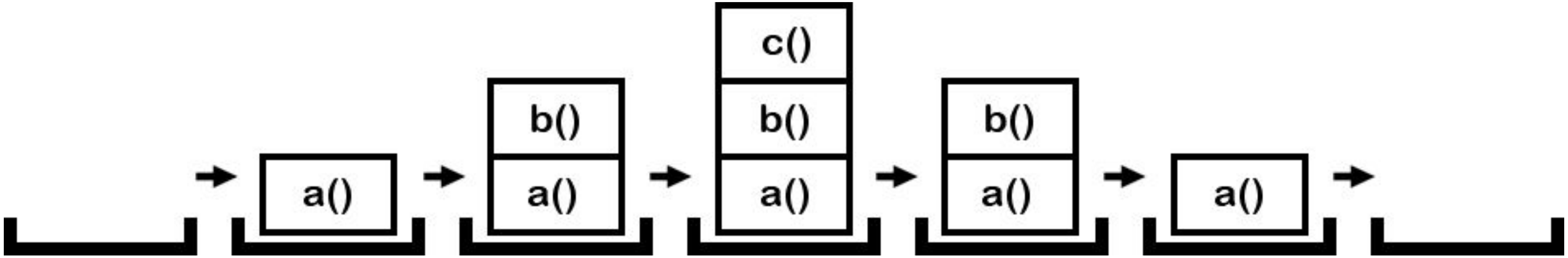
```
Start of a()
Start of b()
Start of c()
End of c()
End of b()
End of a()
```

```python
def a():
    print('Start of a()')
    b()
    print('End of a()')

def b():
    print('Start of b()')
    c()
    print('End of b()')

def c():
    print('Start of c()')
    print('End of c()')

a()
```

```
Start of a()
Start of b()
Start of c()
End of c()
End of b()
End of a()
```

“GOTO Considered Harmful”





The “call stack” is a stack of “frame objects”.
(frame object == a function call)

Recursion is a lot easier to understand if you understand stacks and the call stack.

```python
def a():
    print('Start of a()')
    b()
    print('End of a()')

def b():
    print('Start of b()')
    c()
    print('End of b()')

def c():
    print('Start of c()')
    print('End of c()')

a()
```

```
Start of a()
Start of b()
Start of c()
End of c()
End of b()
End of a()
```

```python
def a():
    print('Start of a()')
    b()
    print('End of a()')

def b():
    print('Start of b()')
    c()
    print('End of b()')

def c():
    print('Start of c()')
    print('End of c()')

a()
```

```
Start of a()
Start of b()
Start of c()
End of c()
End of b()
End of a()
```

“Where’s the call stack?”

Extra Credit

Look up the `inspect` and `traceback` modules.

Doug Hellmann's "Python Module of the Week" blog:

- https://pymotw.com/3/inspect/
- https://pymotw.com/3/traceback/

Recursion is overrated.

Factorial

$5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$

$2! = 2 \times 1 = 2$

$4! = 4 \times 3 \times 2 \times 1 = 24$

Factorial

$5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$

$4! = \phantom{5 \times{}} 4 \times 3 \times 2 \times 1 = 24$

Factorial

$5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$

$4! = \quad 4 \times 3 \times 2 \times 1 = 24$

$5! = 5 \times 4!$

Factorial

$5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$

$4! = 4 \times 3 \times 2 \times 1 = 24$

$5! = 5 \times 4! = 5 \times 24 = 120$

Factorial

Number! = Number × (Number - 1)!

(Factorial has a recursive nature.)

Factorial

Number! = Number × (Number - 1)!

```
def factorial(number):
    return number * factorial(number - 1)

print(factorial(5))
```

Factorial

```
Traceback (most recent call last):
  File "factorial.py", line 4, in <module>
    print(factorial(5))
  File "factorial.py", line 2, in factorial
    return number * factorial(number - 1)
  File "factorial.py", line 2, in factorial
    return number * factorial(number - 1)
  File "factorial.py", line 2, in factorial
    return number * factorial(number - 1)
  [Previous line repeated 996 more times]
RecursionError: maximum recursion depth exceeded
```

Factorial

$$5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$$

Factorial

$$5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$$

$$5! = 5 \times 4 \times 3 \times 2 \times 1 \times 0 \times -1 \times -2 \times \ldots$$

Factorial

$5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$

$5! = 5 \times 4 \times 3 \times 2 \times 1 \times 0 \times -1 \times -2 \times \ldots$

A stack overflow is when a recursive function gets out of control and doesn’t stop recursing.

Factorial

```
Traceback (most recent call last):
  File "factorial.py", line 4, in <module>
    print(factorial(5))
  File "factorial.py", line 2, in factorial
    return number * factorial(number - 1)
  File "factorial.py", line 2, in factorial
    return number * factorial(number - 1)
  File "factorial.py", line 2, in factorial
    return number * factorial(number - 1)
  [Previous line repeated 996 more times]
RecursionError: maximum recursion depth exceeded
```

Factorial

```python
def factorial(number):
    return number * factorial(number - 1)

print(factorial(5))
```

Factorial (hint: 1! = 1)

```
def factorial(number):
    return number * factorial(number - 1)

print(factorial(5))
```

Factorial (hint: 1! = 1)

```
def factorial(number):
    if number == 1:
        return 1


    return number * factorial(number - 1)

print(factorial(5))
```

Factorial (hint: 1! = 1)

```
def factorial(number):
    if number == 1:
        return 1 # BASE CASE

    # RECURSIVE CASE
    return number * factorial(number - 1)

print(factorial(5))
```

Your recursive function must always have at least one base case and one recursive case.

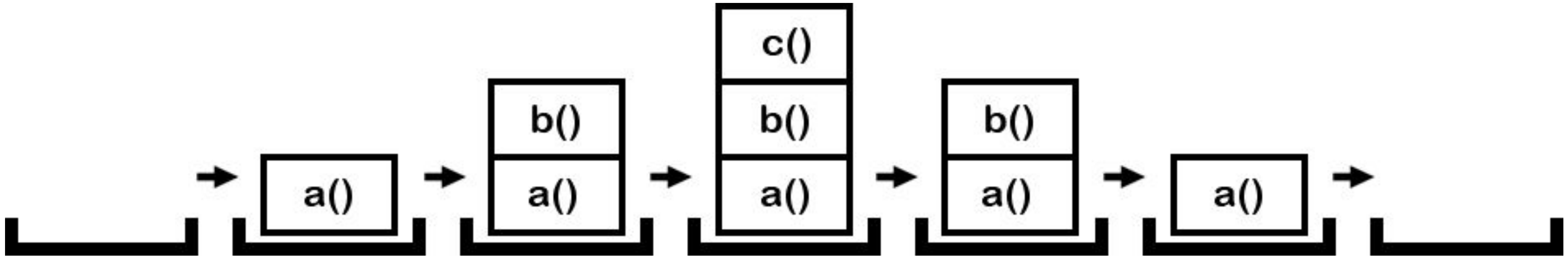
Factorial (hint: 1! = 1)

```
def factorial(number):
    if number == 1:
        return 1 # BASE CASE

    # RECURSIVE CASE
    return number * factorial(number - 1)

print(factorial(5))
```

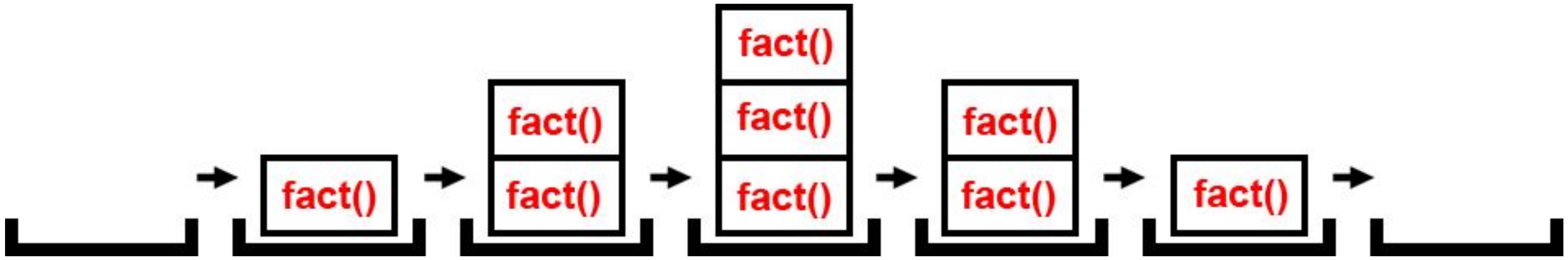
Factorial (hint: 1! = 1)

"Where does the 5 × 4 × 3 × 2 × 1 happen?"

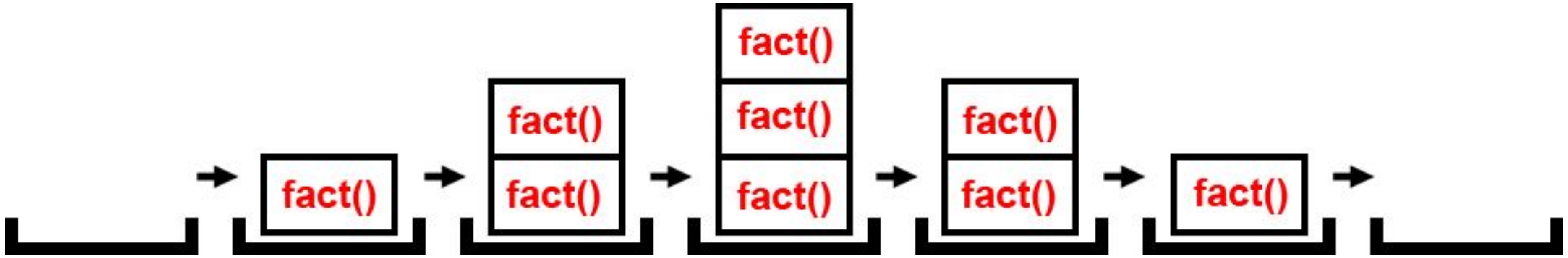
```
def factorial(number):
    if number == 1:
        return 1 # BASE CASE

    # RECURSIVE CASE
    return number * factorial(number - 1)

print(factorial(5))
```



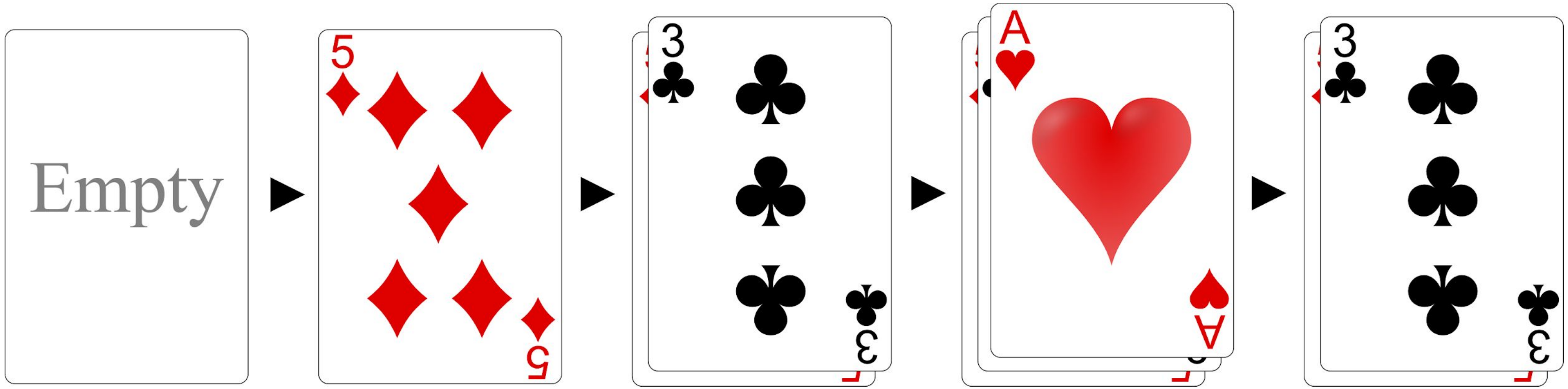
The “call stack” is a stack of “frame objects”.
(frame object == a function call)



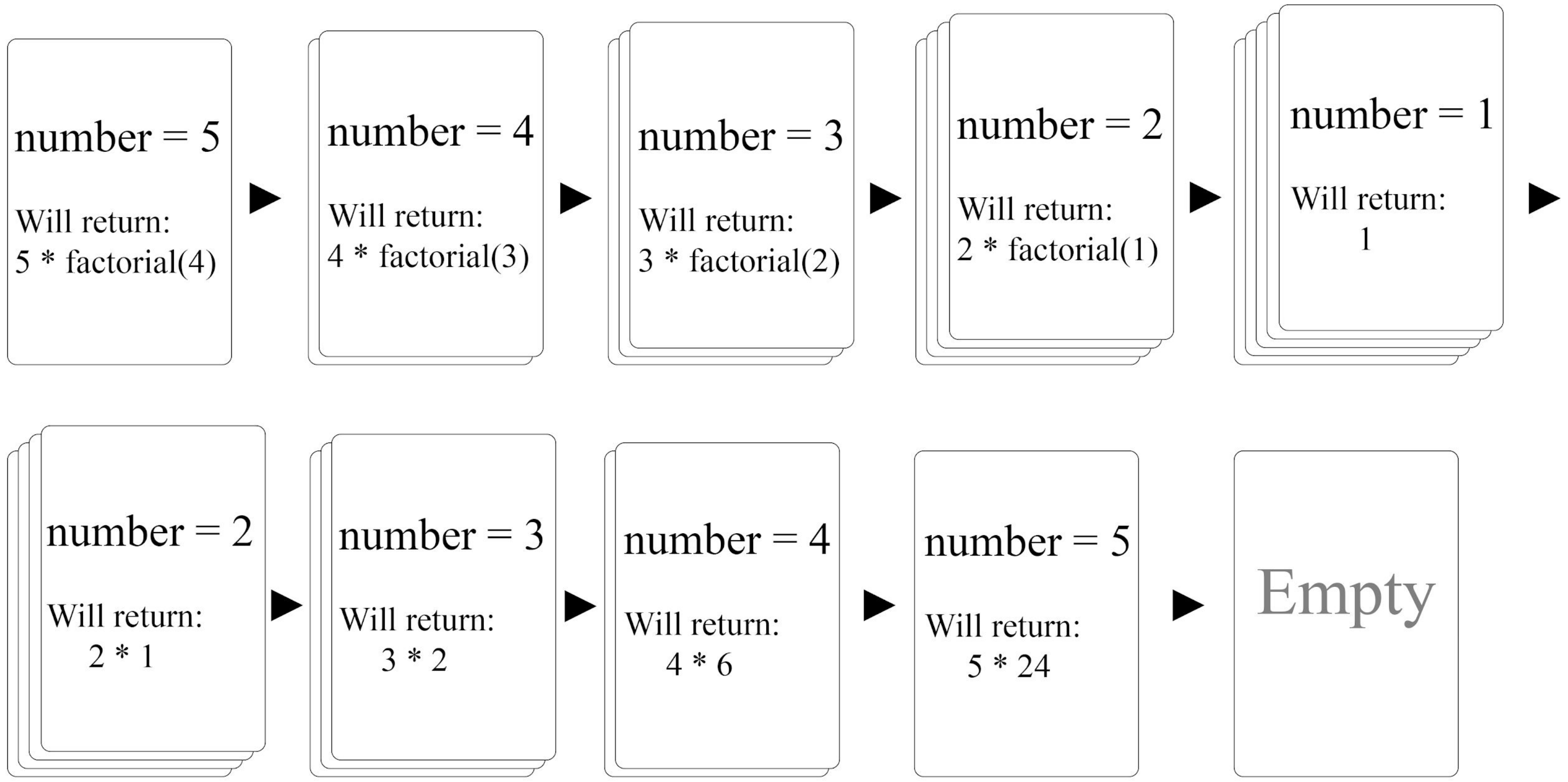
The “call stack” is a stack of “frame objects”.
(frame object == a function call)

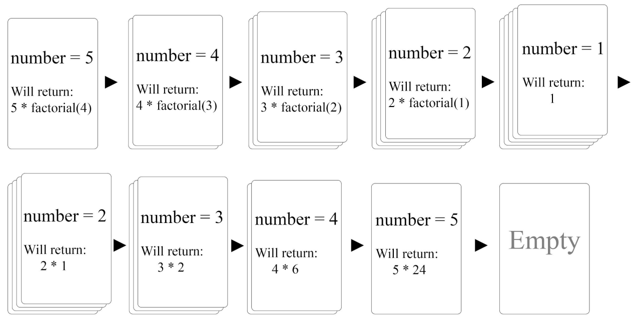


The “call stack” is a stack of “frame objects”.
(frame object == a function call)



Frame objects are where local variables are stored.





```
def factorial(number):
    if number == 1:
        return 1 # BASE CASE

    # RECURSIVE CASE
    return number * factorial(number - 1)
```

```python
# Hard-coded pseudo-recursive algorithm
def factorial5():
    return 5 * factorial4()
def factorial4():
    return 4 * factorial3()
def factorial3():
    return 3 * factorial2()
def factorial2():
    return 2 * factorial1()
def factorial1():
    return 1
print(factorial5())
```

```python
# Iterative factorial algorithm
def factorial(number):
    total = 1
    for i in range(1, number):
        total *= i
    return total

print(factorial(5))
```

```python
# Using iteration to emulate recursion.
callStack = [] # The explicit call stack, which holds "frame objects".
callStack.append({'instrPtr': 'start', 'number': 5}) # "Call" the "factorial() function"
returnValue = None

while len(callStack) > 0:
    # The body of the "factorial() function":

    number = callStack[-1]['number'] # Set number "parameter".
    instrPtr = callStack[-1]['instrPtr']

    if instrPtr == 'start':
        if number == 1:
            # BASE CASE
            returnValue = 1
            callStack.pop() # "Return" from "function call".
            continue
        else:
            # RECURSIVE CASE
            callStack[-1]['instrPtr'] = 'after recursive call'
            # "Call" the "factorial() function":
            callStack.append({'instrPtr': 'start', 'number': number - 1})
            continue
    elif instrPtr == 'after recursive call':
        returnValue = number * returnValue
        callStack.pop()  # "Return from function call".
        continue

print(returnValue)
```

When should we use recursion?

When should we use recursion?

- Never.

When should we use recursion?

When should we use recursion?

- When the problem has a tree-like structure.

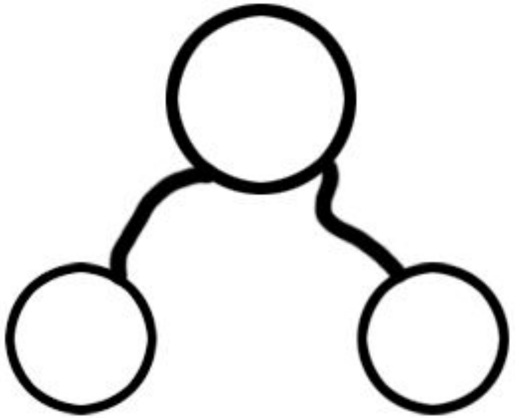
When should we use recursion?

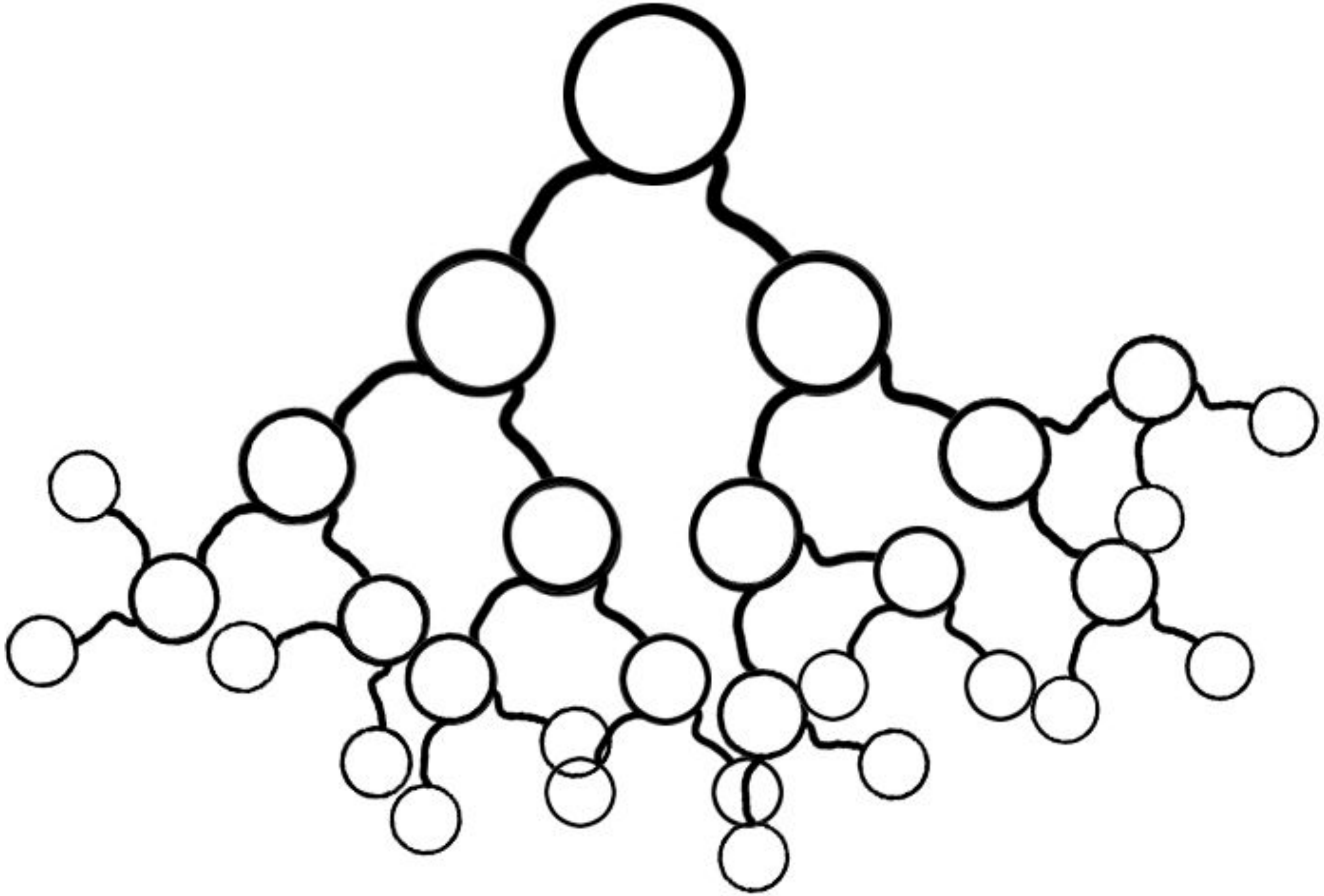
- When the problem has a tree-like structure.
- When the problem requires backtracking.

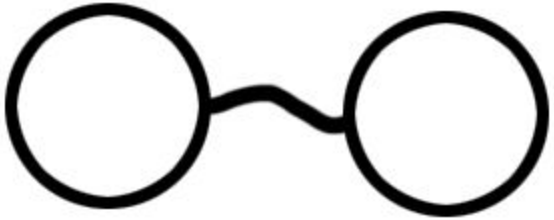
When should we use recursion?

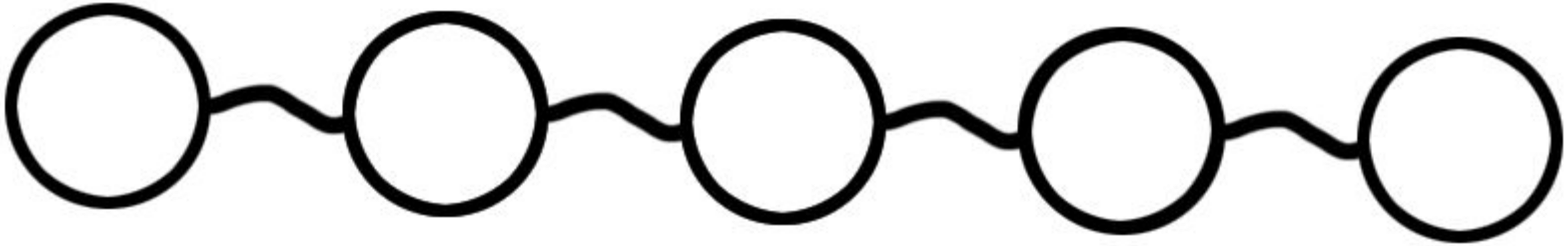
- When the problem has a tree-like structure.
- When the problem requires backtracking.

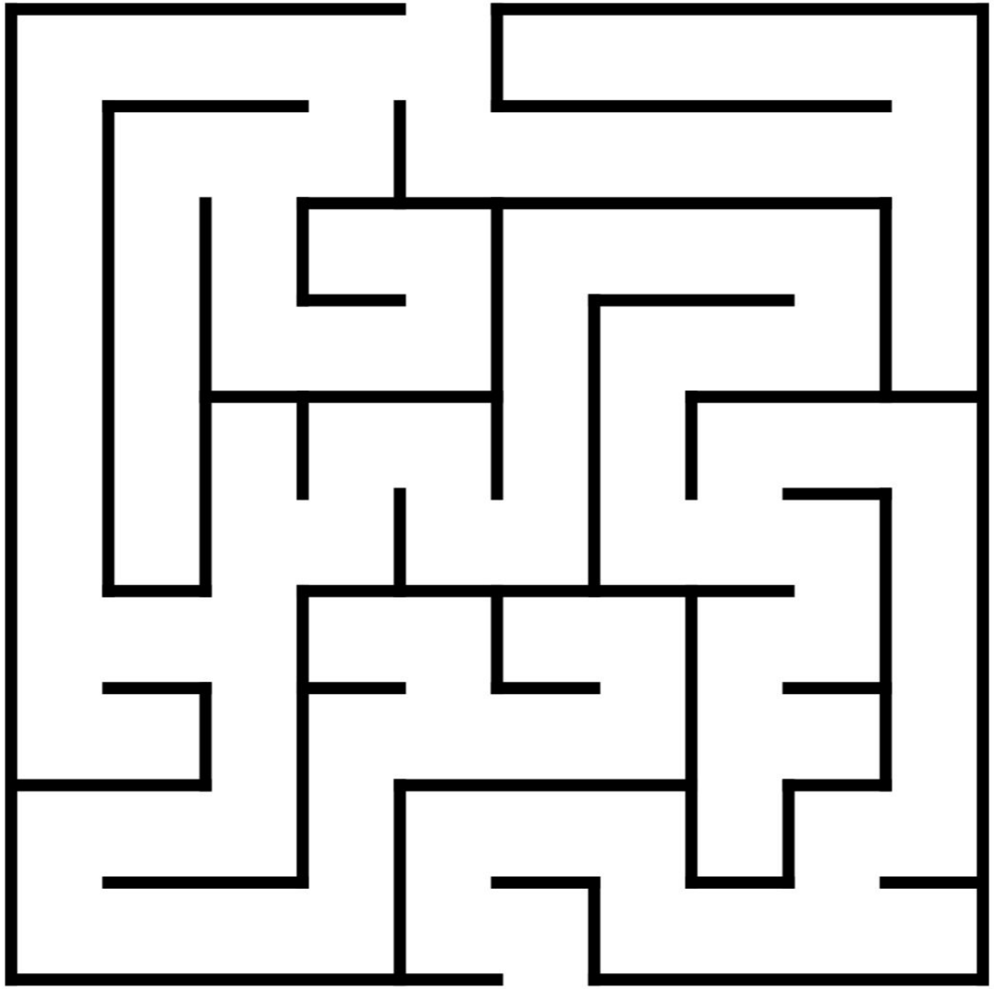
(Both of these are required.)

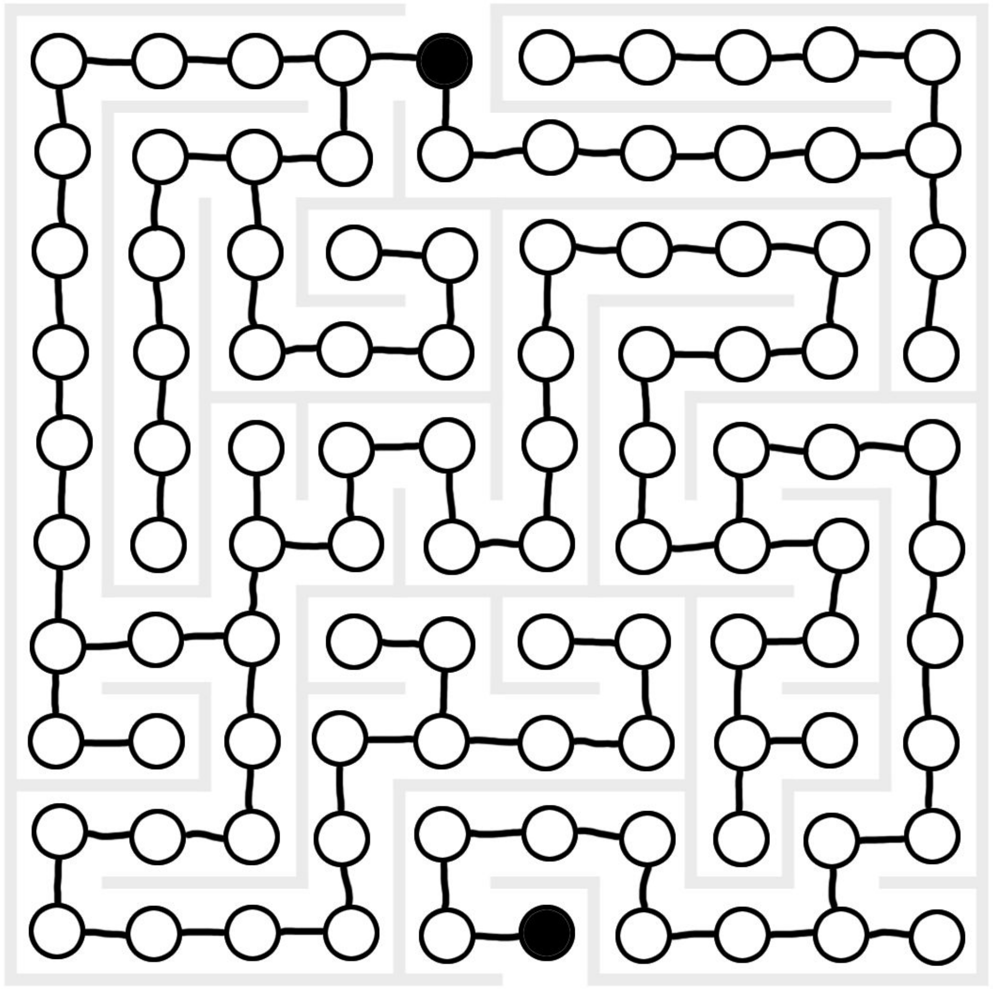


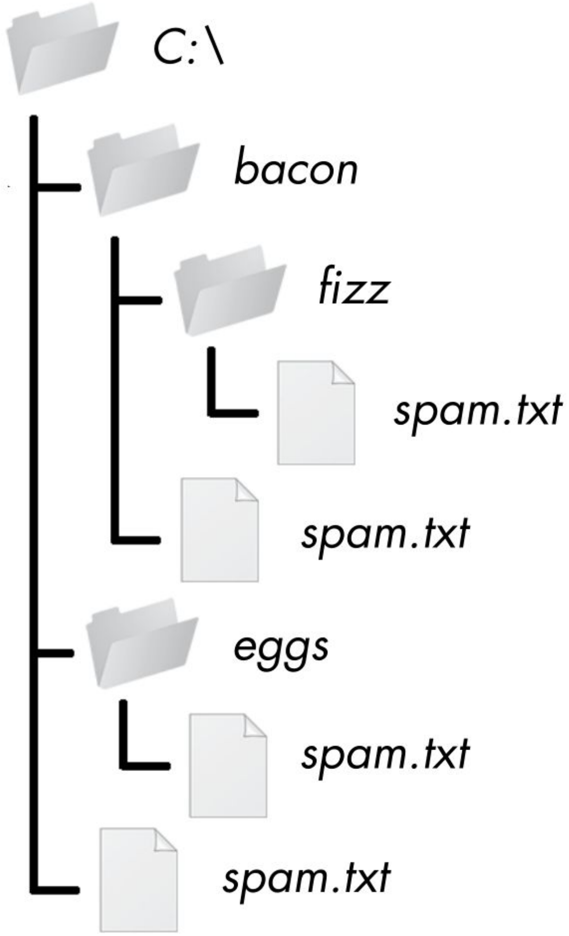











C:\
bacon
fizz
spam.txt
spam.txt
eggs
spam.txt
spam.txt

So when should we use recursion?

- When the problem has a tree-like structure.
- When the problem requires backtracking.

Otherwise, you don’t have to use recursion.

```
factorial(1001)
```

factorial(1001)

```
Traceback (most recent call last):
  File "factorialByRecursion.py", line 8, in <module>
    print(factorial(1001))
  File "factorialByRecursion.py", line 7, in factorial
    return number * factorial(number - 1)
  File "factorialByRecursion.py", line 7, in factorial
    return number * factorial(number - 1)
  File "factorialByRecursion.py", line 7, in factorial
    return number * factorial(number - 1)
  [Previous line repeated 995 more times]
  File "factorialByRecursion.py", line 2, in factorial
    if number == 1:
RecursionError: maximum recursion depth exceeded in comparison
```

Tail Call Optimization/Elimination

What if the problem is big enough that it really does require more than 1000 function calls?

```
factorial(1001)
```

Tail Call Optimization/Elimination

In code, tail call optimization/elimination is when the recursive function call is the last thing in the function before it returns:

```
def recursiveFunc(params):
    # blah blah blah
    return recursiveFunc(params) # RECURSIVE CASE
```

(The recursive function call comes at the “tail” of the function.)

Tail Call Optimization/Elimination

```
return recursiveFunc(params) # RECURSIVE CASE
```

You won’t need to hold on to local variables, because there’s no code after the recursive function call that will need them.

Tail Call Optimization/Elimination

```
return recursiveFunc(params) # RECURSIVE CASE
```

You won’t need to hold on to local variables, because there’s no code after the recursive function call that will need them.

There’s no need to keep the frame object on the call stack.

Tail Call Optimization/Elimination

```
return recursiveFunc(params) # RECURSIVE CASE
```

You won’t need to hold on to local variables, because there’s no code after the recursive function call that will need them.

There’s no need to keep the frame object on the call stack.

You can go beyond 1000 function calls because the call stack isn’t growing.

TCO prevents stack overflows.

Tail Call Optimization/Elimination

Tail call optimization is overrated.

Normal Recursive Factorial Can’t be TC Optimized

```
def factorial(number):
    if number == 1:
        # BASE CASE
        return 1
    else:
        # RECURSIVE CASE
        return number * factorial(number - 1)
print(factorial(5))
```

Factorial with Tail Call Optimization

```
def factorial(number, accumulator=1):
    if number == 0:
        # BASE CASE
        return accumulator
    else:
        # RECURSIVE CASE
        return factorial(number - 1, number * accumulator)
print(factorial(5))
```

Tail call optimization is a compiler trick...

Tail call optimization is a compiler trick…

...that CPython doesn’t implement...

Tail call optimization is a compiler trick…

...that CPython doesn’t implement...

...and never will.

“If you want a short answer, it's simply unpythonic.”

-Guido

http://neopythonic.blogspot.com.au/2009/04/tail-recursion-elimination.html

http://neopythonic.blogspot.com.au/2009/04/final-words-on-tail-calls.html

Fibonacci Sequence

1, 1, 2, 3, 5, 8, 13, 21, 34 ...

Fibonacci Sequence

1, 1, 2, 3, 5, 8, 13, 21, 34 …

fib(1) = 1, fib(2) = 1, fib(3) = 2, fib(4) = 3

Fibonacci Sequence

1, 1, 2, 3, 5, 8, 13, 21, 34 …

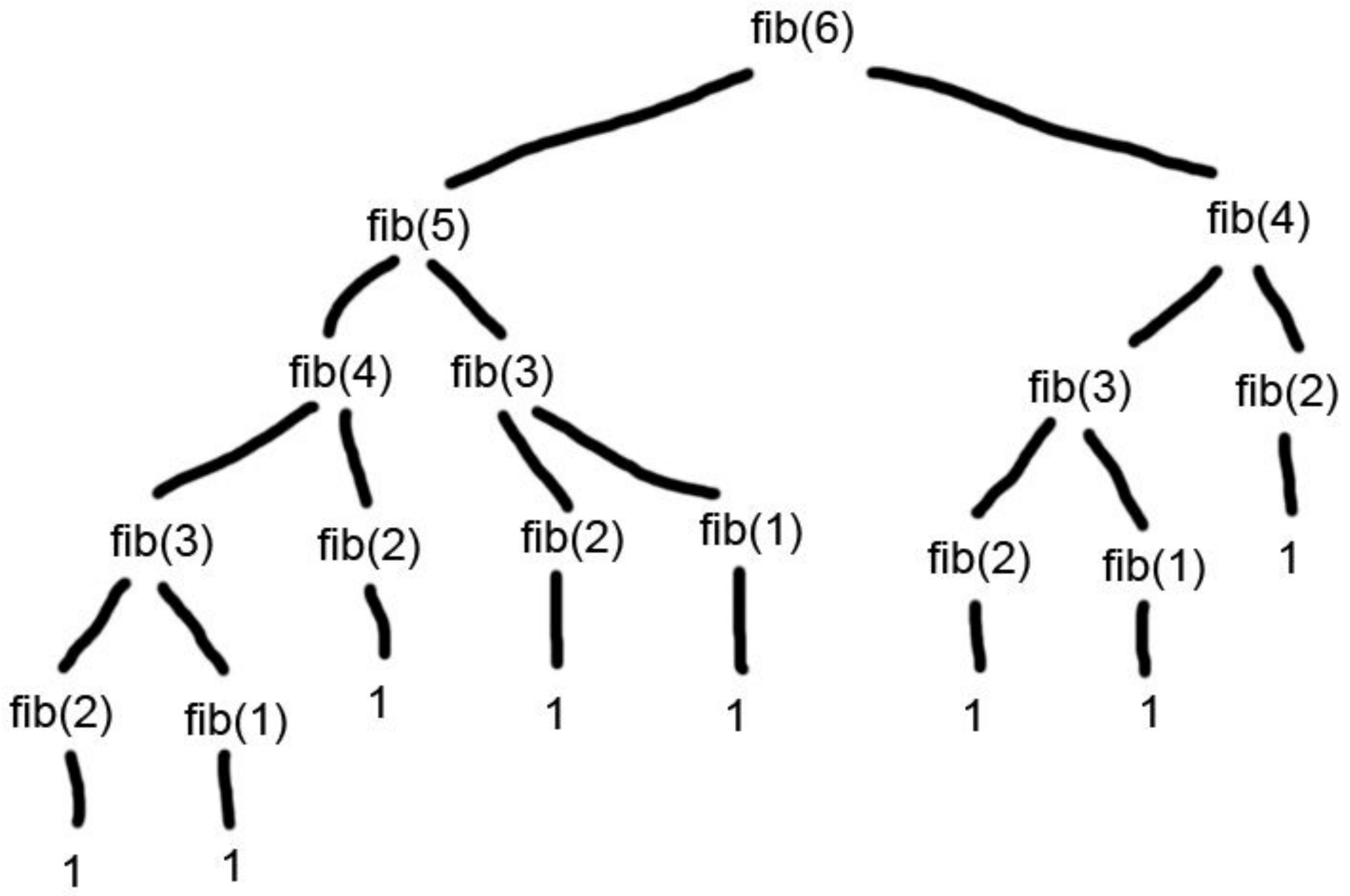
fib(1) = 1, fib(2) = 1, fib(3) = 2, fib(4) = 3

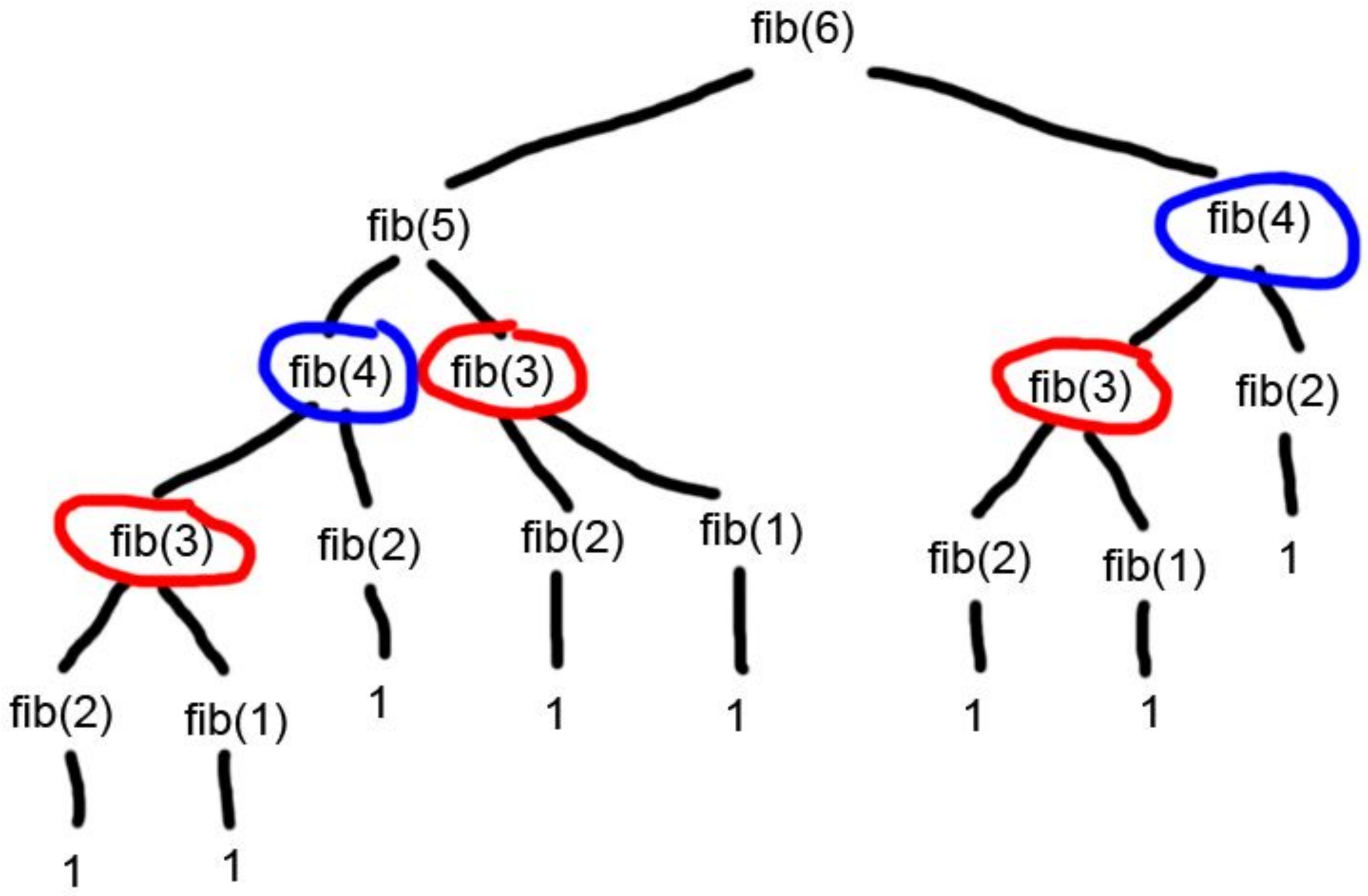
fib(N) = fib(N - 1) + fib(N - 2)

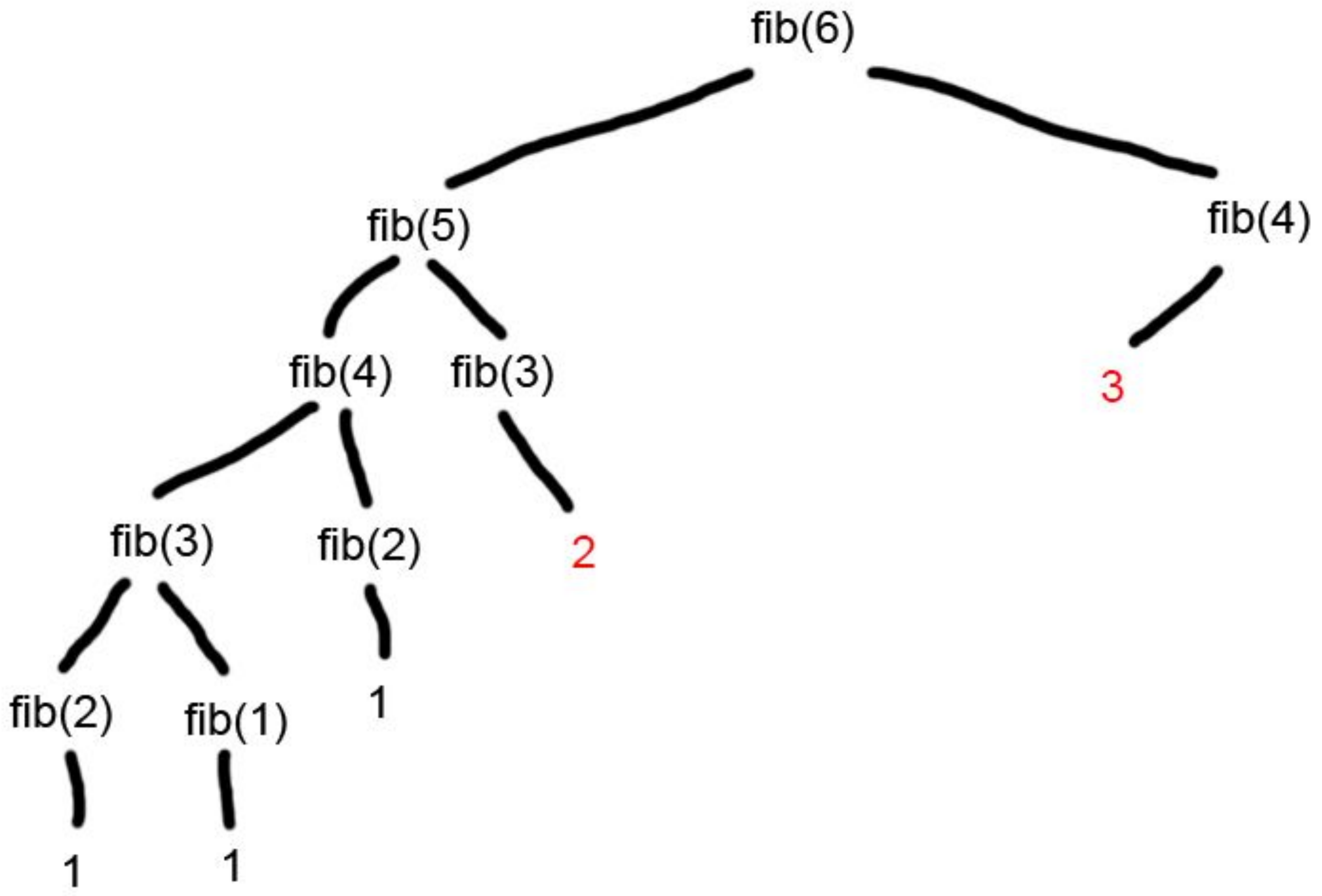
fib(1) and fib(2) = 1

Recursive Fibonacci

```
def fib(nthNumber):
    if nthNumber == 1 or nthNumber == 2:
        # BASE CASE
        return 1
    else:
        # RECURSIVE CASE
        return fib(nthNumber - 2) + fib(nthNumber - 1)
```


fib(6)
fib(5)
fib(4)
fib(4)
fib(3)
fib(3)
fib(2)
fib(3)
fib(2)
fib(2)
fib(1)
fib(2)
fib(1)
1
fib(2)
fib(1)
1
1
1
1
1
1
1


fib(6)
fib(5)
fib(4)
fib(4)
fib(3)
fib(3)
fib(2)
fib(3)
fib(2)
fib(2)
fib(1)
fib(2)
fib(1)
1
fib(2)
fib(1)
1
1
1
1
1
1
1


fib(6)
fib(5)
fib(4)
fib(4)
fib(3)
3
fib(3)
fib(2)
2
fib(2)
fib(1)
1
1
1

Recursive Fibonacci with Memoization

```
FIB_CACHE = {}

def fib(nthNumber):
  if nthNumber in FIB_CACHE:
    return FIB_CACHE[nthNumber]

  if nthNumber == 1 or nthNumber == 2:
    # BASE CASE
    return 1
  else:
    # RECURSIVE CASE
    FIB_CACHE[nthNumber] = fib(nthNumber - 2) + fib(nthNumber -
1)
    return FIB_CACHE[nthNumber]
```

Recursive Fibonacci with Memoization

```
import functools

@functools.lru_cache()
def fib(nthNumber):
    if nthNumber == 1 or nthNumber == 2:
        # BASE CASE
        return 1
    else:
        # RECURSIVE CASE
        return fib(nthNumber - 2) + fib(nthNumber - 1)
```

- To understand recursion, you must first understand stacks.

- To understand recursion, you must first understand stacks.
- Recursion is hard to learn because the call stack is invisible.

- To understand recursion, you must first understand stacks.
- Recursion is hard to learn because the call stack is invisible.
- You always need at least one base case and one recursive case.

- To understand recursion, you must first understand stacks.
- Recursion is hard to learn because the call stack is invisible.
- You always need at least one base case and one recursive case.
- Function calls push a frame object onto the stack, returning pops them off.

- To understand recursion, you must first understand stacks.
- Recursion is hard to learn because the call stack is invisible.
- You always need at least one base case and one recursive case.
- Function calls push a frame object onto the stack, returning pops them off.
- Stack overflows happen when you make too many function calls without returning.

- To understand recursion, you must first understand stacks.
- Recursion is hard to learn because the call stack is invisible.
- You always need at least one base case and one recursive case.
- Function calls push a frame object onto the stack, returning pops them off.
- Stack overflows happen when you make too many function calls without returning.
- Anything you can do with recursion you can do with a loop and a stack.

- To understand recursion, you must first understand stacks.
- Recursion is hard to learn because the call stack is invisible.
- You always need at least one base case and one recursive case.
- Function calls push a frame object onto the stack, returning pops them off.
- Stack overflows happen when you make too many function calls without returning.
- Anything you can do with recursion you can do with a loop and a stack.
- Use recursion only when the problem has a tree-like structure and requires backtracking.

- To understand recursion, you must first understand stacks.
- Recursion is hard to learn because the call stack is invisible.
- You always need at least one base case and one recursive case.
- Function calls push a frame object onto the stack, returning pops them off.
- Stack overflows happen when you make too many function calls without returning.
- Anything you can do with recursion you can do with a loop and a stack.
- Use recursion only when the problem has a tree-like structure and requires backtracking.
- Tail call elimination prevents stack overflows by preventing the call stack from growing. CPython doesn’t implement TCO.

- To understand recursion, you must first understand stacks.
- Recursion is hard to learn because the call stack is invisible.
- You always need at least one base case and one recursive case.
- Function calls push a frame object onto the stack, returning pops them off.
- Stack overflows happen when you make too many function calls without returning.
- Anything you can do with recursion you can do with a loop and a stack.
- Use recursion only when the problem has a tree-like structure and requires backtracking.
- Tail call elimination prevents stack overflows by preventing the call stack from growing. CPython doesn’t implement TCO.
- Memoization can speed up recursive algorithms by caching return values.

@AlSweigart
al@inventwithpython.com
bit.ly/nbpython2018recursion