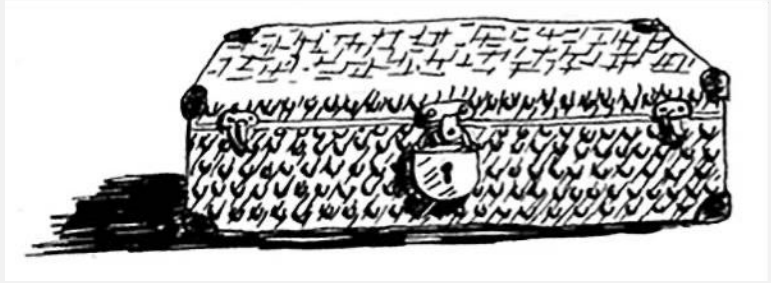
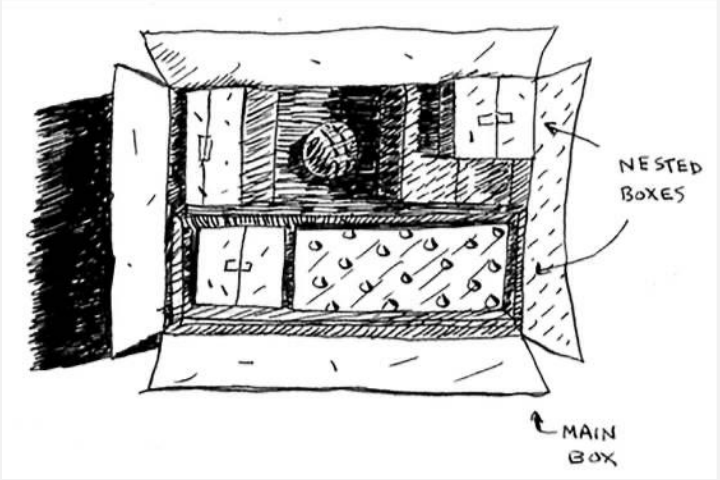
RECURSION

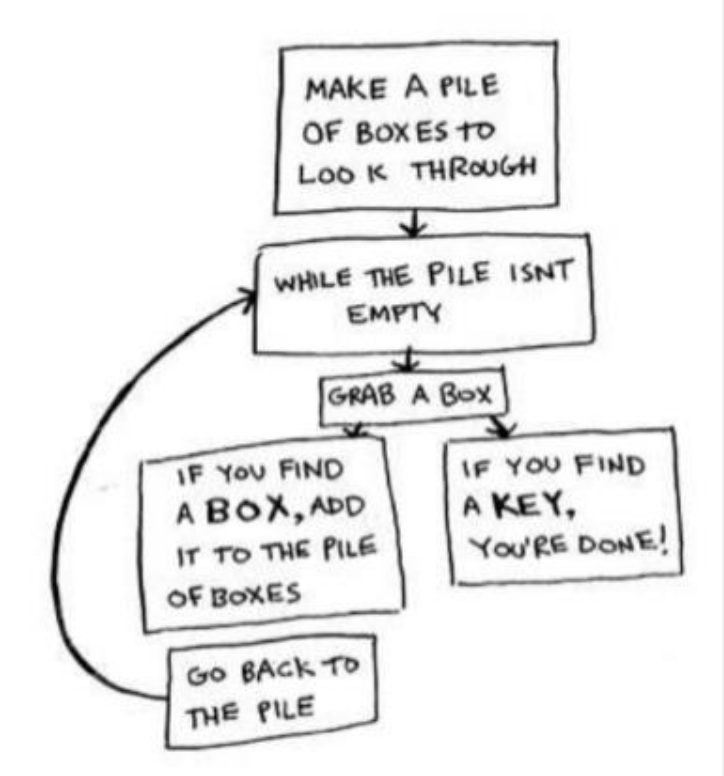
GRANDMA'S ATTIC

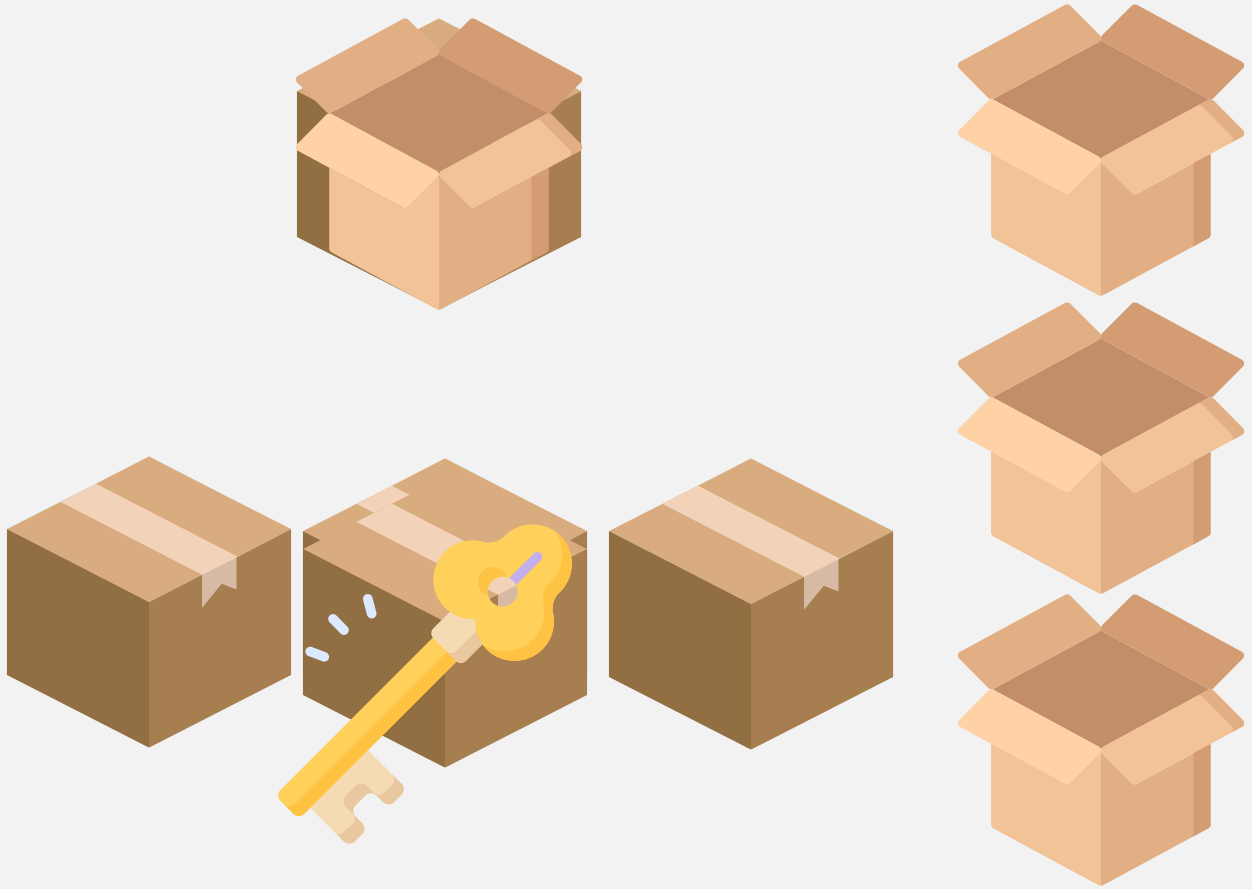




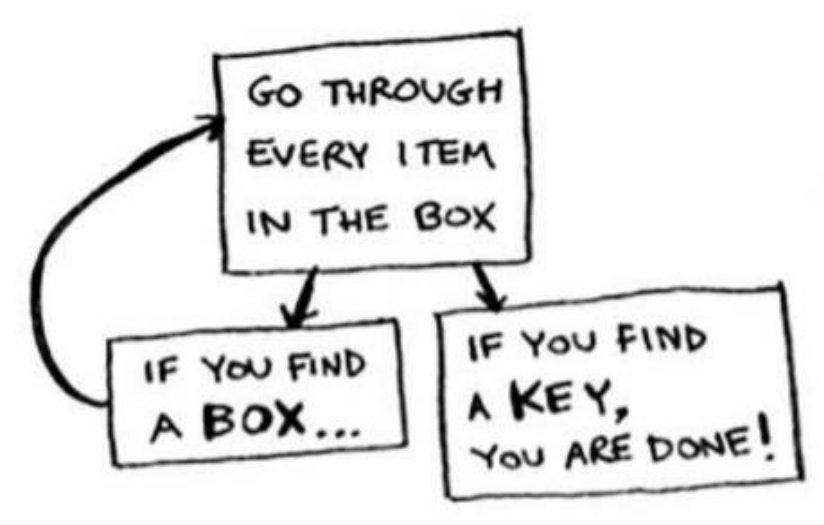
Example and resources copied from Grokking Algorithms book

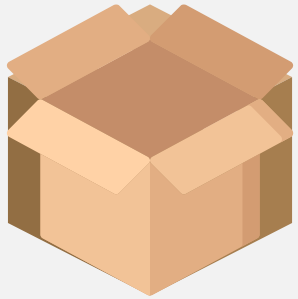
GRANDMA'S ATTIC. ITERATIVE APPROACH





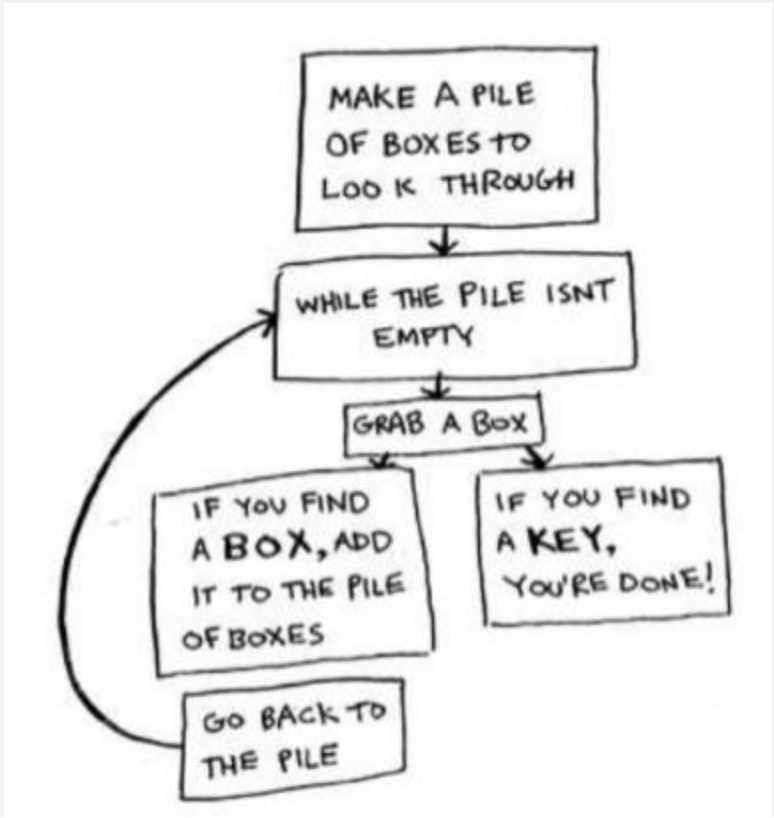
GRANDMA'S ATTIC. RECURSIVE APPROACH





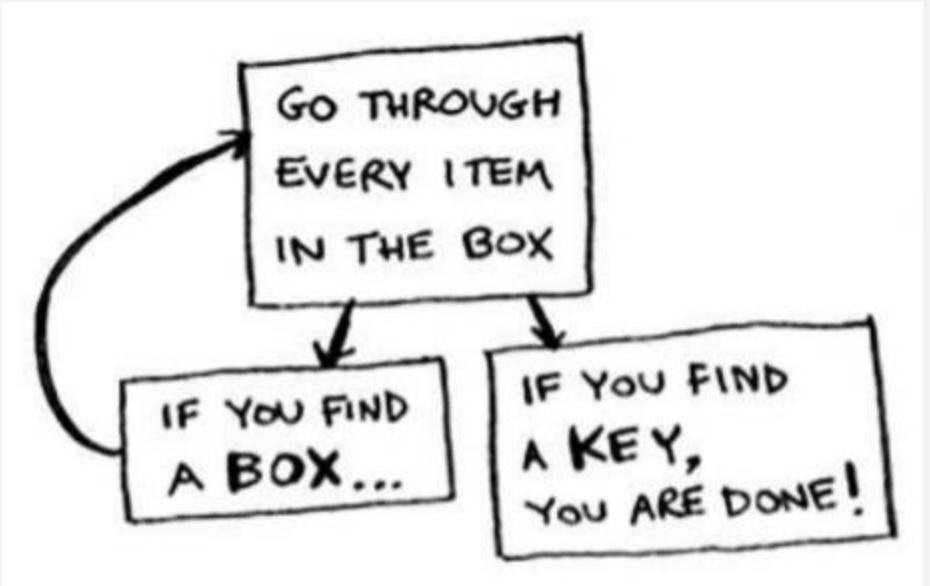
"TALK IS CHEAP
SHOW ME THE CODE"

- Linus Torvalds




```
def look_for_key(main_box):
  pile = main_box.make_a_pile_to_look_through()
  while pile is not empty:
    box = pile.grab_a_box()
    for item in box:
      if item.is_a_box():
        pile.append(item)
      elif item.is_a_key():
        print "found the key!"
```

ITERATIVE




```
def look_for_key(box):
  for item in box:
    if item.is_a_box():
      look_for_key(item)  <-- Recursion!
    elif item.is_a_key():
      print "found the key!"
```

RECURSIVE

“Loops may achieve a performance gain for your program.
Recursion may achieve a performance gain for your programmer.
Choose which is more important in your situation!”

Leigh Caldwell

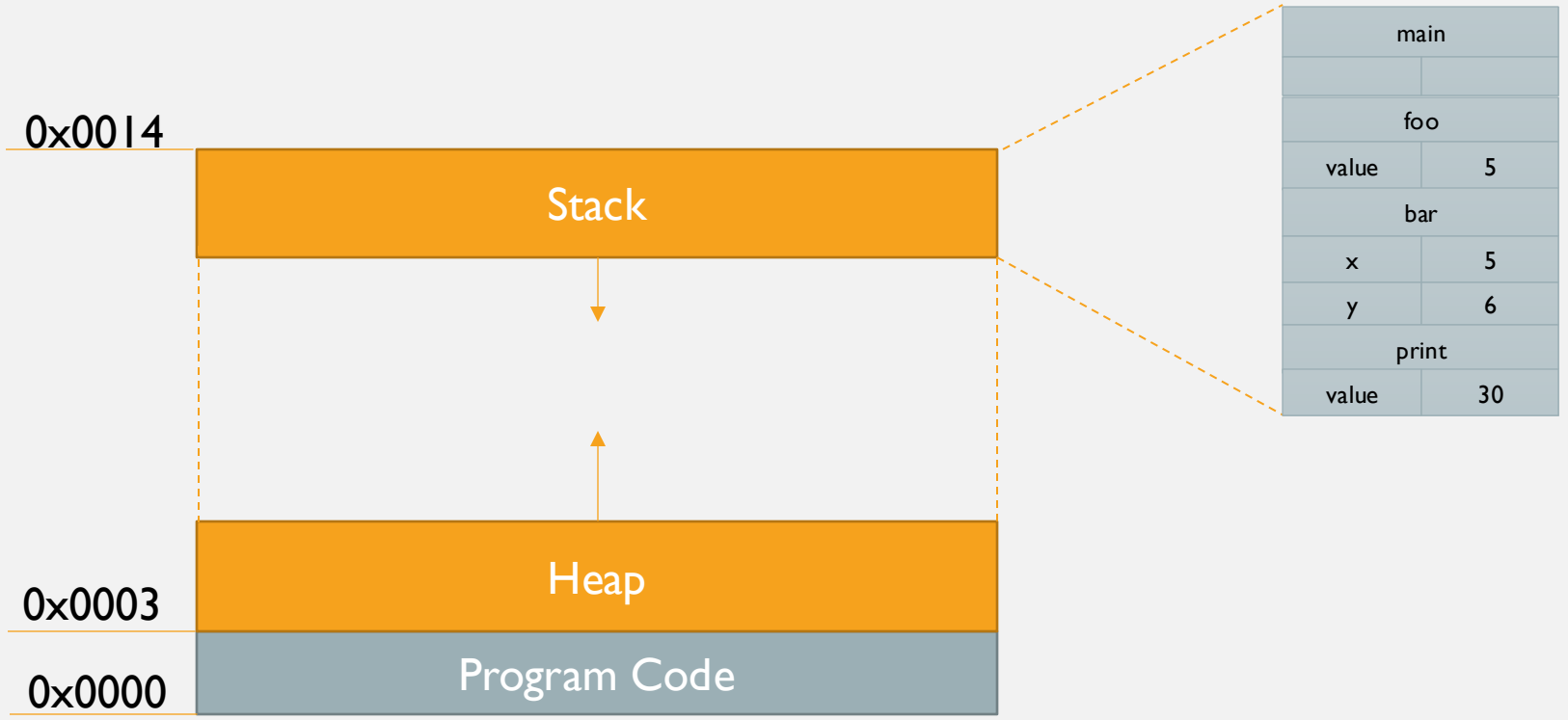
GO RECURSIVE…

Create a function **countdown** that receives an integer as a parameter.
Make it recursively print a countdown starting at the given number.


STACK LEVEL
TOO DEEP

THE PROGRAM STACK

MEMORY LAYOUT



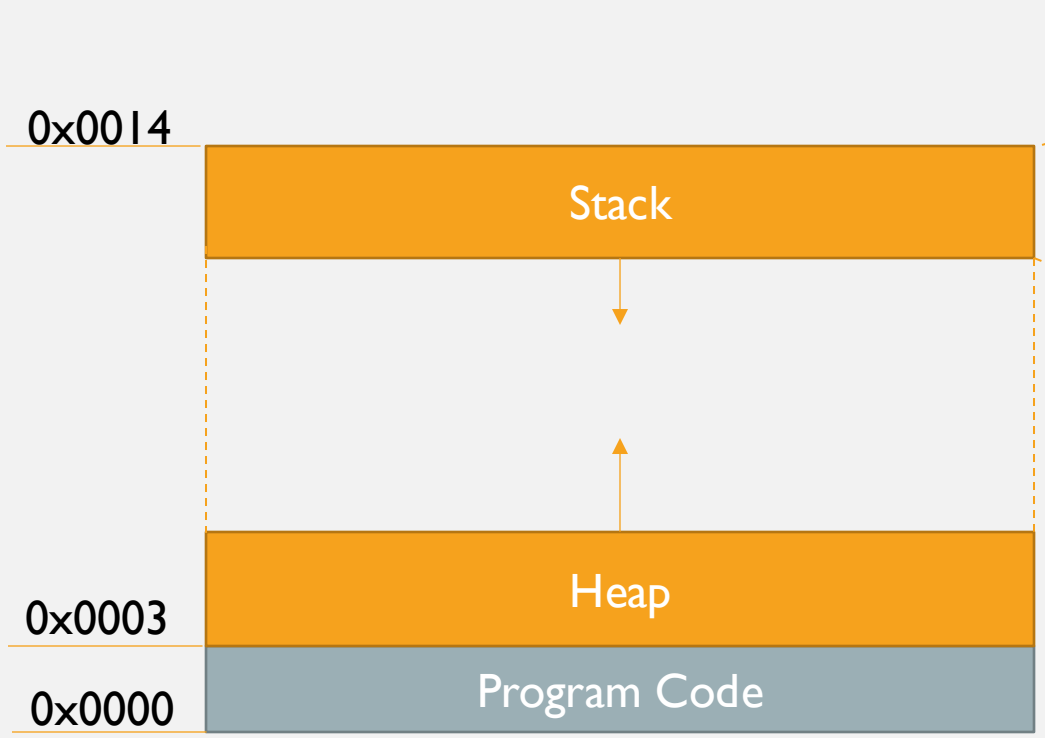

```
def bar(x, y):
  print(x * y)

def foo(value):
  bar(value, value + 1)

def main():
  foo(5)

main()
```

MEMORY LAYOUT



recursive_countdown	
value	999
print	
value	999
recursive_countdown	
value	998
print	
value	998
recursive_countdown	
value	997

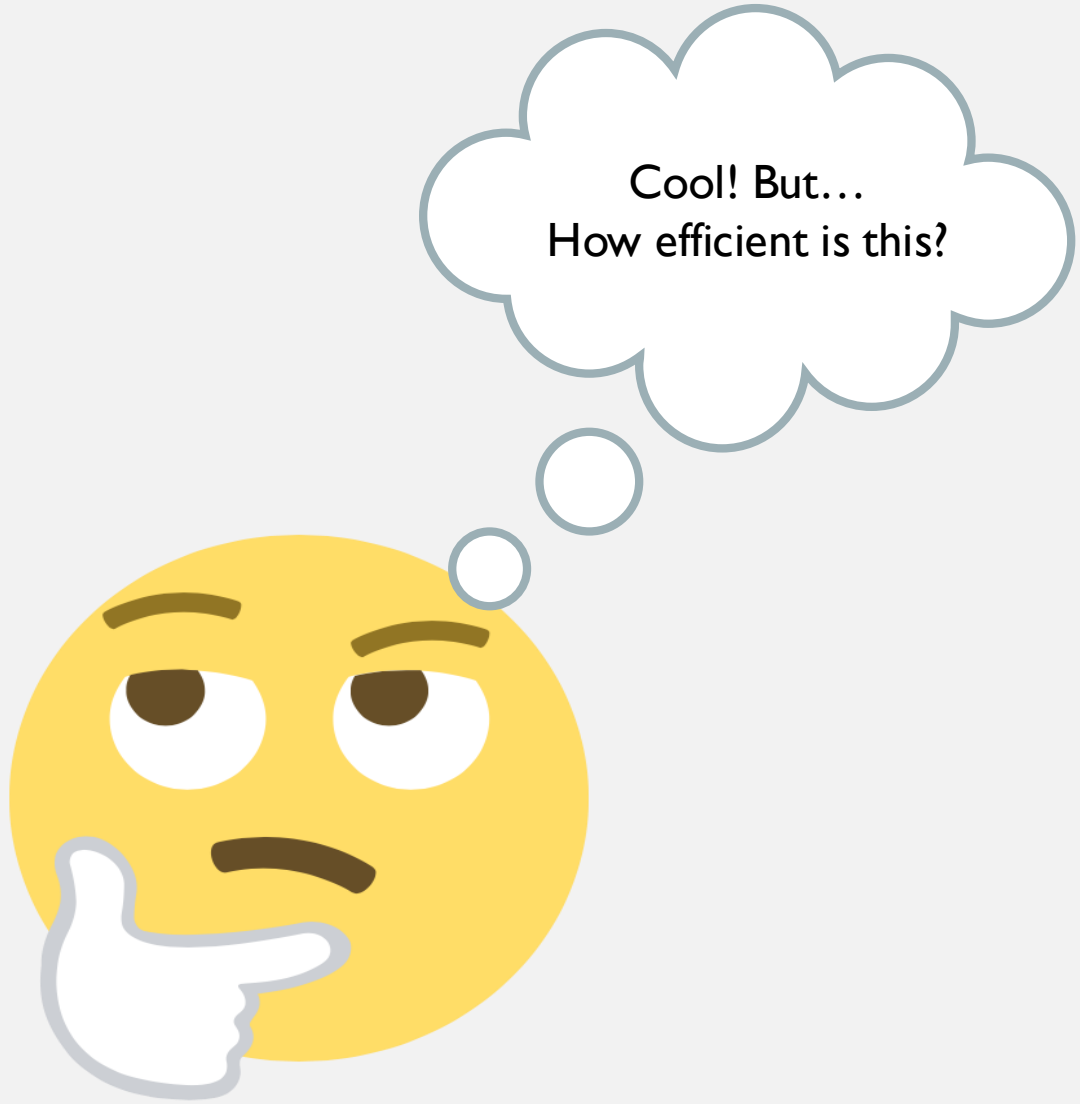
```
def recursive_countdown(value):
  print(value)
  recursive_countdown(value - 1)
```

We need a **Base Case** here!: A condition used to stop the recursion

GO RECURSIVE AGAIN…

Create a function **power** that receives two positive integers, **x** and **y**
Make it recursively calculate the power of **x** elevated to **y**.

What would be the base case?


Cool! But…
How efficient is this?


REMEMBER
MEMORY USAGE
imgflip.com

DIVIDE AND CONQUER

… and combine

DIVIDE AND CONQUER (AND COMBINE)

- It is a programming technique that uses recursion to solve problems. It proceeds in three steps:

1. Divide the main problem into smaller problems, until a base case is found
2. Resolve (Conquer) each subproblem
3. Combine the results of each subproblem until the result for the main problem is composed.

How would you calculate $n!$?

DIVIDE

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

$$5! = 5 * 4!$$

$$5! = 5 * (4 * 3!)$$

$$5! = 5 * (4 * (3 * (2 * (1))))$$

CONQUER

Base case

$$5! = 5 * (4 * (3 * (2 * (1))))$$

COMBINE

```
5! = 5 * (4 * (3 * (2 * (1))))
5! = 5 * (4 * (3 * (2)))
5! = 5 * (4 * (6))
5! = 5 * (24)
5! = 120
```

"TALK IS CHEAP
SHOW ME THE CODE"

- Linus Torvalds

```
def factorial(x):
    if x == 1:
        return 1
    else:
        subproblem_result = factorial(x-1)
        return x * subproblem_result
```

Conquer (base case)

Divide

Combine

REVIEW

RECURSION IS…

- a) A programming technique where a function invokes itself
- b) A problem solving technique where we divide a problem into smaller and easier problems until we find a trivial one
- c) A religion

RECURSION IS

- a) generally more efficient than the iterative approach
- b) generally simpler to reason than the iterative approach
- c) Both

ONE OF THE MAIN CONCERNS TO KEEP IN MIND WHEN USING RECURSION IS

- a) memory consumption
- b) cpu heating
- c) none of the above

DIVIDE AND CONQUER IS…

- a) A programming technique that relies on recursion
- b) A problem solving technique where we divide a problem into smaller and easier problems until we find a trivial one
- c) A religion

FIBONACCI SEQUENCE

- $F(n) = F(n-1) + F(n-2)$
- Only for n > 1
- Code a function to calculate the Fibonacci Sequence

EXERCISE

- Create a function to calculate, recursively, the sum of an array of integers

EXERCISE

- Towers of Hanoi: https://yongdanielliang.github.io/animation/web/TowerOfHanoi.html