Intro to Computer Science

CS-UH 1001

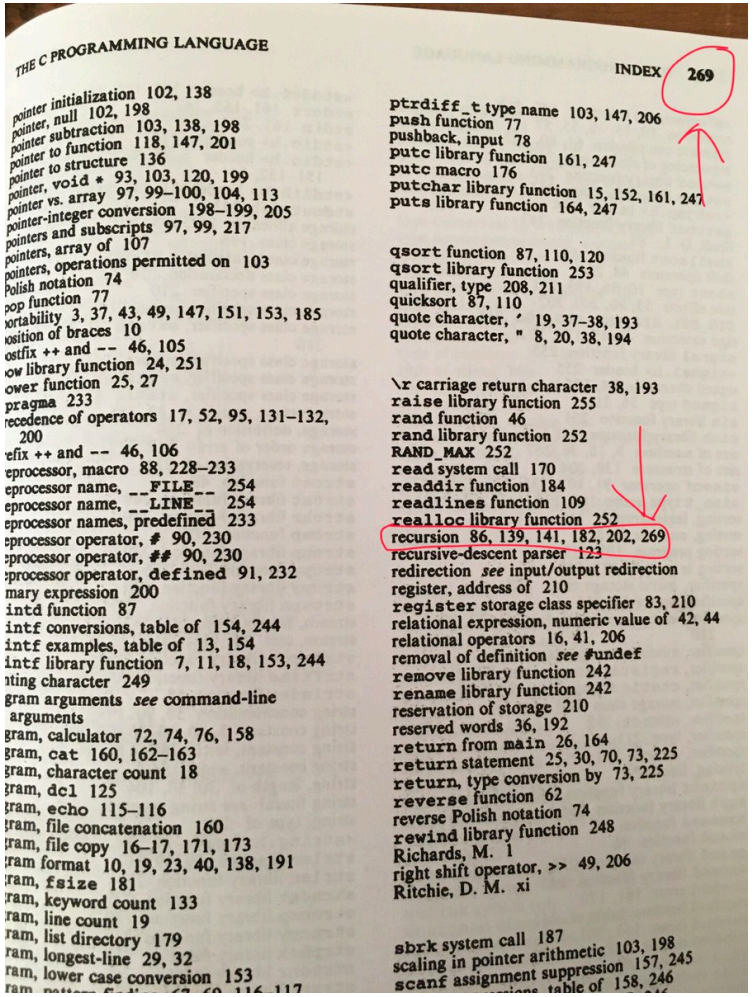
Lecture 20 – Recursions

Recursion

- Recursion is more than a programming topic, it is
  - a method of problem solving
  - a different way of thinking of problems - that’s the tricky part ;)
- Recursion can solve some problems better than iterations (loops)
- Recursion can lead to elegant, simplistic and short code (when used well)
- Recursion is technically simply:
  - A function that calls itself (recursive function)

Recursion in the Real World

- Text Books

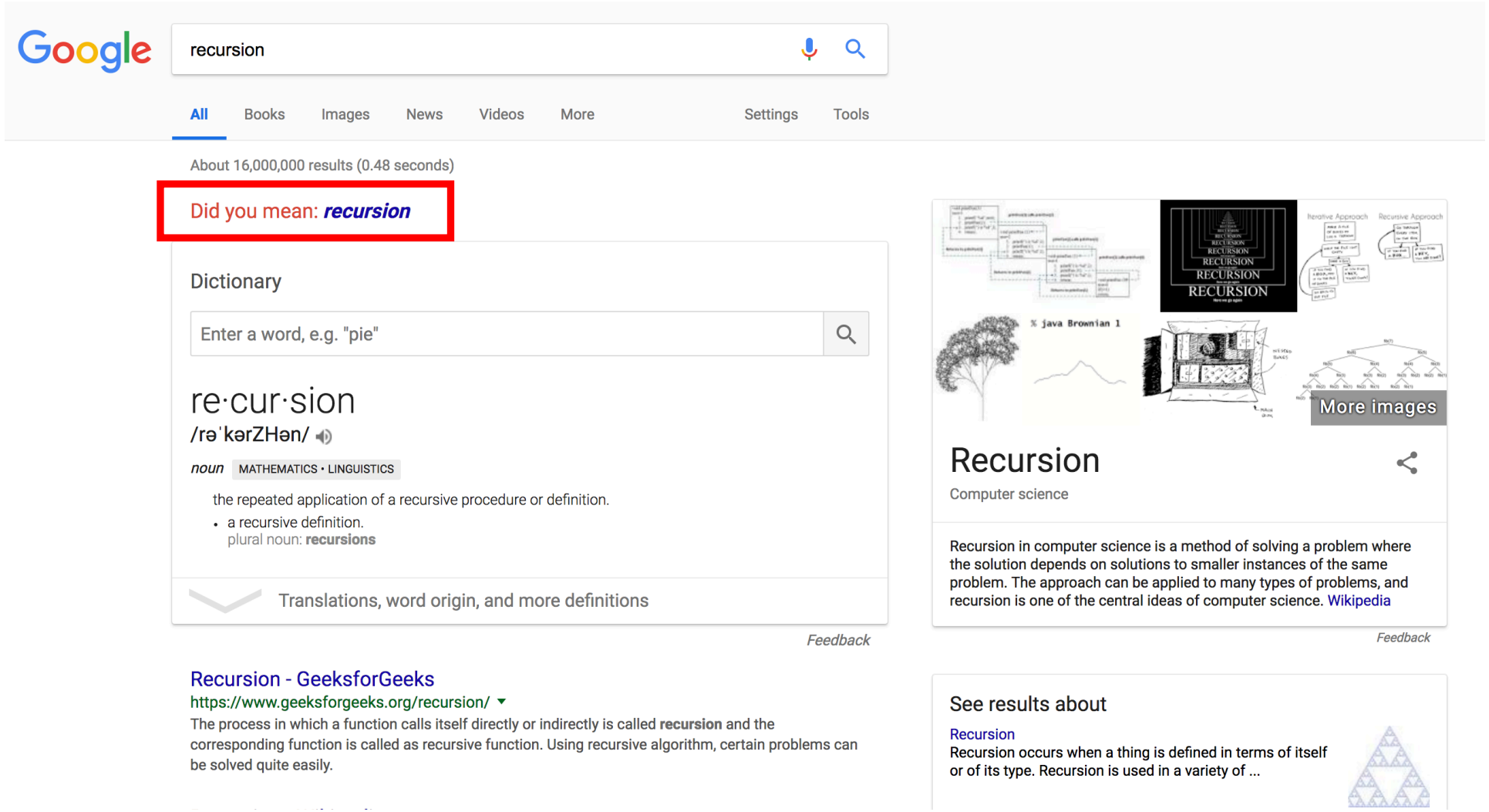


THE C PROGRAMMING LANGUAGE

INDEX 269

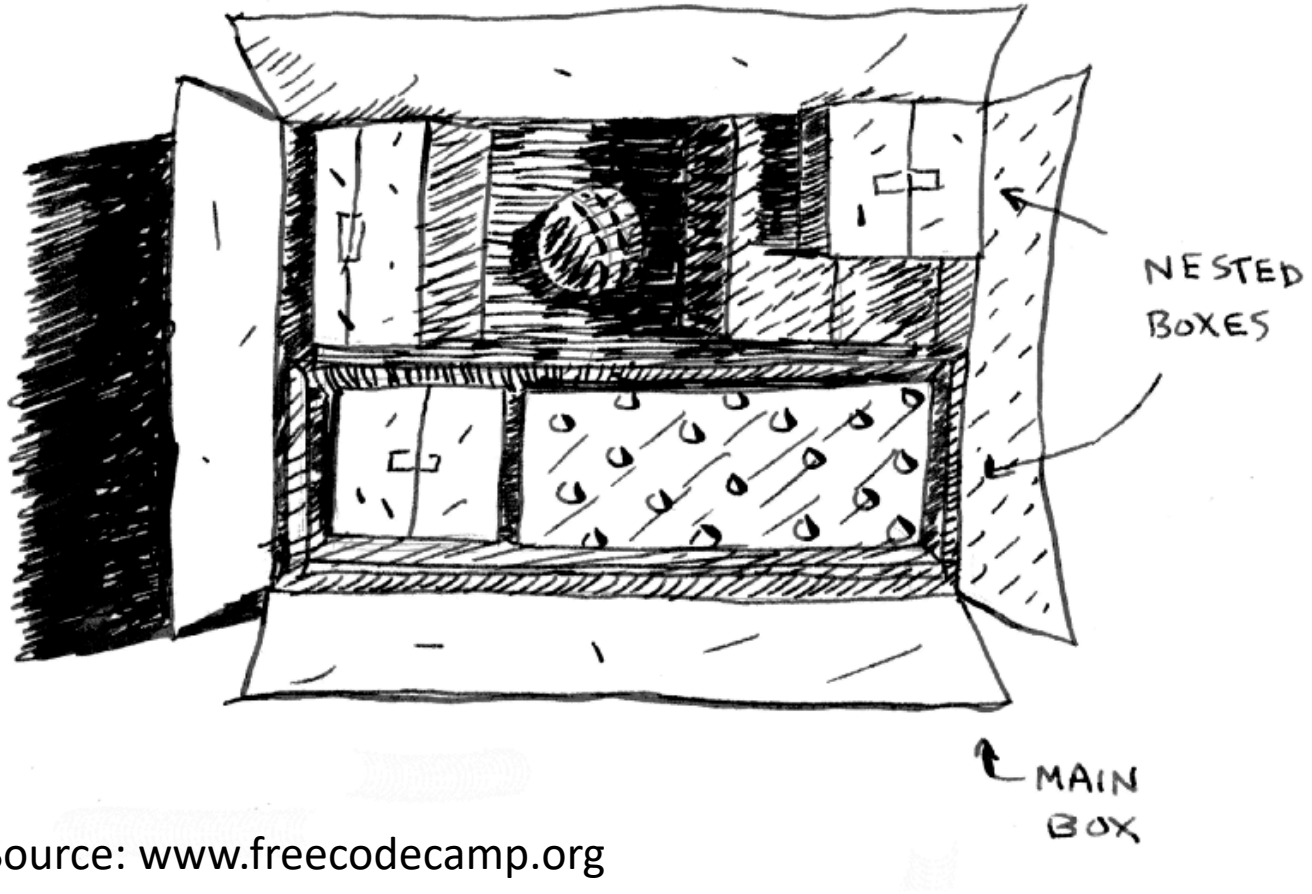
- Shopping

Recursion on google.com



Motivating Example

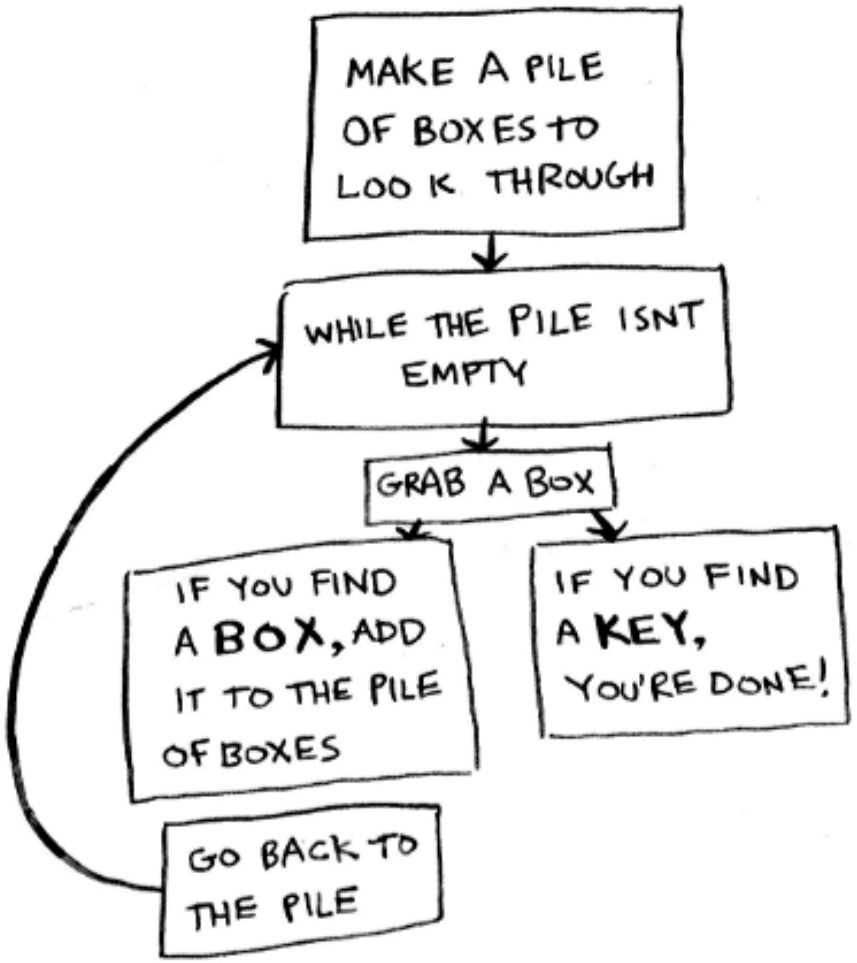
- A key in a box in a box in a box in a box…



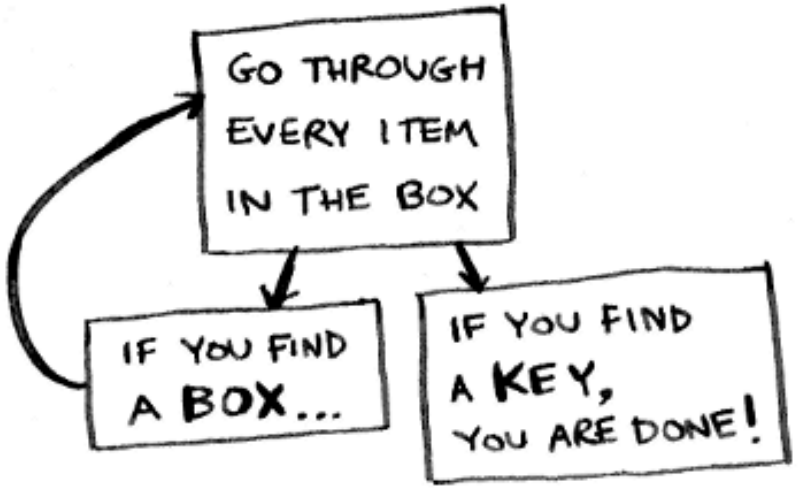
Source: www.freecodecamp.org

Motivating Example

Iterative Approach

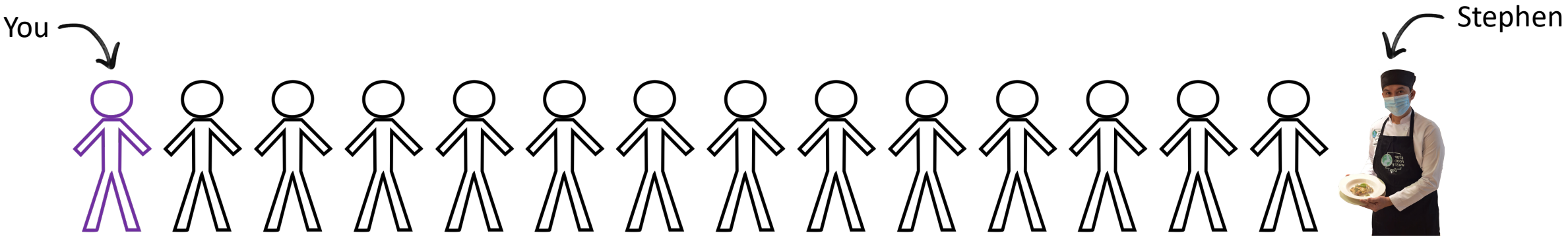


Recursive Approach



Source: www.freecodecamp.org

Another Example (Inspired by Real Events)



- How to find out how many people are in the queue in front of you?
- Assumptions:
  - People are not allowed to move
  - People are only allowed to speak to the person in front of them or behind them

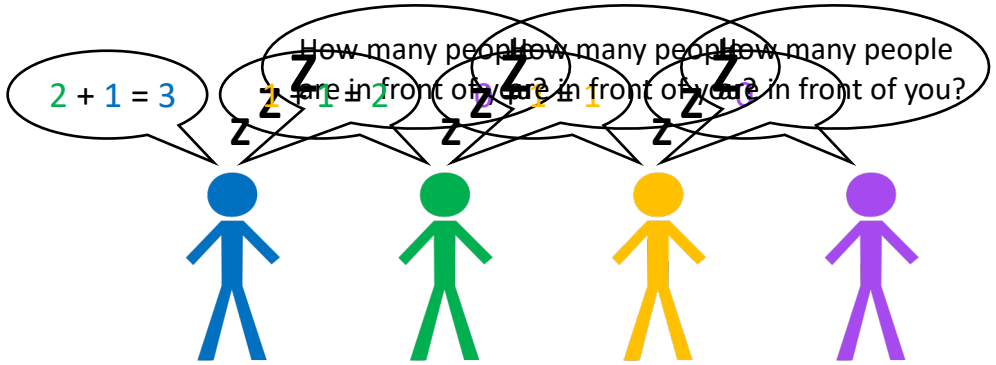
Another Example

- Recursion is all about breaking a big problem into smaller instances of the same problem
- Each person can solve a small part of the problem

- Solution:
  - If there is someone in front of you, ask him/her how many people are in front of him/her
    - When he/she responds with a value N, then you will answer with N+1
  - If there is nobody in front of you/him/her, you/he/she will answer 0

Dry-Run Solution

- Suppose you are 4th in line, and you want to know how many people are in front of you
- Person 4 (You): Tap and ask person 3 how many people are in front and wait for reply
  - Person 3: Tap and ask person 2 and wait for reply
    - Person 2: Tap and ask person 1 and wait for reply
      - Person 1: As person 1 is in front and no one is left to ask, tell person behind (person 2) the answer is 0
    - Person 2: Tell person 3 behind the answer is 0 + 1 = 1
  - Person 3: Tell person 4 behind the answer is 1 + 1 = 2
- Person 4: Just use the answer of person 3 to find the final answer: 2 + 1 = 3



Components of Recursion

- If we analyze our solution, there were two main things that were happening:
  - Simplifying/Reducing the problem in terms of itself and then solving it using the same logic (Recursive case)
  - Simplification does not go on forever (Base case)
    - It breaks/stops when we encounter a problem version which could not be simplified further

Recursion Algorithm

- Recursive case: The set of instructions that will be used over and over
  - Divide: Split the problem into one or more simpler or smaller versions of the problem
  - Call: Recursive call to solve a simpler version of a problem
  - Combine: Combining the solutions of the versions into a solution for the problem/complex version

- Base case: The point where it stops applying the recursive case, the problem is simple enough to be solved directly
- In both cases, return whatever answer we arrived on

- In the queue problem:
  - Recursive case is: Tap person in front of you. Ask how many people are in front of him/her. Wait for their answer and add 1 to their answer
    - Tap person in front of you (Divide)
    - Ask how many people are in front of them (Call)
    - Wait for their answer and add 1 (Combine)
  - Base case is: Person 1. You do not execute the above. The answer is known.
  - If someone asked, tell them how many people are in front of you (return)

Recap Functions

- Function calls are like a detour in the flow of execution:
  - Instead of going to the next statement in the code, the flow jumps to the called function, executes the function body, and then comes back to pick up where it left off
- A return statement ends the execution of a function call
  - Functions can have multiple return statements

Recursion is Simple… Or Not?

- Recursion is a function that calls itself

- Example:

```
def greeting():
    print("Hello World")
    greeting()

greeting()
```

```
Hello World
Hello World
Hello World
Hello World
.
.
.
```

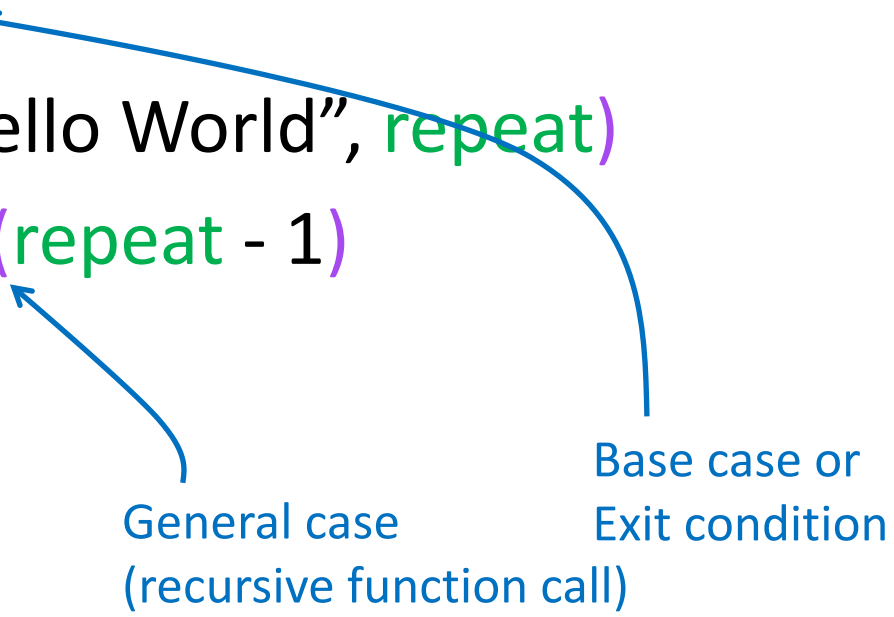
Infinite loop: There is no way for the function to stop executing

Simple Recursion with Base Case

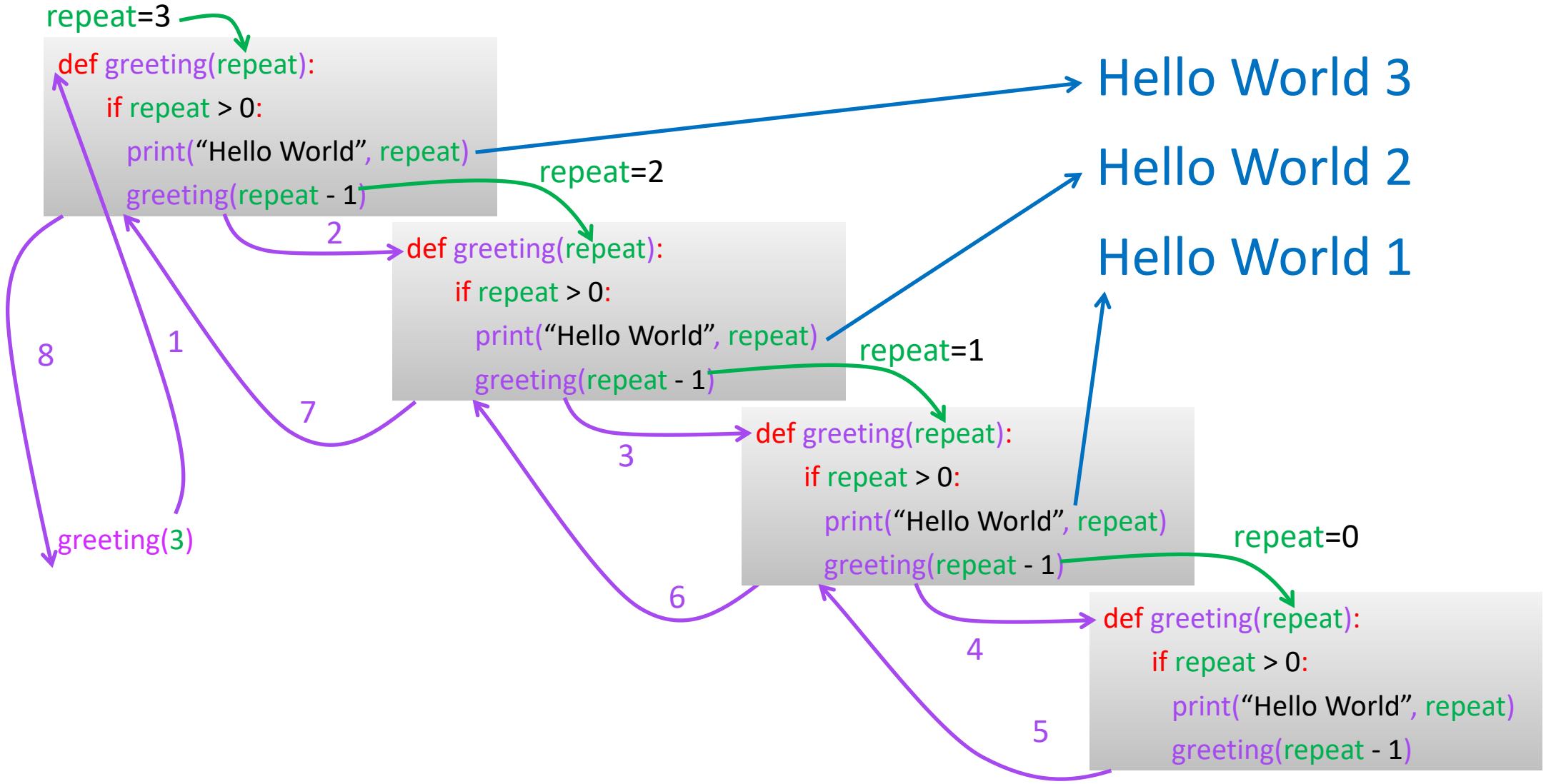
- Every recursion must have a base case!
  - Using function argument(s) to make the problem smaller

```
def greeting(repeat):
    if repeat > 0:
        print("Hello World", repeat)
        greeting(repeat - 1)

greeting(3)
```

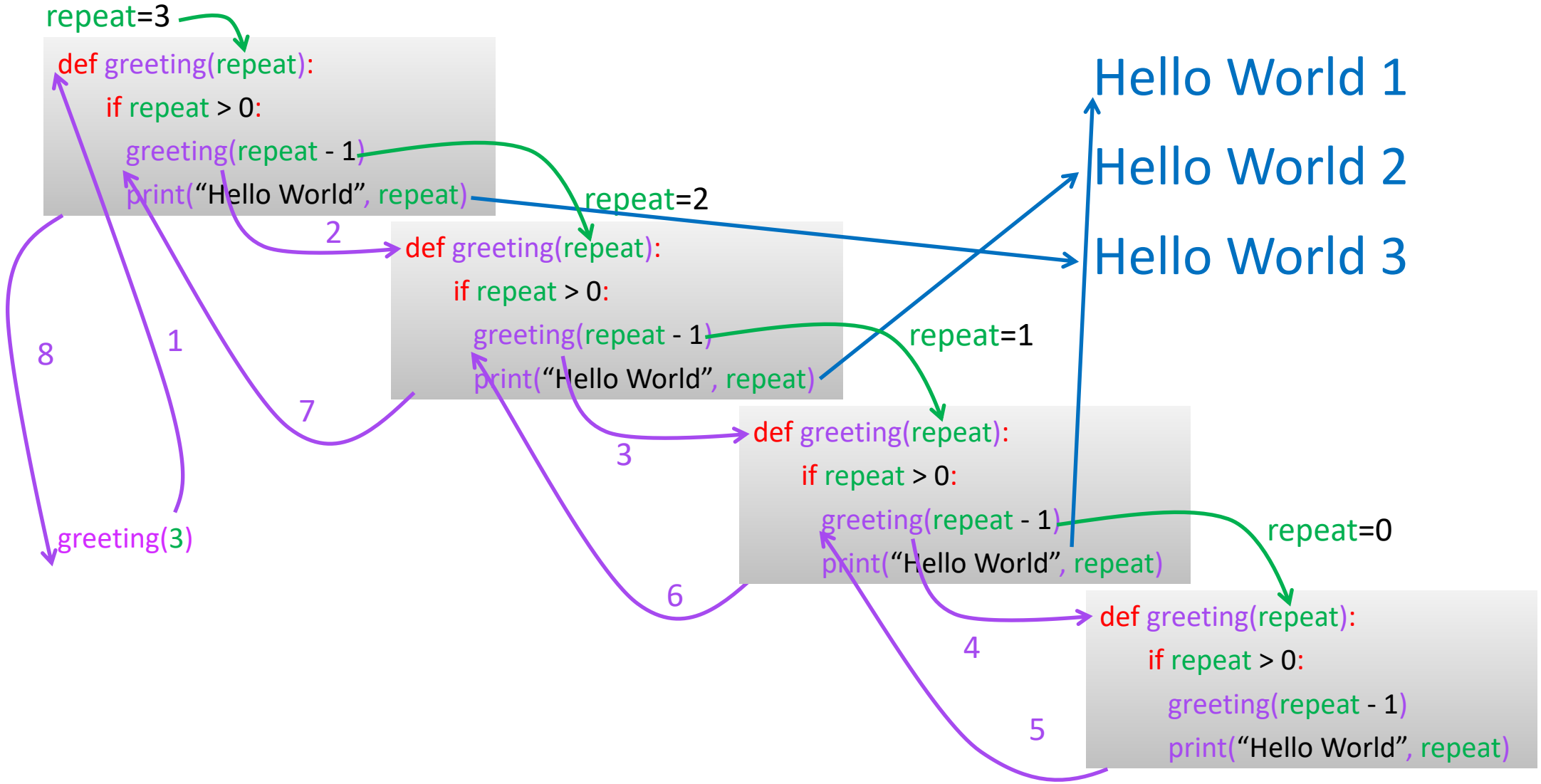
Hello World 3
Hello World 2
Hello World 1



Simple Recursion (Illustration)



Simple Recursion (Illustration)



Solving a Simple Math Problem with Recursion

- A classic example to solve using recursion is factorial (n!)

- Factorial is the product of all positive integers less-than or equal-to a given integer

- Factorial of n! is defined as:
  - If n > 0 then n! = 1 x 2 x 3 x 4 x 5 x …. x n
  - If n = 0 then 0! = 1

Factorial: The Iterative Approach

- Factorial of n! is defined as:
  - If n > 0 then n! = 1 x 2 x 3 x 4 x 5 x …. x n
  - If n = 0 then 0! = 1

- 4! = 4 x 3 x 2 x 1

- Iterative approach:

```
n = 4
factorial = 1
for i in range(1, n+1):
        factorial = factorial * i
```

Factorial: The Recursive Approach

- 4! = 4 x 3 x 2 x 1

- Factorial is recursive by nature:
  - n! = n x (n -1)! ← Recursive case
  - 0! = 1 ← Base case

4! = 4 x 3!
= 4 x 3 x 2!
= 4 x 3 x 2 x 1!
= 4 x 3 x 2 x 1 x 0! ← Base case

↓ Getting closer to the Base case...

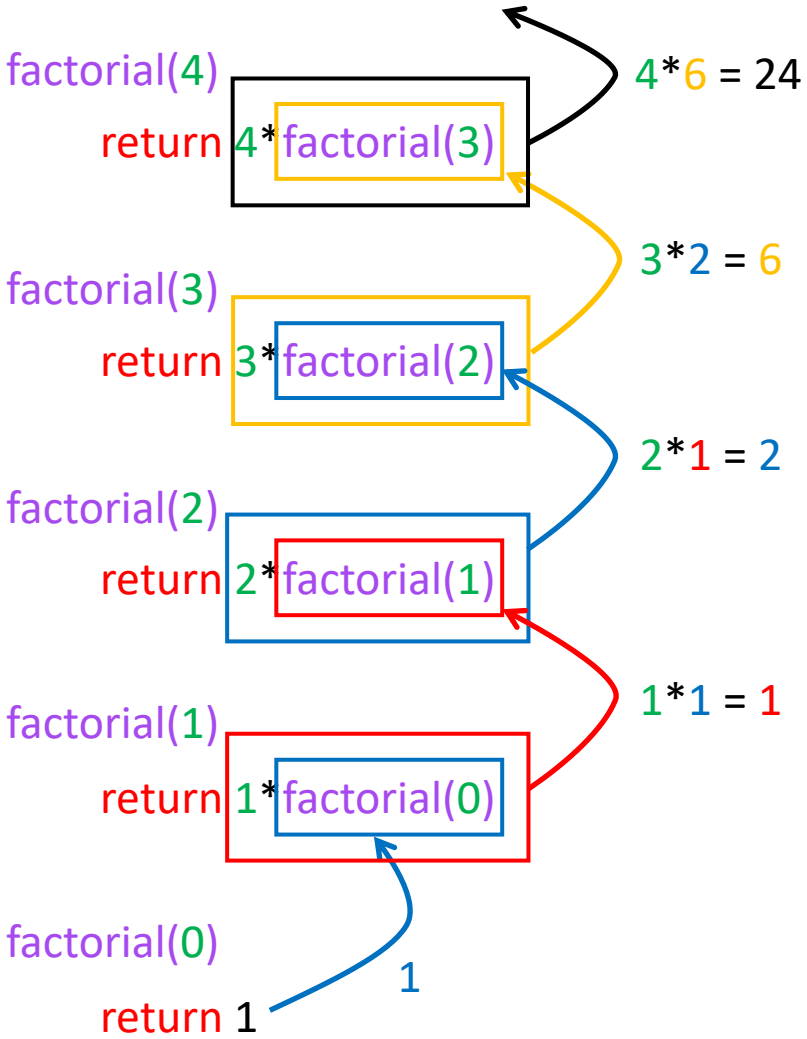
Factorial Using Recursion

```
def factorial(n):
    if n == 0:
        return 1
    return n * factorial(n - 1)
```

Base case:
0! = 1

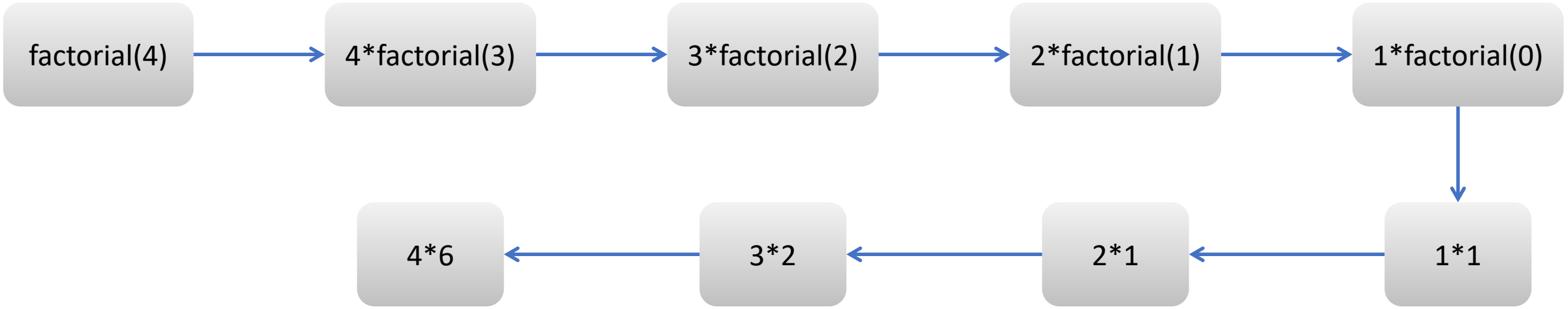
Recursive case: n! = n x (n -1)!
(decrement n towards 0)

Factorial Using Recursion




```
def factorial(n):
        if n == 0:
            return 1
        return n * factorial(n  - 1)
```

Factorial Using Recursion



- This is called linear recursion

Fibonacci Series

- The Fibonacci series is named after the Italian mathematician Leonardo Fibonacci

- 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, …

- Each number is the sum of the previous two numbers

Fibonacci Series

- 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, …

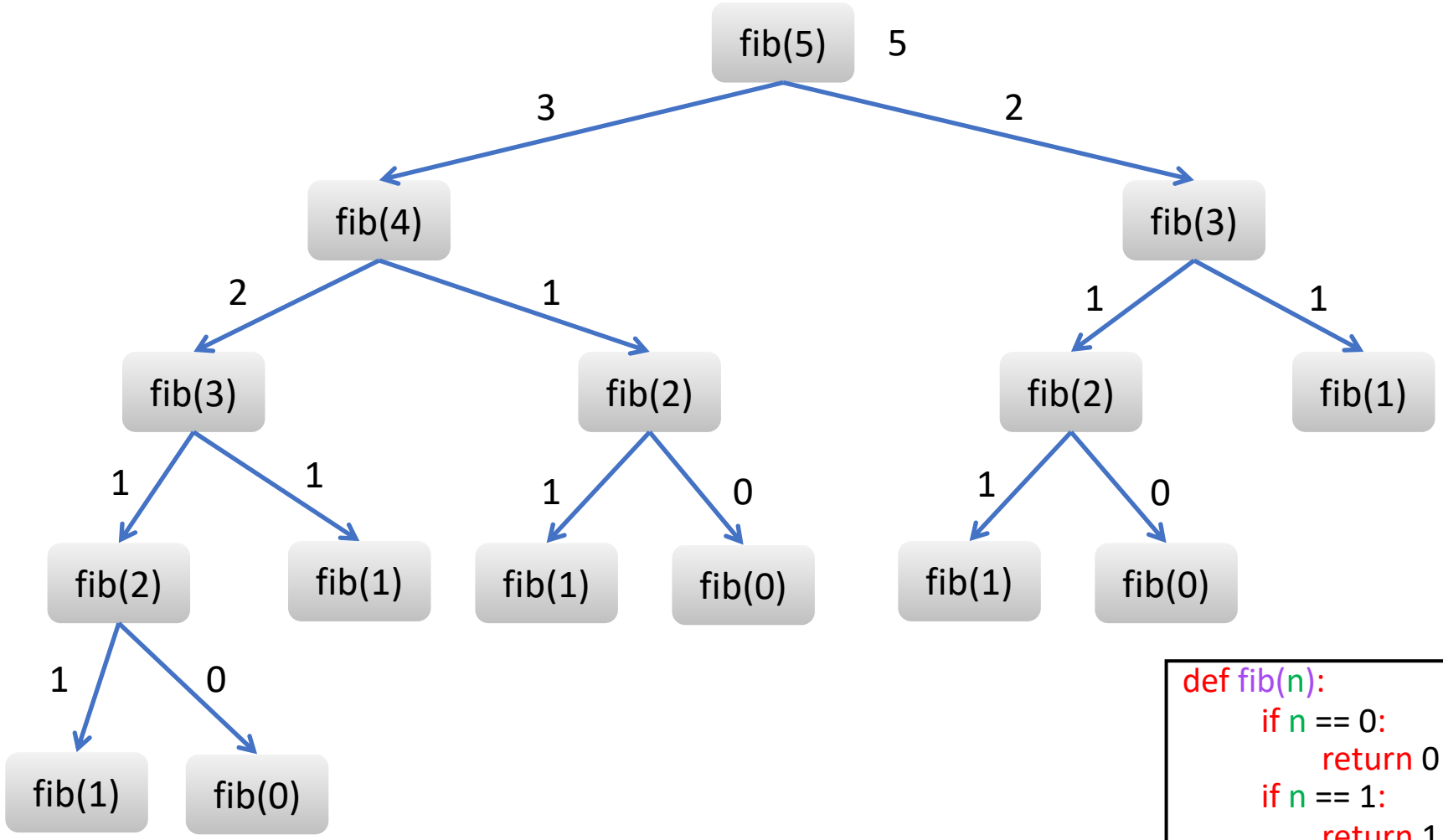
- Mathematically, the sequence fib(n) can be defined as:

  - If n = 0 then fib(n) = 0 ← Base case #1
  - If n = 1 then fib(n) = 1 ← Base case #2
  - If n > 1 then fib(n) = fib(n - 1) + fib(n - 2)

  where n is the nth term of the sequence

- The Fibonacci sequence definition by default has a recursion
- Exercise (ex_20.1): Write a recursive function for fib(n)
  e.g. fib(6) => 8

Example: Fibonacci Series fib(5)



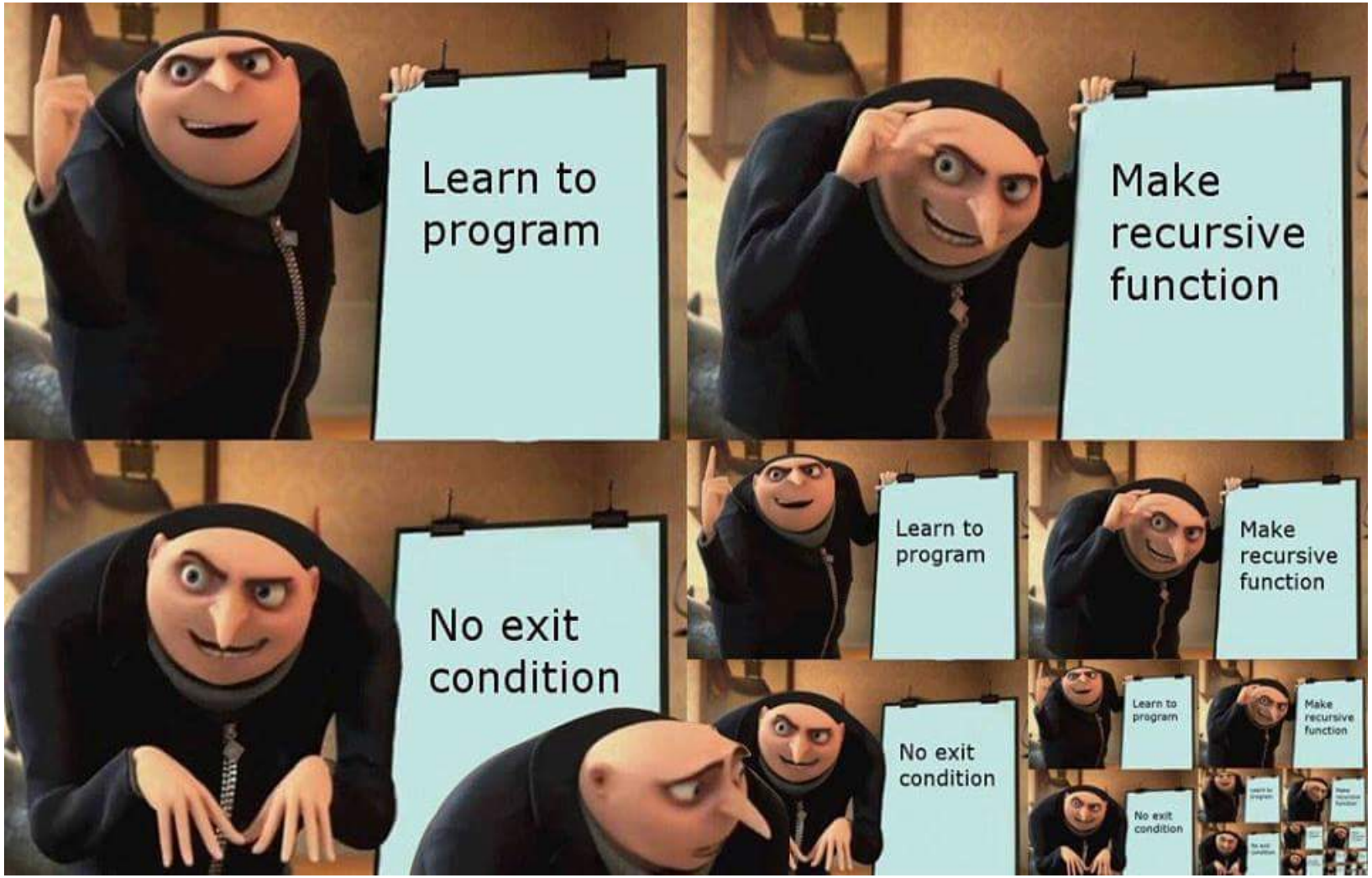
This is called binary/tree recursion

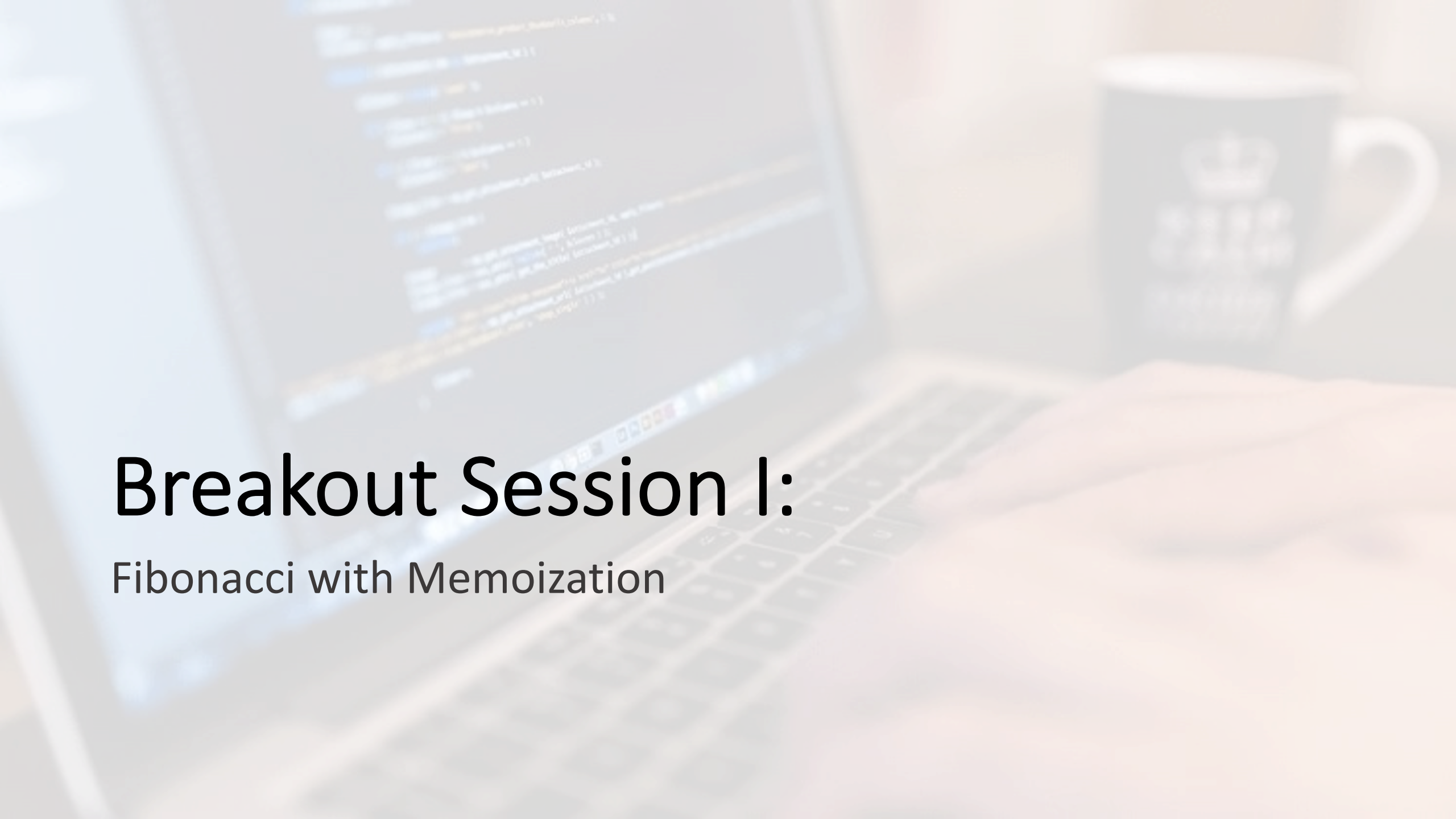
```
def fib(n):
    if n == 0:
        return 0
    if n == 1:
        return 1

    return fib(n - 1) + fib(n - 2)
```

Efficiency of Recursion

- The execution takes long because it computes the same values over and over
- Computation of fib(5) calls fib(2) three times!
- Keeping previously calculated values, such as fib(2), to avoid computing the values more than once improves algorithm performance
- This technique is called Memoization

To understand recursion, you must first understand recursion!





Breakout Session I:

Fibonacci with Memoization

Exercise: Fibonacci with Memoization

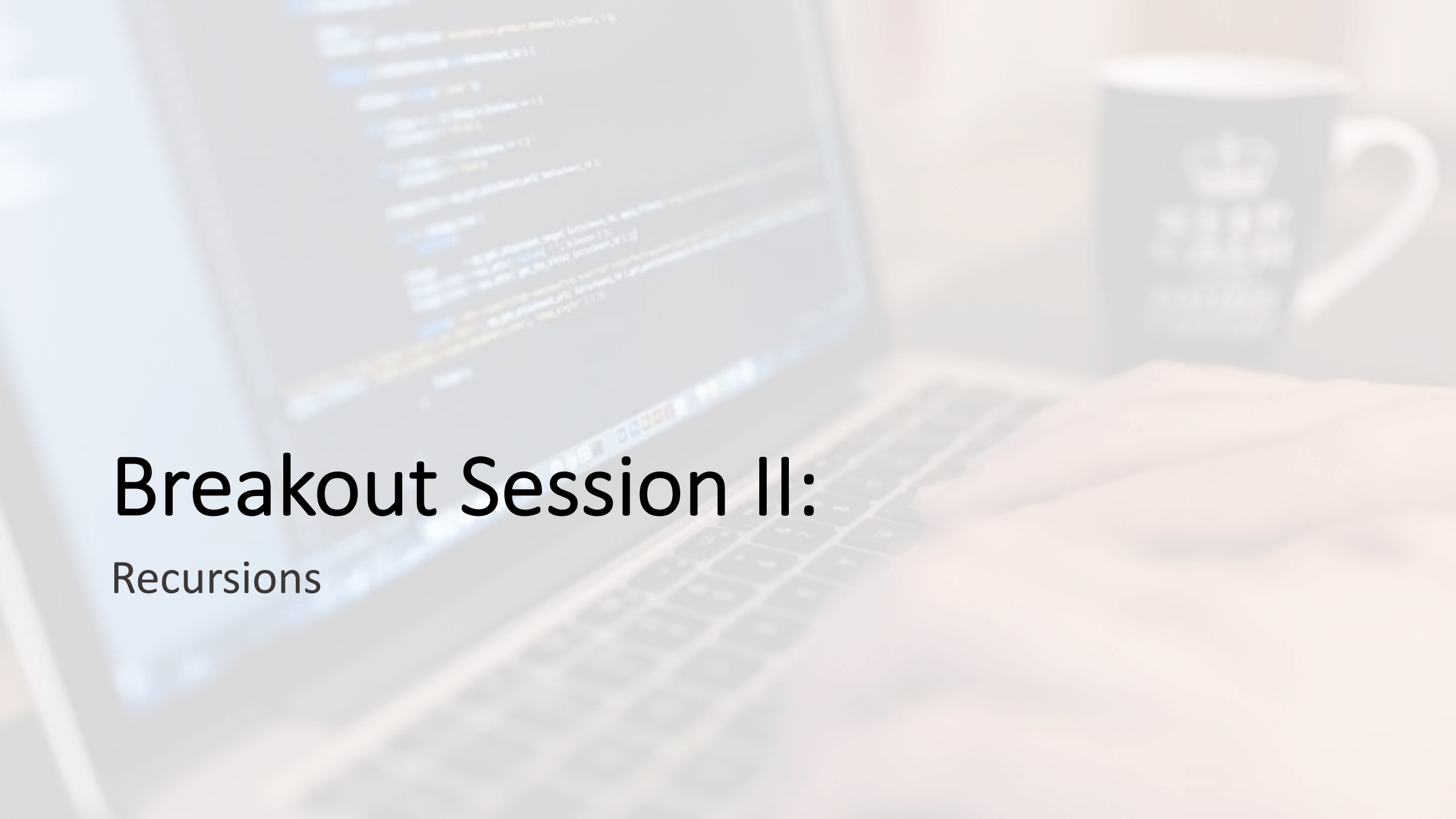
- Modify the recursive function for fib(n) so that it uses Memoization

- Compare the execution time by using

```
import time
start = time.time()
print(fib(30))
end = time.time()
print(end - start)
```

Common Errors

- Infinite recursion:
  - A function calling itself over and over with no end in sight
  - The computer needs some amount of memory for book keeping (stack call) during each call
  - After some number of calls, all available memory for this purpose is exhausted
  - Your program shuts down and reports a “stack overflow”
- Causes:
  - The arguments don’t get simpler or because a special terminating case is missing



Breakout Session II:

Recursions

Sum of n

- Write a recursive function that sums n numbers

  sum(n) = 1 + 2 + 3 + ... + (n-1) + n

- For example:

```
print(sum_of_n(5))
15
```

Power of n

- Write a recursive function that calculates x^n
- Mathematically, the power function can be described as:

$$x^n = \begin{cases} 1 & if\ n = 0 \\ x * x^{n-1} & if\ n > 0 \end{cases}$$

- Assume x and n are both positive integers

Reversing a List

- Write a recursive function that reverses a list

- For example:

```
my_list = [1,2,3,4]
print(recursive_reverse(my_list))
[4,3,2,1]
```