Heuristics

Lab Section 4

For students with Programming background

- Do the coin change exercises on last slide
- coins.py
- Send me code/output screenshots on different inputs
- Also send me an explanation of how did you approach/solve the problem

Today's Lab

- We will explore these exercises and concepts:
- modules
- Random library
- Recursion
- Fibonacci
- Factorial
- Coin change problem

Modules

- A ***module*** is a Python file that contains function definitions and other statements
  - Named just like a regular Python file:

    `myModule.py`

- Python provides many useful modules for us
- We can also create our own if we want

Importing Modules

- To use a module, we must first ***import*** it

- Where does Python look for module files?

- In the current directory
- In a list of pre-defined directories
  - These directories are where libraries like **`random`** and **`calendar`** are stored

Importing Modules

- To import modules, use this command:

```
import moduleName
```

- This imports the entire module of that name
  - Every single thing in the file is now available
  - This includes functions, data types, constants, etc.

`import`

- To use the things we've imported this way, we need to append the filename and a period to the front of its name ("**`moduleName.`**")

- To access a function called **`function`**:

```
moduleName.function()
```

Calendar Module Example

```
import calendar
exCal = calendar.TextCalendar()
printCal = exCal.formatmonth(2016, 11)
print(printCal)
```

```
    November 2016
Mo Tu We Th Fr Sa Su
    1  2  3  4  5  6
 7  8  9 10 11 12 13
14 15 16 17 18 19 20
21 22 23 24 25 26 27
28 29 30
```

“Random” Numbers

Random Numbers

- Random numbers are useful for many things
  - Like what?
  - Cryptography
  - Games of chance
  - Procedural generation
    - Minecraft levels, snowflakes in Frozen
- Random numbers generated by computers can only be ***pseudo*** random

Pseudo Randomness

- “Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin.”
  – *John von Neumann*

- Pseudorandom appears to be random, but isn’t
  - Mathematically generated, so it can’t be
  - Called a Random Number Generator (RNG)

Seeding for Randomness

- The RNG isn’t truly random
  - The computer uses a “seed” in an attempt to be as random as possible
- By default, the seed is the system time
  - Changes every time the program is run
- We can set our own seed
  - Use the `random.seed()` function

Seeding for Randomness

- Same seed means same “random” numbers
  - Good for testing, allow identical runs

```
random.seed(7)
random.seed("hello")
```

- 7 always gives .32, .15, .65, .07
- “hello” always gives .35, .66, .54, .13

Seeding with User Input

- Can allow the user to choose the seed
  - Gives user more control over how program runs

```
random.seed(userSeedChoice)
```

- Can also explicitly seed the system time
  - Give the **`seed()`** function **`None`** or nothing

```
random.seed(None)
random.seed()
```

How Seeds Work

- “Resets” the random number generator each time it is seeded

- Should only seed once per program
- Seeding and calling gives the same number

```
>>> random.seed(3)
>>> random.random()     0.2379646270918913 7
>>> random.seed(3)
>>> random.random()     0.2379646270918913 7
```

Generating Random Integers

- **`random.randrange()`**
- Works the same as normal **`range()`**
  - Start, stop, and step

```
>>> random.seed("dog")
>>> random.randrange(2, 21, 4)          14
>>> random.randrange(2, 21, 4)          6
>>> random.randrange(2, 21, 4)          10
>>> random.randrange(2, 21, 4)          10
>>> random.randrange(6)                        5
>>> random.randrange(6)                        4
```

Generating Random Floats

- **`random.random()`**
- Returns a random float from 0.0 up to (but not including) 1.0

```
>>> random.seed(201)
>>> random.random()
   0.06710225875940379
>>> random.random()
   0.3255995543326774
>>> random.random()
   0.0036753697681032316
>>> random.random()
   0.28279809896785435
```

Generating Random Options

- **`random.choice()`**
- Takes in a list, returns one of the options at random

```
>>> dogs = ["Yorkie", "Xolo", "Westie",
"Vizsla"]
>>> random.seed(11.2016)
>>> random.choice(dogs)          'Xolo'
>>> random.choice(dogs)          'Westie'
>>> random.choice(dogs)          'Vizsla'
>>> random.choice(dogs)          'Westie'
```

Shuffling Options randomly

- **`random.shuffle()`**
- **Shuffling a list of objects** means changing the position of the elements of the sequence
- Takes in a list and shuffles it, it does not return anything but change the original list

```
>>> dogs = ["Yorkie", "Xolo", "Westie", "Vizsla"]
>>> random.shuffle()
>>> print(dogs)
["Xolo", "Yorkie", "Vizsla", "Westie"]
```

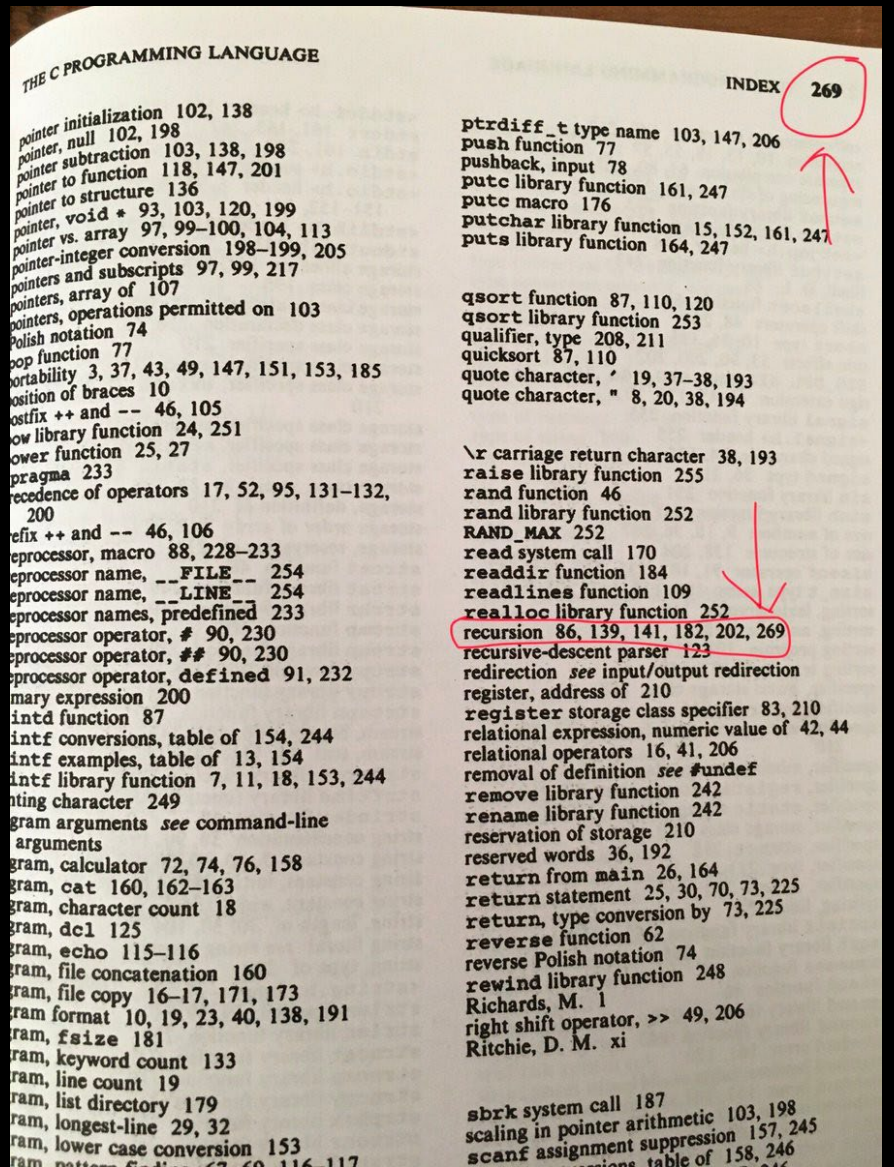
Generating PINs

- Write a program that stores usernames and their PINs in a dictionary
- Ask the user for their username
  - If it exists, tell them their pin code
  - If it doesn't exist, create one using random
    - Tell the user what their new temporary pin is
- Pin should be between 0000 and 9999

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 Text Books



Recursion on google.com

Google

recursion

All Books Images News Videos More Settings Tools

About 16,000,000 results (0.48 seconds)

Did you mean: ***recursion***

Dictionary

Enter a word, e.g. "pie"

re·cur·sion

/rəˈkərZHən/

noun MATHEMATICS · LINGUISTICS

the repeated application of a recursive procedure or definition.

- a recursive definition.
  plural noun: **recursions**

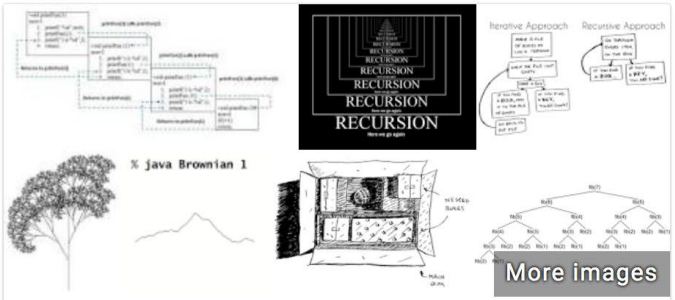
Translations, word origin, and more definitions

Feedback

Recursion - GeeksforGeeks

https://www.geeksforgeeks.org/recursion/

The process in which a function calls itself directly or indirectly is called **recursion** and the corresponding function is called as recursive function. Using recursive algorithm, certain problems can be solved quite easily.



Recursion

Computer science

Recursion in computer science is a method of solving a problem where the solution depends on solutions to smaller instances of the same problem. The approach can be applied to many types of problems, and recursion is one of the central ideas of computer science. Wikipedia

Feedback

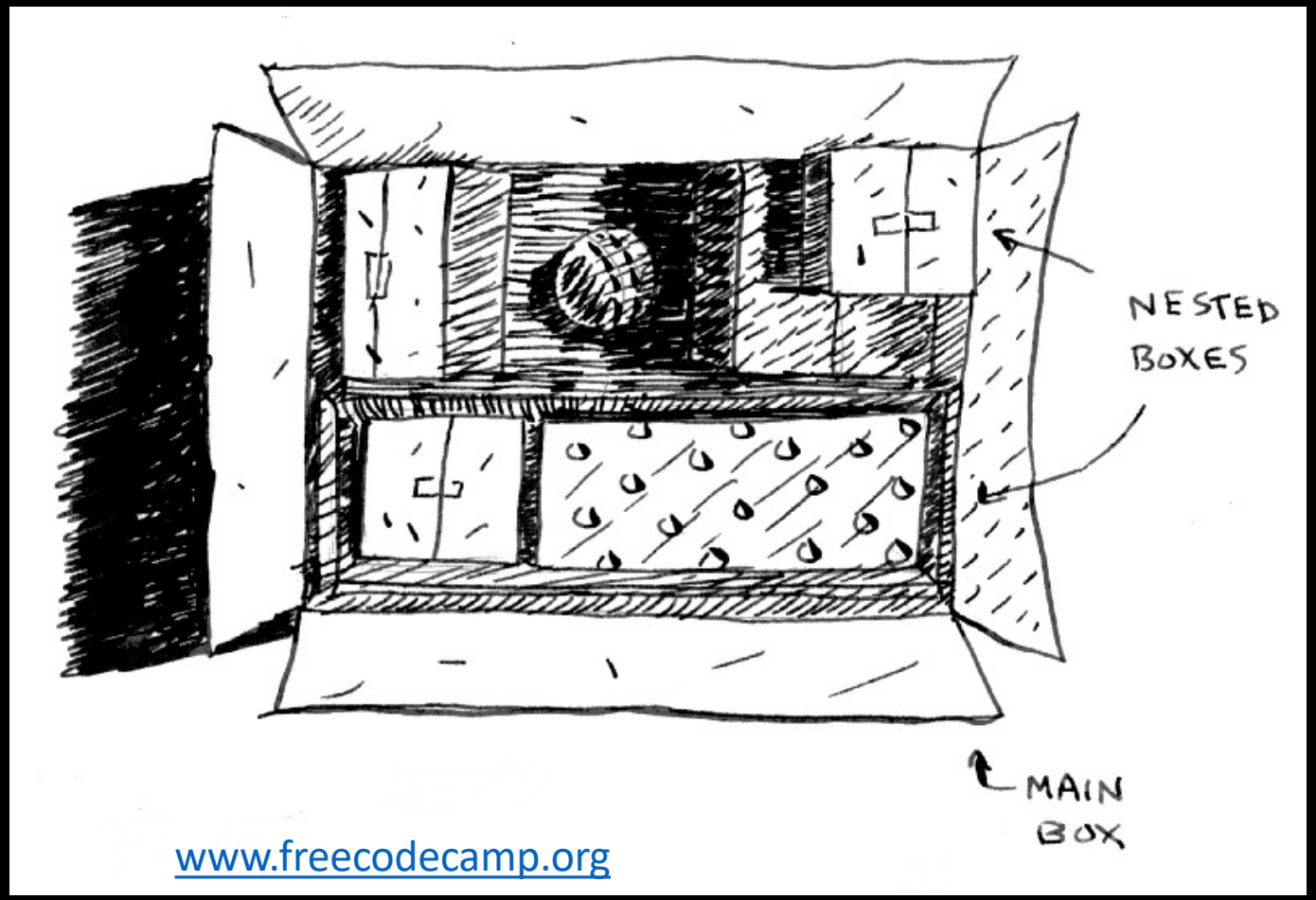
See results about

Recursion

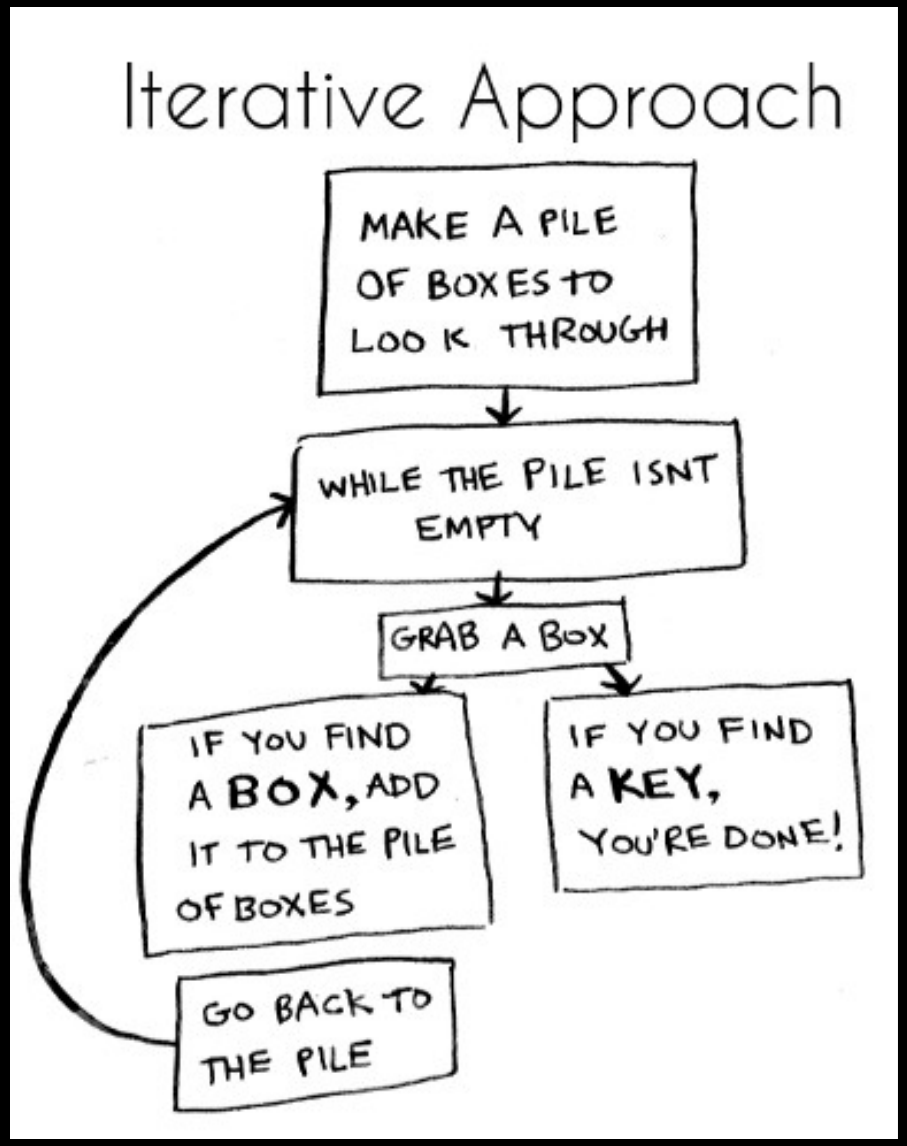
Recursion occurs when a thing is defined in terms of itself or of its type. Recursion is used in a variety of ...

Motivating Example

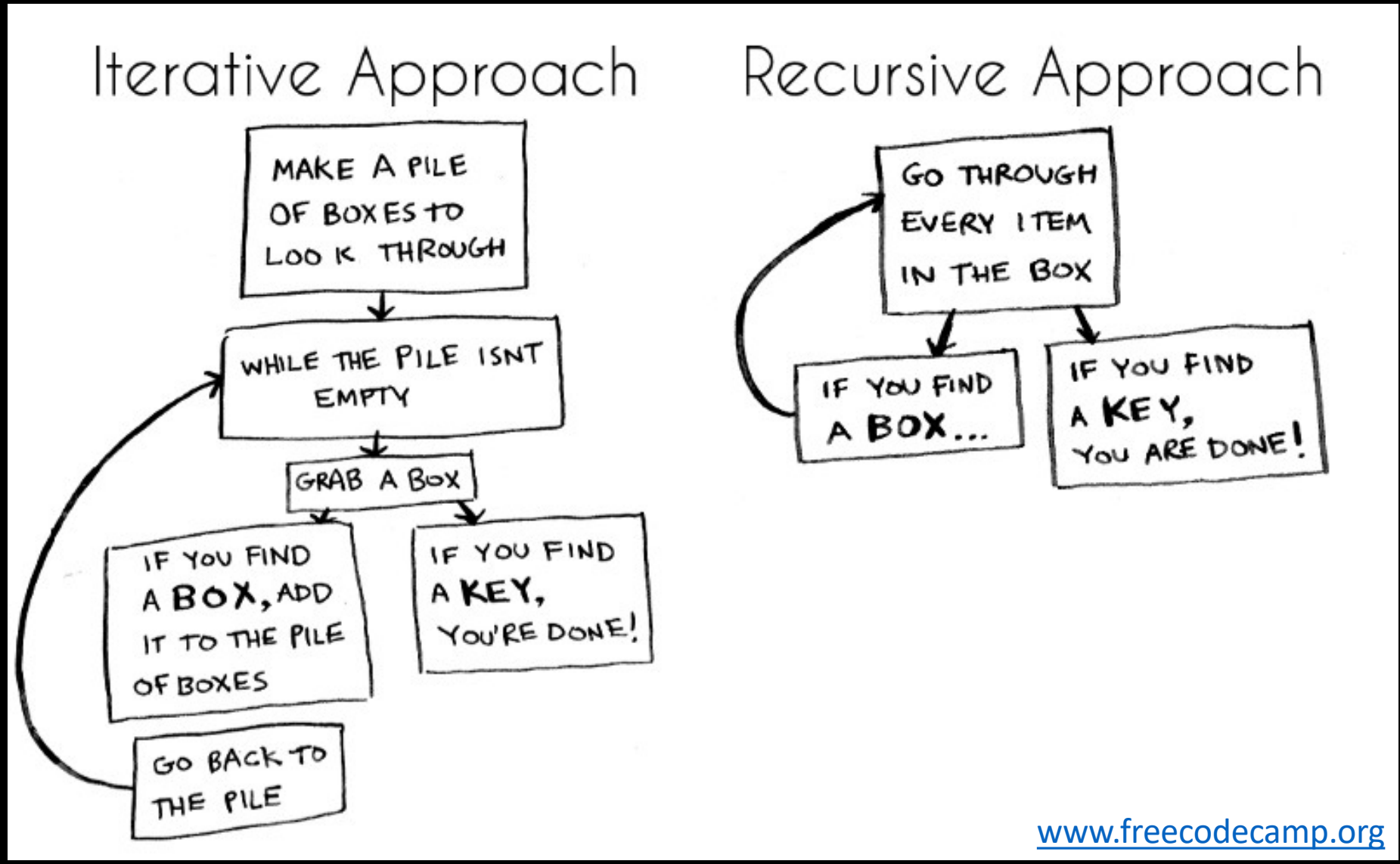
- A key in a box in a box in a box in a box…



Motivating Example



Motivating Example



www.freecodecamp.org

Another Example

- How to find out how many people are in the queue in front of you?
- Assumptions:
  - Your vision is poor. You can not look far
  - You are not allowed to move
  - You are only allowed to speak to the person in front of you or behind you

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 N+1
  - If there is nobody in front of you, you will answer 0

Dry/Run Solution

- Suppose you are 4th in line, and you are determining how many people are in front to you
- Person 4: Tap person 3 shoulder, Ask people in front and wait
  - Person 3: Tap and ask person 2 and wait
    - Person 2: Tap and ask person 1 and wait
      - Person 1 : As person 1 is in front and no one is there to tap their shoulder. Tell person behind it is 0
    - Person 2: Tell person behind it is 0+1 = 1
  - Person 3: Tell person behind it is 1+1 = 2
- Person 4: Just use the answer of person 3 to find the final answer, which is 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 same logic (Recursive Case)
  - Simplification doesn't go on forever, (Base Case)
    - It breaks/stop when encounter a problem version which couldn’t 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 you stop applying the recursive case, the problem is simple enough to be solved directly
- In both cases, we 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 them. Wait for their answer and add 1
    - 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
  - If someone asked, tell them how many people are in front of you (return)

Recursion is Simple

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 not way for the function to stop executing

Recursion with Base Case

Every recursion must have a base case!

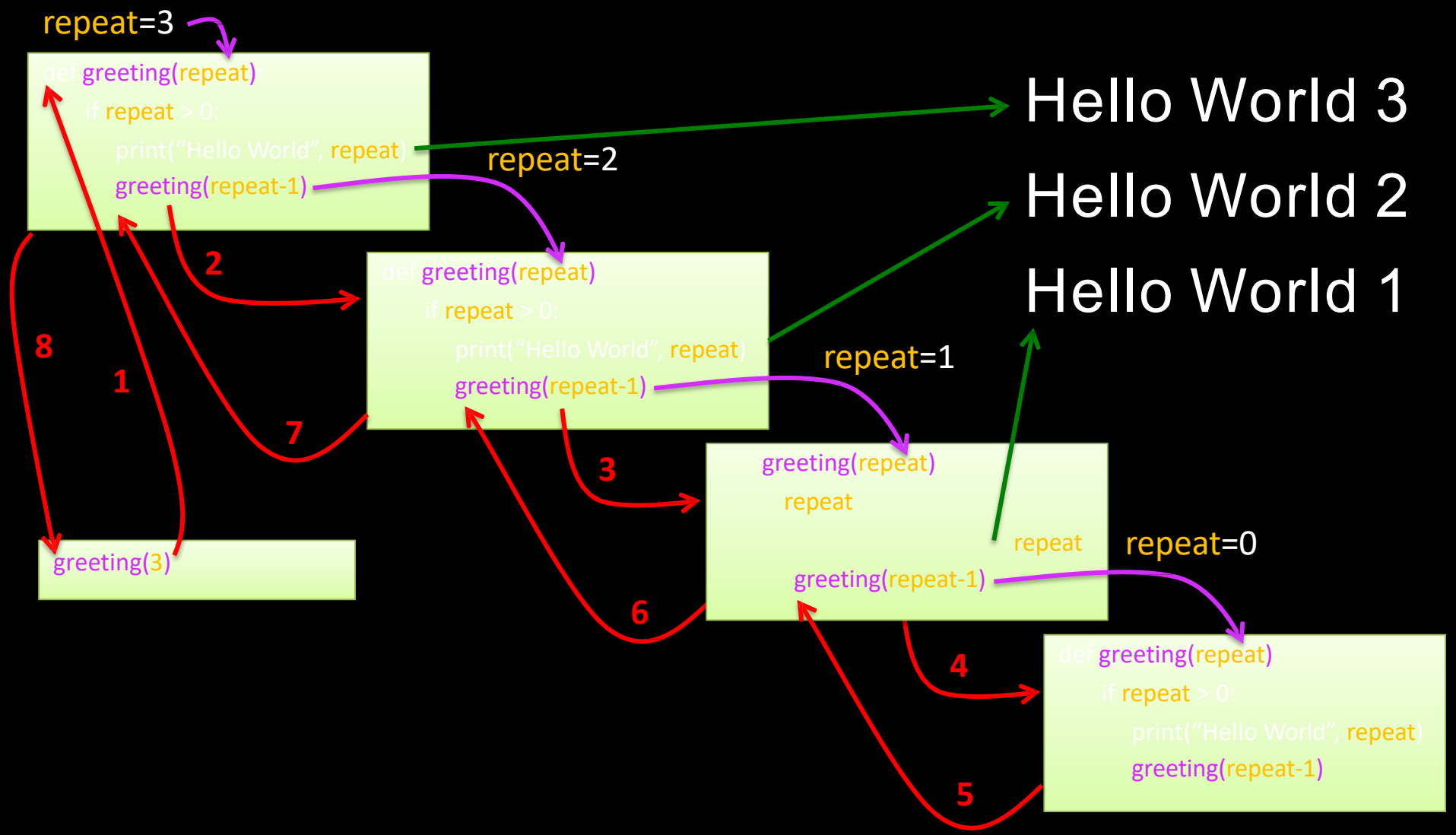
```
def greeting(repeat):
    if repeat > 0:
        print("Hello World", repeat)
        greeting(repeat - 1)

greeting(3)
```

Hello World 3
Hello World 2
Hello World 1

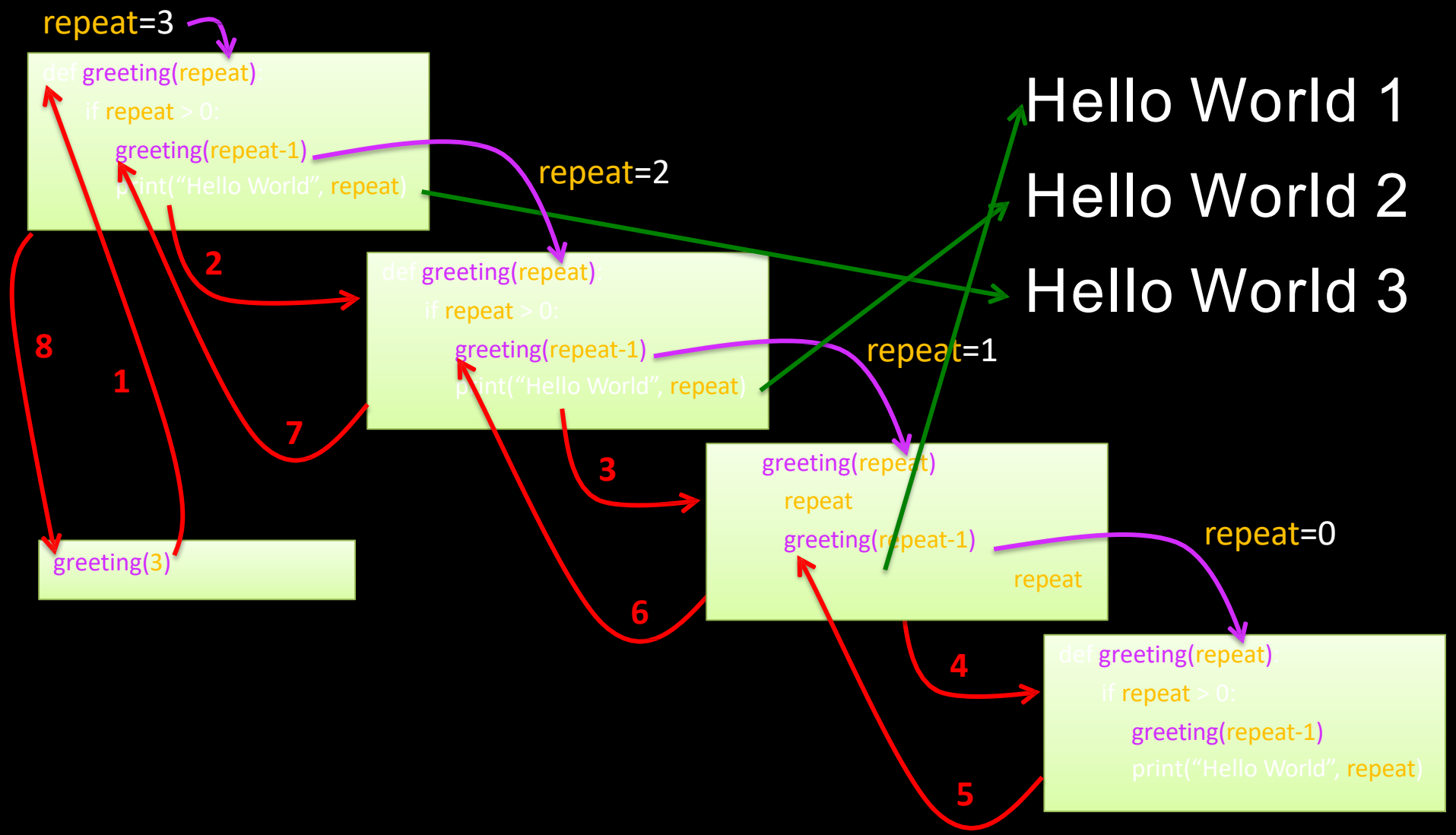
General case (recursive function call)

Base case or Exit condition

Simple Recursion (Illustration)



Simple Recursion (Illustration)



Solving a Simple 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

- 4! = 4 x 3 x 2 x 1

- Iterative approach:

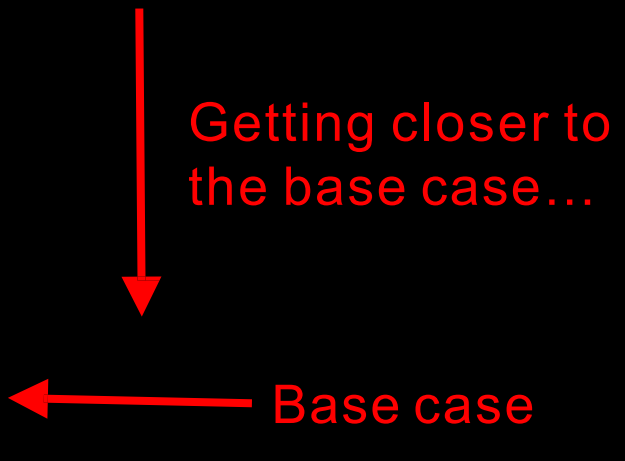
```
n = 4
factorial = 1
for i in range(1, n+1):
    factorial = factorial * i
```

Factorial: The Recursive Approach

- 4! = 4 x 3 x 2 x 1

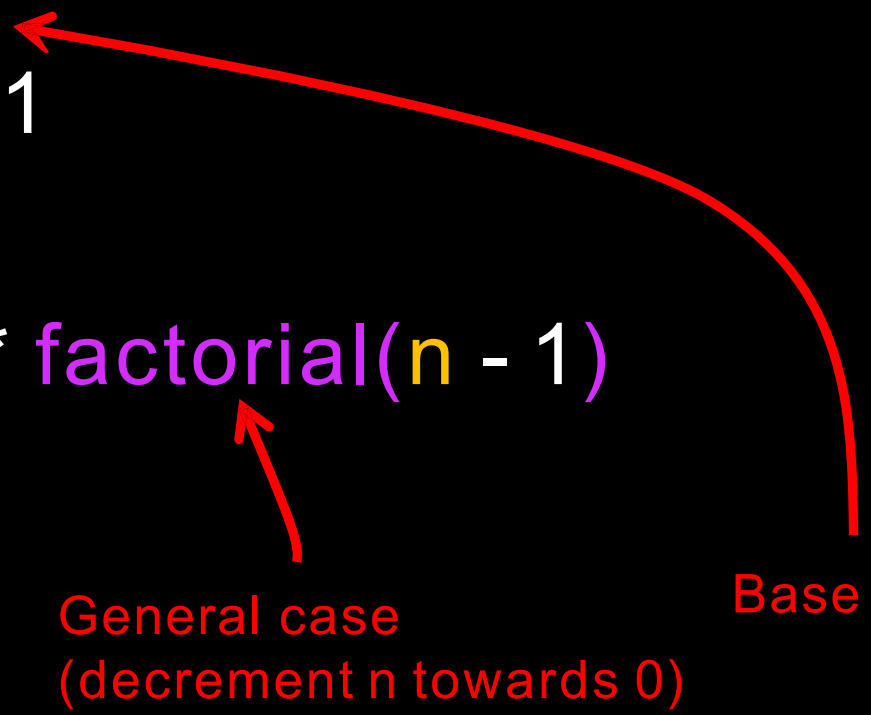
- Factorial is recursive by nature
  - n! = n x (n -1)!
  - 0! = 1

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!

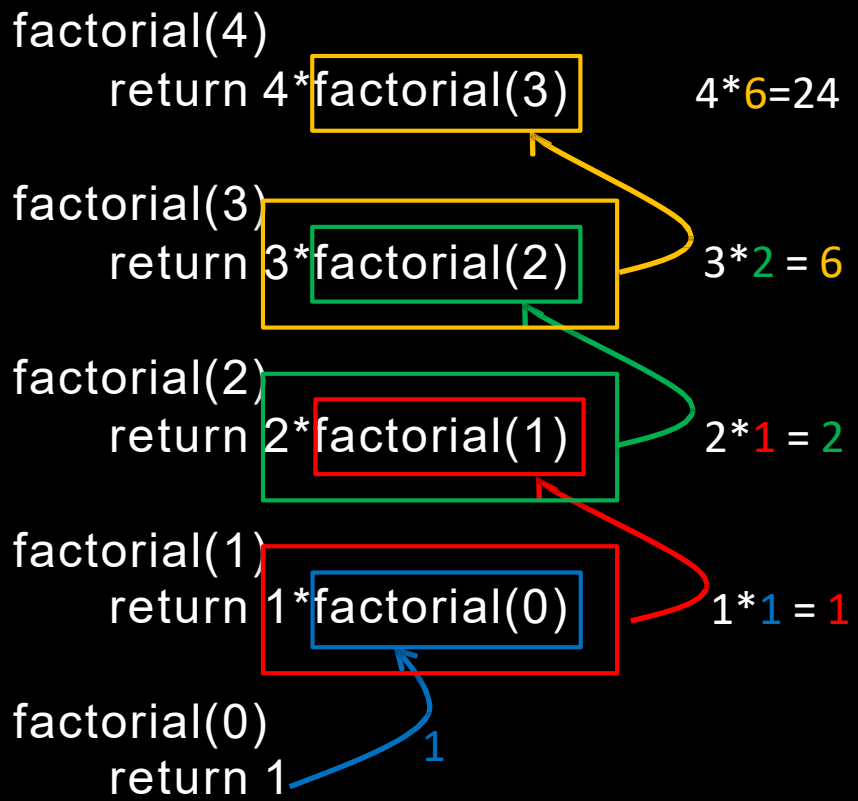


Factorial using Recursion

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

    return n * factorial(n - 1)
```

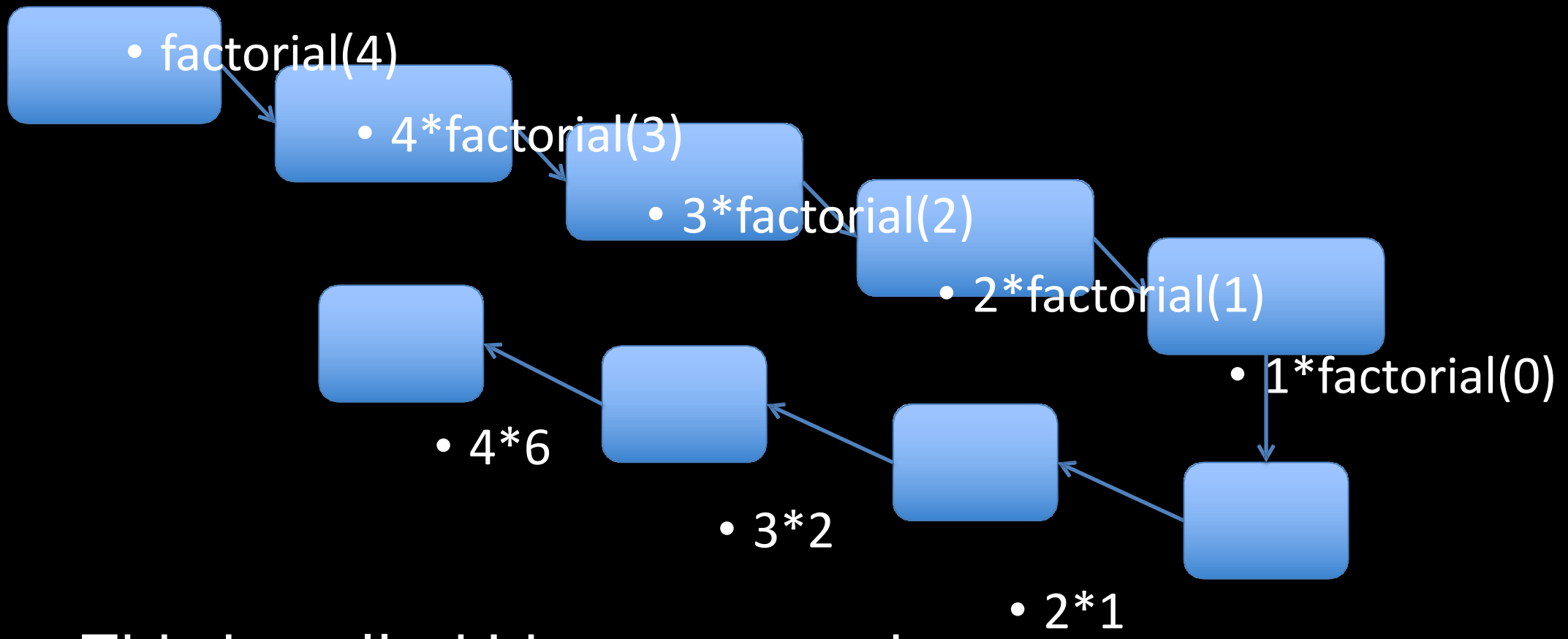


Factorial using Recursion




```
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

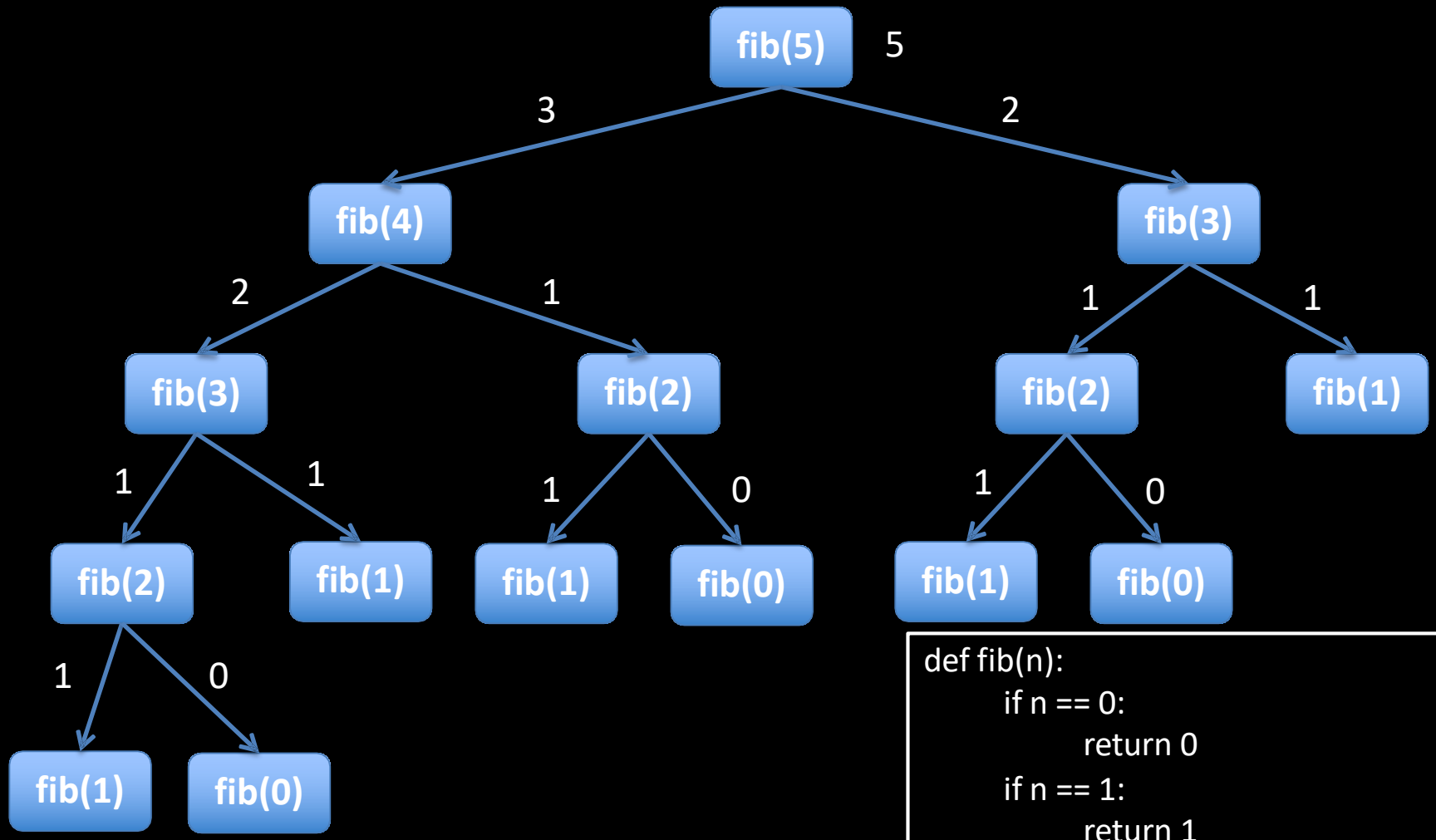
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, …

- 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_21.1): Write a recursive function for fib(n) e.g. fib(6) => 8

Example: Fibonacci Series fib(5)




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

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

This is called Binary recursion

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!

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)
```

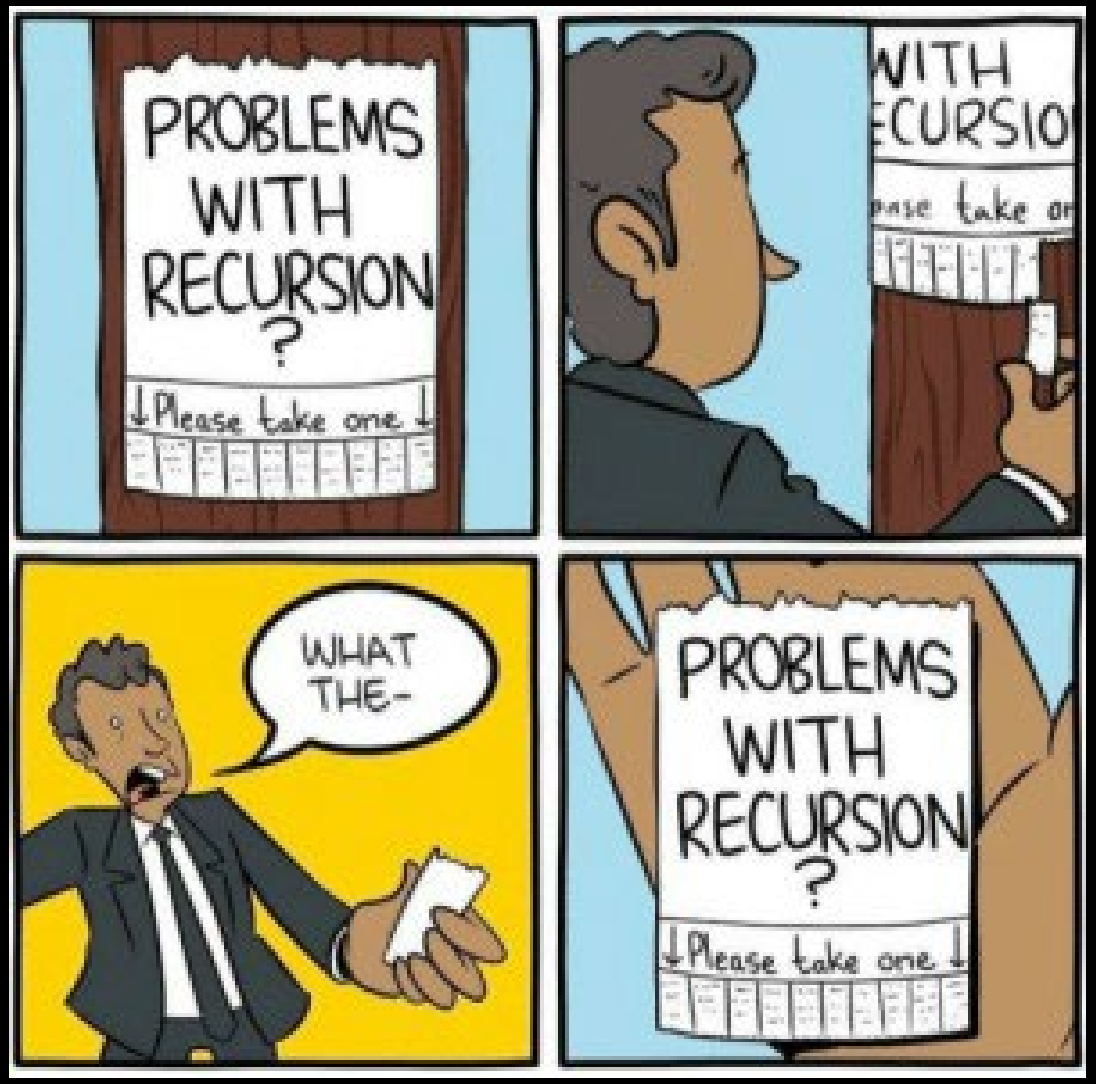
Common Errors

- 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

Reverse List

- Write a recursive function that reverses a list
- For example:

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

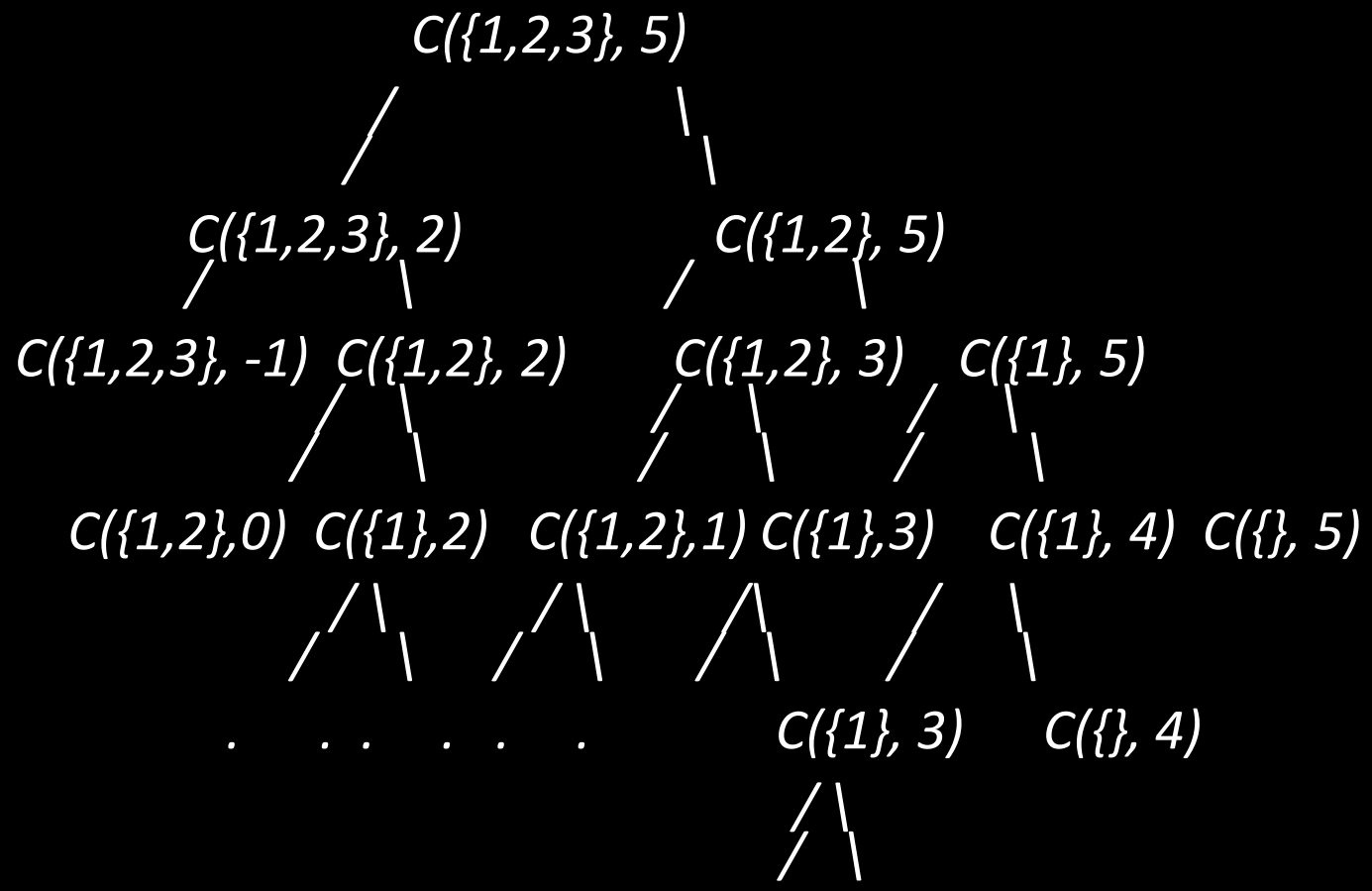

PROBLEMS WITH RECURSION ?
↓Please take one↓
WITH
ECURSIO
ase take o
WHAT THE-
PROBLEMS WITH RECURSION ?
↓Please take one↓

coins.py

- Given an amount and the denominations of coins available, determine how many ways change can be made for the amount. There is a limitless supply of each coin type. Optional: Also, modify your code to print the denomination used.
- Examples:
- Input: coins[] = [8, 3, 2 , 1], V = 15
- Output: There are 35 different ways to make sum using the given denominations.
- Denomination used: All of them.
- Input: coins[] = {9, 1, 2, 3}, V = 4
- Output: There are 4 different ways to make sum using the given denominations.
- ***Explanation:*** *there are four solutions: {1, 1, 1, 1}, {1, 1, 2}, {2, 2}, {1, 3}.*
- Denomination used: 1,2,3

coins.py

- Break the coin change problem
- *C() –> count()*



coins.py

- Follow the below steps to Implement the idea:
- We have 2 choices for a coin of a particular denomination, either i) to include, or ii) to exclude.
- If we are at coins[n-1], we can take as many instances of that coin (unbounded inclusion) i.e **count(coins, n, sum – coins[n-1])**; then we move to coins[n-2].
- After moving to coins[n-2], we can’t move back and can’t make choices for coins[n-1] i.e **count(coins, n-1, sum)**.
- Finally, as we have to find the total number of ways, so we will add these 2 possible choices, i.e **count(coins, n, sum – coins[n-1]) + count(coins, n-1, sum);**

End