3.1

Recursion Fundamentals

Lesson Plan

- [10] Icebreaker
- [15] Recursion Review
- [15] Examples

- [15] Branching

- [20] Practice Problems

- [20] Practice Problems

Icebreaker

I need a brave volunteer…

to guess the number I'm thinking of. I'll tell you with each guess if my number is higher or lower.

Recursion

Guessing a number

To find my number in the range [1, 100]:
Guess the midpoint (50)
    If my number is higher, the new range is [51, 100]
    If my number is lower, the new range is [1, 50]

To find my number in the range [0, 50]:
You're now solving the same problem, but in a smaller range

What is Recursion?

Recursion is recursion. Okay, but what is recursion?

What is Recursion?

Recursion is recursion. Okay, but what is recursion?
    Recursion is recursion. Okay, but what is recursion?

What is Recursion?

Recursion is recursion. Okay, but what is recursion?
    Recursion is recursion. Okay, but what is recursion?
        Recursion is recursion. Okay, but what is recursion?

What is Recursion?

Recursion is recursion. Okay, but what is recursion?
    Recursion is recursion. Okay, but what is recursion?
        Recursion is recursion. Okay, but what is recursion?
            Recursion is recursion. Okay, but what is recursion?

What is Recursion?

Recursion is recursion. Okay, but what is recursion?
    Recursion is recursion. Okay, but what is recursion?
        Recursion is recursion. Okay, but what is recursion?
            Recursion is recursion. Okay, but what is recursion?
                Recursion is recursion. Okay, but what is recursion?

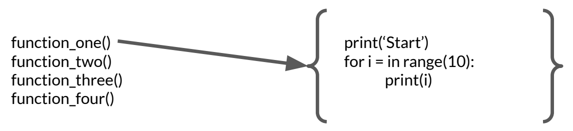
What is Recursion?

Recursion is recursion. Okay, but what is recursion?
    Recursion is recursion. Okay, but what is recursion?
        Recursion is recursion. Okay, but what is recursion?
            Recursion is recursion. Okay, but what is recursion?
                Recursion is recursion. Okay, but what is recursion?

Using a function that calls itself on identical, but smaller, sub-problems to solve the original problem.

Iteration vs. Recursion

The code we have seen so far is ***iterative***. It's a step-by-step, sequential process.



```
function_one()
function_two()
function_three()
function_four()
```

```
print('Start')
for i = in range(10):
        print(i)
```

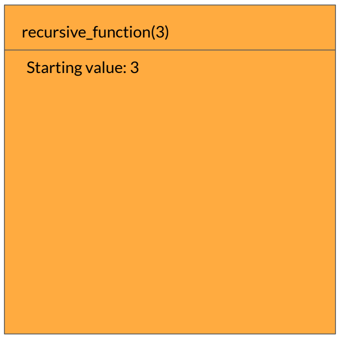
The order of operations is clear from the code, you follow it from top to bottom, and you can trace it step-by-step.

Recursion: a function that calls itself

```
def recursive_function(x) {
      print("Starting value: ", x)
      recursive_function(x)
      print("Done with value: ", x)
}
```

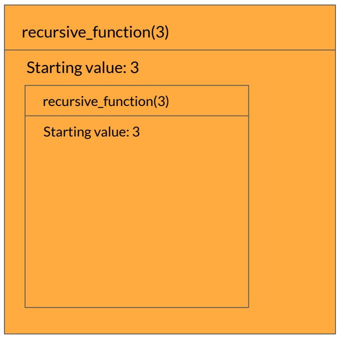
Recursion: a function that calls itself

```
def recursive_function(x) {
      print("Starting value: ", x)
      recursive_function(x)
      print("Done with value: ", x)
}
```



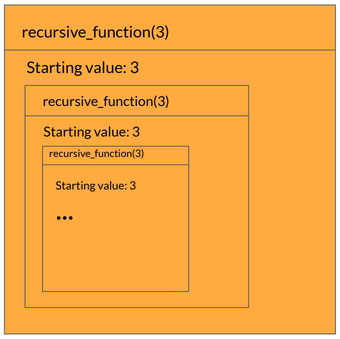
Recursion: a function that calls itself

```
def recursive_function(x) {
      print("Starting value: ", x)
      recursive_function(x)
      print("Done with value: ", x)
}
```



Recursion: a function that calls itself

```
def recursive_function(x) {
      print("Starting value: ", x)
      recursive_function(x)
      print("Done with value: ", x)
}
```

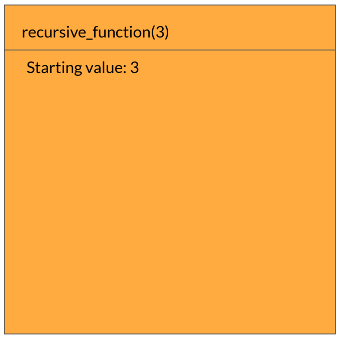


Recursion: sub-problems

```
def recursive_function(x) {
      print("Starting value: ", x)
      recursive_function(x - 1)
      print("Done with value: ", x)
}
```

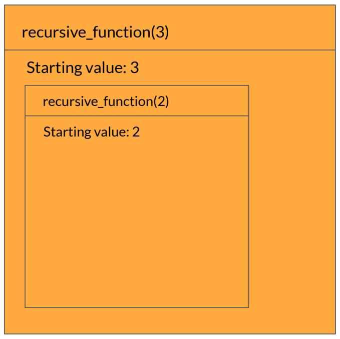
Recursion: sub-problems

```
def recursive_function(x) {
      print(“Starting value: ”, x)
      recursive_function(x - 1)
      print(“Done with value: ”, x)
}
```



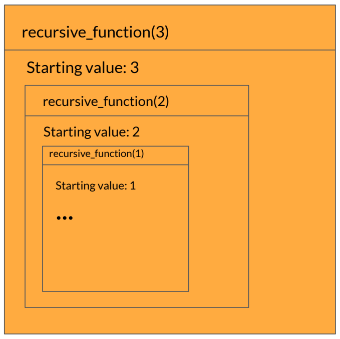
Recursion: sub-problems

```
def recursive_function(x) {
      print(“Starting value: ”, x)
      recursive_function(x - 1)
      print(“Done with value: ”, x)
}
```



Recursion: sub-problems

```
def recursive_function(x) {
      print(“Starting value: ”, x)
      recursive_function(x - 1)
      print(“Done with value: ”, x)
}
```

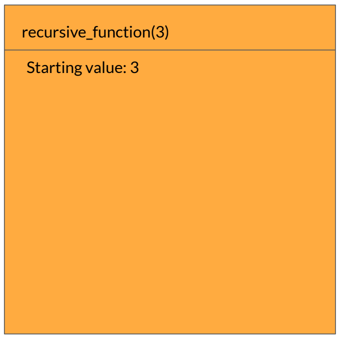


Recursion: identical, but smaller, sub-problems

```
def recursive_function(x) {
      if x < = 1:
            print(“Base case!”)
      else:
            print(“Starting value: ”, x)
            recursive_function(x - 1)
            print(“Done with value: ”, x)
}
```

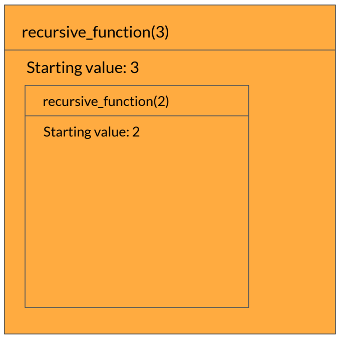
Recursion: identical, but smaller, sub-problems

```
def recursive_function(x) {
      if x < = 1:
            print(“Base case!”)
      else:
            print(“Starting value: ”, x)
            recursive_function(x - 1)
            print(“Done with value: ”, x)
}
```



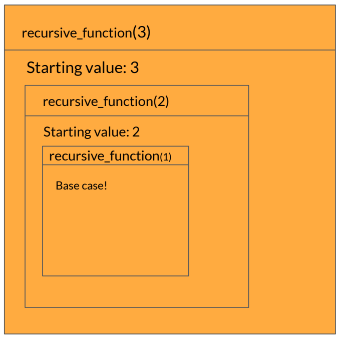
Recursion: identical, but smaller, sub-problems

```
def recursive_function(x) {
      if x < = 1:
            print(“Base case!”)
      else:
            print(“Starting value: ”, x)
            recursive_function(x - 1)
            print(“Done with value: ”, x)
}
```



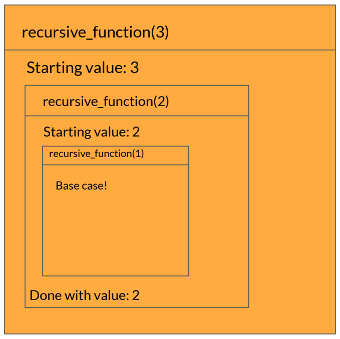
Recursion: identical, but smaller, sub-problems

```
def recursive_function(x) {
      if x < = 1:
            print("Base case!")
      else:
            print("Starting value: ", x)
            recursive_function(x - 1)
            print("Done with value: ", x)
}
```



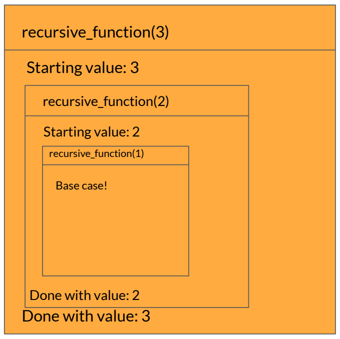
Recursion: identical, but smaller, sub-problems

```
def recursive_function(x) {
      if x < = 1:
            print("Base case!")
      else:
            print("Starting value: ", x)
            recursive_function(x - 1)
            print("Done with value: ", x)
}
```



Recursion: identical, but smaller, sub-problems

```
def recursive_function(x) {
      if x < = 1:
            print("Base case!")
      else:
            print("Starting value: ", x)
            recursive_function(x - 1)
            print("Done with value: ", x)
}
```



Recursion: Two Cases

Base case: The smallest, simplest sub-problem. Can be solved immediately.
Recursive case: The function calls itself on a smaller sub-problem. Needs to approach the base case.

```
def recursive_function(x) {
      if x < = 1:
            print("Base case!")
      else:
            print("Starting value: ", x)
            recursive_function(x - 1)
            print("Done with value: ", x)
}
```

Example: Factorial

n! = n * (n - 1) * (n - 2) * (n - 3) * … * 3 * 2 * 1

Example: Factorial!

n! = n * (n - 1) * (n - 2) * (n - 3) * … * 3 * 2 * 1
5! = 5 * 4 * 3 * 2 * 1

Example: Factorial

n! = n * (n - 1) * (n - 2) * (n - 3) * … * 3 * 2 * 1
5! = 5 * 4 * 3 * 2 * 1
5! = 5 * (4 * 3 * 2 * 1)

Example: Factorial

n! = n * (n - 1) * (n - 2) * (n - 3) * … * 3 * 2 * 1
5! = 5 * 4 * 3 * 2 * 1
5! = 5 * (4 * 3 * 2 * 1)
5! = 5 * (4!)

Example: Factorial

n! = n * (n - 1) * (n - 2) * (n - 3) * … * 3 * 2 * 1
5! = 5 * 4 * 3 * 2 * 1
5! = 5 * (4 * 3 * 2 * 1)
5! = 5 * (4!)

Example: Factorial

n! = n * (n - 1) * (n - 2) * (n - 3) * … * 3 * 2 * 1
5! = 5 * 4 * 3 * 2 * 1
5! = 5 * (4 * 3 * 2 * 1)
5! = 5 * (4!)
1! = 1

Let's code it!

How does it work? - Call Stack Overview

- When a function is called in Python, Python allocates an area of memory that is set aside to keep track of a functions call in progress. This is called a **Call Stack Frame** or just **Stack Frame**
- A Stack Frame is born when a function is called, and dies when the function returns!
- Stack Frames are commonly composed of:
    - Local variables
    - Arguments passed into the method
    - Information about the caller's stack frame
    - The return address
        - What the program should do after the function returns (i.e.: where it should "return to").

Let's try another example!

Can we find the maximum element of a list recursively?

(yes, we can)

Let's code it!

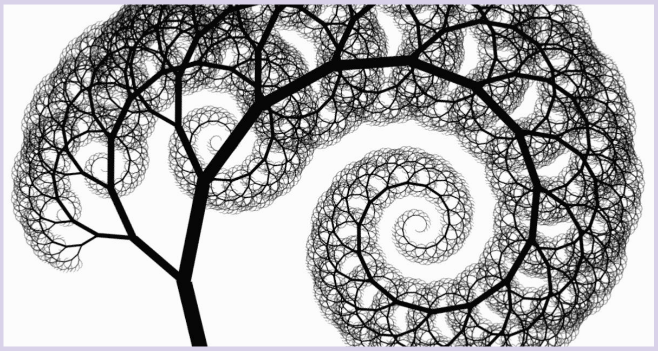
Practice Problems: Recursion

You will be working in teams of 3 or 4. Use the table below to figure out what your role is. [repl.it]

Role	Responsibilities	Assignment Criteria
Captain	Share screen, write code, keep track of time, ensure all team members participate	Person who has been assigned this role the least number of times
Tester	Plays devil's advocate, design test cases, determine algorithm complexity (time and space)	Person who has been assigned this role the least number of times
Presenter	Explain solution to the class, present the team's algorithm design decisions, state solution's complexity (time and space), share one thing the team learned from the problem	Person who has been assigned this role the least number of times

If there are ties, get creative and come up with a way to break them (i.e., sort yourselves by last name, distance to Google Austin, etc.)
If there are 4 members in your team, you should have two Testers

Recursion: Branching



Example: Fibonacci

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

Each number in the sequence is the sum of the two previous numbers.

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

Example: Fibonacci

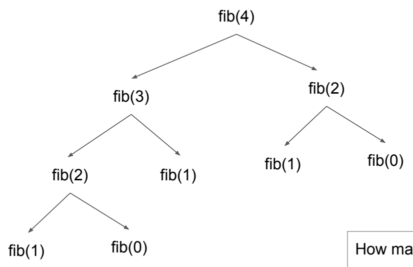
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, …

The first two elements are 0 and 1!

```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    return fib(n - 1) + fib(n - 2)
```

Tracing + Time and Space Complexity

Fibonacci



How many function calls have we made?	2^n
What is the maximum depth of calls?	n

Practice Problems: Recursion Branching

You will be working in teams of 3 or 4. Use the table below to figure out what your role is [repl.it]

Role	Responsibilities	Assignment Criteria
Captain	Share screen, write code, keep track of time, ensure all team members participate	Person who has been assigned this role the least number of times
Tester	Plays devil's advocate, design test cases, determine algorithm complexity (time and space)	Person who has been assigned this role the least number of times
Presenter	Explain solution to the class, present the team's algorithm design decisions, state solution's complexity (time and space), share one thing the team learned from the problem	Person who has been assigned this role the least number of times

If there are ties, get creative and come up with a way to break them (i.e., sort yourselves by last name, distance to Google Austin, etc.)
If there are 4 members in your team, you should have two Testers

Helper Functions

Making Change

```
def change(x):
    if x < 0:
        return False
    if x == 0:
        return True
    return change(x - 7) or change(x - 11) or change(x - 19)
```

What if we're only allowed to use 5 coins total?

First define a helper function

```
def change(x):
    def change_helper(x):
        if x < 0:
            return False
        if x == 0:
            return True
        return change(x - 7) or change(x - 11) or change(x - 19)

    return change_helper(x)
```

Introduce a new parameter

```
def change(x):
    def change_helper(x, n):
        if x < 0:
            return False
        if x == 0:
            return True
        return change(x - 7) or change(x - 11) or change(x - 19)

    return change_helper(x, 0)
```

It's important not to change function headers when provided!

Introduce a new parameter

```
def change(x):
    def change_helper(x, n):
        if x < 0 or n > 5:
            return False
        if x == 0:
            return True
        return change(x - 7) or change(x - 11) or change(x - 19)

    return change_helper(x, 0)
```

It’s important not to change function headers when provided!

Pass the new parameter through recursive calls

```
def change(x):
    def change_helper(x, n):
        if x < 0 or n > 5:
            return False
        if x == 0:
            return True
        return change(x - 7, n + 1) or change(x - 11, n + 1) or change(x - 19, n + 1)

    return change_helper(x, 0)
```

It’s important not to change function headers when provided!

Helper Functions - Another Example