Data Structures & Algorithms

Topic 05: Recursion

version 1.0

(Content adapted from:

- *Data Structures & Algorithms using Python. Rance D. Necaise, Wiley, 1st Edition, 2011.*
- *Data Structures & Algorithms in Python. Michael T. Goodrich, Roberto Tamassia & Michael H. Goldwasser, Wiley, 1st Edition, 2013.)*

Recursion

Image source: https://www.therussianstore.com/blog/the-history-of-nesting-dolls

Learning Outcomes :Recursion

- Recursion
  - Recursive Functions
  - Classic example: Factorial function
  - Properties of Recursion
- Recursion Call Tree
- Run Time Stack

Recursion

- **Recursion** is a process of solving problems by subdividing a larger problem into smaller versions of the itself and then solving the smaller, more trivial parts.

- Recursion is a **powerful programming and problem-solving tool**, BUT **not always the most efficient**.

- However, in some instances, recursion is the implementation of choice as it allows us to **easily develop a solution for a complicated problem** that may otherwise be difficult to solve.

Video on Recursion - Factorial

In the video, duration: 12 Mins

The instructor will:
- Explain how to compute Factorial in Maths
- Show how to implement Factorial using Recursion
- Define the Base Case and Recursive Case
- Code the example
- Explain the steps very clearly to you

- Reference: https://youtu.be/B0NtAFf4bvU

Recursive Functions

- A function that calls itself is known as a **recursive function**.

- Classic example – the **Factorial** function:
  - n! = 1· 2· 3· ··· · (n-1)· n

Recursive definition

$$n! = \begin{cases} 1, & \text{if } n = 0 \\ n * (n-1)!, & \text{if } n > 0 \end{cases}$$

Python implementation

```
# Factorial function
# Assuming n >= 0
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)
```

Properties of Recursion

- All recursive solutions must satisfy three rules or properties:
  - A recursive solution must contain a **base case**
  - A recursive solution must contain a **recursive case**.
  - A recursive solution must **make progress towards the base case**.

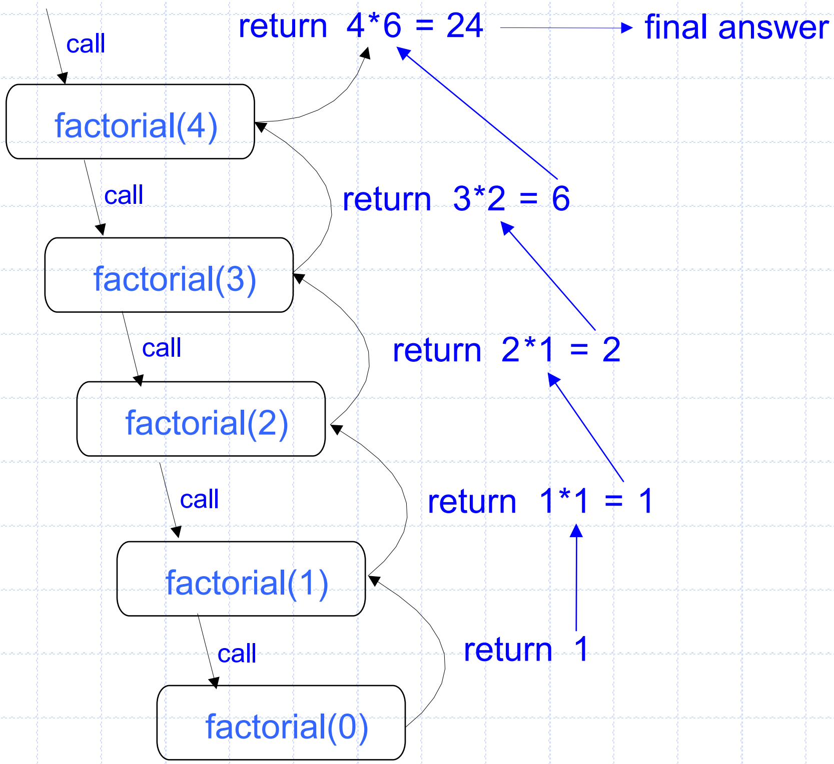
Properties of Recursion

- **Base case:**
  - Terminating case and represents the smallest subdivision of the problem.
  - Signals the end of the recursive calls.
  - In `factorial()`, the base case occurred when `n = 0`.
- **Recursive case:**
  - The case when the recursive function calls itself.
  - In `factorial()`, the recursive case occurred when `n > 0`.

- How does `factorial()` ensure that it makes progress towards the base case? And what will happen when it does not?

Visualizing Recursion

- Recursion call tree
  - A box for each recursive call
  - An arrow from each caller to callee
  - An arrow from each callee to caller showing return value
- Example



Recursion Call Tree

```
# A sample program
# containing three functions.
def main():
    y = foo( 3 )
    bar( 2 )

def foo( x ):
    if x % 2 != 0:
        return 0
    else:
        return x + foo( x-1 )

def bar( n ):
    if n > 0:
        print( n )
        bar( n-1 )

main()
```

Draw the recursion call tree.

What is the output of the program?

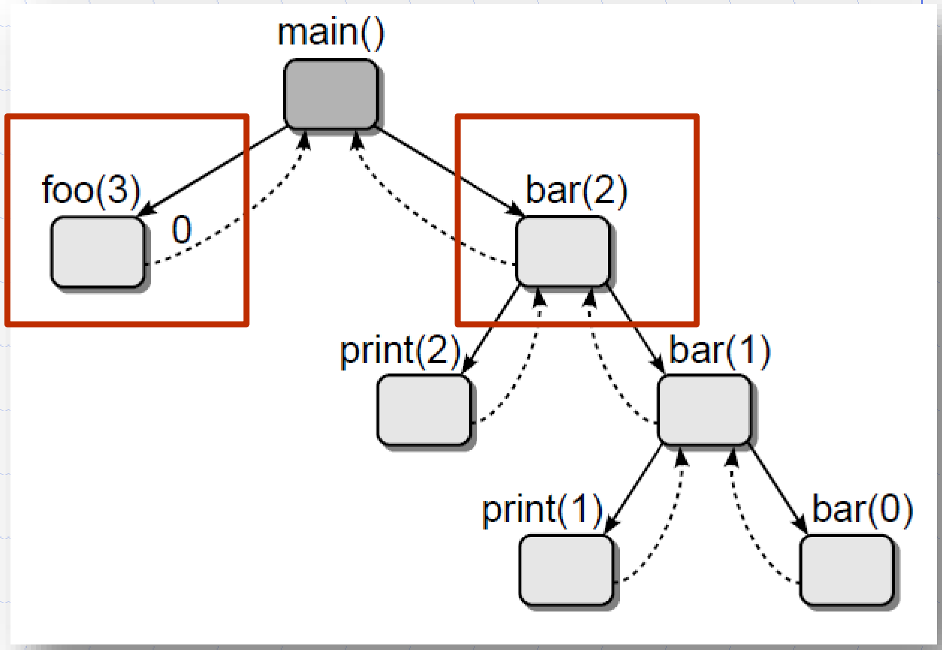
Recursion Call Tree

```python
# A sample program
# containing three functions.
def main():
    y = foo( 3 )
    bar( 2 )

def foo( x ):
    if x % 2 != 0:
        return 0
    else:
        return x + foo( x-1 )

def bar( n ):
    if n > 0:
        print( n )
        bar( n-1 )

main()
```



NOTE: The edges are listed left to right in the order the calls are made.

Behind the scenes …

- Each time a function is called, an **activation record** is created to maintain information related to the function:
  - **Return address** – location of the next instruction to be executed when the function terminates
  - **Local variables**
- When the function terminates, the activation record is destroyed.

Behind the scenes …

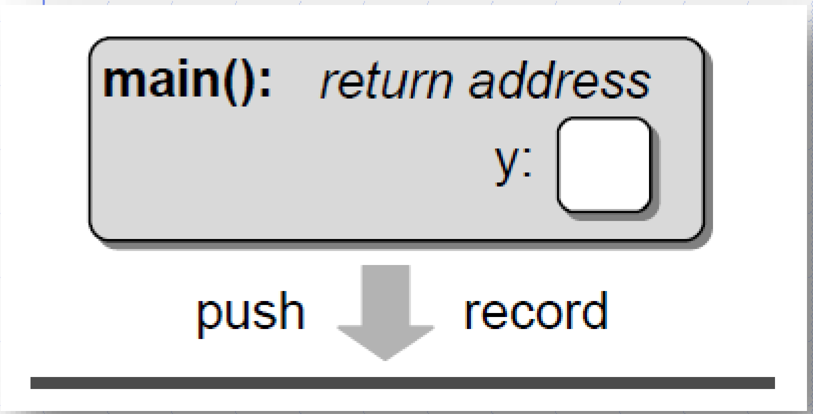
- System must:
  - Manage the collection of activation records.
  - Remember the order in which they were created. **WHY?**

Allow the system to backtrack and return to the next statement in the previous function when an invoked function terminates.

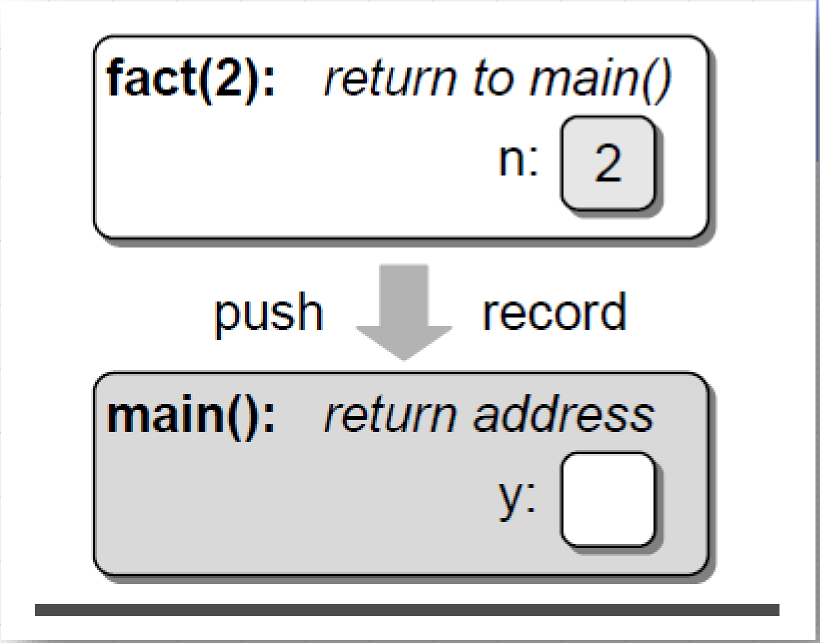
Using a Run Time Stack ← Remember Stacks? LIFO? A data structure to be discussed in a later topic.

Run Time Stack

```
# Consider executing this code:
def main():
    y = fact( 2 )
```

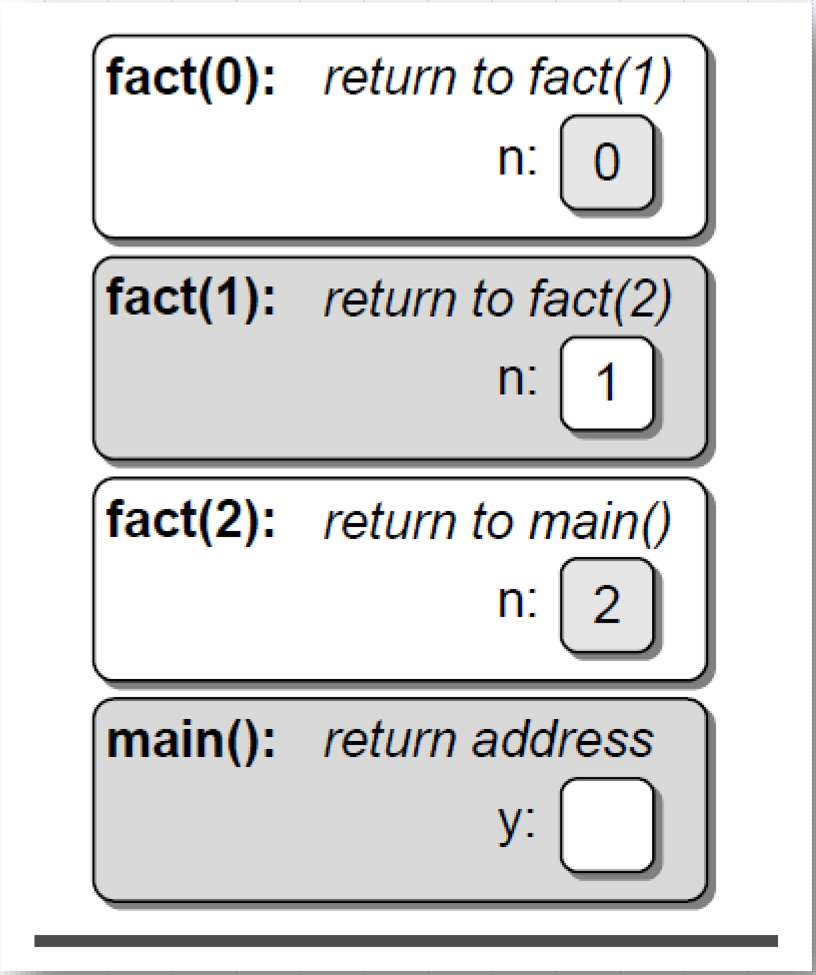


When the `main()` routine is executed, the first activation record is created and pushed onto the stack.



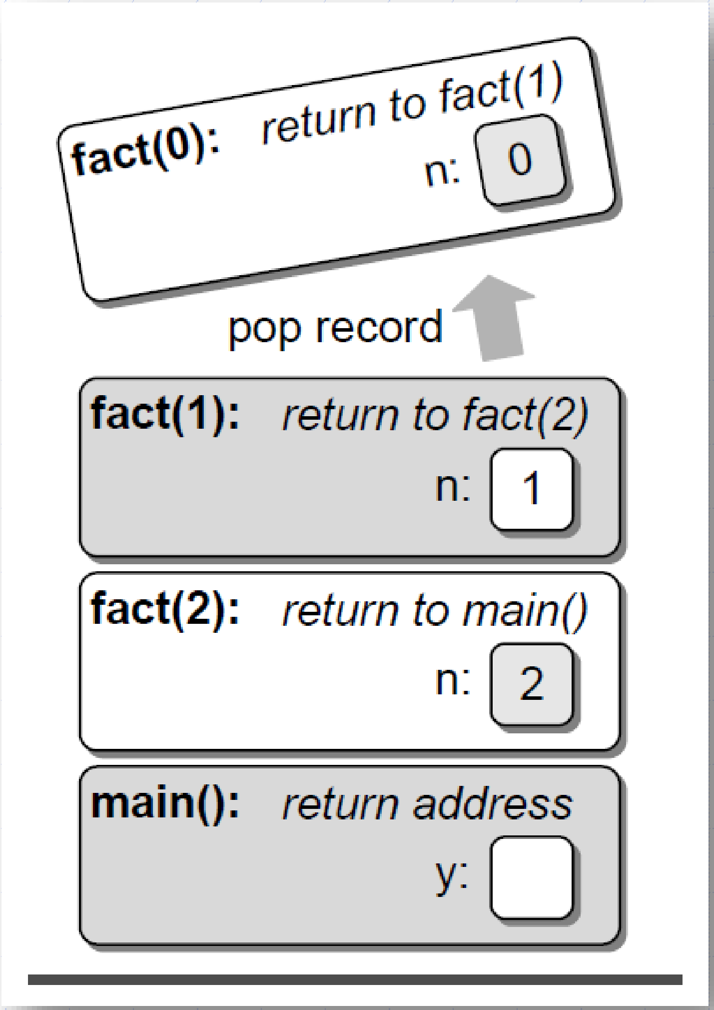
When the factorial function, `fact()`, is called, the second activation record is created and pushed onto the stack.

Run Time Stack



The factorial function, `fact()`, is recursively called until the base case is reached with a value of `n = 0`.

Run Time Stack



Potential problem:

Recursive call with huge n, number of iterations, it may result in Stack overflowed

Summary :Recursion

- Definition of Recursion Function and its properties (Base and Recursive cases)
- Draw a Recursion Call Tree
- Understand the Run Time Stack
- Applications of Recursion