Computer Programming I

Recursion

What is Recursion?

- A function that calls itself, either directly or indirectly, is called a recursive function.
- This technique is very similar to looping.
  - The code inside the function is performed over and over again (each time the function is called), typically on an easier and easier sub-problem.
- This technique plays an important role in programming
  - It is particularly useful when a problem can be broken down into strictly smaller problems of the same nature as the original

Recursive Design

- Two things are required for recursion to work:
  1. A termination condition that allows the function to exit **without** calling itself → Base case.
  2. Some assurance that we will approach this condition → recursive call on a smaller problem.
- We use some form of conditional branching to decide which gets executed: base case or recursive call
- A recursive function may have:
  - More that one base case
  - More than one recursive call

Recursive Design

- We might consider a recursive solution to a problem if we can come up with a solution that:
  a. Has some "basic" solutions for some inputs
  b. For other inputs we can describe the solution to the problem in terms of the solution of an easier/smaller problem of the same type.

Example: Sum of Digits

- How can we sum the digits of a number:

```
sumOfDigits(1234)
```

- Can we describe a recursive solution?
  - To sum the digits in a number you would sum all the digits except the last one, and then add the last one in.
  - The "sum of all digits except one" suggests recursion since you are performing the same process on a smaller part of the problem

```
sumOfDigits(1234)  = sumOfDigits(123) + 4
                   = sumOfDigits(12) + 3 + 4
                   = sumOfDigits(1) + 2 + 3 + 4
                   = 1 + 2 + 3 + 4
```

Example: Sum of Digits

- What is the base case and what should we return then?
  - When a number is < 10 (single digit) just return the number
- What is the recursive relationship?
  - Add the last digit to the result of summing the remaining digits.

Example: Sum of Digits

```
def sumOfDigits(number) :
    if  number < 0 :   #take care of negative numbers
        return ( sumOfDigits( -number ) )
    if number < 10 :  #base case: a single digit number
        return number
    else:
        #recursive call: Add the last digit to the result
        #of summing the remaining digits
        return (number % 10 + sumOfDigits(number // 10))
```

Why Use Recursion?

- Since we said recursion is similar to looping the question is: why use recursion?
  - In fact any recursive algorithm can be expressed without using recursion and instead using a loop.
- Worse yet, function calls can get expensive
  - We need to save variable states etc.

Why Use Recursion?

- So why use recursion at all?
  - A recursive solution may be less labor intensive than another solution
  - Some problems have naturally recursive solutions
    - Many math problems are recursive by nature
  - Some recursive algorithms may be simpler to code or to understand than a different approach

Example: Fibonacci Numbers

- Sometimes we encounter problems that have a recursive definition already built into them.
  - Therefore it may be tough to come up with a non-recursive solution
- A classic example is the computation of a Fibonacci number which is mathematically defined as:
  - `f(n) = 0` `if n = 1`
  - `f(n) = 1` `if n = 2`
  - `f(n) = f(n-1) + f (n-2)` `if n > 2`
- This definition results in two recursive calls every time.

Example: Fibonacci Numbers

```
def fibonacci(number) :
    if number == 1 :        #base case
        return 0
    elif number == 2  :     #base case
        return 1
    else: #recursive calls: when n > 2
        return (fibonacci(number - 1) + fibonacci(number - 2))
```

Example: factorial

The factorial *n!* of a non-negative integer *n* is defined as:

$$n! = \begin{cases} 1, & n = 0 \\ 1 * 2 * 3 * \ldots * n, & n > 0 \end{cases}$$

Thus:

0! = 1

1! = 1

2! = 1 * 2 = 2

3! = 1 * 2 * 3 = 6

4! = 1 * 2 * 3 * 4 = 24

5! = 1 * 2 * 3 * 4 * 5 = 120

Example: factorial

We can express recursively the definition of $n!$ as follows:

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

Thus:

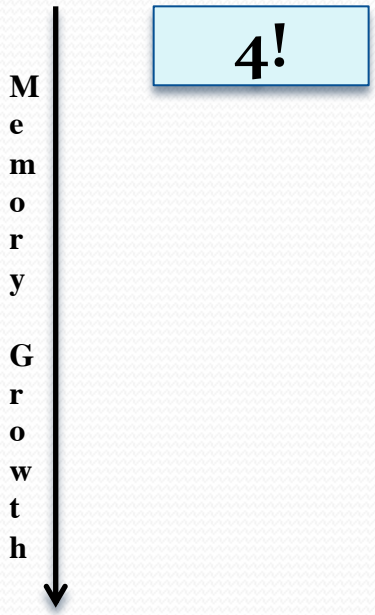
$5! = 5 * 4! = 120$

Example: factorial

```
def factorial (number) :
    if (number == 0):  # Base case
        return 1
    else :  # Recursive call
        return (number * factorial(number - 1) )
```

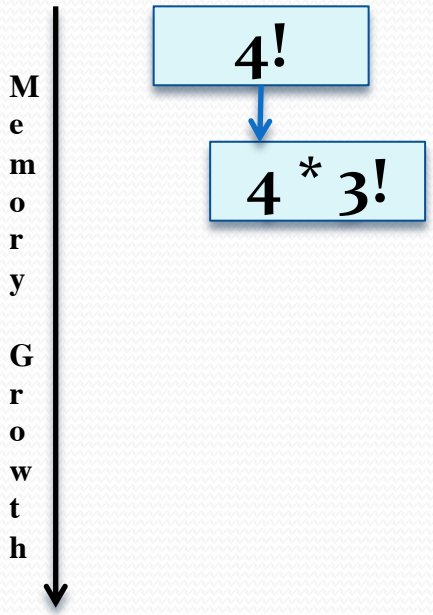
Recursive function Invocations

- What exactly happens when the recursive factorial function executes?
- Let's consider the case when the program computes 4!



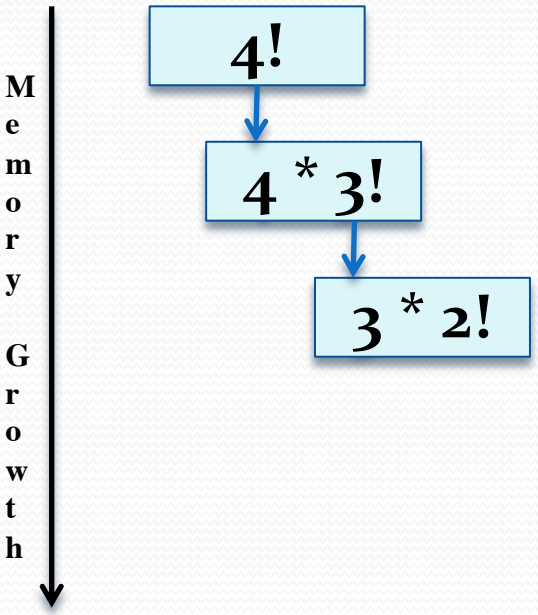
Recursive function Invocations

- What exactly happens when the recursive factorial function executes?
- Let's consider the case when the program computes 4!



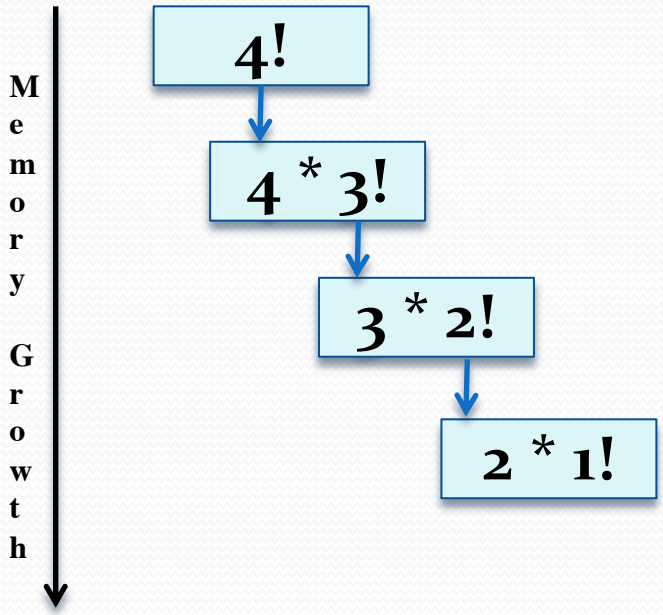
Recursive function Invocations

- What exactly happens when the recursive factorial function executes?
- Let's consider the case when the program computes 4!



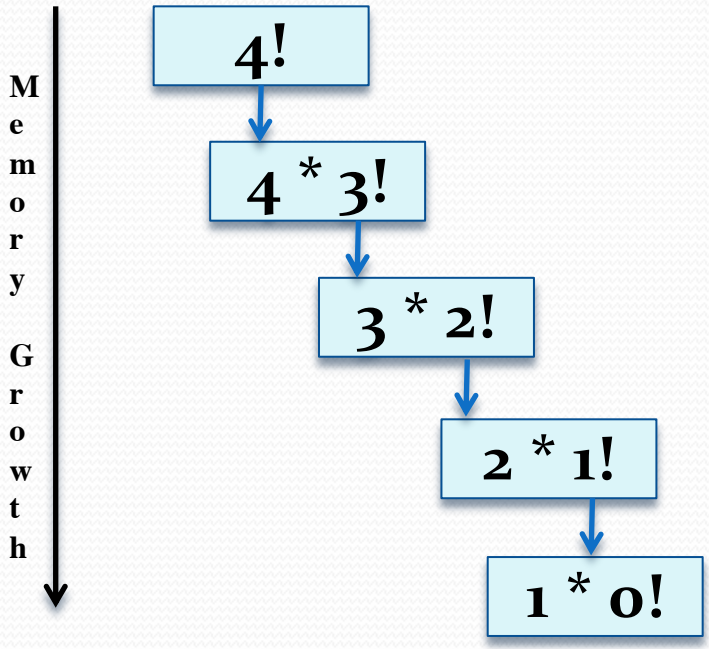
Recursive function Invocations

- What exactly happens when the recursive factorial function executes?
- Let's consider the case when the program computes 4!



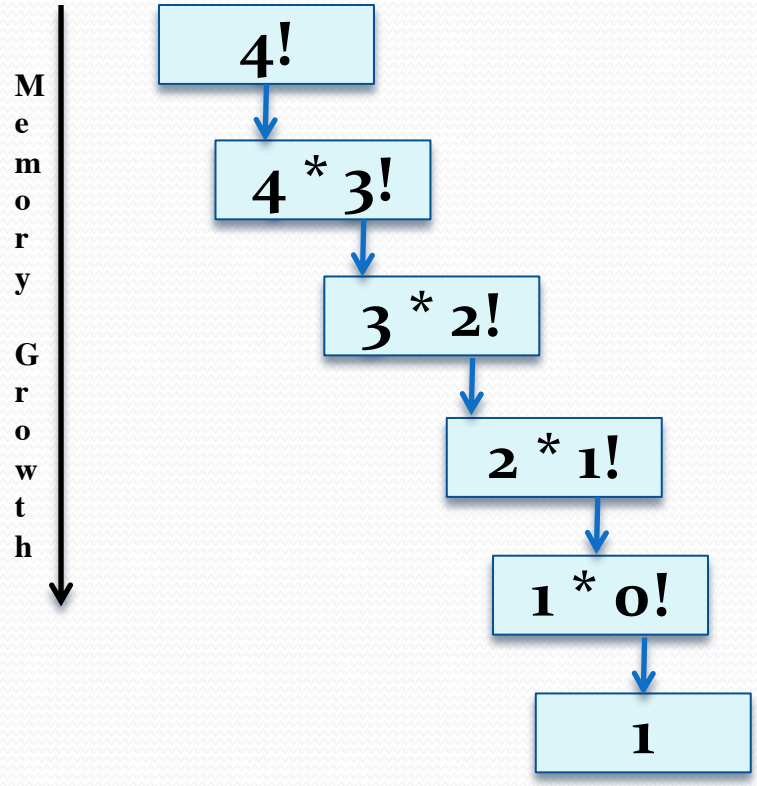
Recursive function Invocations

- What exactly happens when the recursive factorial function executes?
- Let's consider the case when the program computes 4!



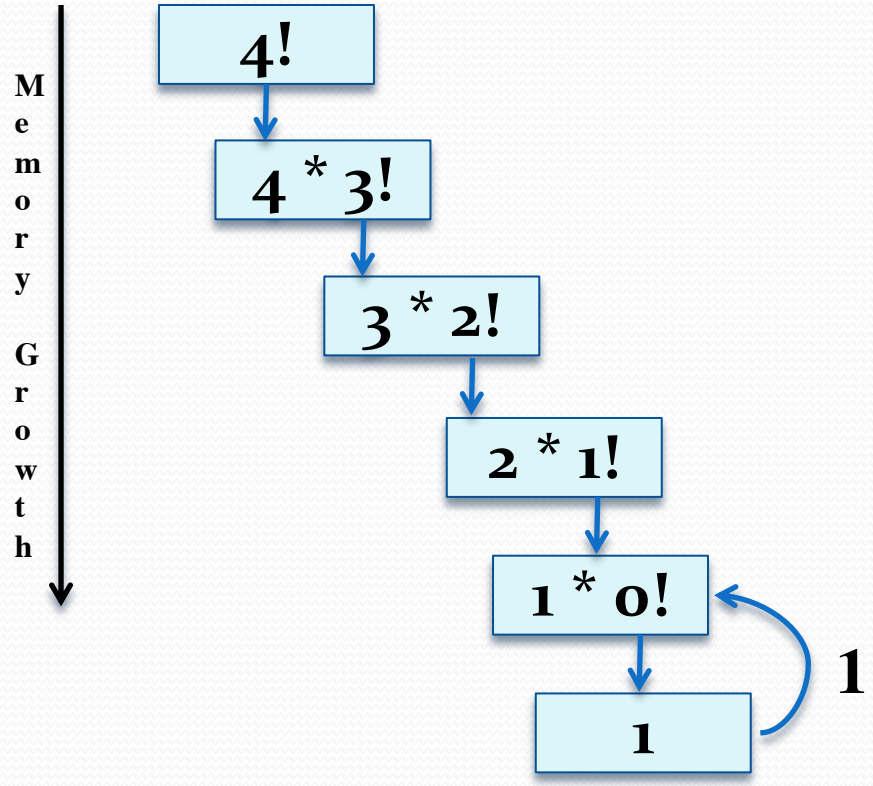
Recursive function Invocations

- What exactly happens when the recursive factorial function executes?
- Let's consider the case when the program computes 4!



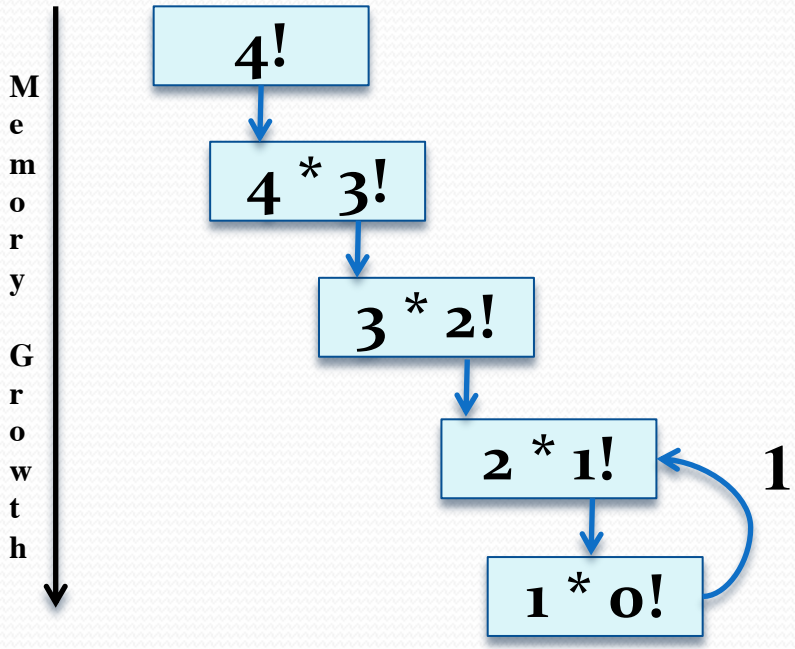
Recursive function Invocations

- What exactly happens when the recursive factorial function executes?
- Let's consider the case when the program computes 4!



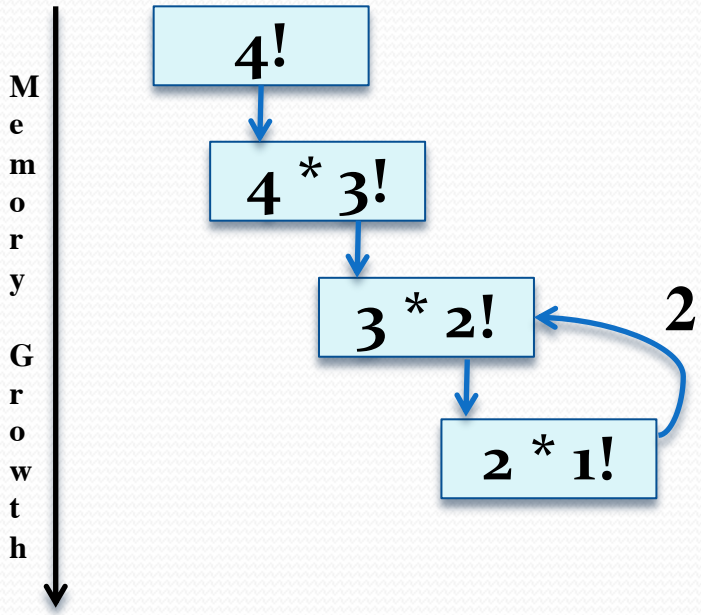
Recursive function Invocations

- What exactly happens when the recursive factorial function executes?
- Let's consider the case when the program computes 4!



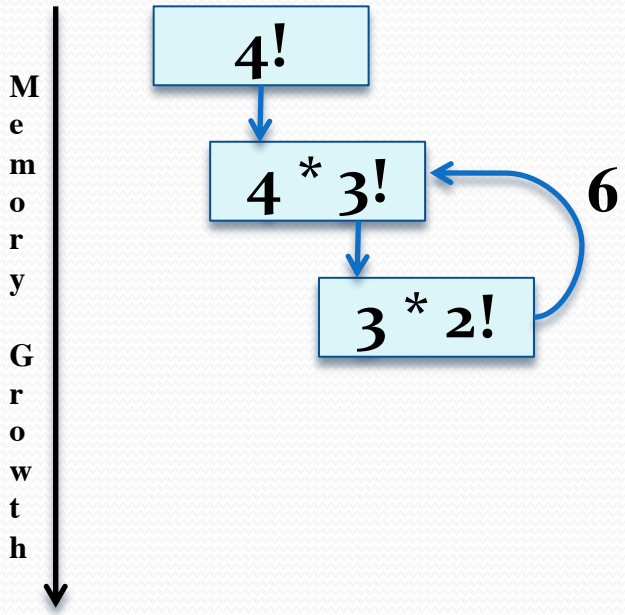
Recursive function Invocations

- What exactly happens when the recursive factorial function executes?
- Let's consider the case when the program computes 4!



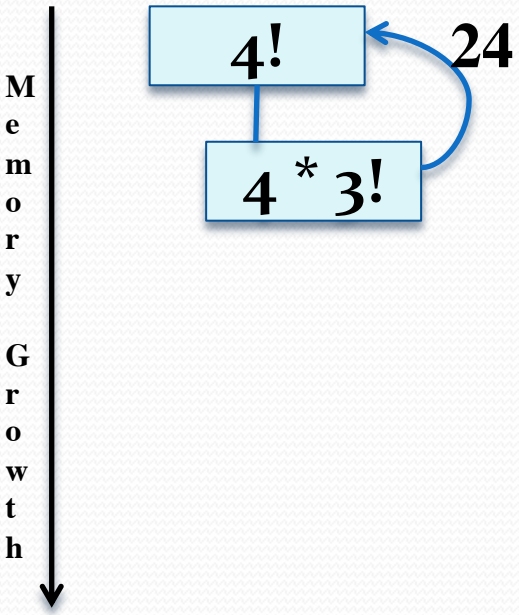
Recursive function Invocations

- What exactly happens when the recursive factorial function executes?
- Let's consider the case when the program computes 4!



Recursive function Invocations

- What exactly happens when the recursive factorial function executes?
- Let's consider the case when the program computes 4!



Recursive function Invocations

- What exactly happens when the recursive factorial function executes?
- Let's consider the case when the program computes 4!

4!

Result is 24

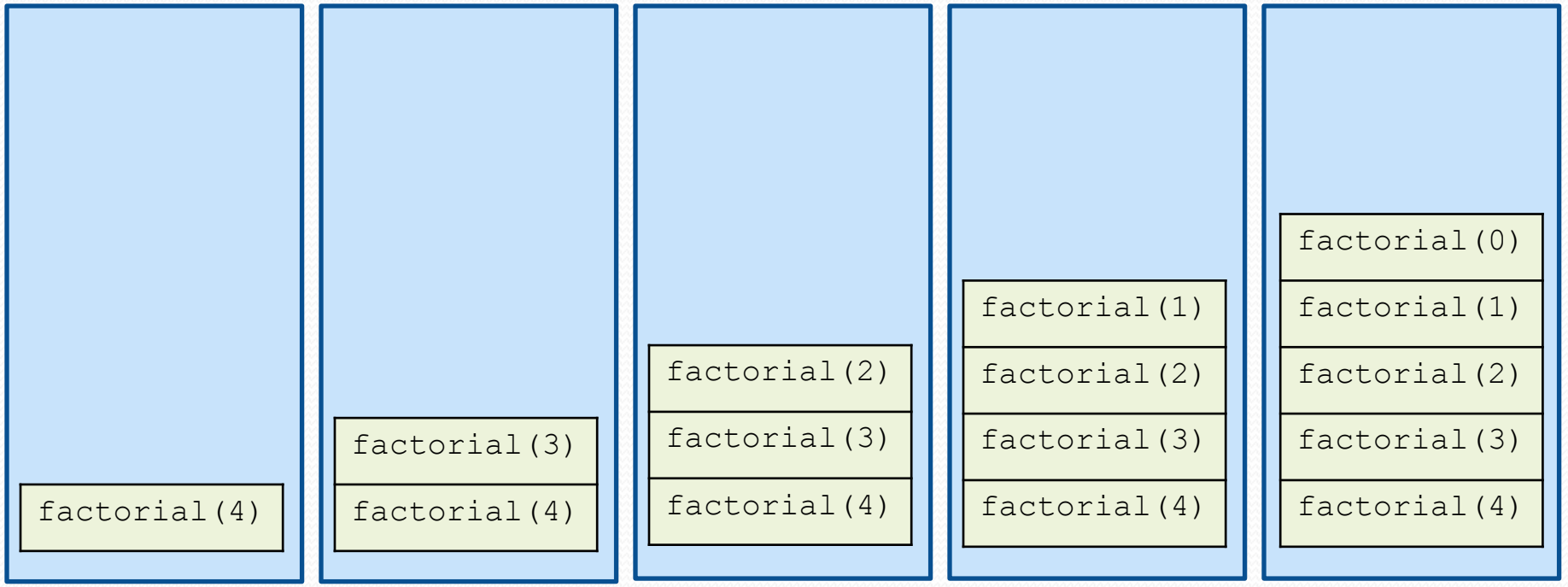
So, what happens when a recursive function executes?

Recursion is a form of Iteration

- Code is repeatedly executed.
- The call stack tracks function calls waiting for return values.

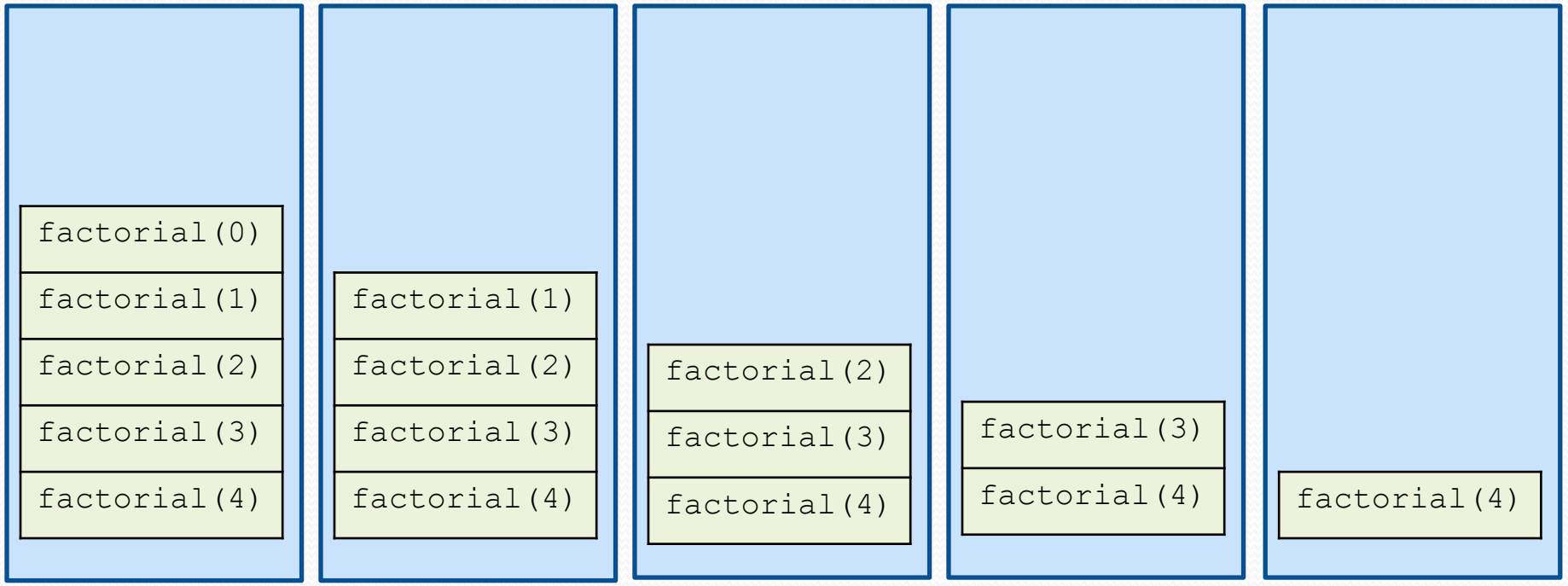
Recursion: the call stack

```
result = factorial (4)
```



Recursion: the call stack

```
result = factorial (4)
```



Tail Recursion

- Tail Recursion: Recursive function that does no use its return value in computations.
  - A recursive function that does not modify it's return value is Tail Recursive
  - A Tail Recursive function does not need to store the entire call stack.
- Tail Recursion allows for less memory use.

Tail Recursive factorial

```
def factorial_V2(number, result = 1) :
    if (number == 0):  # Base case
        return result
    else :  # Recursive call
        return factorial_V2(number - 1, result * number)
```

Tail Recursive factorial

- Let's consider the case when the program computes 4!
- We don't need to do math on the result, so the computer does not need to remember it.

```
factorial_V2(4) # result = 1 as default argument
```

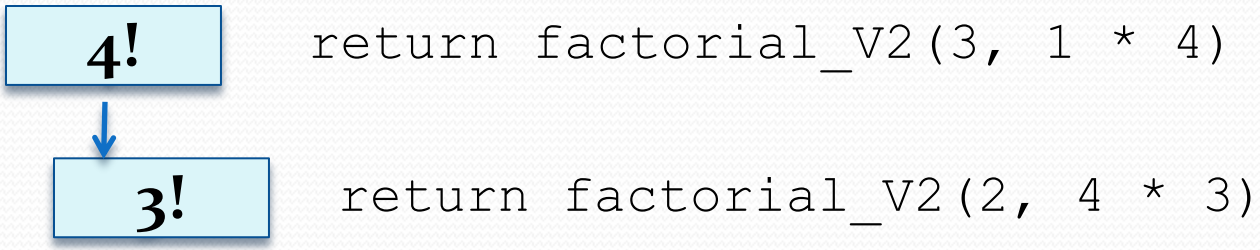
4!

```
return factorial_V2(3, 1 * 4)
```

Tail Recursive factorial

- We don't need to do math on the result, so the computer does not need to remember it.

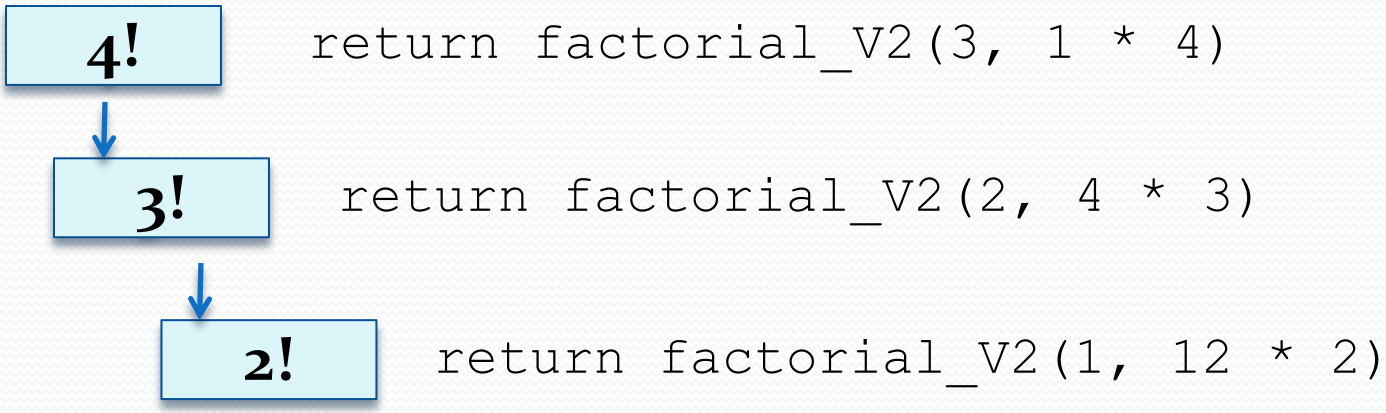
```
factorial_V2(4) # result = 1
```



Tail Recursive factorial

- We don't need to do math on the result, so the computer does not need to remember it.

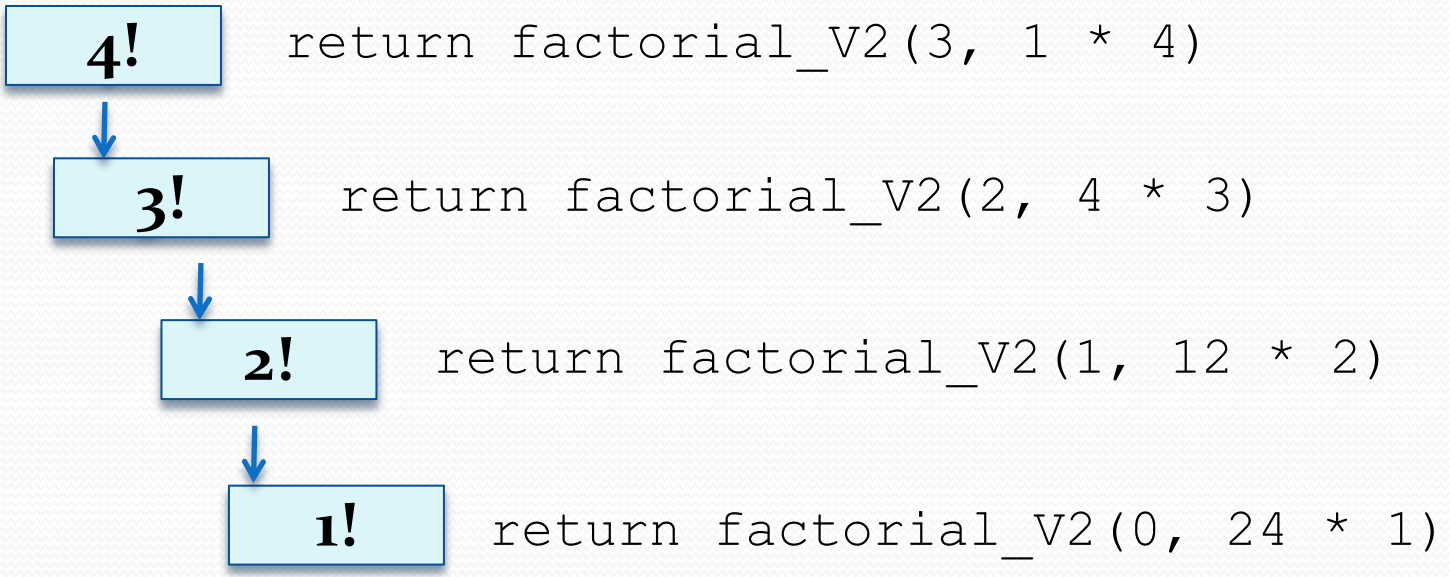
```
factorial_V2(4) # result = 1
```



Tail Recursive factorial

- We don't need to do math on the result, so the computer does not need to remember it.

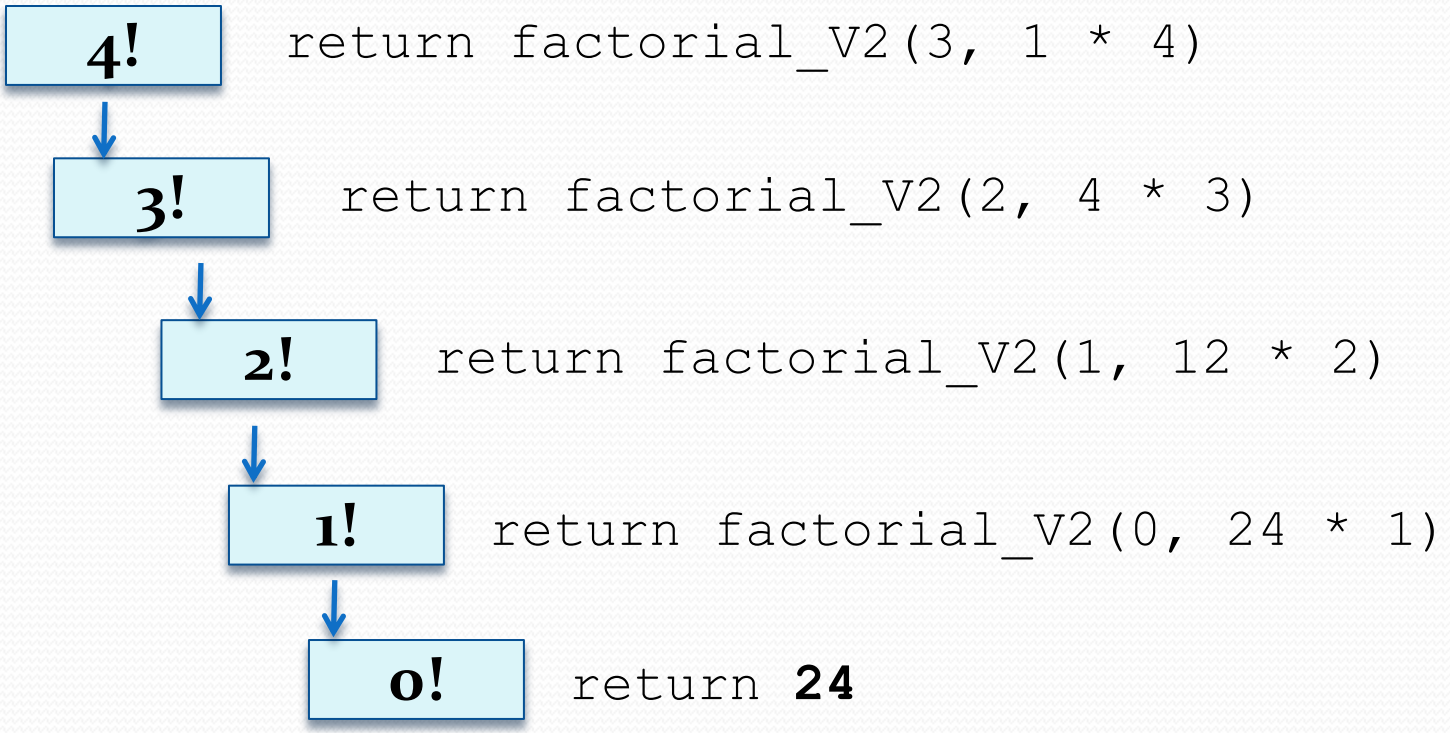
```
factorial_V2(4) # result = 1
```



Tail Recursive factorial

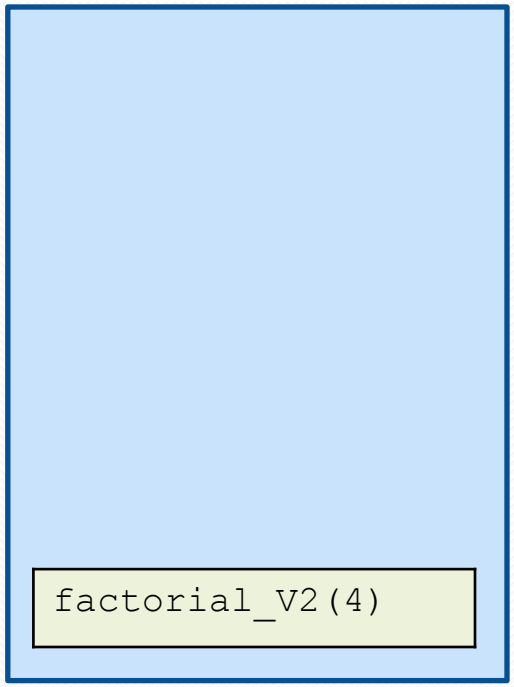
- We don't need to do math on the result, so the computer does not need to remember it.

```
factorial_V2(4) # result = 1
```



Tail Recursion: the call stack

```
factorial_V2(4) #result = 1
```



The tail recursion re-uses a single stack frame

Dangers

- There are several dangers associated with recursion that programmers need to be aware of

1. Recursive programs can fail to terminate
   - Remember that your function must have code that handles the termination conditions
     - There must be some way for the program to exit without calling itself again
2. Stack Overflow
   - Remember that every function is a separate task that the computer must keep track of.
   - The computer manages a list of tasks that it can maintain, but this list only has a limited amount of space.
   - Should a recursive function require many copies of itself to solve a problem, the computer may not be able to handle that many tasks, causing a system error.

Dangers

3. Out of memory Error
   - Every time a function is called it must allocate computer memory to copy the values of all the parameter variables.
   - If a recursive function has many parameters, and/or these parameters are memory intensive, the recursive calls can eat away at the computer's memory until there is no more, causing a system error