CMPUT 274

Recursion

Topics Covered:

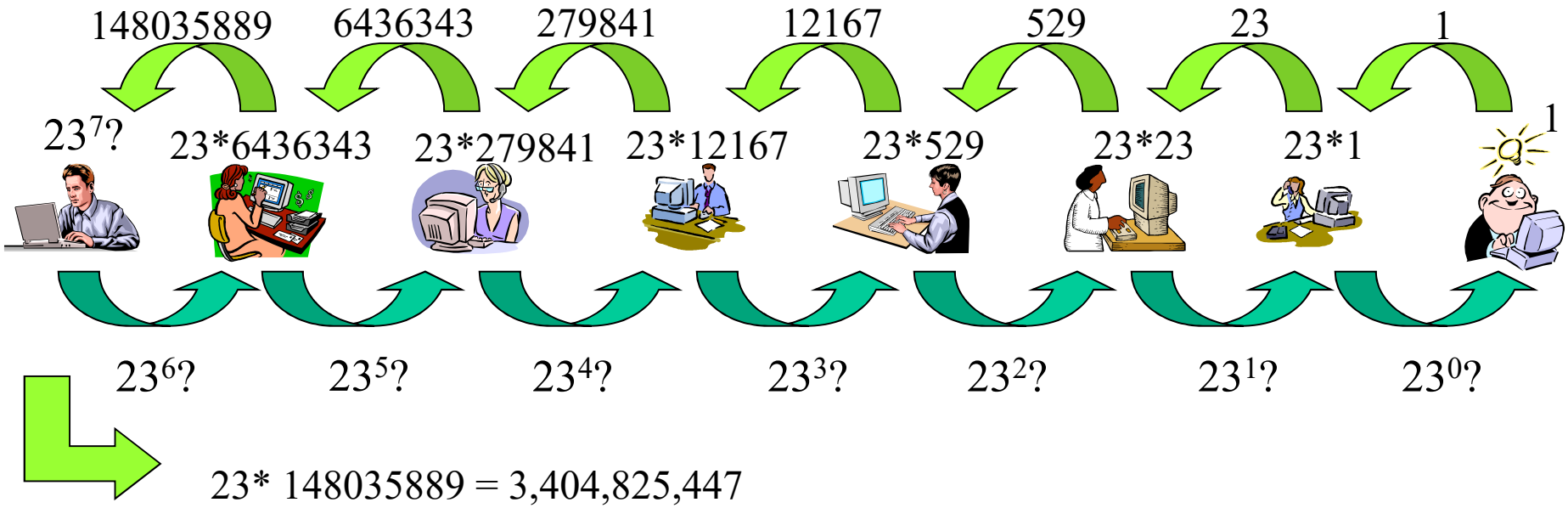
- What is recursion?
- Conditions for termination
- Stack frames

Recursion

- **Recursion** occurs when a function (or method) calls itself, either directly or indirectly.
- If a problem can be resolved by solving a simple part of it and resolving the rest of the big problem the same way, we can write a function that solves the simple part of the problem then calls itself to resolve the rest of the problem.
- This is called a **recursive function**.

Recursive Function Example

- Suppose we want to calculate 23^7. We know that 23^7 is $23*23^6$. If we know the solution for 23^6 we would know the solution for 23^7.



Recursive Methods

- For recursion to **terminate**, two conditions must be met:
  - there must be one or more simple cases that do **NOT** make recursive calls. (base case)
  - the recursive call must somehow be simpler than the original call. (Change the state to move towards the base case.)

Example: Factorial

- For example, how can we write a recursive function that computes the factorial of an Integer:

  0! = 1

  1! = 1

  2! = 2*1 = 2 ➔ 2! = 2*1!

  3! = 3*2*1 = 6 ➔ 3! = 3*2!

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

- The last observation, together with the simple cases, is the basis for a recursive function.

Factorial Function

$n! = n*(n-1)!$

```
def factorial(number):
    '''
    Return the factorial of number.
    '''
    if (number == 0 or number== 1): # base case
        answer = 1
  else:
      answer = number * factorial(number-1)

  return answer
```

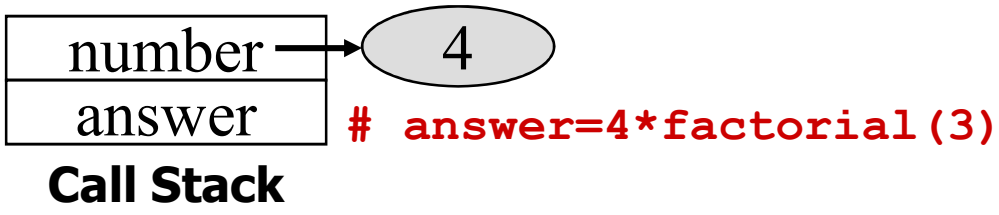
Function Activations and Frames

- When a function is invoked, a **frame** or **stack frame** corresponding to that function is created and pushed onto the call stack.
- The frame stores all of the local variables associated with that function call.
- The frame is created when the function is invoked, and destroyed when the function finishes.
- If a function is invoked again, a new frame is created for it with all its local variables.

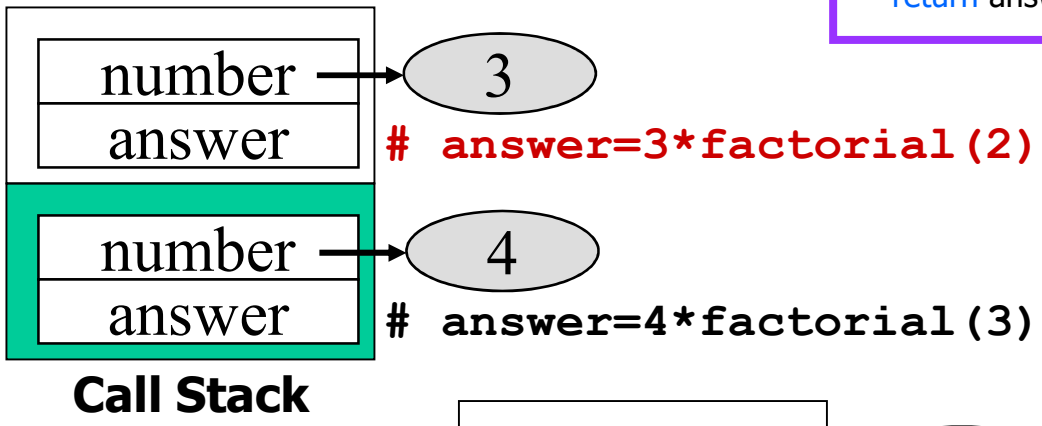
Multiple Activations of a Function

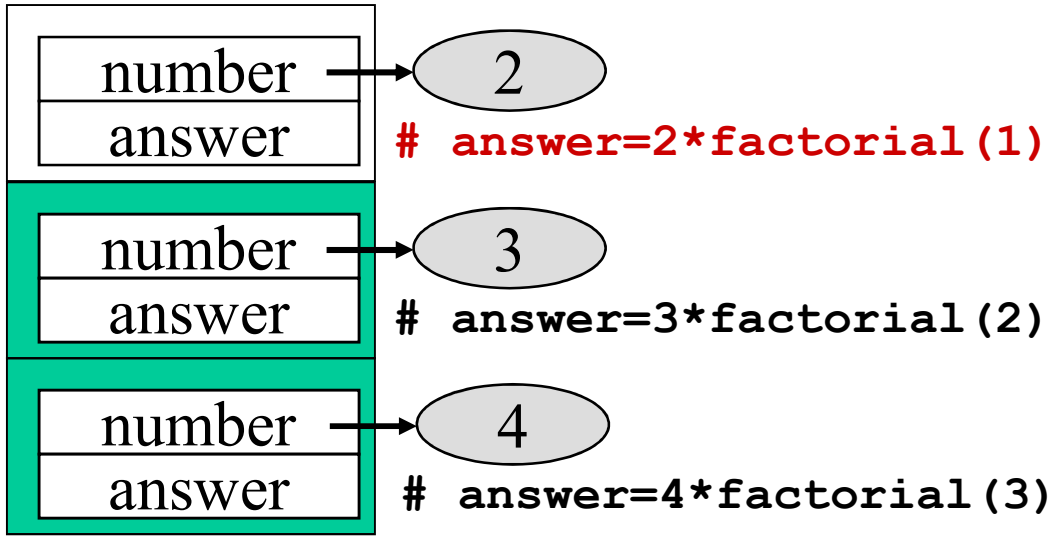
- When we invoke a recursive function, the function becomes active.
- Before it is finished, it makes a recursive call to the same function.
- This means that when recursion is used, there is more than one copy of the same function active at once.
- Therefore, each active function has its own frame which contains independent copies of its local variables.
- These frames are stored on the call stack.

Calling `factorial(4)`



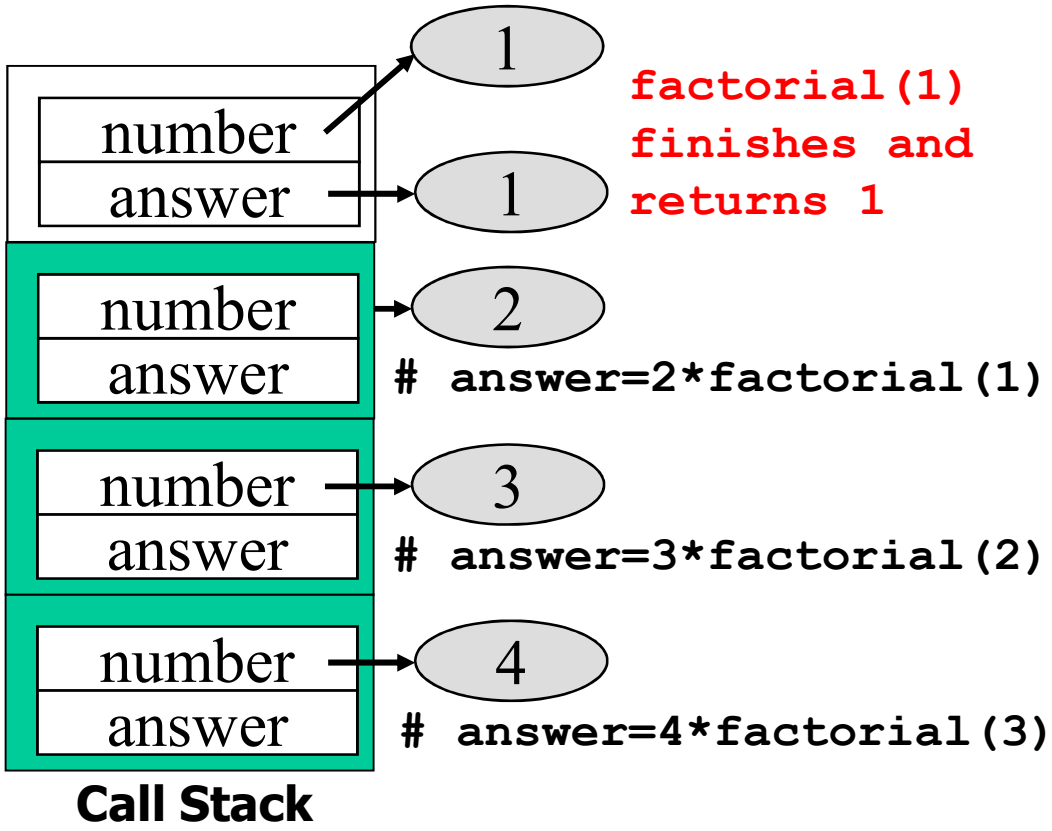

```
def factorial(number):
    if (number == 0 or number== 1):
        answer = 1
    else:
        answer = number * factorial(number-1)
    return answer
```





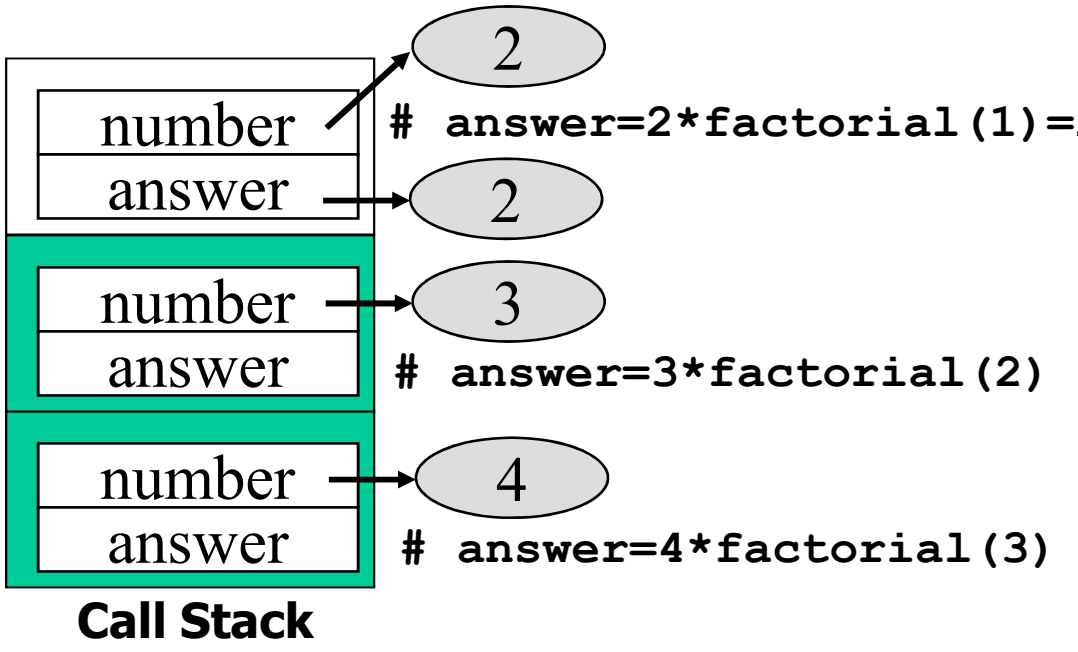
Calling `factorial(4)`

```
def factorial(number):
    if (number == 0 or number== 1):
        answer = 1
    else:
        answer = number * factorial(number-1)
    return answer
```



Calling `factorial(4)`

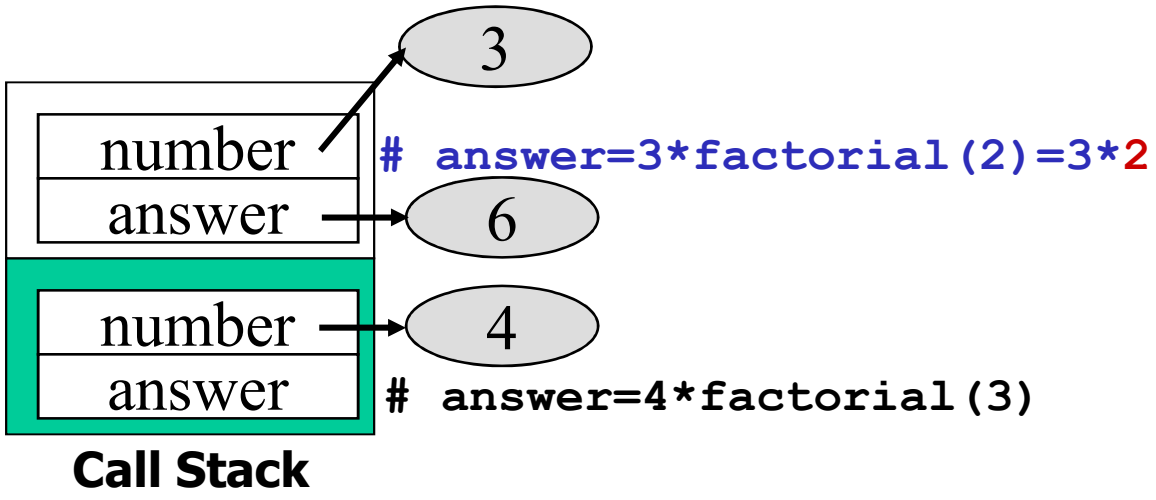
```
def factorial(number):
    if (number == 0 or number== 1):
        answer = 1
    else:
        answer = number * factorial(number-1)
    return answer
```



factorial(2) finishes and returns 2

Calling `factorial(4)`

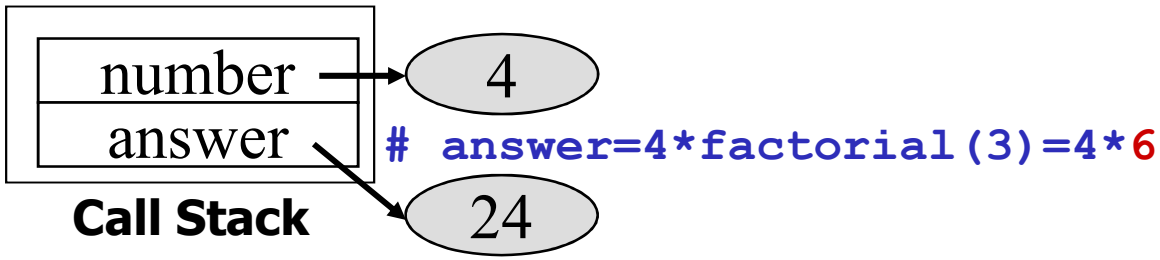
```
def factorial(number):
    if (number == 0 or number== 1):
        answer = 1
    else:
        answer = number * factorial(number-1)
    return answer
```



`factorial(3)` finishes and returns 6

Calling `factorial(4)`

```
def factorial(number):
    if (number == 0 or number== 1):
        answer = 1
    else:
        answer = number * factorial(number-1)
    return answer
```




```
# answer=4*factorial(3)=4*6
```

```
factorial(4)
finishes and
returns 24
```