Lecture 27
Recursion

Objectives

- To understand recursion.
- To understand when to use recursion.
- To take examples using recursion.

Recursive Problem-Solving

Algorithm: Factorial – find the factorial for x

```
if n == 0:
    return the factorial of zero    [0! = 1]
else
    find factorial of (n-1)         [(n-1)!]
    return n * (n-1)!
```

- This version has no loop, and seems to refer to itself! What's going on??

Recursive Definitions

- A description of something that refers to itself is called a *recursive* definition.
- In the last example, the factorial algorithm uses its own description – a "call" to factorial "recurs" inside of the definition – hence the label "recursive definition."

Recursive Definitions

- In mathematics, recursion is frequently used. The most common example is the factorial:

- For example, $5! = 5(4)(3)(2)(1)$, or
  $5! = 5(4!)$

$$n! = n(n-1)(n-2)...(1)$$

Recursive Definitions

- In other words, $n! = n(n-1)!$
- Or

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

- This definition says that 0! is 1, while the factorial of any other number is that number times the factorial of one less than that number.

Recursive Definitions

- Our definition is recursive, but definitely not circular. Consider 4!
  - $4! = 4(4\text{-}1)! = 4(3!)$
  - *What is 3!? We apply the definition again*

    $4! = 4(3!) = 4[3(3\text{-}1)!] = 4(3)(2!)$
  - *And so on…*

    $4! = 4(3!) = 4(3)(2!) = 4(3)(2)(1!) = 4(3)(2)(1)(0!) = 4(3)(2)(1)(1) = 24$
- Factorial is not circular because we eventually get to 0!, whose definition does not rely on the definition of factorial and is just 1. This is called a *base case* for the recursion.
- When the base case is encountered, we get a closed expression that can be directly computed.

Recursive Definitions

- All good recursive definitions have these two key characteristics:

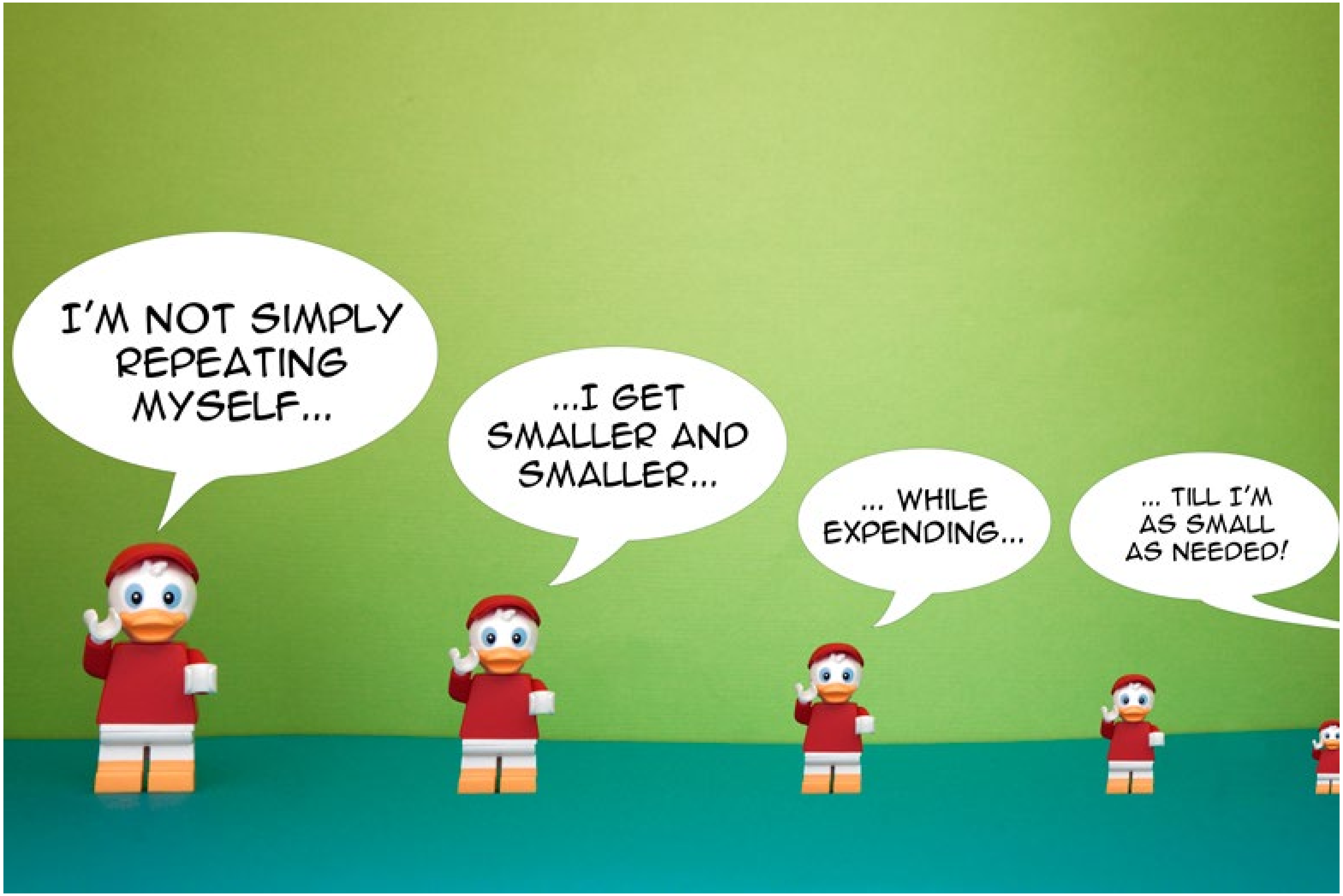
    1. *There are one or more base cases for which no recursion is applied.*

    2. *All chains of recursion eventually end up at one of the base cases.*

    Put differently, each iteration must drive the computation toward a base case.

- The simplest way for these two conditions to occur is for each recursion to act on a smaller version of the original problem. A very small version of the original problem that can be solved without recursion becomes the base case.

Recursion


I'M NOT SIMPLY REPEATING MYSELF...
...I GET SMALLER AND SMALLER...
... WHILE EXPENDING...
... TILL I'M AS SMALL AS NEEDED!

Recursive Functions

- Factorial can be calculated using a loop accumulator.

```
def fact_loop(n):
    ans = 1
    for i in range(n,1,-1):
        ans *= i
    return ans
```

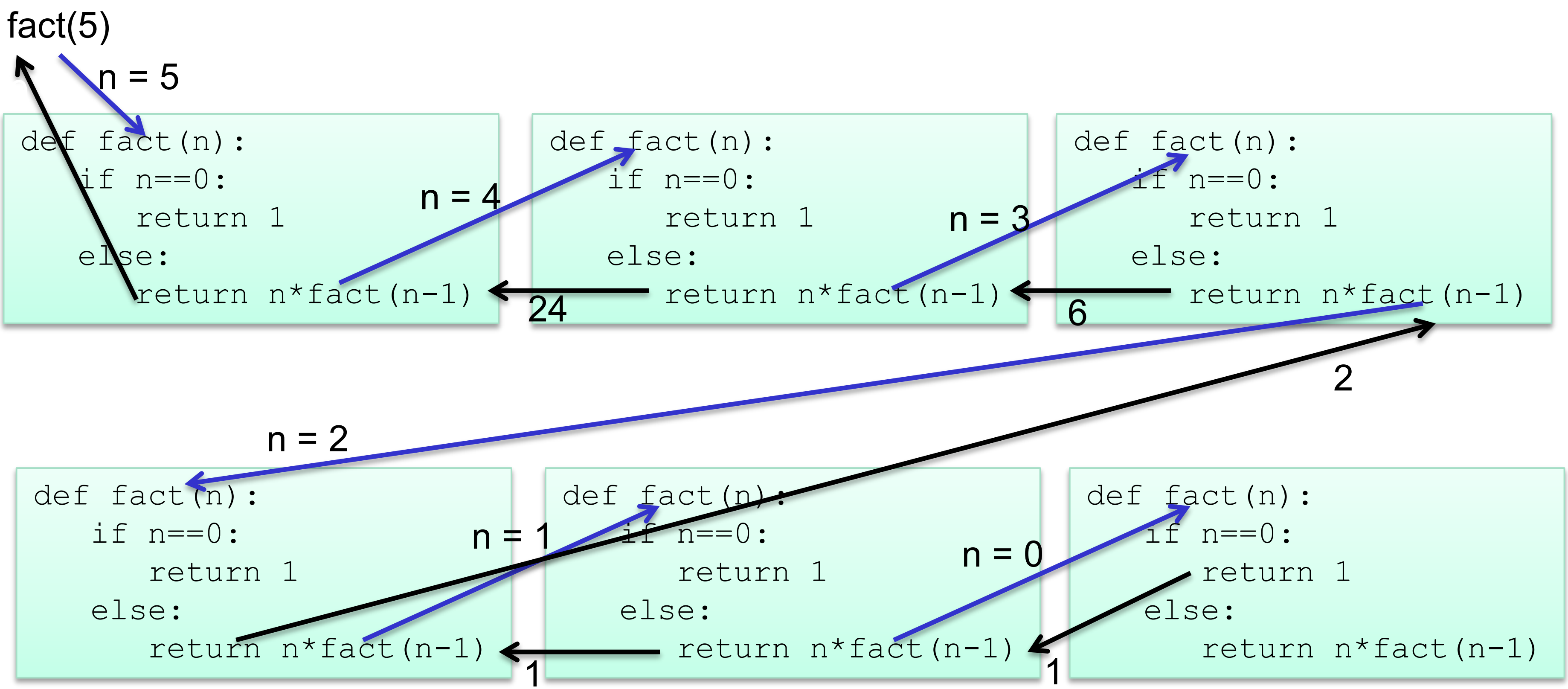
- If factorial is written as a separate recursive function:

```
def fact(n):
    if n == 0:
        return 1
    else:
        return n * fact(n-1)
```

Recursive Functions

- We've written a function that calls *itself*, i.e. a *recursive function*.
- The function first checks to see if we're at the base case `(n==0)`. If so, return `1`. Otherwise, return the result of multiplying *n* by the factorial of *n*−*1*, `fact(n-1)`.
- Remember that each call to a function starts that function anew, with its own copies of local variables and parameters.

Recursive Functions



Example: Binary Search

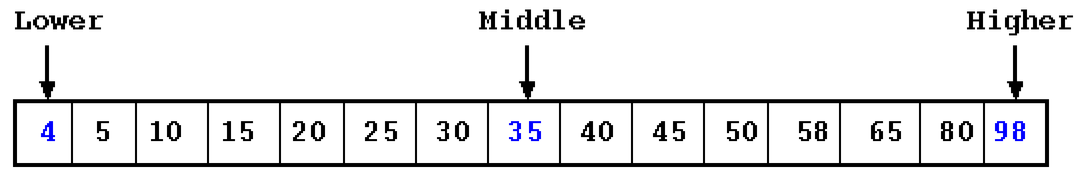
- In the last lecture, we learned how to perform binary search using a loop.
- If you haven't noticed already, we can perform binary search recursively.
- In binary search, we look at the middle value first, then we either search the lower half or upper half of the array.
- There are two base cases (to stop recursion/searching):
  - *when the target value is found*
  - *when we have run out of places to look.*

Example: Binary Search

- The recursive calls will cut the search in half each time by specifying the range of locations that are not searched and may contain the target value.
- Each invocation of the search routine will search the list between the given *low* and *high* parameters.

Example: Binary Search

```
def recBinSearch(x, nums, low, high):
    if low > high:          # No place left to look, return -1
        return -1
    mid = (low + high)//2
    item = nums[mid]
    if item == x:
        return mid
    if x < item:          # Look in lower half
        return recBinSearch(x, nums, low, mid-1)
                          # Look in upper half
    return recBinSearch(x, nums, mid+1, high)
```



We can then call the binary search with a generic search wrapping function

```
def search(x, nums):
 return recBinSearch(x,nums,0,len(nums)-1)
```

Summary

- We learned the concept of recursion.
- We took the example of factorial and binary search.