Recursion

The Recursion Pattern

- **Recursion**: when a method calls itself
- Classic example--the factorial function:
  - n! = 1· 2· 3· ··· · (n-1)· n
- Recursive definition:

$$f(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot f(n-1) & else \end{cases}$$

- As a Python method:

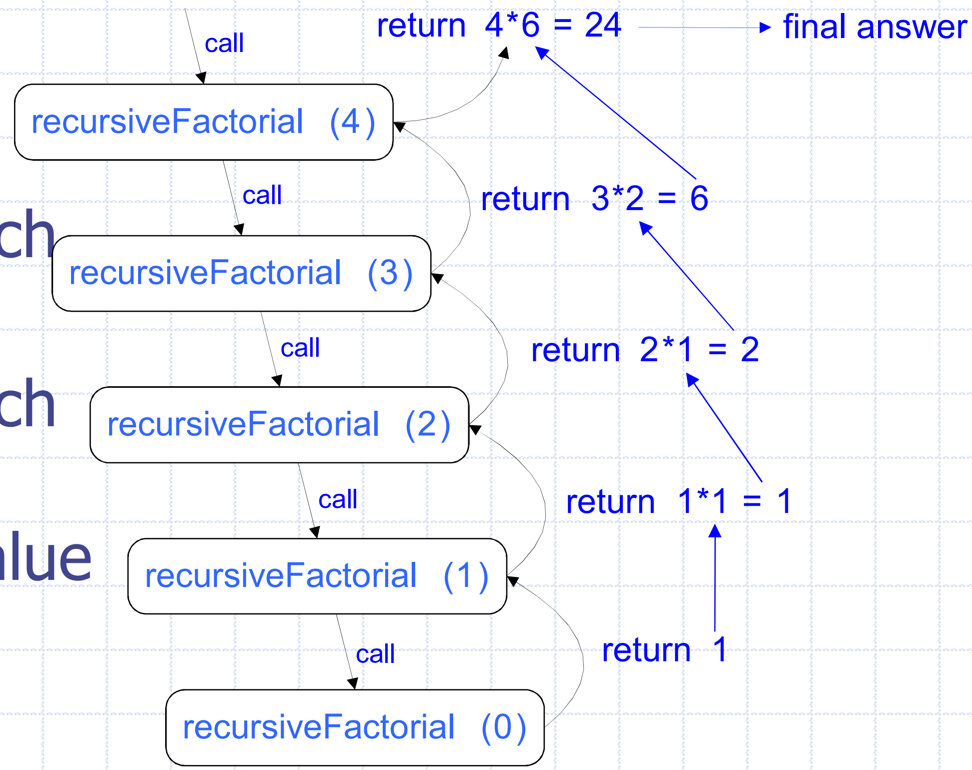
```
def factorial(n):
  if n == 0:
    return 1
  else:
    return n * factorial(n−1)
```

Content of a Recursive Method

- Base case(s)
  - Values of the input variables for which we perform no recursive calls are called base cases (there should be at least one base case).
  - Every possible chain of recursive calls must eventually reach a base case.
- Recursive calls
  - Calls to the current method.
  - Each recursive call should be defined so that it makes progress towards a base case.

Visualizing Recursion

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



Example: English Ruler

- Print the ticks and numbers like an English ruler:

```
---- 0          ----- 0          --- 0
-               -                -
--              --               --
-               -                -
---             ---              --- 1
-               -                -
--              --               --
-               -                -
---- 1          ----             --- 2
-               -                -
--              --               --
-               -                -
---             ---              --- 3
-               -
--              --
-               -
---- 2          ----- 1
```

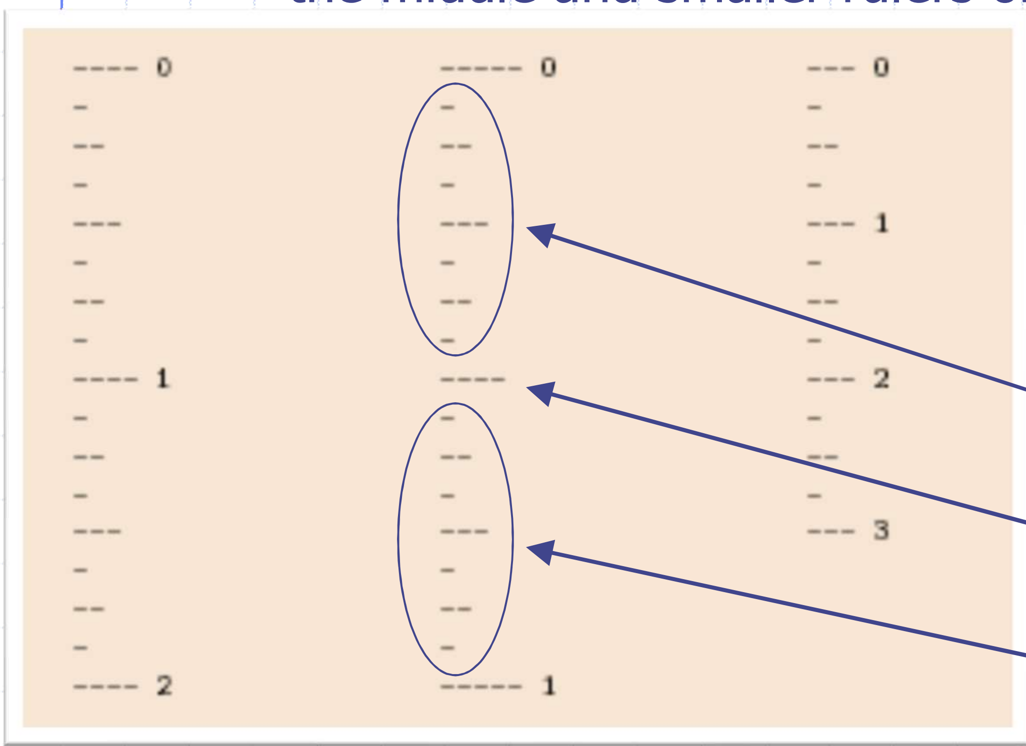
Slide by Matt Stallmann included with permission.

Using Recursion

drawTicks(length)

Input: length of a ‘tick’

Output: ruler with tick of the given length in the middle and smaller rulers on either side



drawTicks(length)

if(length > 0) then

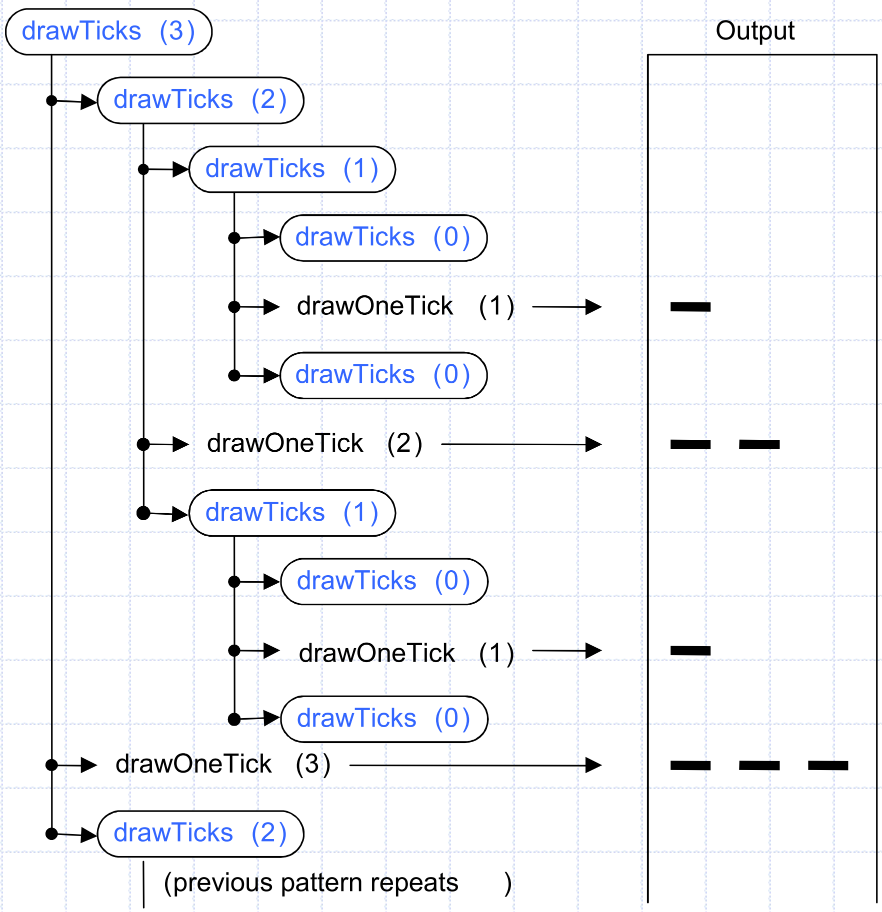
drawTicks(length – 1)

draw tick of the given length

drawTicks(length – 1)

Recursive Drawing Method

- The drawing method is based on the following recursive definition
- An interval with a central tick length L $\geq$1 consists of:
  - An interval with a central tick length L–1
  - An single tick of length L
  - An interval with a central tick length L–1



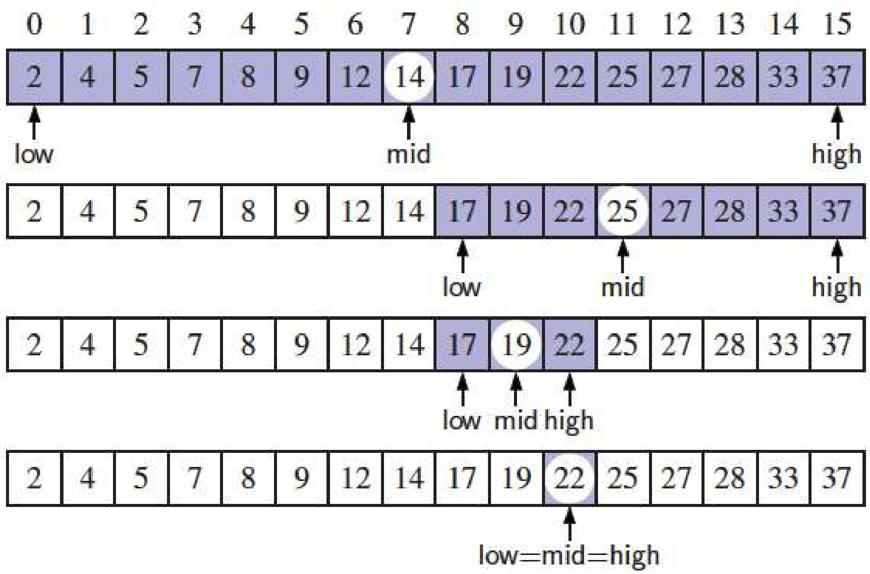
A Recursive Method for Drawing Ticks on an English Ruler

```python
def draw_line(tick_length, tick_label=''):
  """Draw one line with given tick length (followed by optional label)."""
  line = '-' * tick_length
  if tick_label:
    line += ' ' + tick_label
  print(line)

def draw_interval(center_length):
  """Draw tick interval based upon a central tick length."""
  if center_length > 0:                  # stop when length drops to 0
    draw_interval(center_length - 1)     # recursively draw top ticks
    draw_line(center_length)             # draw center tick
    draw_interval(center_length - 1)     # recursively draw bottom ticks

def draw_ruler(num_inches, major_length):
  """Draw English ruler with given number of inches, major tick length."""
  draw_line(major_length, '0')           # draw inch 0 line
  for j in range(1, 1 + num_inches):
    draw_interval(major_length - 1)      # draw interior ticks for inch
    draw_line(major_length, str(j))      # draw inch j line and label
```

Note the two recursive calls

Visualizing Binary Search

- We consider three cases:
  - If the target equals data[mid], then we have found the target.
  - If target < data[mid], then we recur on the first half of the sequence.
  - If target > data[mid], then we recur on the second half of the sequence.



Binary Search

- Search for an integer, target, in an ordered list.

```
def binary_search(data, target, low, high):
  """Return True if target is found in indicated portion of a Python list.

  The search only considers the portion from data[low] to data[high] inclusive.
  """
  if low > high:
    return False                          # interval is empty; no match
  else:
    mid = (low + high) // 2
    if target == data[mid]:               # found a match
      return True
    elif target < data[mid]:
      # recur on the portion left of the middle
      return binary_search(data, target, low, mid − 1)
    else:
      # recur on the portion right of the middle
      return binary_search(data, target, mid + 1, high)
```

Linear Recursion

- Test for base cases
  - Begin by testing for a set of base cases (there should be at least one).
  - Every possible chain of recursive calls must eventually reach a base case, and the handling of each base case should not use recursion.
- Recur once
  - Perform a single recursive call
  - This step may have a test that decides which of several possible recursive calls to make, but it should ultimately make just one of these calls
  - Define each possible recursive call so that it makes progress towards a base case.

Example of Linear Recursion

Algorithm LinearSum(*A*, *n*):

Input:

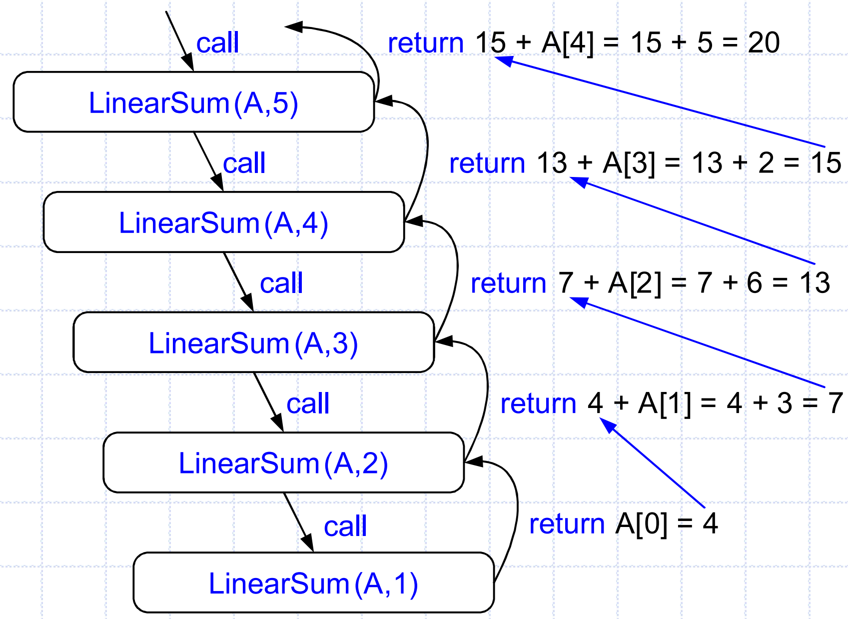
A integer array *A* and an integer *n* = 1, such that *A* has at least *n* elements

Output:

The sum of the first *n* integers in *A*

if *n* = 1 **then**
  return *A*[0]
else
  return LinearSum(*A*, *n* - 1) + *A*[*n* - 1]

Example recursion trace:



Reversing an Array

Algorithm ReverseArray(A, i, j):

Input: An array A and nonnegative integer indices i and j

Output: The reversal of the elements in A starting at index i and ending at j

if $i < j$ **then**

Swap $A[i]$ and $A[j]$

ReverseArray(A, $i + 1$, $j - 1$)

return

Defining Arguments for Recursion

- In creating recursive methods, it is important to define the methods in ways that facilitate recursion.
- This sometimes requires we define additional paramaters that are passed to the method.
- For example, we defined the array reversal method as ReverseArray(*A, i, j*), not ReverseArray(*A*).
- Python version:

```python
def reverse(S, start, stop):
  """Reverse elements in implicit slice S[start:stop]."""
  if start < stop - 1:                        # if at least 2 elements:
    S[start], S[stop-1] = S[stop-1], S[start] # swap first and last
    reverse(S, start+1, stop-1)               # recur on rest
```

Tail Recursion

- Tail recursion occurs when a linearly recursive method makes its recursive call as its last step.
- The array reversal method is an example.
- Such methods can be easily converted to non-recursive methods (which saves on some resources).
- Example:

Algorithm IterativeReverseArray(A, i, j):

Input: An array A and nonnegative integer indices i and j

Output: The reversal of the elements in A starting at index i and ending at j

while $i < j$ **do**

Swap $A[i]$ and $A[j]$

$i = i + 1$

$j = j - 1$

return

Binary Recursion

- Binary recursion occurs whenever there are **two** recursive calls for each non-base case.
- Example from before: the DrawTicks method for drawing ticks on an English ruler.

```
---- 0          ----- 0         --- 0
-               -               -
--              --              --
-               -               -
---             ---             --- 1
-               -               -
--              --              --
-               -               -
---- 1          ----            --- 2
-               -               -
--              --              --
-               -               -
---             ---             --- 3
-               -
--              --
-               -
---- 2          ----- 1
```

Another Binary Recusive Method

- Problem: add all the numbers in an integer array A:

Algorithm BinarySum(*A, i, n*):
Input: An array *A* and integers *i* and *n*
Output: The sum of the *n* integers in *A* starting at index *i*
if $n = 1$ **then**
return *A*[*i*]
return BinarySum(*A, i, n/* 2) + BinarySum(*A, i* + *n/* 2, *n/* 2)

- **Example trace:**