CS134 Lecture 21: Graphical Recursion
Announcements & Logistics

- **Lab 7** today and tomorrow: focuses on **recursion**
  - Please write/print the **pre lab** before you come to lab
  - Partner lab: you and your partner have to attend the same lab section
  - We will be collecting it at the start of lab.
  - Prelab is an **individual** assignment
    - You may discuss with your partner after submitting it
- **HW 6** due @ 10 pm
  - We made a mistake on one question — Glow is now fixed. The question no longer counts against your quiz score.

Do You Have Any Questions?
Maximum coverage at **3.27 pm** at Williams College

For the 2.30pm lab folks: we can walk out to watch for a few mins, so bring your eclipse viewing glasses!
Last Time: Recursive Approach to Problem Solving

• A recursive function is a function **that calls itself**

• A recursive approach to problem solving has two main parts:
  
  • **Base case(s).** When the problem is **so small**, we solve it directly, without having to reduce it any further
  
  • **Recursive step.** Does the following things:
    
    • Performs an action that contributes to the solution
    
    • **Reduces** the problem to a smaller version of the same problem, and calls the function on this **smaller subproblem**
    
    • The recursive step is a form of "wishful thinking" (also called the inductive hypothesis)
Today’s Plan

• Introduction to Turtle
• Graphical recursion examples
• Understanding function **invariance** and why it matters when doing recursion
The Turtle Module

• Turtle is a **graphics module** first introduced in the 1960s by computer scientists Seymour Papert, Wally Feurzig, and Cynthia Solomon.

• It uses a programmable cursor — fondly referred to as the “turtle” — to draw on a Cartesian plane (x and y axis.)
Turtle In Python

- **turtle** is available as a built-in module in Python. See the [Python turtle module API](#) for details.
- Basic turtle commands:

  Use `from turtle import *` to use these commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fd(dist)</code></td>
<td>turtle moves <strong>forward</strong> by <code>dist</code></td>
</tr>
<tr>
<td><code>bk(dist)</code></td>
<td>turtle moves <strong>backward</strong> by <code>dist</code></td>
</tr>
<tr>
<td><code>lt(angle)</code></td>
<td>turtle turns <strong>left</strong> <code>angle</code> degrees</td>
</tr>
<tr>
<td><code>rt(angle)</code></td>
<td>turtle turns <strong>right</strong> <code>angle</code> degrees</td>
</tr>
<tr>
<td><code>up()</code></td>
<td>(pen <strong>up</strong>) turtle raises pen in belly</td>
</tr>
<tr>
<td><code>down()</code></td>
<td>(pen <strong>down</strong>) turtle lowers pen from belly</td>
</tr>
<tr>
<td><code>shape(shp)</code></td>
<td>sets the turtle's <strong>shape</strong> to <code>shp</code></td>
</tr>
<tr>
<td><code>speed(spd)</code></td>
<td>sets the turtle's <strong>speed</strong> 1-10 (slow-fast). 0 skips animation.</td>
</tr>
<tr>
<td><code>home()</code></td>
<td>turtle returns to (0,0) (center of screen)</td>
</tr>
<tr>
<td><code>clear()</code></td>
<td>delete turtle drawings; no change to turtle's state</td>
</tr>
<tr>
<td><code>reset()</code></td>
<td>delete turtle drawings; <code>reset</code> turtle's state</td>
</tr>
<tr>
<td><code>setup(width, height)</code></td>
<td>create a turtle window of given <code>width</code> and <code>height</code></td>
</tr>
</tbody>
</table>
Basic Turtle Movement

- `forward(dist)` or `fd(dist)`,
- `left(angle)` or `lt(angle)`,
- `right(angle)` or `rt(angle)`,
- `backward(dist)` or `bk(dist)`

```python
# set up a 400x400 turtle window
setup(400, 400)
reset()

fd(100)  # move the turtle forward 100 pixels
lt(90)   # turn the turtle 90 degrees to the left
fd(100)  # move forward another 100 pixels

# complete a square
lt(90)
fd(100)
lt(90)
lt(90)
fd(100)
done()
```
We can write functions that use turtle commands to draw shapes.

For example, here’s a function that draws a square of the desired size:

```python
def draw_square(length):
    # a loop that runs 4 times
    # and draws each side of the square
    for i in range(4):
        fd(length)
        lt(90)
done()
```

```python
setup(400, 400)
reset()
draw_square(150)
```
Drawing Basic Shapes With Turtle

• How about drawing polygons?

```python
def draw_polygon(length, num_sides):
    for i in range(num_sides):
        fd(length)
        lt(360/num_sides)
    done()
```

draw_polygon(80, 3)  # Triangle

draw_polygon(80, 10)  # Octagon
Adding Color!

- What if we wanted to add some color to our shapes?

```python
def draw_polygon_color(length, num_sides, color):
    # set the color we want to fill the shape with
    # color is a string
    fillcolor(color)

    begin_fill()
    for i in range(num_sides):
        fd(length)
        lt(360/num_sides)
    end_fill()
    done()

draw_polygon_color(80, 10, "gold")
draw_polygon_color(80, 10, "purple")
```
Recursive Figures With Turtle

• Let’s explore how to draw pretty recursive pictures with Turtle
• We’ll start with figures that only require recursive calls
• Below we have a set of concentric circles of alternating colors
• How is this recursive?
Example:
Concentric Circles
Concentric Circles

- Function definition

```python
concentric_circles(radius, gap)
```

- **radius**: radius of the outermost circle
- **gap**: width of gap between circles
Concentric Circles With No Colors

- Let’s first think about the circles without colors.
- **Base case**: radius of the circle is so small it’s not worth drawing
- **Recursive step**:
  - Draw a single circle of radius $r$, increment total by 1
  - Recursively draw concentric circles starting with an outer circle of a slightly smaller radius $r-g$ (where $g$ is any positive number you want to shrink the radius by, or the “gap” between the circles)
- Let's also count the number of circles we draw to understand the process

Counting the number of circles isn't necessary for drawing pictures, but it does make debugging easier!
Concentric Circles

```python
def concentric_circles(radius, gap):
    '''draw concentric circles and return # circles drawn'''
    # base case, don't draw anything, return 0
    if radius < gap:
        return 0
    else:
        # tell the turtle draw a circle
        circle(radius)

        # recursive function call; draw smaller circles
        num = concentric_circles(radius-gap, gap)

        # we drew one circle in this step, plus however many we
        # drew recursively, so return 1 + num
        return 1 + num
```

• Are we done?
Concentric Circles

print("Num Circles:", concentric_circles(300, 30))

Num Circles: 10

- Pretty picture, and almost there! But not quite right. What happened?
Concentric Circles

```python
print("Num Circles:", concentric_circles(300, 30))
```
 Num Circles: 10

• We need to reposition the turtle after each recursive call.
```python
def concentric_circles(radius, gap):
    # base case, don't draw anything
    if radius < gap:
        return 0
    else:
        # pen down, draw circle
down()
circle(radius)

        # pen up, ensure the turtle doesn't draw while repositioning
up()

        # reposition the turtle for the next circle
lt(90)
fd(gap)
rt(90)

        # recursive function call; draw smaller circles
num = concentric_circles(radius-gap, gap)

        # we drew one circle in this step, plus however many we
        # drew recursively, so return 1 + num
return 1 + num
```
Great! Now let's add some color.
Concentric Circles With Colors

- Function definition

\texttt{concentric\_circles(radius, gap, color\_outer, color\_inner)}

- \texttt{radius}: radius of the outermost circle
- \texttt{gap}: width of the gap between circles
- \texttt{color\_outer}: color of the outermost circle
- \texttt{color\_inner}: color that alternates with color\_outer
Concentric Circles: Adding Color

• Base case and recursive case stay the same
• How do we achieve the alternating colors?
• Just swap the order in the recursive call
  • \texttt{color\_outer} becomes \texttt{color\_inner} and vice versa
• Let's also write a helper function to draw a circle filled in with some color to clean up the recursive function itself
```python
def draw_disc(radius, color):
    # put the pen down
    down()

    # set the color
    fillcolor(color)

    # draw the circle
    begin_fill()
    circle(radius)
    end_fill()

    # put the pen up
    up()
```

Helper Function

Draw circle of a given radius and fill it with color.

Turtle: pen commands

- `down()`
- `up()`

Starting position of turtle

$(0, -radius)$
def concentric_circles_color(radius, gap, color_outer, color_inner):
    # Recursive function to draw concentric circles with alternating colors
    if radius < gap:
        return 0
    else:
        draw_disc(radius, color_outer)
        lt(90)
        fd(gap)
        rt(90)
        num = concentric_circles_color(radius-gap, gap, color_inner, color_outer)
        return 1 + num
Concentric Circles

print("Num circles:", concentric_circles_color(300, 30, "gold", "purple"))

Num Circles: 10
Function Frame Model: concentric_circles
```python
def concentric_circles(radius, gap, color_outer, color_inner):
    """Recursive function to draw concentric circles"""
    if radius < gap:
        return 0
    else:
        draw_disc(radius, color_out)
        lt(90); fd(gap); rt(90)
        num = concentric_circles(radius-gap, gap, color_in, color_out)
        return 1 + num
```
def concentric_circles(radius, gap, color_out, color_in):
    """Recursive function to draw concentric circles"""
    if radius < gap:
        return 0
    else:
        draw_disc(radius, color_out)
        lt(90); fd(gap); rt(90)
        num = concentric_circles(radius - gap, gap, color_in, color_out)
        return 1 + num

>>> concentric_circles(18, 5, "purple", "gold"
def concentric_circles(radius, gap, color_out, color_in):
    """Recursive function to draw concentric circles"""
    if radius < gap:
        return 0
    else:
        draw_disc(radius, color_out)
        lt(90); fd(gap); rt(90)
        num = concentric_circles(radius-gap, gap, color_in, color_out)
        return 1 + num

>>> concentric_circles(18, 5, "purple", "gold")

contrc_circles(18,5,'p','g')

radius 18  gap 5

if rad < gap:
    return 0
else:
    draw_disc(rad, clr_o)
    lt(90); fd(gap); rt(90)
    num = concentric_circles(rad-g, g, clr_i, clr_o)
    return 1 + num
```python
def concentric_circles(radius, gap, color_out, color_in):
    """Recursive function to draw concentric circles"""
    if radius < gap:
        return 0
    else:
        draw_disc(radius, color_out)
        lt(90); fd(gap); rt(90)
        num = concentric_circles(radius - gap, gap, color_in, color_out)
        return 1 + num
```

```python
>>> concentric_circles(18, 5, "purple", "gold")
13
```
```python
def concentric_circles(radius, gap, color_out, color_in):
    """Recursive function to draw concentric circles"""
    if radius < gap:
        return 0
    else:
        draw_disc(radius, color_out)
        lt(90); fd(gap); rt(90)
        num = concentric_circles(radius - gap, gap, color_in, color_out)
        return 1 + num
```

```python
>>> concentric_circles(18, 5, "purple", "gold")
```

```
constrc_circles(18,5,'p','g')

radius 18  gap 5

if rad < gap:
    return 0
else:
    draw_disc(rad, clr_o)
    lt(90); fd(gap); rt(90)
    num = concentric_circles(rad-g, g, clr_i, clr_o)
    return 1 + num
```

```
constrc_circles(13,5,'g','p')

radius 13  gap 5

if rad < gap:
    return 0
else:
    draw_disc(rad, clr_o)
    lt(90); fd(gap); rt(90)
    num = concentric_circles(rad-g, g, clr_i, clr_o)
    return 1 + num
```

```
constrc_circles(8,5,'p','g')

radius 8  gap 5

if rad < gap:
    return 0
else:
    draw_disc(rad, clr_o)
    lt(90); fd(gap); rt(90)
    num = concentric_circles(rad-g, g, clr_i, clr_o)
    return 1 + num
```
```python
def concentric_circles(radius, gap, color_out, color_in):
    """Recursive function to draw concentric circles""
    if radius < gap:
        return 0
    else:
        draw_disc(radius, color_out)
        lt(90); fd(gap); rt(90)
        num = concentric_circles(radius - gap, gap, color_in, color_out)
        return 1 + num

>>> concentric_circles(18, 5, "purple", "gold")

contrc_circles(18, 5, 'p', 'g')
contrc_circles(13, 5, 'g', 'p')
contrc_circles(8, 5, 'p', 'g')
```

```python
def concentric_circles(radius, gap, color_out, color_in):
    """Recursive function to draw concentric circles""
    if radius < gap:
        return 0
    else:
        draw_disc(rad, clr_o)
        lt(90); fd(gap); rt(90)
        num = concentric_circles(rad - gap, gap, clr_i, clr_o)
        return 1 + num
```
```python
def concentric_circles(radius, gap, color_out, color_in):
    """Recursive function to draw concentric circles"""
    if radius < gap:
        return 0
    else:
        draw_disc(radius, color_out)
        lt(90); fd(gap); rt(90)
        num = concentric_circles(radius-gap, gap, color_in, color_out)
        return 1 + num

>>> concentric_circles(18, 5, "purple", "gold")
```

- **contrc_circles(18, 5, 'p', 'g')**
  - **radius**: 18
  - **gap**: 5
  - If radius < gap:
    - Return 0
  - Else:
    - Draw disc with radius 18, color_out
    - Turn left 90 degrees, forward gap, right 90 degrees
    - Recursively call concentric_circles with radius = radius - gap, gap, color_in, color_out
    - Return 1 + num

- **contrc_circles(13, 5, 'g', 'p')**
  - **radius**: 13
  - **gap**: 5
  - If radius < gap:
    - Return 0
  - Else:
    - Draw disc with radius 13, color_out
    - Turn left 90 degrees, forward gap, right 90 degrees
    - Recursively call concentric_circles with radius = radius - gap, gap, color_in, color_out
    - Return 1 + num

- **contrc_circles(8, 5, 'p', 'g')**
  - **radius**: 8
  - **gap**: 5
  - If radius < gap:
    - Return 0
  - Else:
    - Draw disc with radius 8, color_out
    - Turn left 90 degrees, forward gap, right 90 degrees
    - Recursively call concentric_circles with radius = radius - gap, gap, color_in, color_out
    - Return 1 + num
```python
def concentric_circles(radius, gap, color_out, color_in):
    """Recursive function to draw concentric circles"""
    if radius < gap:
        return 0
    else:
        draw_disc(radius, color_out)
        lt(90); fd(gap); rt(90)
        num = concentric_circles(radius-g, g, clr_i, clr_o)
        return 1 + num

>>> concentric_circles(18, 5, "purple", "gold")
```
def concentric_circles(radius, gap, color_out, color_in):
    """Recursive function to draw concentric circles"""
    if radius < gap:
        return 0
    else:
        draw_disc(radius, color_out)
        lt(90); fd(gap); rt(90)
        num = concentric_circles(radius-g, g, clr_i, clr_o)
    return 1 + num

>>> concentric_circles(18, 5, "purple", "gold")

contrc_circles(18, 5, 'p', 'g')

contrc_circles(13, 5, 'g', 'p')

contrc_circles(8, 5, 'p', 'g')
def concentric_circles(radius, gap, color_out, color_in):
    """Recursive function to draw concentric circles"""
    if radius < gap:
        return 0
    else:
        draw_disc(radius, color_out)
        lt(90); fd(gap); rt(90)
        num = concentric_circles(radius - gap, gap, color_in, color_out)
        return 1 + num

>> concentric_circles(18, 5, "purple", "gold")

```python
>>> concentric_circles(18, 5, "purple", "gold")
```

```
if rad < gap:
    return 0
else:
    draw_disc(rad, clr_o)
    lt(90); fd(gap); rt(90)
    num = concentric_circles(rad - gap, gap, clr_i, clr_o)
    return 1 + num
```
```python
def concentric_circles(radius, gap, color_out, color_in):
    """Recursive function to draw concentric circles"""
    if radius < gap:
        return 0
    else:
        draw_disc(radius, color_out)
        lt(90); fd(gap); rt(90)
        num = concentric_circles(radius - gap, gap, color_in, color_out)
        return 1 + num
```

```bash
>>> concentric_circles(18, 5, "purple", "gold")
```
```python
def concentric_circles(radius, gap, color_out, color_in):
    '''Recursive function to draw concentric circles'''
    if radius < gap:
        return 0
    else:
        draw_disc(radius, color_out)
        lt(90); fd(gap); rt(90)
        num = concentric_circles(radius-gap, gap, color_in, color_out)
        return 1 + num

>>> concentric_circles(18, 5, "purple", "gold")
```
```python
def concentric_circles(radius, gap, color_out, color_in):
    """Recursive function to draw concentric circles"""
    if radius < gap:
        return 0
    else:
        draw_disc(radius, color_out)
        lt(90); fd(gap); rt(90)
        num = concentric_circles(radius - gap, gap, color_in, color_out)
        return 1 + num
```

```python
>>> concentric_circles(18, 5, "purple", "gold")
```

```python
contrc_circles(18, 5, 'p', 'g')
radius 18 gap 5
if rad < gap:
    return 0
else:
    draw_disc(rad, clr_o)
    lt(90); fd(gap); rt(90)
    num = concentric_circles(rad - gap, gap, clr_i, clr_o)
    return 1 + num
```

```python
contrc_circles(13, 5, 'g', 'p')
radius 13 gap 5
if rad < gap:
    return 0
else:
    draw_disc(rad, clr_o)
    lt(90); fd(gap); rt(90)
    num = concentric_circles(rad - gap, gap, clr_i, clr_o)
    return 1 + num
```

```python
contrc_circles(8, 5, 'p', 'g')
radius 8 gap 5
if rad < gap:
    return 0
else:
    draw_disc(rad, clr_o)
    lt(90); fd(gap); rt(90)
    num = concentric_circles(rad - gap, gap, clr_i, clr_o)
    return 1 + num
```

```python
contrc_circles(3, 5, 'g', 'p')
radius 3 gap 5
if rad < gap:
    return 0
else:
    draw_disc(rad, clr_o)
    lt(90); fd(gap); rt(90)
    num = concentric_circles(rad - gap, gap, clr_i, clr_o)
    return 1 + num
```
Invariance of Functions

• Where does the turtle end up in this example with `concentric_circles_color`?

• The turtle does not end where it starts

```python
def concentric_circles_color(radius, gap, color_outer, color_inner):
    """
    Recursive function to draw concentric circles with alternating colors
    """
    # base case, don't draw anything, return 0
    if radius < gap:
        return 0
    else:
        draw_disc(radius, color_outer)
        lt(90)
        fd(gap)
        rt(90)
        num = concentric_circles_color(radius-gap, gap, color_inner, color_outer)
        return 1 + num

concentric_circles(18, 5, 'purple', 'gold')
```
Example: Nested Circles
Invariance of Recursive Functions

• Let's do an example with multiple recursive calls: nested circles
Multiple Recursive Calls

- **Example:** Nested circles function definition

```
nested_circles(radius, min_radius, color_out, color_alt)
  • radius: radius of the outermost circle
  • min_radius: minimum radius of any circle
  • color_out: color of the outermost circle
  • color_alt: color that alternates with colorOut
```
Nested Circles

• **Base case?**
  • When radius becomes less than min\_radius
  • Don’t draw anything return 0

• **Recursive case**
  • Draw the outer circle, add one to total
  • Position turtle for recursive calls
starting position of turtle

\text{nested\_circles}(300, 150)
Nested Circles

- **Base case?**
  - When radius becomes less than minRadius
  - Don’t draw anything return 0

- **Recursive case**
  - Draw the outer circle, add one to total
  - Position turtle for recursive calls
  - How many recursive calls do we need?
    - Two! Right subcircle and left subcircle
Nested Circles

- **Recursive case**
  - Draw the outer circle, add one to total
  - Position turtle for right recursive subcircle

```python
def nested_circles(radius, min_radius, color_out, color_alt):
    if radius < min_radius:
        return 0
    else:
        # contribute to the solution
        draw_disc(radius, color_out)

        # save half of radius
        half_radius = radius/2

        # position the turtle to draw right subcircle
        lt(90); fd(half_radius); rt(90); fd(half_radius)

        # draw right subcircle recursively
        right = nested_circles(half_radius, min_radius, color_alt, color_out)
```
Nested Circles

- **Recursive case**
  - Move the turtle to draw left subcircle recursively
  - (continued from previous slide)

```python
# draw right subcircle recursively
geright = nested_circles(half_radius, min_radius, color_alt, color_out)

# position turtle for left subcircle
bk(radius)

# draw left subcircle recursively
left = nested_circles(half_radius, min_radius, color_alt, color_out)

# add one to our count of subcircles
return 1 + num
```
Nested Circles

• **Recursive case**
  
  • Are we done? Let’s try it!
Nested Circles

- **Recursive case**
  - What happened?!
  - We made assumptions about where the turtle started, that wasn't true!
  - Need turtle to end where it started
  - This is called *position invariance*
Invariance of Functions

• A function is **invariant** if the state of the object is the same *before* and *after* the function is invoked

• Right now our `nested_circles` function is not invariant with respect to the position of the turtle
  
  • That is, the turtle does not end were it starts

• How can we make it invariant by returning the turtle to starting position?

```python
def nested_circles(radius, min_radius, color_out, color_alt):
    if radius < min_radius:
        return 0
    else:
        draw_disc(radius, color_out)
        h_r = radius/2
        lt(90); fd(h_r); rt(90); fd(h_r)
        right = nested_circles(h_r, min_radius, color_alt, color_out)
        bk(radius)
        left = nested_circles(h_r, min_radius, color_alt, color_out)
        fd(h_r); lt(90); bk(h_r); rt(90)
    return 1 + right + left
```
Maintaining Invariance

• Any turtle movements that happen before the recursive call should be “undone” after the recursive call to maintain proper invariance

• **Rule of thumb:** always return turtle to its starting position

```python
def nested_circles(radius, min_radius, color_out, color_alt):
    if radius < min_radius:
        return 0
    else:
        # contribute to the solution
        draw_disc(radius, color_out)

        # save half of radius
        half_radius = radius/2

        # position the turtle to draw right subcircle
        lt(90); fd(half_radius); rt(90); fd(half_radius)

        # draw right subcircle recursively
        right = nested_circles(half_radius, min_radius, color_alt, color_out)

        # position turtle for left subcircle
        bk(radius)

        # draw left subcircle recursively
        left = nested_circles(half_radius, min_radius, color_alt, color_out)

        # bring turtle back to start position
        fd(half_radius); lt(90); bk(half_radius); rt(90)

        # return total number of circles drawn
        return 1 + right + left
```
def nested_circles(radius, min_radius, color_out, color_alt):
    if radius < min_radius:
        return 0
    else:
        # contribute to the solution
        draw_disc(radius, color_out)

        # save half of radius
        half_radius = radius / 2

        # position the turtle to draw right subcircle
        lt(90); fd(half_radius); rt(90); fd(half_radius)

        # draw right subcircle recursively
        right = nested_circles(half_radius, min_radius, color_alt, color_out)

        # position turtle for left subcircle
        bk(radius)

        # draw left subcircle recursively
        left = nested_circles(half_radius, min_radius, color_alt, color_out)

        # bring turtle back to start position
        fd(half_radius); lt(90); bk(half_radius); rt(90)

        # return total number of circles drawn
        return 1 + right + left

Maintaining Invariance

• Move turtle back to starting position to maintain invariance
Invariance of Recursive Functions

• Why do we care about invariance?

  • Though not always necessary for correctness, it is a good property to maintain in recursive functions

  • Our graphical functions will not always work properly if they are not invariant
Recursive Trees
Recursive Trees

- Let's draw some trees using turtle graphics and recursion
One more recursive example: Trees

- Example: Draw recursive trees as shown; count and return # branches drawn
- What is our base case? Recursive case?
- Note: Assume turtle starts facing north
def tree(trunk_len, angle, shrink_factor, min_len):
    # trunk_len length of the main (vertical) trunk
    # angle branching angle (angle between a trunk and its
    # right or left branch)
    # shrink_factor factor by which each subsequent branch
    shrinks by
    # min_len minimum branch length in our tree
```python
def tree(trunk_len, angle, shrink_factor, min_len):
    '''Draw tree and return number of branches drawn including trunk'''
    # Base case: trunk_len < min_len
    # return 0, don't draw anything!
    # Recursive case:
    # Draw trunk

    # Position for Right branch: Turn right angle
    # Right branch → shrink trunk, pass along other variables

    # Position for Left branch: Turn left angle*2
    # Left branch → shrink trunk, pass along other variables

    # Maintain invariance
    # Turn right, then back up to starting position

    # return 1 (for the trunk we drew), plus the sum of the branches
```
def tree(trunk_len, angle, shrink_factor, min_len):
    '''Draw tree and return number of branches drawn including trunk'''
    if (trunk_len < min_len):  # Base case
        return 0
    else:
        # Draw trunk
        fd(trunk_len)

        # Right branch
        rt(angle)
        right_branch = tree(trunk_len*shrink_factor, angle, shrink_factor, min_len)

        # Left branch
        lt(angle*2)
        left_branch = tree(trunk_len*shrink_factor, angle, shrink_factor, min_len)

        # Maintain invariance
        rt(angle); bk(trunk_len)

        return 1 + right_branch + left_branch
Recursion: Wrap Up
What’s The Big Deal With Recursion?

• Why choose recursion over iteration?
  • Some problems have a *natural recursive structure*
  • Using recursion on them leads to elegant and concise solutions
  • Fewer lines of code often correlates with less debugging!
• We will use recursion to search and sort in a few weeks
• Recursion also helps us build and maintain complex data structures
• Downsides: Recursive approaches often have more computational overhead
  • Steeper learning curve (but can be very rewarding once you get the hang of it)
  • To understand recursion you must understand recursion…