CS 134:
Graphical Recursion

Announcements & Logistics

- **Lab 7** has been posted: focuses on recursion
  - Please complete Task 0 before you come to lab!
- **HW 6** due Monday @ 11 pm: covers sorting, dictionaries, sets, tuples
- **Scheduled final:** Sun, May 22, 9:30 am, details TBD

- **CS TA applications** due Apr 22nd

Do You Have Any Questions?

Last Time

- Discussed more examples with recursion & recursive approach to problem solving and compared it with iterative approaches
- Recursion helps us better appreciate how to break down a problem into smaller pieces — **decomposition** or **divide and conquer** — which is a key concept in computer science. We see more of it in **CS 136**, **256**
- Finally, even if you never write a recursive program, others will! Can you find the base case and recursive step?

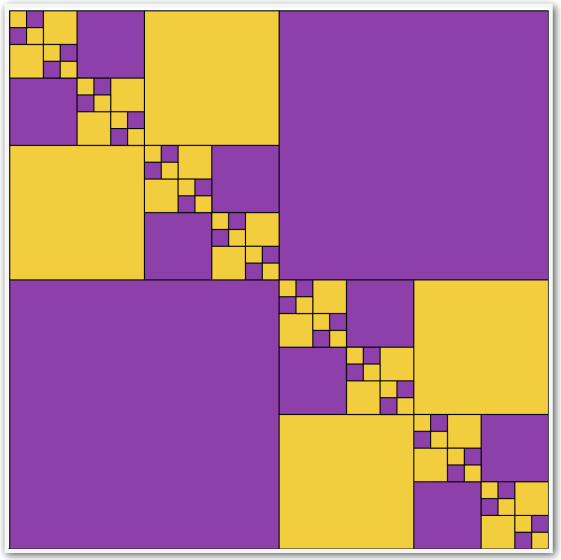
Algorithm 1 CAPA

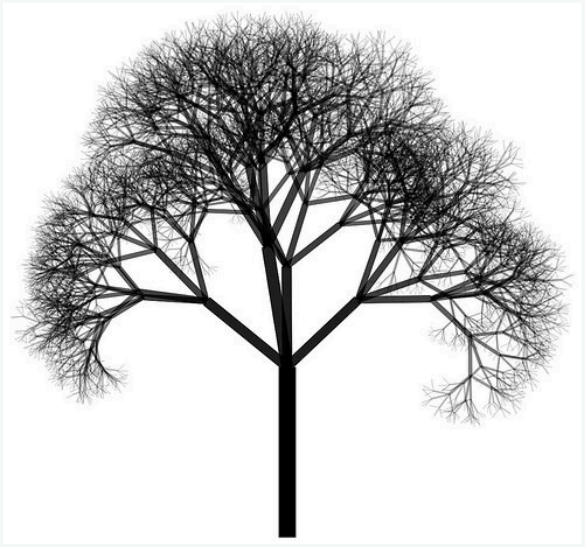
Input: The original variable set V, algorithm A_g.
Output: The causal graph G.
Find a causal partitioning $\{V_1, V_2, V_3\}$ on V.
if $\max\{|V_1|, |V_2|, |V_3|\} = |V|$ **then**
    Return G by running algorithm A_g on V.
else
    G_1=**CAPA**(V_1, A_g, δ),
    G_2=**CAPA**(V_2, A_g, δ),
    G_3=**CAPA**(V_3, A_g, δ).
    Return G by merging G_1, G_2 and G_3.
end if

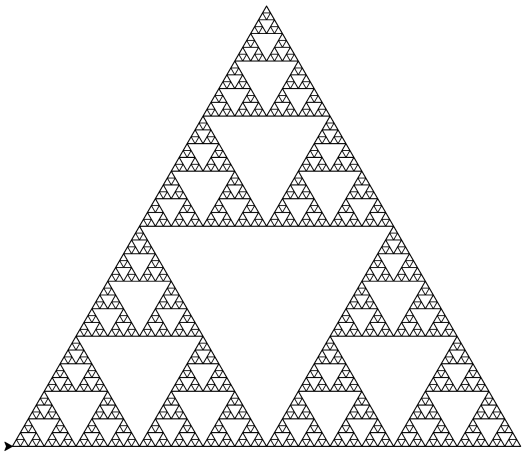
- Recursively Learning Causal Structures Using Regression-Based Conditional Independence Test by Zhang et al (2019)

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:

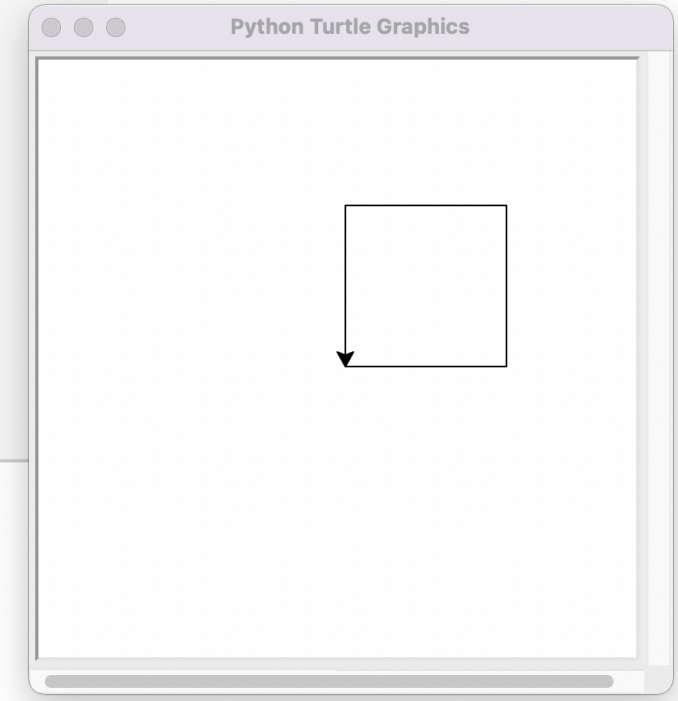
Use `from turtle import *` to use these commands:

`fd(dist)`	turtle moves forward by *dist*
`bk(dist)`	turtle moves backward by *dist*
`lt(angle)`	turtle turns left *angle* degrees
`rt(angle)`	turtle turns right *angle* degrees
`up()`	(pen up) turtle raises pen in belly
`down()`	(pen down) turtle lower pen in belly
`pensize(width)`	sets the thickness of turtle's pen to *width*
`pencolor(color)`	sets the color of turtle's pen to *color*
`shape(shp)`	sets the turtle's shape to *shp*
`home()`	turtle returns to (0,0) (center of screen)
`clear()`	delete turtle drawings; no change to turtle's state
`reset()`	delete turtle drawings; reset turtle's state
`setup(width,height)`	create a turtle window of given *width* and *height*

Basic Turtle Movement

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

```
# set up a 500x500 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)
fd(100)
```



Drawing Basic Shapes With Turtle

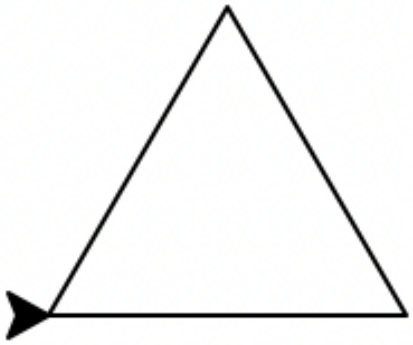
- We can write functions that use turtle commands to draw shapes.
- For example, here's a function that draws a square of the desired size

```
def drawSquare(length):

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

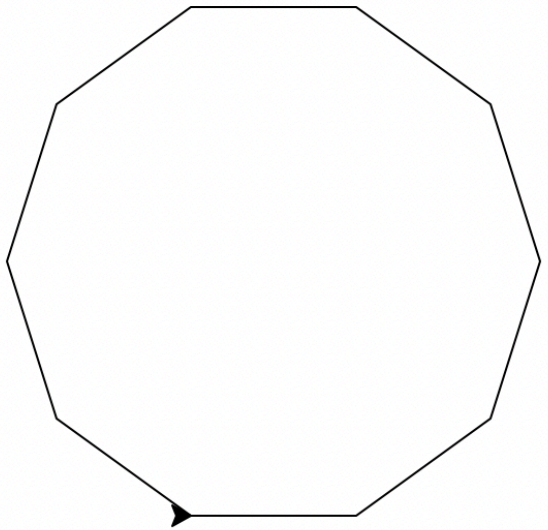
Drawing Basic Shapes With Turtle

- How about drawing polygons?

```
def drawPolygon(length, numSides):

    for i in range(numSides):
        fd(length)
        lt(360/numSides)
```



```
drawPolygon(80, 3)
```

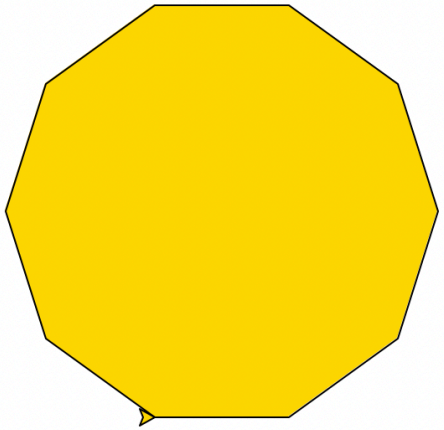


```
drawPolygon(80, 10)
```

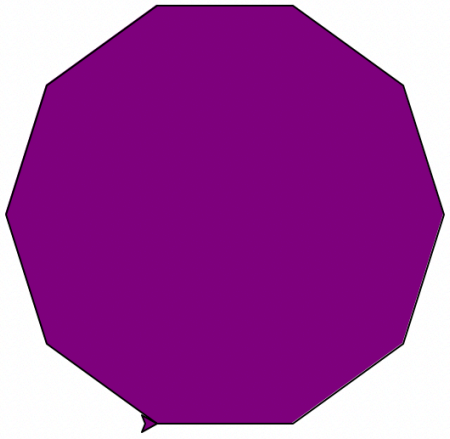
Adding Color!

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

```
def drawPolygonColor(length, numSides, color):

    # set the color we want to fill the shape with
    fillcolor(color)

    begin_fill()
    for i in range(numSides):
        fd(length)
        lt(360/numSides)
    end_fill()
```



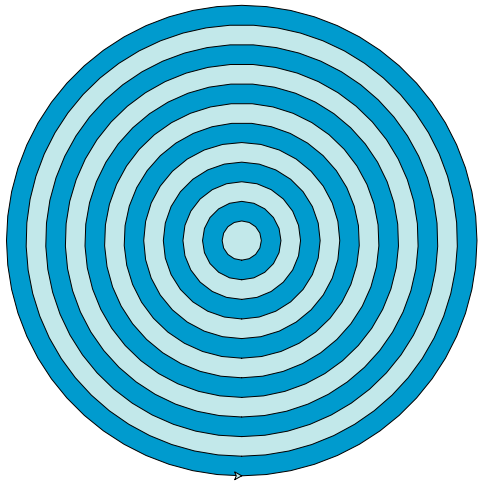
```
drawPolygonColor(80, 10, "gold")
```



```
drawPolygonColor(80, 10, "purple")
```

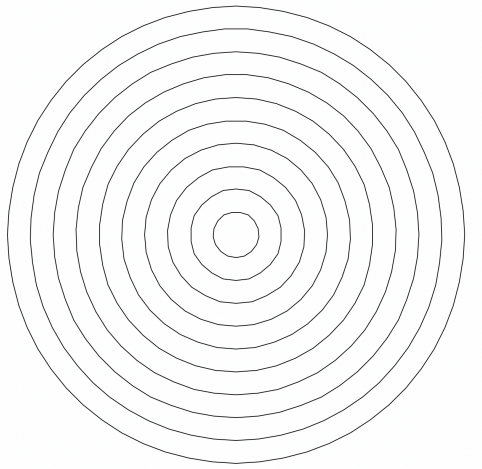
Recursive Figures With Turtle

- 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



Concentric Circles With No Colors

- Recursive idea: we have circles within circles, and each circle becomes successively smaller. Let's first discuss the circles with no coloring involved
- Base case: radius of the circle is so small it's not worth drawing
- Recursive step:
  - Draw a single circle of radius `r`,
  - Draw concentric circles starting with an outer circle of a slightly smaller radius `r-c` (where `c` is any positive number you want to shrink the radius by)

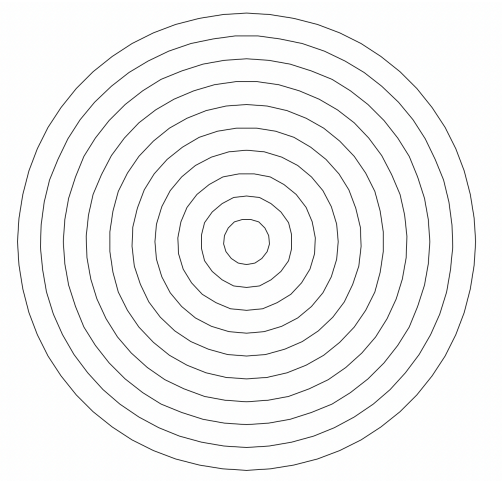


Concentric Circles

- Function definition of the recursive function.

```
concentricCircles(radius, thickness)
```

  - `radius`: radius of the outermost circle
  - `thickness`: thickness of the band between circles



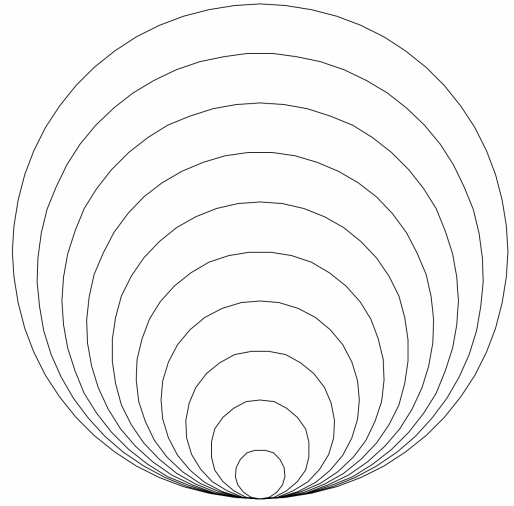
Concentric Circle

```
def concentricCircles(radius, thickness):

    # base case do nothing
    if radius < thickness:
        pass
    else:
        # tell the turtle to start drawing and draw a circle
        circle(radius)

        # recursive function call
        concentricCircles(radius-thickness, thickness)
```

- Are we done?

Concentric Circles

```
concentricCircles(300, 30)
```



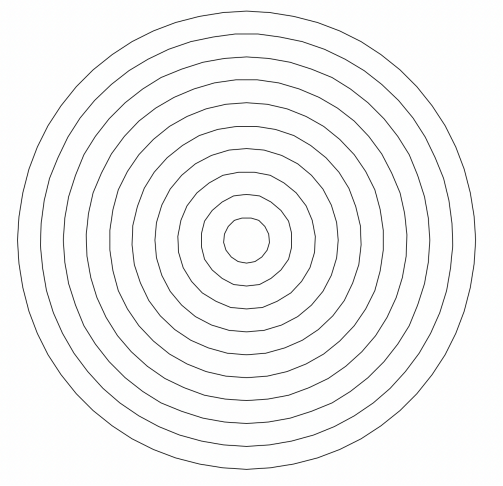
- Pretty picture, and almost there! But we also need to reposition the turtle slightly with each recursive call.

Concentric Circle

```
def concentricCircles(radius, thickness):

    # base case do nothing
    if radius < thickness:
        pass
    else:
        # tell the turtle to start drawing and draw a circle
        down()
        circle(radius)

        # reposition the turtle for the next circle
        up() # ensure the turtle doesn't draw while repositioning
        lt(90)
        fd(thickness)
        rt(90)

        # recursive function call
        concentricCircles(radius-thickness, thickness)
```

Concentric Circles

```
concentricCircles(300, 30)
```



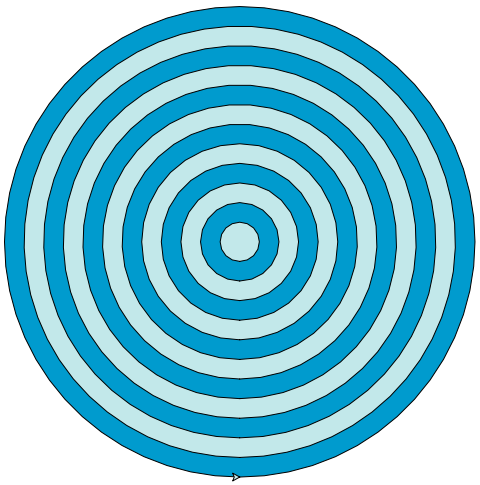
- Great! Now let's add some color.

Concentric Circles With Colors

- Function definition of the recursive function.

```
concentricCircles(radius, thickness, colorOuter, colorInner)
```

- `radius`: radius of the outermost circle
- `thickness`: thickness of the band between circles
- `colorOuter`: color of the outermost circle
- `colorInner`: color that alternates with colorOuter

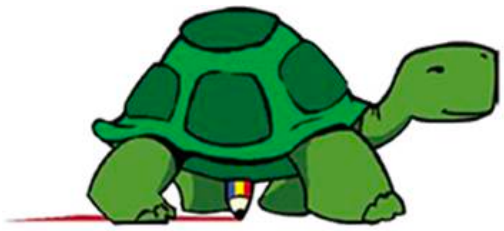


Concentric Circles: Adding Color

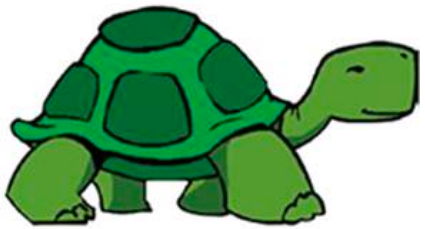
- Pretty much everything about the base case and recursive step remains the same. Except now on each recursive call we just swap the color parameters!
  - colorOuter becomes colorInner and vice versa
- We'll also write a helper function to draw a circle filled in with some color to clean up the recursive function itself

Helper Function

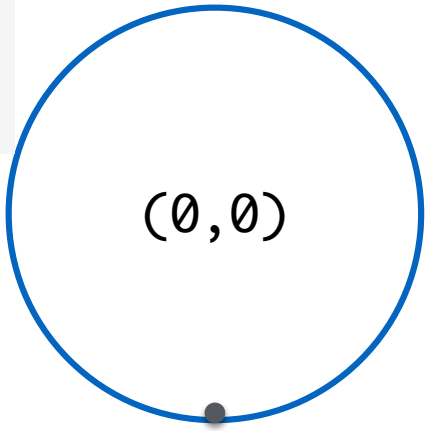
```
def drawDisc(radius, color):
    """
    Draw circle of a given radius
    and fill it with a given color
    """

    # put the pen down
    down()

    # set the color
    fillcolor(color)

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

    # put the pen up
    up()
```



Turtle.PenDown()



Turtle.PenUp()

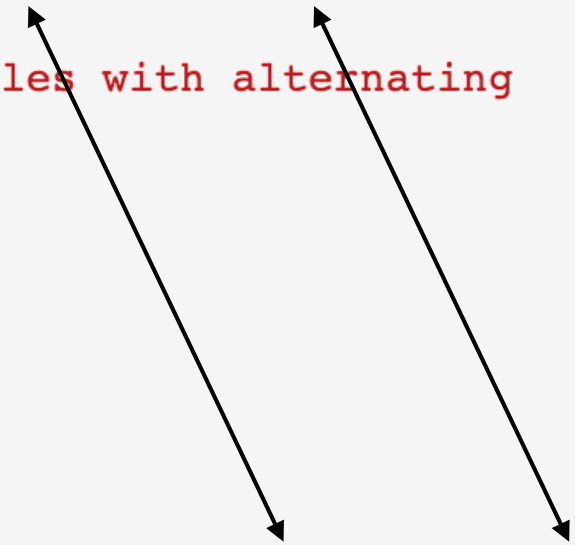


Starting position of turtle

(0, -radius)

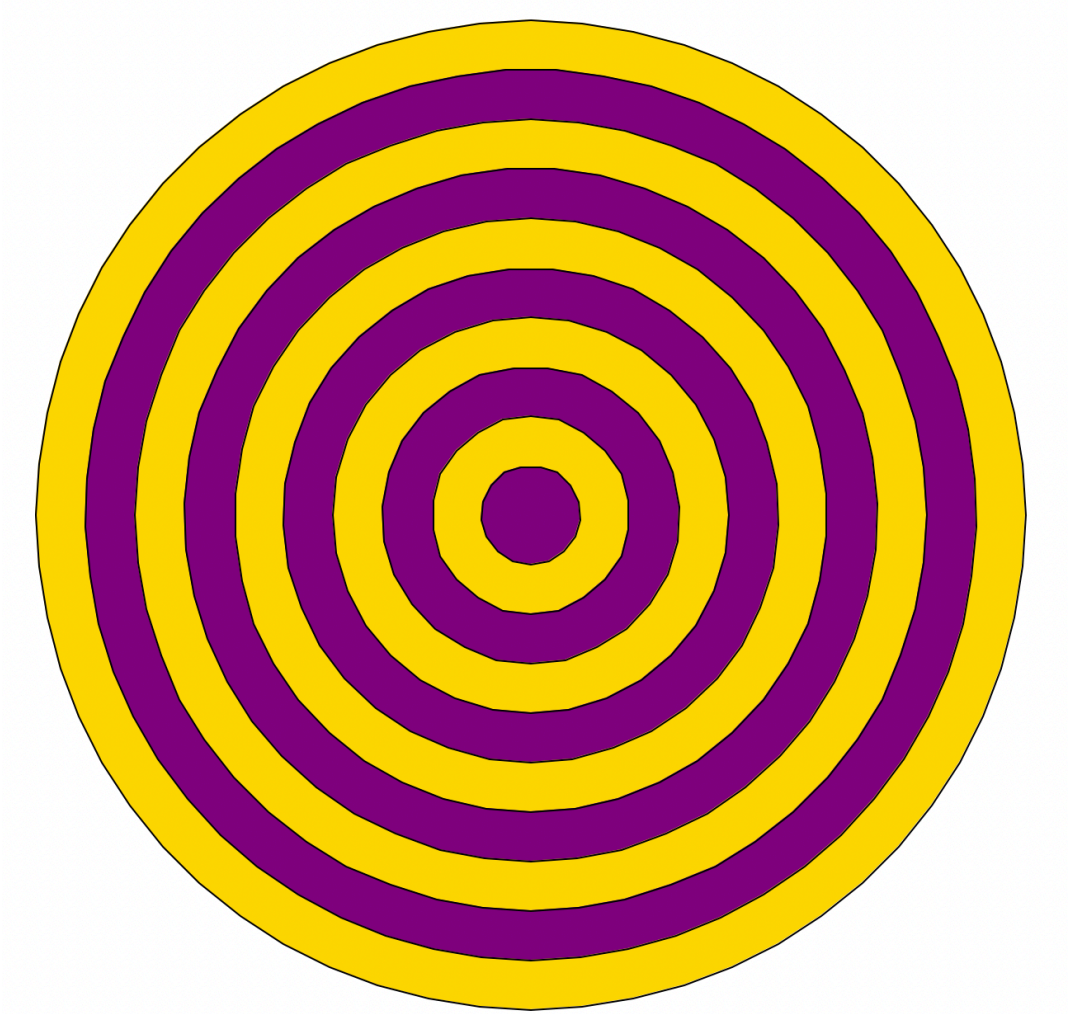
The Recursive Function

```
def concentricCircles(radius, thickness, colorOuter, colorInner):
    """
    Recursive function to draw concentric circles with alternating
    color
    """
    if radius < thickness:
        pass
    else:
        drawDisc(radius, colorOuter)
        lt(90)
        fd(thickness)
        rt(90)
        concentricCircles(radius-thickness, thickness, colorInner, colorOuter)
```



Concentric Circles

```
concentricCircles(300, 30, "gold", "purple")
```



Invariance of Functions

- A function is **invariant** relative to an object's state if the state of the object is the same *before* and *after* a function is invoked
- Right now our `concentricCircles` function is not invariant with respect to the position of the turtle (the turtle does not end were it starts)
- How can we make it invariant, that is, return the turtle to starting position?

```
def concentricCircles(radius, thickness, colorOuter, colorInner):
    """
    Recursive function to draw concentric circles with alternating
    color
    """
    if radius < thickness:
        pass
    else:
        drawDisc(radius, colorOuter)
        lt(90)
        fd(thickness)
        rt(90)
        concentricCircles(radius-thickness, thickness, colorInner, colorOuter)
```

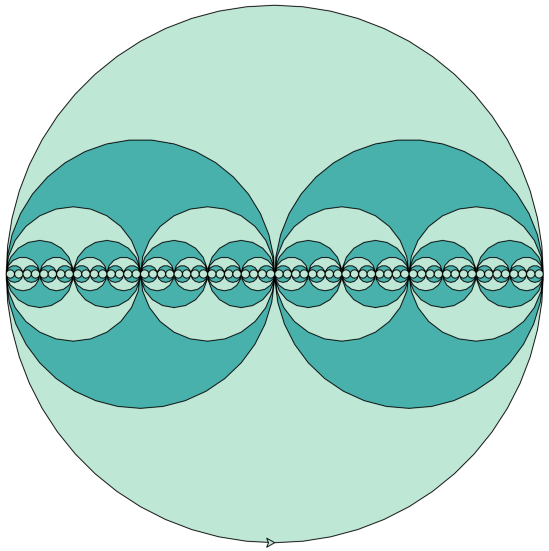
Invariant Concentric Circles

- Ensuring that we "undo" turtle movements that happened before the recursive call, after the recursive call, results in invariance
- Rule of thumb: always return turtle to starting position

```
def concentricCircles(radius, thickness, colorOuter, colorInner):
    """
    Recursive function to draw concentric circles with alternating
    color
    """
    if radius < thickness:
        pass
    else:
        drawDisc(radius, colorOuter)
        lt(90)
        fd(thickness)
        rt(90)
        concentricCircles(radius-thickness, thickness, colorInner, colorOuter)
        # move turtle back to starting position
        lt(90)
        bk(thickness)
        rt(90)
```

Invariance of Recursive Functions

- Why do we care about **invariance**?
  - It is a good property to have for recursive functions
  - Is not crucial for correctness when we have a single recursive call
  - However, with multiple recursive calls, our graphical functions will not work properly if it they are not invariant
- Let's do an example with multiple recursive calls
  - Nested circles (see picture)

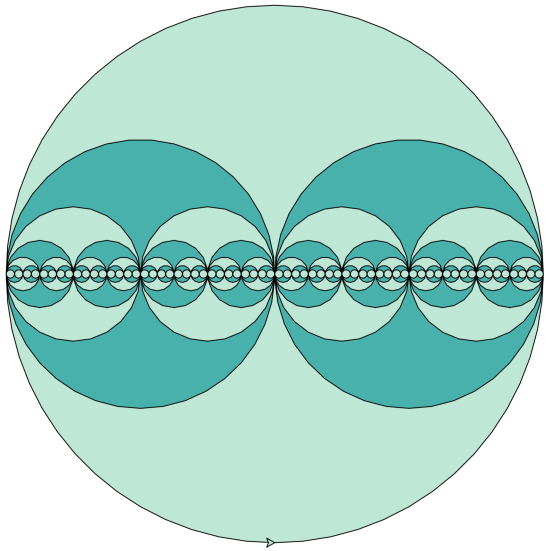


Multiple Recursive Call

- **Example:** Nested circles. Write the following recursive function:

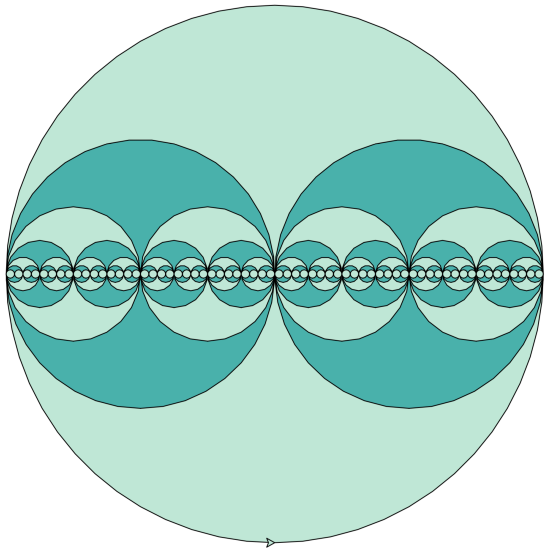
```
nestedCircles(radius, minRadius, colorOut, colorAlt)
```

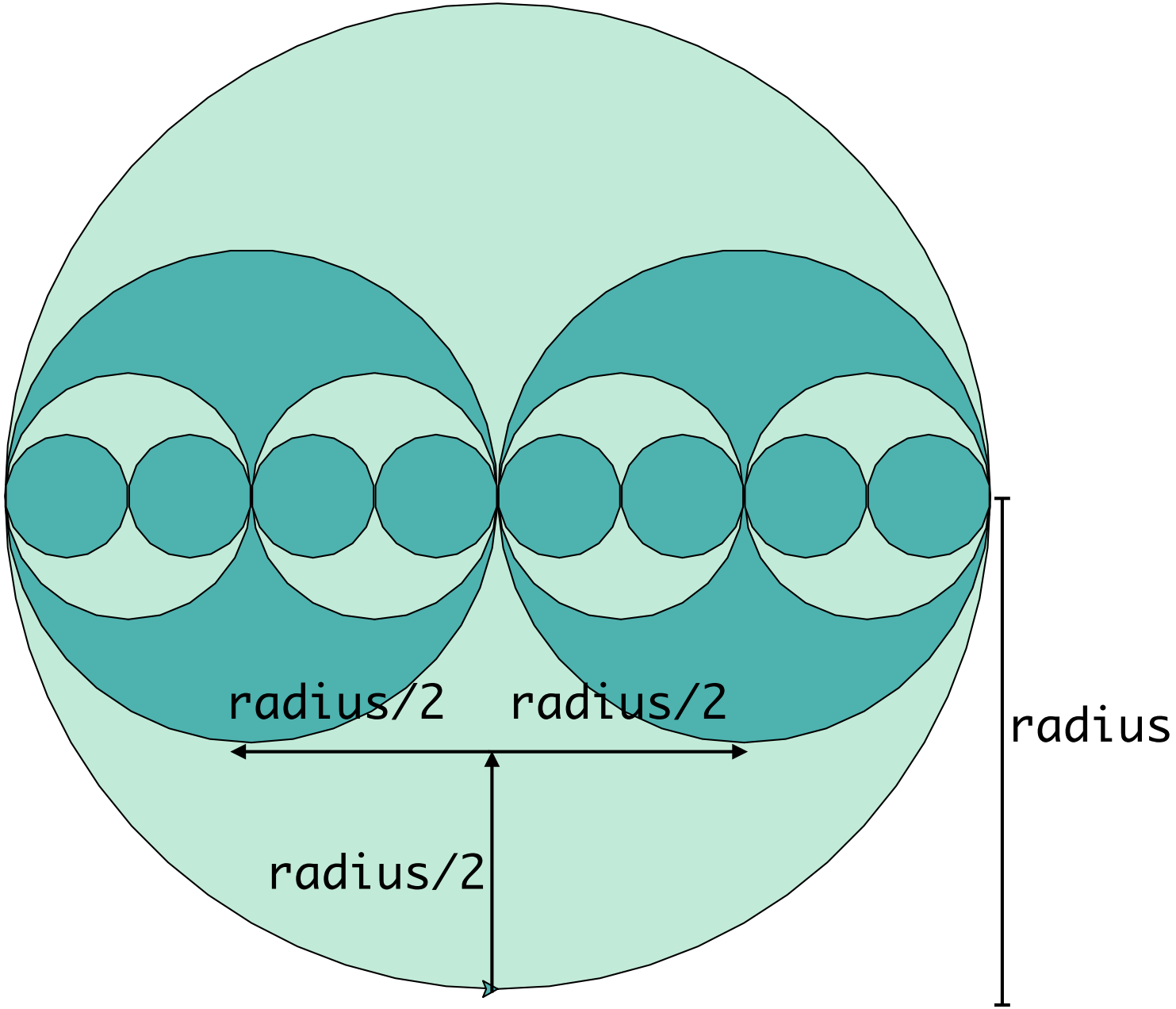
  - `radius`: radius of the outermost circle
  - `minRadius`: minimum radius of any circle
  - `colorOut`: color of the outermost circle
  - `colorAlt`: color that alternates with `colorOut`



Nested Circles

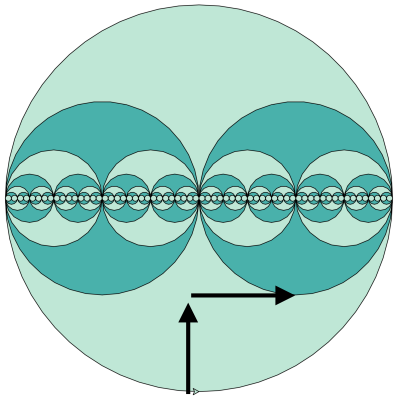
- **Base case?**
  - When radius becomes less than minRadius
- **Recursive case**
  - Draw the outer circle
  - Position turtle for recursive calls
  - How many recursive calls?






```
nestedCircles(300, 37.5)
```

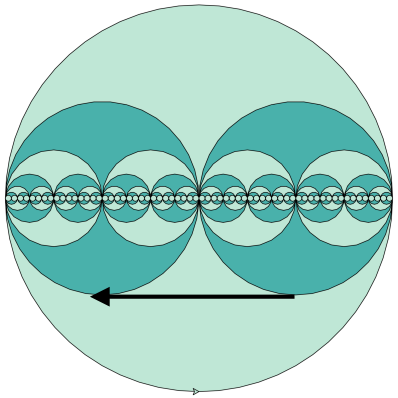
Nested Circles



- **Recursive case**
  - Draw the outer circle
  - Position turtle for right recursive subcircle

```
def nestedCircles(radius, minRadius, colorOut, colorAlt):
    if radius < minRadius:
        pass # do nothing
    else:
        # contribute to the solution
        drawDisc(radius, colorOut)

        # save half of radius
        halfRadius = radius/2

        # position the turtle at the right place
        lt(90); fd(halfRadius); rt(90); fd(halfRadius)

        # draw right subcircle recursively
        nestedCircles(halfRadius, minRadius, colorAlt, colorOut)
```

Nested Circles



- **Recursive case**
  - Move the turtle to draw left subcircle recursively

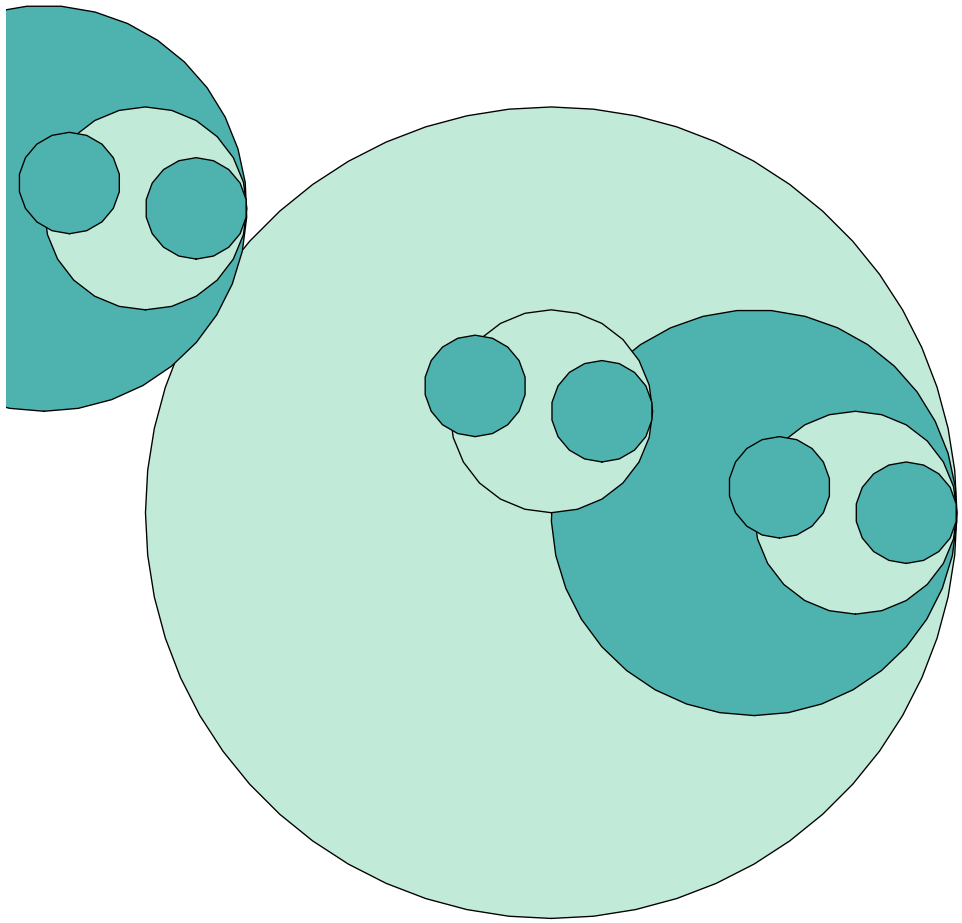
```
def nestedCircles(radius, minRadius, colorOut, colorAlt):
    if radius < minRadius:
        pass # do nothing
    else:
        # contribute to the solution
        drawDisc(radius, colorOut)

        # save half of radius
        halfRadius = radius/2

        # position the turtle at the right place
        lt(90); fd(halfRadius); rt(90); fd(halfRadius)

        # draw right subcircle recursively
        nestedCircles(halfRadius, minRadius, colorAlt, colorOut)

        # position turtle for left subcircle
        bk(radius)

        # draw left subcircle recursively
        nestedCircles(halfRadius, minRadius, colorAlt, colorOut)
```

Nested Circles

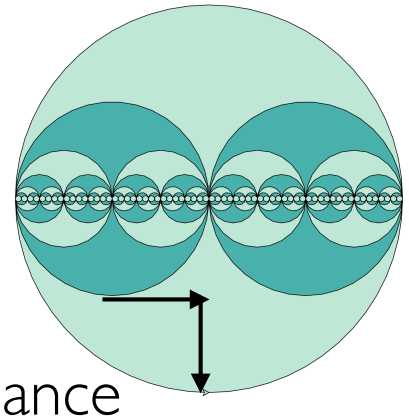
- **Recursive case**
  - Are we done? Let's try it!

Nested Circles

- **Recursive case**
  - Invariance matters! We **must** return the turtle to its starting state to make sure subsequent recursive calls behave correctly

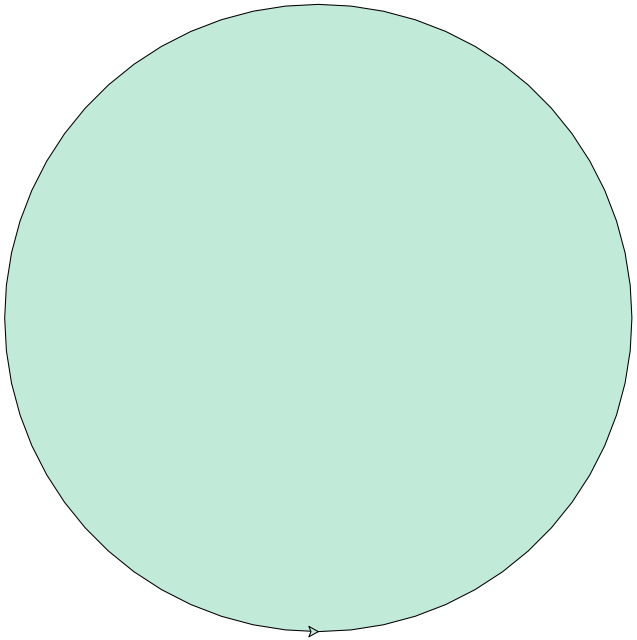


Nested Circles

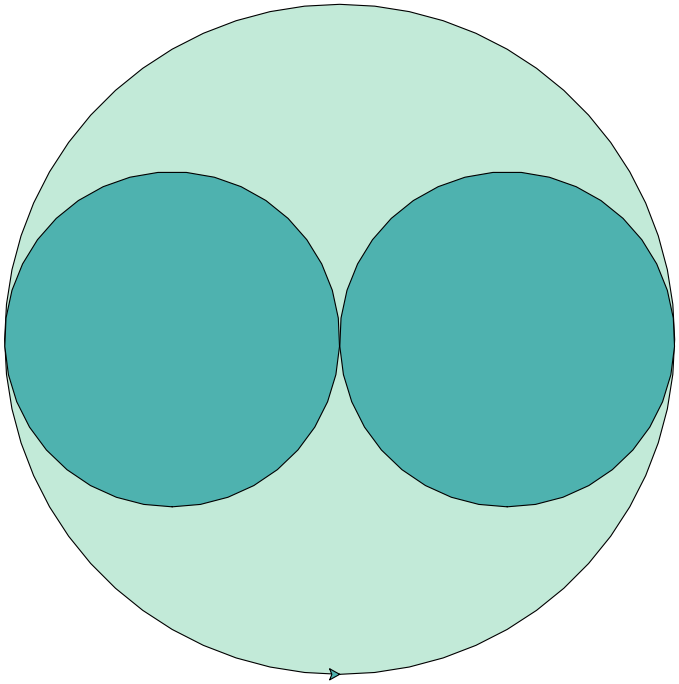


- **Recursive case**
  - Move turtle back to starting position to maintain invariance

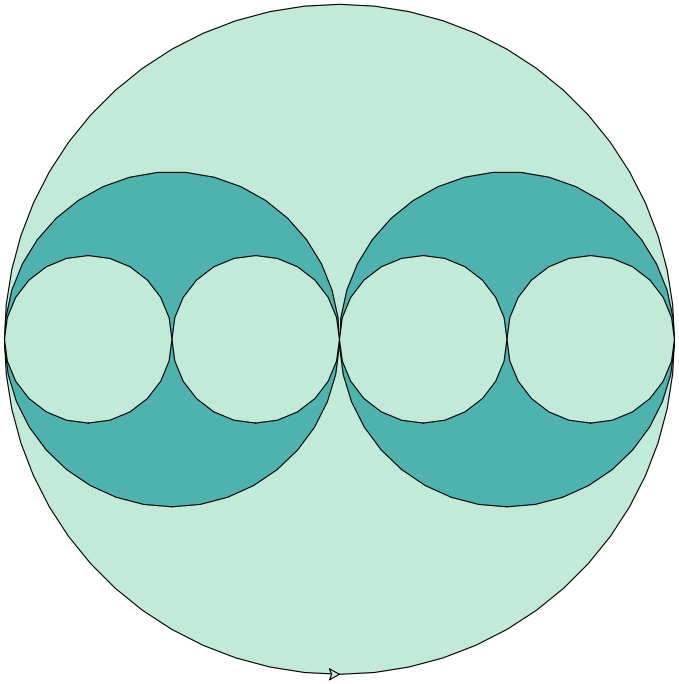
```
def nestedCircles(radius, minRadius, colorOut, colorAlt):
    if radius < minRadius:
        pass # do nothing
    else:
        # contribute to the solution
        drawDisc(radius, colorOut)

        # save half of radius
        halfRadius = radius/2

        # position the turtle at the right place
        lt(90); fd(halfRadius); rt(90); fd(halfRadius)

        # draw right subcircle recursively
        nestedCircles(halfRadius, minRadius, colorAlt, colorOut)

        # position turtle for left subcircle
        bk(radius)

        # draw left subcircle recursively
        nestedCircles(halfRadius, minRadius, colorAlt, colorOut)

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



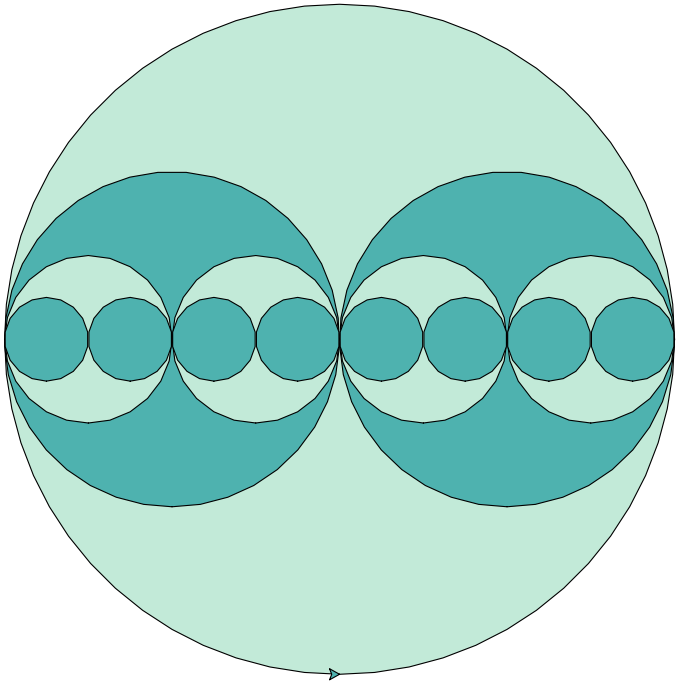
nestedCircles(300, 300)



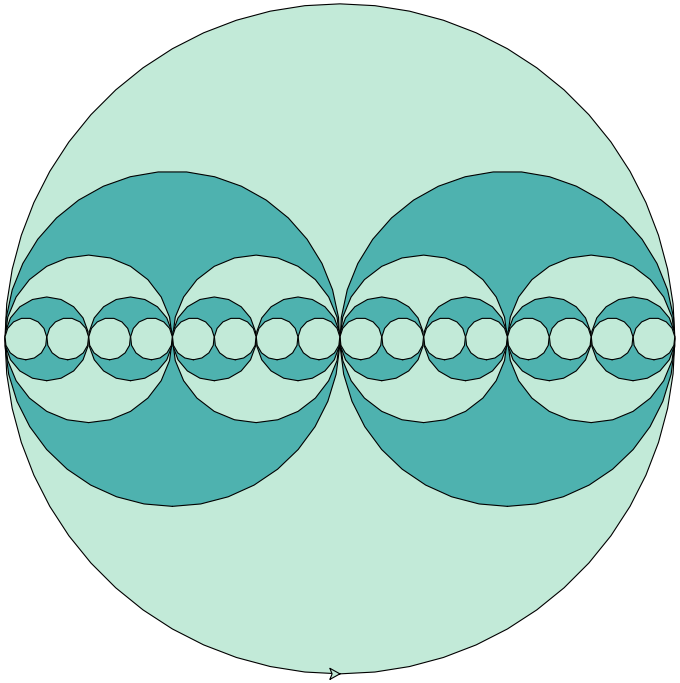
nestedCircles(300, 150)

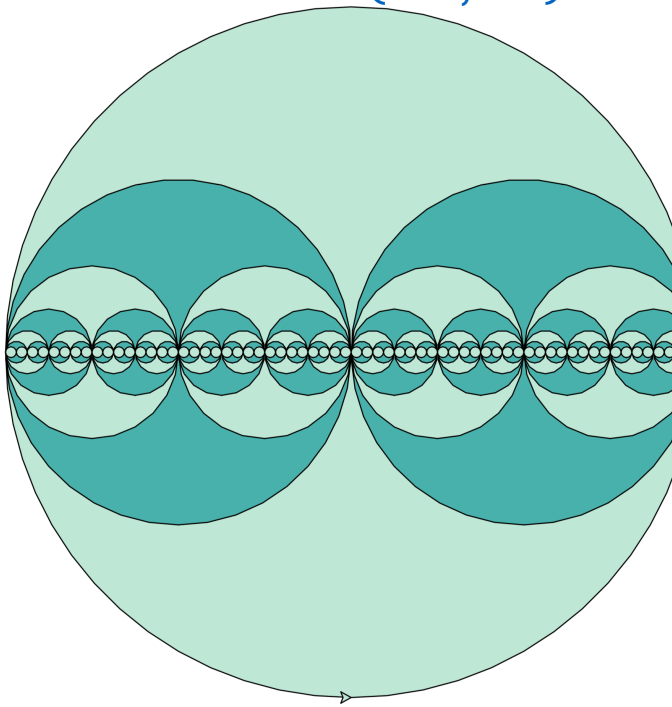


nestedCircles(300, 75)



nestedCircles(300, 75)





Next Time

- Next time: We'll wrap up recursion with a few more examples!