# CS134:
# Recursion Wrap-up

# Announcements & Logistics

- **Lab 7 is today/tomorrow,** due Wed/Thur at 10pm
  - Reminder: Attendance in lab is required
  - You're putting yourself at a disadvantage if you don't attend
  - Next few labs are conceptually trickier than earlier labs; make the most of your time in your lab session
- **HW 6** due tonight at 10pm
- **HW 7** will be posted on Wed, due Mon at 10pm
- **Midterms** are coming back at the end of class
  - Avg: 85%  (this is a little higher than usual)
  - Please come talk to us during student help hours if you got < 70% or if you have questions

# An Aside:
# Preregistration Info

# Next Steps

- Logical next steps if want to explore more CS is to take CS136
- Taught in Java
- Super useful and fun class
- Can also take Math 200

## CS136 :: Data Structures & Advanced Prog

This course builds on the programming skills acquired in Computer Science 134. It couples work on program design, analysis, and verification with an introduction to the study of data structures. Data structures capture common ways in which to store and manipulate data, and they are important in the construction of sophisticated computer programs. Students are introduced to some of the most important and frequently used data structures: lists, stacks, queues, trees, hash tables, graphs, and files. Students will be expected to write several programs, ranging from very short programs to more elaborate systems. Emphasis will be placed on the development of clear, modular programs that are easy to read, debug, verify, analyze, and modify.

# After CS136

- After CS136 you should take 237 and 256

## CS237 :: Computer Organization

This course studies the basic instruction set architecture and organization of a modern computer. It provides a programmer's view of how computer systems execute programs, store information, and communicate. Over the semester the student learns the fundamentals of translating higher level languages into assembly language, and the interpretation of machine languages by hardware. At the same time, a model of computer hardware organization is developed from the gate level upward.

## CS256 :: Algorithm Design & Analysis

This course investigates methods for designing efficient and reliable algorithms. By carefully analyzing the structure of a problem within a mathematical framework, it is often possible to dramatically decrease the computational resources needed to find a solution. In addition, analysis provides a method for verifying the correctness of an algorithm and accurately estimating its running time and space requirements. We will study several algorithm design strategies that build on data structures and programming techniques introduced in Computer Science 136. These include induction, divide-and-conquer, dynamic programming, and greedy algorithms. Additional topics of study include algorithms on graphs and strategies for handling potentially intractable problems.

# Declaring the Major

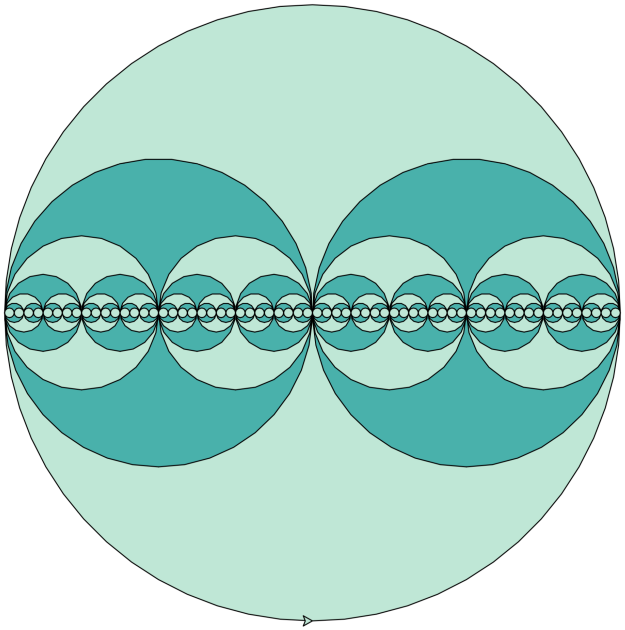
- Complete **two CS courses** by end of 2nd year
- Also **satisfy the Discrete Math requirement** by end of 2nd year
    - Take (and pass) Math 200 for credit, or
    - Pass the Discrete Math Proficiency Exam
- Only the first course in the major may be taken Pass/Fail
- Courses with grades below C- may not be used toward the major (at time of declaration)

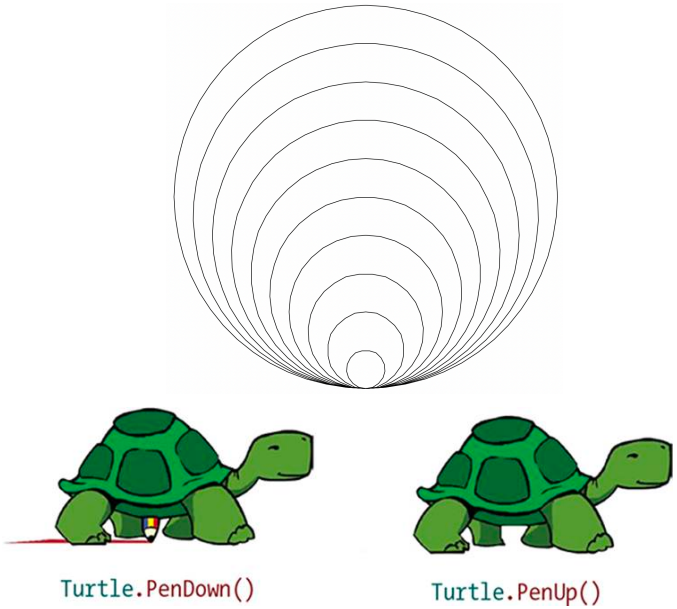
# Completing the Major

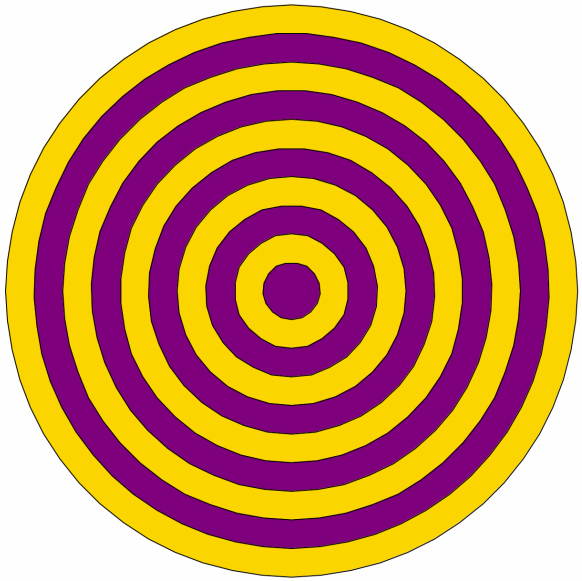
- **Intro**: CS134, CS136
- **Core**: CS237, CS256, CS334, CS361
- **Electives**: two (or more) CS electives numbered 300+
- **Math**: Discrete math proficiency, one (or more) Math/Stat course labeled 200+
- **Colloquium**: attendance at 20 (or more) colloquia. Start now!
- No double counting! If you are majoring in Stat or Math, you cannot count any classes towards both majors
- In total:
    - 8 computer science courses (core + eligible electives)
    - ~2 math courses (200/DPME + 200+)

# Last Time

- Learned about the Turtle graphics package in Python
- Looked at graphical recursion examples (and drew some pretty pictures)



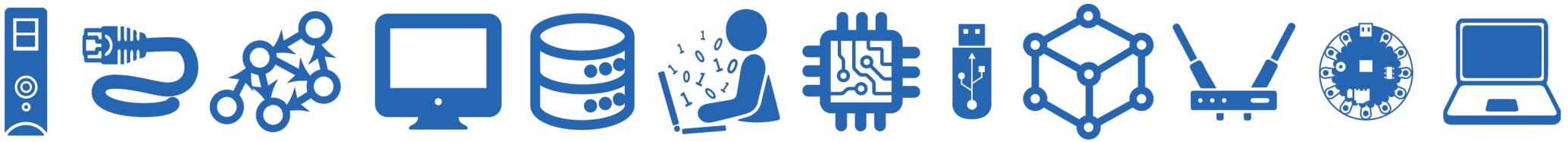


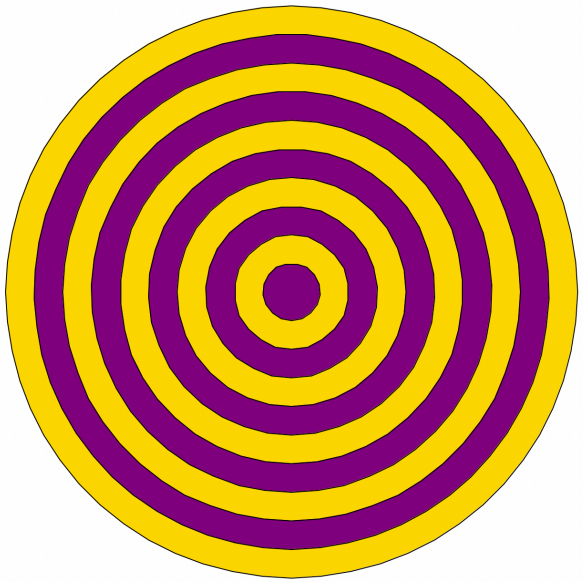
# Today's Plan

- Look at and review a few more graphical examples
- Compare iterative vs. recursive ideas and discuss trade offs

# Function Frame Model:
# `concentricCircles`



```
def concentricCircles(radius, gap, colorOuter, colorInner):
    """Recursive function to draw concentric circles"""
 if radius < gap:
  return 0
 else:
  drawDisc(radius, colorOuter)
  lt(90); fd(gap); rt(90)
  num = concentricCircles(radius-gap, gap, colorInner, colorOuter)
  lt(90); bk(gap); rt(90)
  return 1 + num
```



```
def concentricCircles(radius, gap, colorOuter, colorInner):
    """Recursive function to draw concentric circles"""
  if radius < gap:
    return 0
  else:
    drawDisc(radius, colorOuter)
    lt(90); fd(gap); rt(90)
    num = concentricCircles(radius-gap, gap, colorInner, colorOuter)
    lt(90); bk(gap); rt(90)
    return 1 + num
```

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

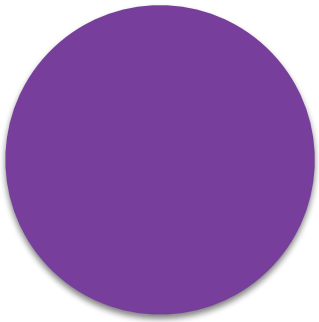
```
def concentricCircles(radius, gap, colorOuter, colorInner):
    """Recursive function to draw concentric circles"""
  if radius < gap:
    return 0
  else:
    drawDisc(radius, colorOuter)
    lt(90); fd(gap); rt(90)
    num = concentricCircles(radius-gap, gap, colorInner, colorOuter)
    lt(90); bk(gap); rt(90)
    return 1 + num
```

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

**concentricCircles(18, 5, …)**

radius 18 gap 5

```
if radius < gap:
  return 0
else:
  drawDisc(radius, colorOuter)
  lt(90); fd(gap); rt(90)
  num = concentricCircles(…)
  lt(90); bk(gap); rt(90)
  return 1 + num
```



```
def concentricCircles(radius, gap, colorOuter, colorInner):
    """Recursive function to draw concentric circles"""
  if radius < gap:
    return 0
  else:
    drawDisc(radius, colorOuter)
    lt(90); fd(gap); rt(90)
    num = concentricCircles(radius-gap, gap, colorInner, colorOuter)
    lt(90); bk(gap); rt(90)
    return 1 + num
```

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

**concentricCircles(18, 5, …)**

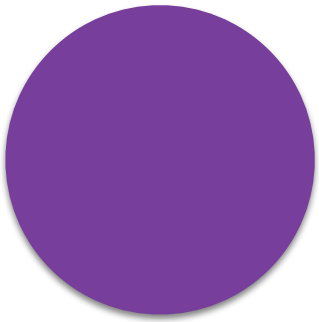
radius 18 gap 5

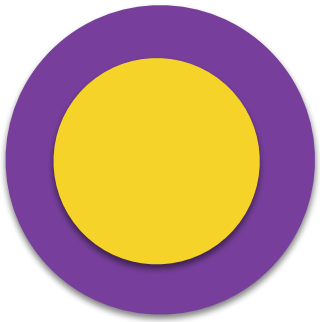
```
if radius < gap:
  return 0
else:
  drawDisc(radius, colorOuter)
  lt(90); fd(gap); rt(90)
  num = concentricCircles(…)
  lt(90); bk(gap); rt(90)
  return 1 + num
```

**concentricCircles(13, 5, …)**

radius 13 gap 5

```
if radius < gap:
  return 0
else:
  drawDisc(radius, colorOuter)
  lt(90); fd(gap); rt(90)
  num = concentricCircles(…)
  lt(90); bk(gap); rt(90)
  return 1 + num
```





```
def concentricCircles(radius, gap, colorOuter, colorInner):
    """Recursive function to draw concentric circles"""
  if radius < gap:
    return 0
  else:
    drawDisc(radius, colorOuter)
    lt(90); fd(gap); rt(90)
    num = concentricCircles(radius-gap, gap, colorInner, colorOuter)
    lt(90); bk(gap); rt(90)
    return 1 + num
```

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

**concentricCircles(18, 5, …)**

radius 18 gap 5

```
if radius < gap:
  return 0
else:
  drawDisc(radius, colorOuter)
  lt(90); fd(gap); rt(90)
  num = concentricCircles(…)
  lt(90); bk(gap); rt(90)
  return 1 + num
```

**concentricCircles(13, 5, …)**

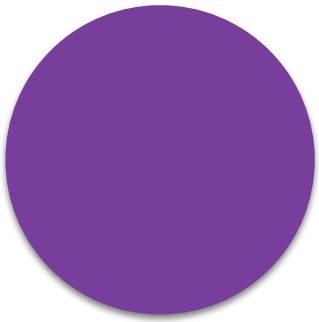
radius 13 gap 5

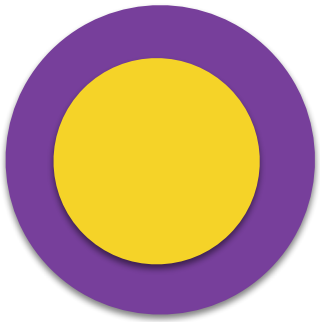
```
if radius < gap:
  return 0
else:
  drawDisc(radius, colorOuter)
  lt(90); fd(gap); rt(90)
  num = concentricCircles(…)
  lt(90); bk(gap); rt(90)
  return 1 + num
```

**concentricCircles(8, 5, …)**

radius 8 gap 5

```
if radius < gap:
  return 0
else:
  drawDisc(radius, colorOuter)
  lt(90); fd(gap); rt(90)
  num = concentricCircles(…)
  lt(90); bk(gap); rt(90)
  return 1 + num
```





```
def concentricCircles(radius, gap, colorOuter, colorInner):
    """Recursive function to draw concentric circles"""
  if radius < gap:
    return 0
  else:
    drawDisc(radius, colorOuter)
    lt(90); fd(gap); rt(90)
    num = concentricCircles(radius-gap, gap, colorInner, colorOuter)
    lt(90); bk(gap); rt(90)
    return 1 + num
```

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

**concentricCircles(3, 5, …)**

radius 3 gap 5

```
if radius < gap:
  return 0
else:
  drawDisc(radius, colorOuter)
  lt(90); fd(gap); rt(90)
  num = concentricCircles(…)
  lt(90); bk(gap); rt(90)
  return 1 + num
```

**concentricCircles(18, 5, …)**

radius 18 gap 5

```
if radius < gap:
  return 0
else:
  drawDisc(radius, colorOuter)
  lt(90); fd(gap); rt(90)
  num = concentricCircles(…)
  lt(90); bk(gap); rt(90)
  return 1 + num
```

**concentricCircles(13, 5, …)**

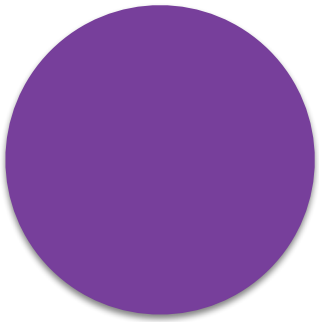
radius 13 gap 5

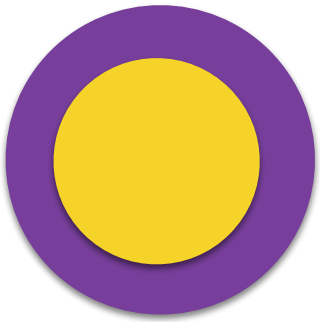
```
if radius < gap:
  return 0
else:
  drawDisc(radius, colorOuter)
  lt(90); fd(gap); rt(90)
  num = concentricCircles(…)
  lt(90); bk(gap); rt(90)
  return 1 + num
```

**concentricCircles(8, 5, …)**

radius 8 gap 5

```
if radius < gap:
  return 0
else:
  drawDisc(radius, colorOuter)
  lt(90); fd(gap); rt(90)
  num = concentricCircles(…)
  lt(90); bk(gap); rt(90)
  return 1 + num
```





```
def concentricCircles(radius, gap, colorOuter, colorInner):
    """Recursive function to draw concentric circles"""
  if radius < gap:
    return 0
  else:
    drawDisc(radius, colorOuter)
    lt(90); fd(gap); rt(90)
    num = concentricCircles(radius-gap, gap, colorInner, colorOuter)
    lt(90); bk(gap); rt(90)
    return 1 + num
```

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

**concentricCircles(18, 5, …)**

radius 18 gap 5

```
if radius < gap:
  return 0
else:
  drawDisc(radius, colorOuter)
  lt(90); fd(gap); rt(90)
  num = concentricCircles(…)
  lt(90); bk(gap); rt(90)
  return 1 + num
```

**concentricCircles(13, 5, …)**

radius 13 gap 5

```
if radius < gap:
  return 0
else:
  drawDisc(radius, colorOuter)
  lt(90); fd(gap); rt(90)
  num = concentricCircles(…)
  lt(90); bk(gap); rt(90)
  return 1 + num
```

**concentricCircles(8, 5, …)**

radius 8 gap 5

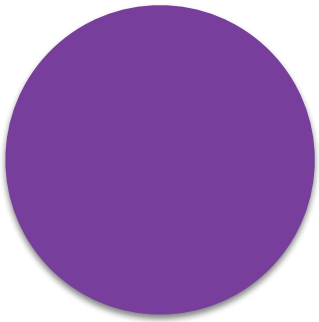
```
if radius < gap:
  return 0
else:
  drawDisc(radius, colorOuter)
  lt(90); fd(gap); rt(90)
  num = concentricCircles(…)
  lt(90); bk(gap); rt(90)
  return 1 + num
```

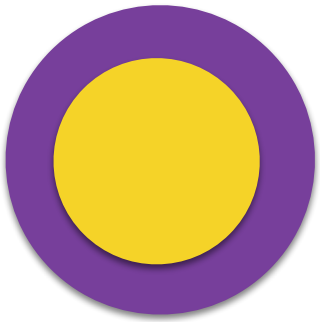
0

**concentricCircles(3, 5, …)**

radius 3 gap 5

```
if radius < gap:
  return 0
else:
  drawDisc(radius, colorOuter)
  lt(90); fd(gap); rt(90)
  num = concentricCircles(…)
  lt(90); bk(gap); rt(90)
  return 1 + num
```





```
def concentricCircles(radius, gap, colorOuter, colorInner):
    """Recursive function to draw concentric circles"""
  if radius < gap:
    return 0
  else:
    drawDisc(radius, colorOuter)
    lt(90); fd(gap); rt(90)
    num = concentricCircles(radius-gap, gap, colorInner, colorOuter)
    lt(90); bk(gap); rt(90)
    return 1 + num
```

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

**concentricCircles(18, 5, …)**

radius 18 gap 5

```
if radius < gap:
  return 0
else:
  drawDisc(radius, colorOuter)
  lt(90); fd(gap); rt(90)
  num = concentricCircles(…)
  lt(90); bk(gap); rt(90)
  return 1 + num
```

**concentricCircles(13, 5, …)**

radius 13 gap 5

```
if radius < gap:
  return 0
else:
  drawDisc(radius, colorOuter)
  lt(90); fd(gap); rt(90)
  num = concentricCircles(…)
  lt(90); bk(gap); rt(90)
  return 1 + num
```

**concentricCircles(8, 5, …)**

radius 8 gap 5

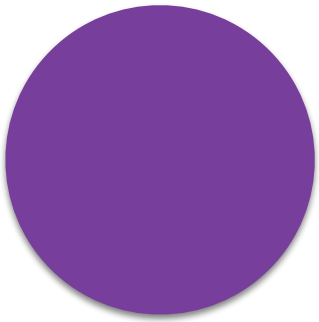
```
if radius < gap:
  return 0
else:
  drawDisc(radius, colorOuter)
  lt(90); fd(gap); rt(90)
  num = concentricCircles(…)
  lt(90); bk(gap); rt(90)
  return 1 + num
```

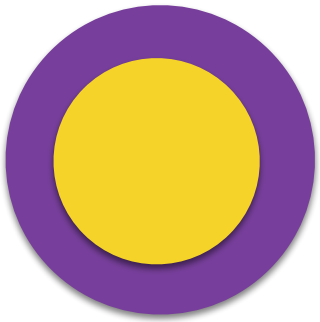
0

**concentricCircles(3, 5, …)**

radius 3 gap 5

```
if radius < gap:
  return 0
else:
  drawDisc(radius, colorOuter)
  lt(90); fd(gap); rt(90)
  num = concentricCircles(…)
  lt(90); bk(gap); rt(90)
  return 1 + num
```





```
def concentricCircles(radius, gap, colorOuter, colorInner):
    """Recursive function to draw concentric circles"""
  if radius < gap:
    return 0
  else:
    drawDisc(radius, colorOuter)
    lt(90); fd(gap); rt(90)
    num = concentricCircles(radius-gap, gap, colorInner, colorOuter)
    lt(90); bk(gap); rt(90)
    return 1 + num
```

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

**concentricCircles(18, 5, …)**

radius 18 gap 5

```
if radius < gap:
  return 0
else:
  drawDisc(radius, colorOuter)
  lt(90); fd(gap); rt(90)
  num = concentricCircles(…)
  lt(90); bk(gap); rt(90)
  return 1 + num
```

**concentricCircles(13, 5, …)**

radius 13 gap 5

```
if radius < gap:
  return 0
else:
  drawDisc(radius, colorOuter)
  lt(90); fd(gap); rt(90)
  num = 1
  lt(90); bk(gap); rt(90)
  return 1 + num
```

**concentricCircles(8, 5, …)**

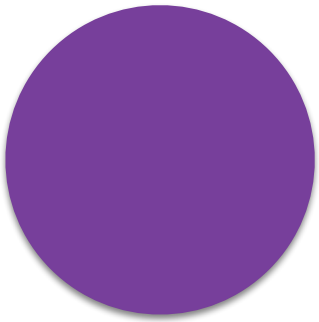
radius 8 gap 5

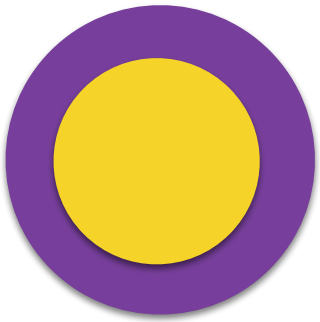
```
if radius < gap:
  return 0
else:
  drawDisc(radius, colorOuter)
  lt(90); fd(gap); rt(90)
  num = 0
  lt(90); bk(gap); rt(90)
  return 1 + num
```

**concentricCircles(3, 5, …)**

radius 3 gap 5

```
if radius < gap:
  return 0
else:
  drawDisc(radius, colorOuter)
  lt(90); fd(gap); rt(90)
  num = concentricCircles(…)
  lt(90); bk(gap); rt(90)
  return 1 + num
```





```
def concentricCircles(radius, gap, colorOuter, colorInner):
    """Recursive function to draw concentric circles"""
  if radius < gap:
    return 0
  else:
    drawDisc(radius, colorOuter)
    lt(90); fd(gap); rt(90)
    num = concentricCircles(radius-gap, gap, colorInner, colorOuter)
    lt(90); bk(gap); rt(90)
    return 1 + num
```

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

**concentricCircles(18, 5, …)**

radius 18 gap 5

```
if radius < gap:
  return 0
else:
  drawDisc(radius, colorOuter)
  lt(90); fd(gap); rt(90)
  num = concentricCircles(…)
  lt(90); bk(gap); rt(90)
  return 1 + num
```

**concentricCircles(13, 5, …)**

radius 13 gap 5

```
if radius < gap:
  return 0
else:
  drawDisc(radius, colorOuter)
  lt(90); fd(gap); rt(90)
  num = 1 ntricCircles(…)
  lt(90); bk(gap); rt(90)
  return 1 + num
```

**concentricCircles(8, 5, …)**

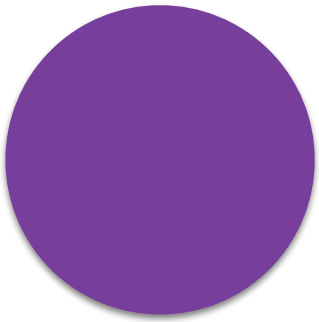
radius 8 gap 5

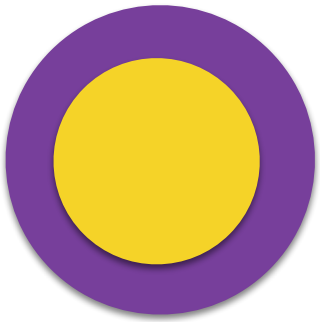
```
if radius < gap:
  return 0
else:
  drawDisc(radius, colorOuter)
  lt(90); fd(gap); rt(90)
  num = 0 ntricCircles(…)
  lt(90); bk(gap); rt(90)
  return 1 + num
```

**concentricCircles(3, 5, …)**

radius 3 gap 5

```
if radius < gap:
  return 0
else:
  drawDisc(radius, colorOuter)
  lt(90); fd(gap); rt(90)
  num = concentricCircles(…)
  lt(90); bk(gap); rt(90)
  return 1 + num
```





```
def concentricCircles(radius, gap, colorOuter, colorInner):
    """Recursive function to draw concentric circles"""
  if radius < gap:
    return 0
  else:
    drawDisc(radius, colorOuter)
    lt(90); fd(gap); rt(90)
    num = concentricCircles(radius-gap, gap, colorInner, colorOuter)
    lt(90); bk(gap); rt(90)
    return 1 + num
```

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

**concentricCircles(18, 5, …)**

radius 18 gap 5

```
if radius < gap:
  return 0
else:
  drawDisc(radius, colorOuter)
  lt(90); fd(gap); rt(90)
  num = concentricCircles(…)
  lt(90); bk(gap); rt(90)
  return 1 + num
```

2

**concentricCircles(13, 5, …)**

radius 13 gap 5

```
if radius < gap:
  return 0
else:
  drawDisc(radius, colorOuter)
  lt(90); fd(gap); rt(90)
  num = concentricCircles(…)
  lt(90); bk(gap); rt(90)
  return 1 + num
```

1

**concentricCircles(8, 5, …)**

radius 8 gap 5

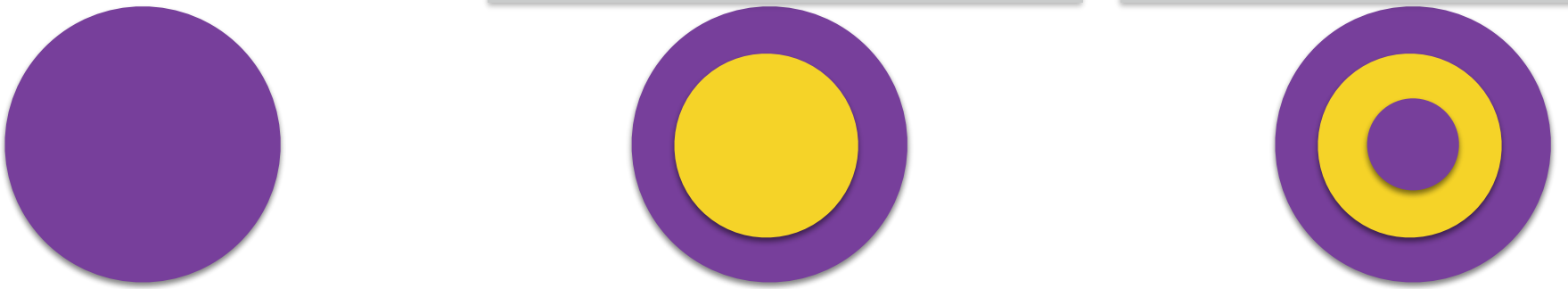
```
if radius < gap:
  return 0
else:
  drawDisc(radius, colorOuter)
  lt(90); fd(gap); rt(90)
  num = concentricCircles(…)
  lt(90); bk(gap); rt(90)
  return 1 + num
```

0

**concentricCircles(3, 5, …)**

radius 3 gap 5

```
if radius < gap:
  return 0
else:
  drawDisc(radius, colorOuter)
  lt(90); fd(gap); rt(90)
  num = concentricCircles(…)
  lt(90); bk(gap); rt(90)
  return 1 + num
```



```
def concentricCircles(radius, gap, colorOuter, colorInner):
    """Recursive function to draw concentric circles"""
  if radius < gap:
    return 0
  else:
    drawDisc(radius, colorOuter)
    lt(90); fd(gap); rt(90)
    num = concentricCircles(radius-gap, gap, colorInner, colorOuter)
    lt(90); bk(gap); rt(90)
    return 1 + num
```

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

**concentricCircles(18, 5, …)**

radius 18 gap 5

```
if radius < gap:
  return 0
else:
  drawDisc(radius, colorOuter)
  lt(90); fd(gap); rt(90)
  num = concentricCircles(…)
  lt(90); bk(gap); rt(90)
  return 1 + num
```

2

**concentricCircles(13, 5, …)**

radius 13 gap 5

```
if radius < gap:
  return 0
else:
  drawDisc(radius, colorOuter)
  lt(90); fd(gap); rt(90)
  num = concentricCircles(…)
  lt(90); bk(gap); rt(90)
  return 1 + num
```

1

**concentricCircles(8, 5, …)**

radius 8 gap 5

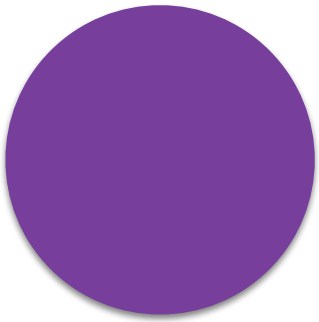
```
if radius < gap:
  return 0
else:
  drawDisc(radius, colorOuter)
  lt(90); fd(gap); rt(90)
  num = concentricCircles(…)
  lt(90); bk(gap); rt(90)
  return 1 + num
```

0

**concentricCircles(3, 5, …)**

radius 3 gap 5

```
if radius < gap:
  return 0
else:
  drawDisc(radius, colorOuter)
  lt(90); fd(gap); rt(90)
  num = concentricCircles(…)
  lt(90); bk(gap); rt(90)
  return 1 + num
```



```
def concentricCircles(radius, gap, colorOuter, colorInner):
    """Recursive function to draw concentric circles"""
  if radius < gap:
    return 0
  else:
    drawDisc(radius, colorOuter)
    lt(90); fd(gap); rt(90)
    num = concentricCircles(radius-gap, gap, colorInner, colorOuter)
    lt(90); bk(gap); rt(90)
    return 1 + num
```

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

3

**concentricCircles(18, 5, …)**

radius 18 gap 5

```
if radius < gap:
  return 0
else:
  drawDisc(radius, colorOuter)
  lt(90); fd(gap); rt(90)
  num = concentricCircles(…)
  lt(90); bk(gap); rt(90)
  return 1 + num
```

2

**concentricCircles(13, 5, …)**

radius 13 gap

```
if radius < gap:
  return 0
else:
  drawDisc(radius, colorOuter)
  lt(90); fd(gap); rt(90)
  num = concentricCircles(…)
  lt(90); bk(gap); rt(90)
  return 1 + num
```

1

**concentricCircles(8, 5, …)**

radius 8 gap 5

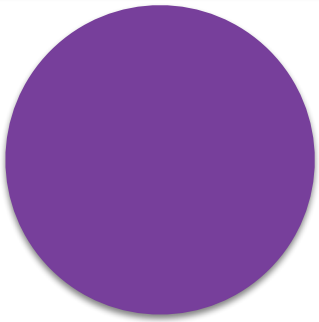
```
if radius < gap:
  return 0
else:
  drawDisc(radius, colorOuter)
  fd(gap); rt(90)
  concentricCircles(…)
  bk(gap); rt(90)
  return 1 + num
```

0

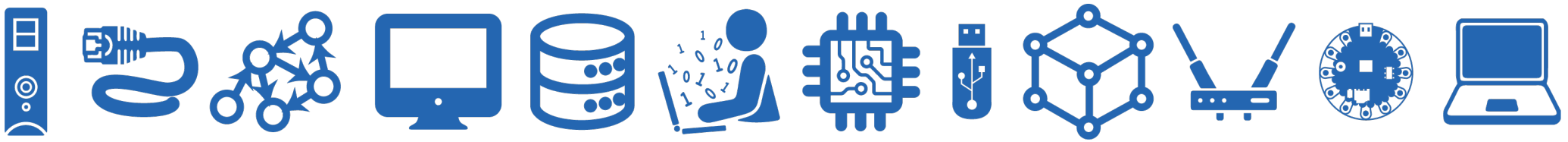
**concentricCircles(3, 5, …)**

radius 3 gap 5

```
if radius < gap:
  return 0
else:
  drawDisc(radius, colorOuter)
  lt(90); fd(gap); rt(90)
  num = concentricCircles(…)
  lt(90); bk(gap); rt(90)
  return 1 + num
```



# Nested Circles



# Nested Circles

```
def nestedCircles(radius, minRadius, colorOut, colorAlt):
    if radius < minRadius:
        return 0
    else:
        # contribute to the solution
        drawDisc(radius, colorOut)

        # save half of radius
        halfRadius = radius/2

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

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

        # position turtle for left subcircle
        bk(radius)

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

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

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

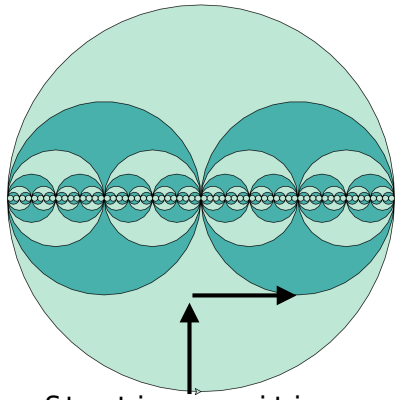
**Base case, return 0**

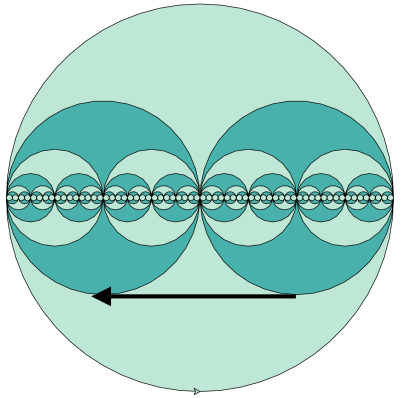
**Draw big circle**

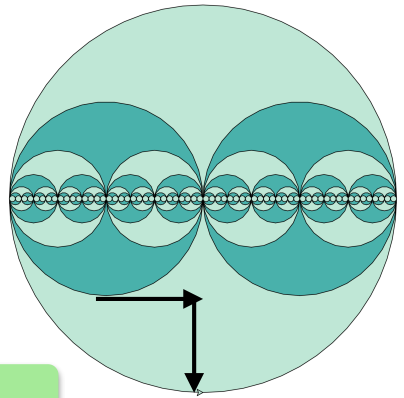
**Draw right circle(s)**

**Draw left circle(s)**

**Return num of circles drawn**

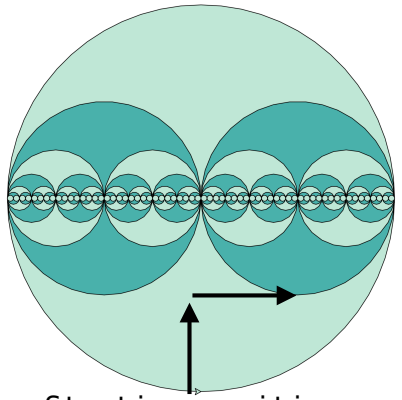


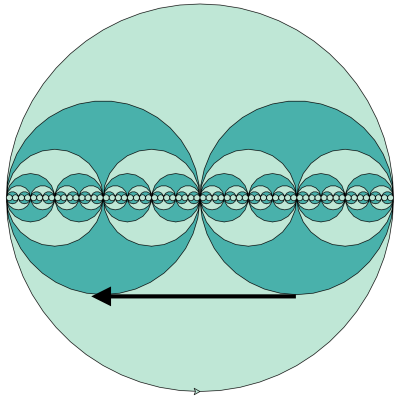



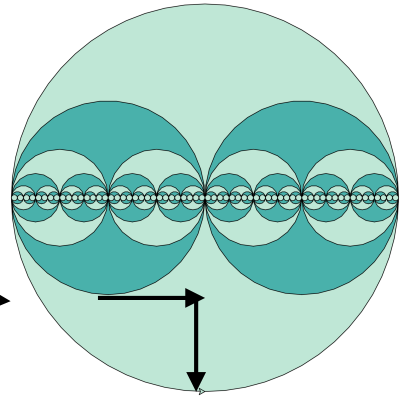


# Reposition Turtle

```
def nestedCircles(radius, minRadius, colorOut, colorAlt):
    if radius < minRadius:
        return 0
    else:
        # contribute to the solution
        drawDisc(radius, colorOut)

        # save half of radius
        halfRadius = radius/2

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

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

        # position turtle for left subcircle
        bk(radius)

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

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

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



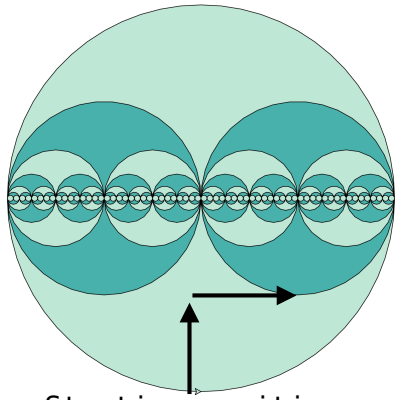


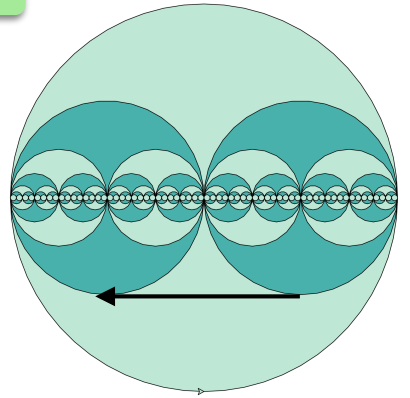

**Maintain invariance**

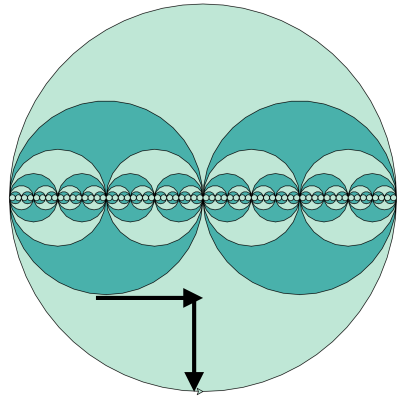
# Swap Colors

```
def nestedCircles(radius, minRadius, colorOut, colorAlt):
    if radius < minRadius:
        return 0
    else:
        # contribute to the solution
        drawDisc(radius, colorOut)

        # save half of radius
        halfRadius = radius/2

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

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

        # position turtle for left subcircle
        bk(radius)

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

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

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

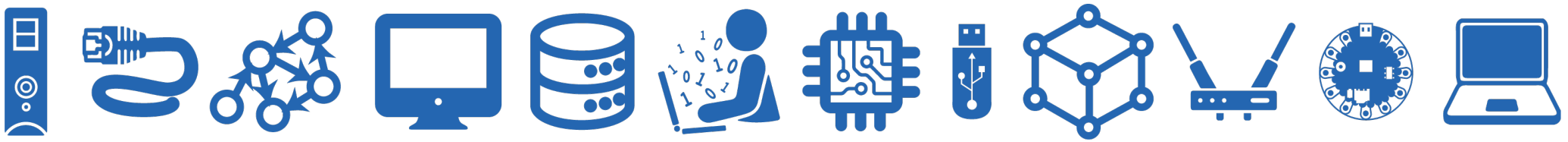
**Alternating colors**





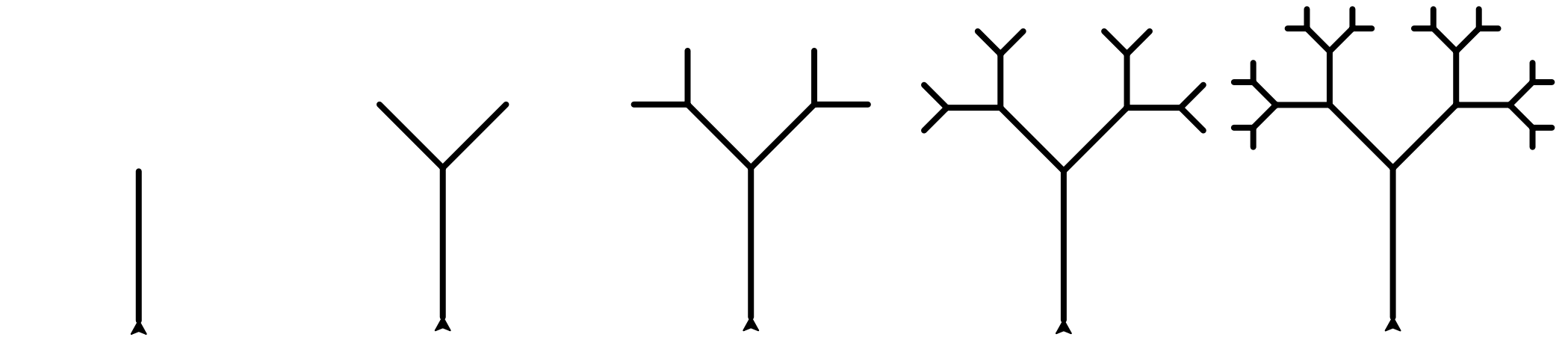
# Recursive Trees



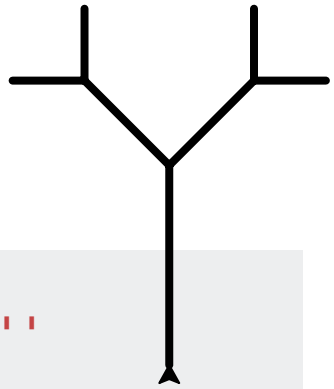
# One more recursive example: Trees

- We can draw more than just circles!
- Suppose we want to draw recursive trees and count branches
- What is our base case? Recursive case?
- Note: Assume turtle starts facing north

```
def tree(trunkLen, angle, shrinkFactor, minLength):
    # trunkLen is the trunk length of the main (vertical) trunk
    # angle is the branching angle, or the angle between a trunk and its
      right or left branch
    # shrinkFactor specifies how much smaller each subsequent branch is in length
    # minLength is the minimum branch length in our tree
```



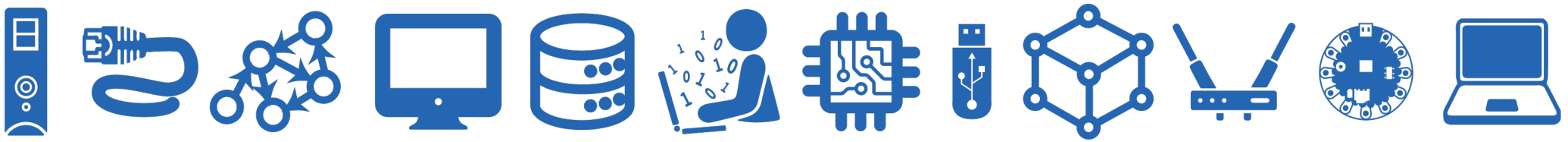
# Tree



```
def tree(trunkLen, angle, shrinkFactor, minLength):
    '''Draw tree and return number of branches drawn including trunk'''
    if (trunkLen < minLength): # Base case
        return 0
    else:
        # Draw trunk
        fd(trunkLen)

        # Right branch
        rt(angle)
        rightBranch = tree(trunkLen*shrinkFactor, angle, shrinkFactor, minLength)

        # Left branch
        lt(angle*2)
        leftBranch = tree(trunkLen*shrinkFactor, angle, shrinkFactor, minLength)

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

        return 1 + rightBranch + leftBranch
```

# Recursion vs. Iteration:

# `sumList`



# `sumList`

- **Goal:** Write a function to sum up a list of numbers
- Iterative approach? (i.e., using loops?)

# Iterative Approach to `sumList`

- **Goal:** Write a function to sum up a list of numbers
- Iterative approach:

```
def sumListIterative(numList):
    sum = 0
    for num in numList:
        sum += num
    return sum
```

```
>>> sumListIterative([3, 4, 20, 12, 2, 20])
61
```

# `sumList`

- **Goal:** Write a function to sum up a list of numbers
- Recursive approach?

# Recursive approach to `sumList`

- **Base case:**
  - `numList` is empty, return 0
- **Recursive rule:**
  - Return first element of `numList` plus result from calling `sumList` on rest of the elements of the list.
- Example: Suppose `numList = [6, 3, 6, 5]`
  - `sumList([6, 3, 6, 5]) = 6 + sumList([3, 6, 5])`
  - `sumList([3, 6, 5]) = 3 + sumList([6, 5])`
  - `sumList([6, 5]) = 6 + sumList([5])`
  - `sumList([5]) = 5 + sumList([])`
- For the base case we have `sum([])` returns `0`

# Recursive approach to `sumList`

- **Base case:**
  - `numList` is empty, return 0
- **Recursive rule:**
  - Return first element of `numList` plus result from calling `sumList` on rest of the elements of the list.
- Example: Suppose `numList = [6, 3, 6, 5]`
  - `sumLi` 20 `6, 3, 6, 5]) = 6 + sumLis` 14 `3, 6, 5])`
  - `sumLi` 14 `3, 6, 5]) = 3 + sumL` 11 `[6, 5])`
  - `sumLi` 11 `6, 5]) = 6 + sum` 5 `([5])`
  - `sumLi` 5 `5]) = 5 + su` 0 `t([])`
- For the base case we have `sum([])` returns `0`

# Recursive approach to `sumList`

```
def sumList(numList):
    """Returns sum of given list"""
    if numList == []:
        return 0
    else:
        return numList[0] + sumList(numList[1:])
```

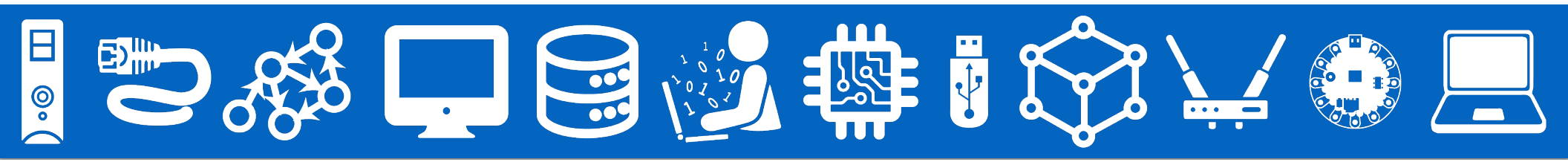
```
>>> sumList([3, 4, 20, 12, 2, 20])
61
```

# Pros and Cons of Recursion

- **Pros:**
  - Can lead to syntactically simpler, shorter, more elegant programs
  - Many tasks, such as exploring and building complex data structures, are best written as recursive programs
  - You will see recursive code or pseudocode out in the real world
- **Cons:**
  - Recursive approaches often have more computational overhead than iterative ones because of repeated function calls
  - Recursion has a steeper learning curve (but can be very rewarding once you get the hang of it)
  - To understand recursion you must understand recursion… (an old CS folklore joke about the steep learning curve)

# The end!

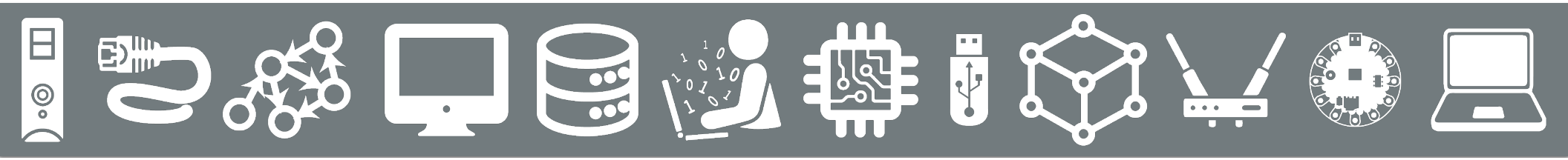
# CS134:
# Lab 7



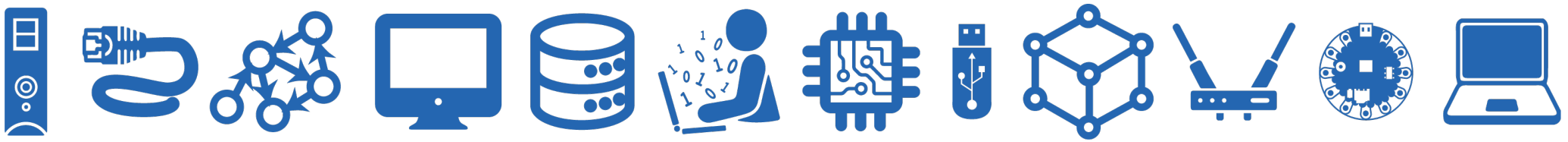
# Lab 7 Overview

- **Recursion!!!!**
- Pre-lab exercise: `sumDigits(num)`
  - Similar to `sumList(numList)`, from class, but *with a twist*!
- Bedtime Story
  - Similar to `countUp` and `countDown` from class, but *with a big twist*!
- Recursive Squares
  - Similar to `concentricCircles`, from class, but *with a twist*!
- Square quilt
  - Similar to `nestedCircles`, from class, but *with a twist*!
  - Read the handout carefully. Make sure you draw all four quadrants!
- Shrub
  - Similar to `tree`, from class, but *with a twist*!

# *Leftover Slides*



# Why Recursion?

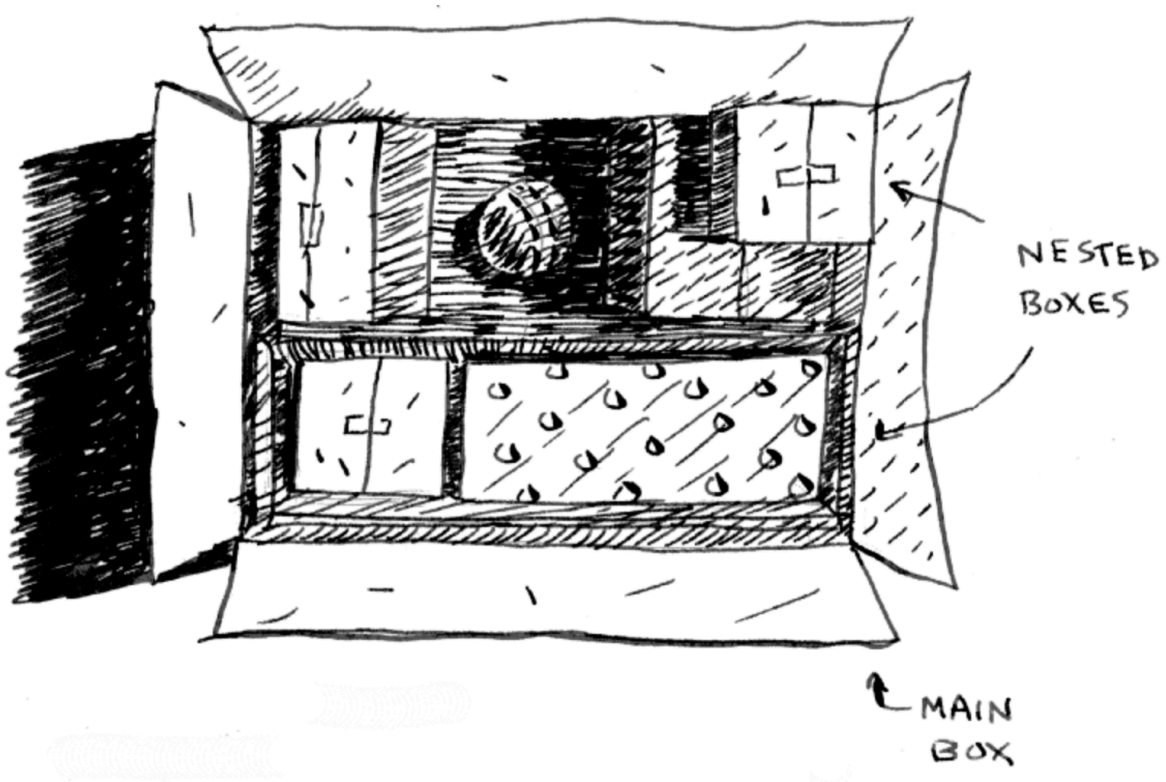


# What's The Big Deal With Recursion?

- Why choose recursion over iteration?
- The recursive solution can be more elegant, resulting in fewer lines of code
- Fewer lines of code often correlates with less debugging!
- Let's consider a simple real world example

# A Simple Real World Task

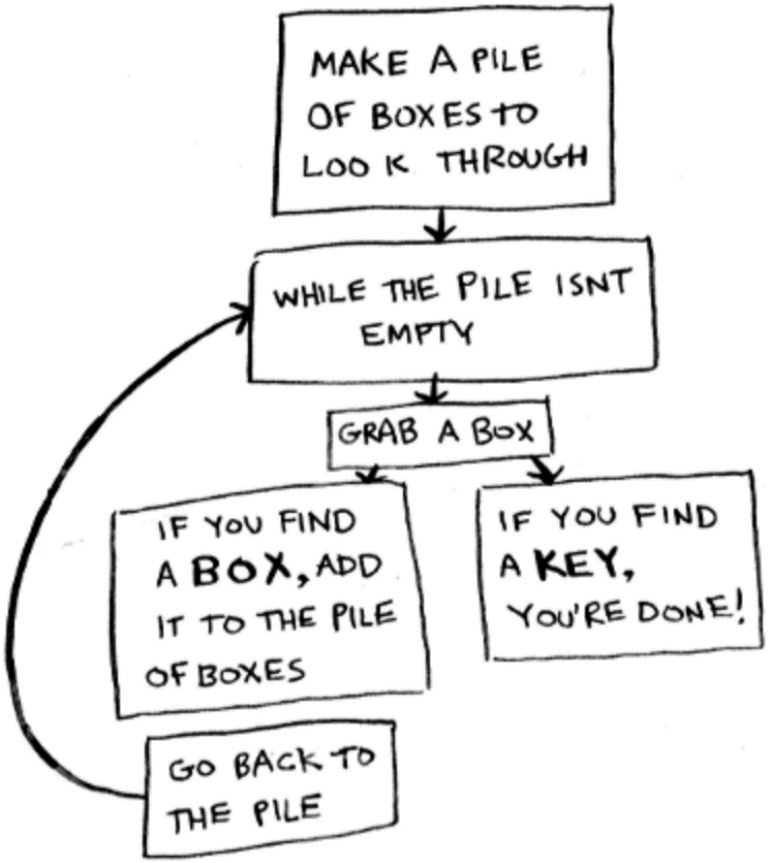
- Consider trying to find a key that is lost in a pile of boxes within boxes.
- (This task is quite similar to trying to find a file on your computer!)



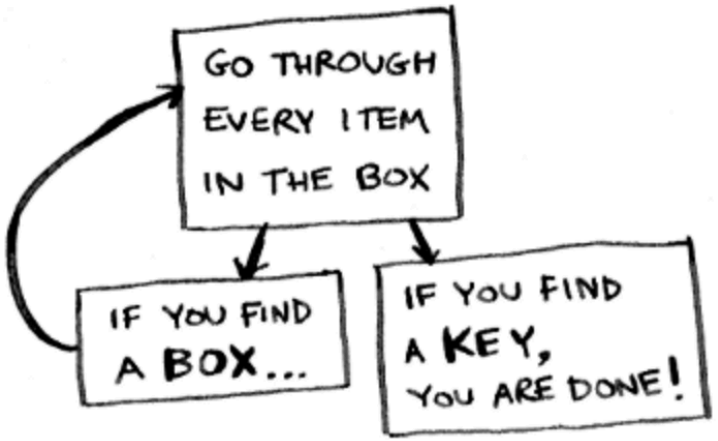

Credit to Aditya Bhargava for the nice illustrations

# Comparing Approaches To Finding The Key

- In this case, it's much easier to describe the algorithm using a recursive approach

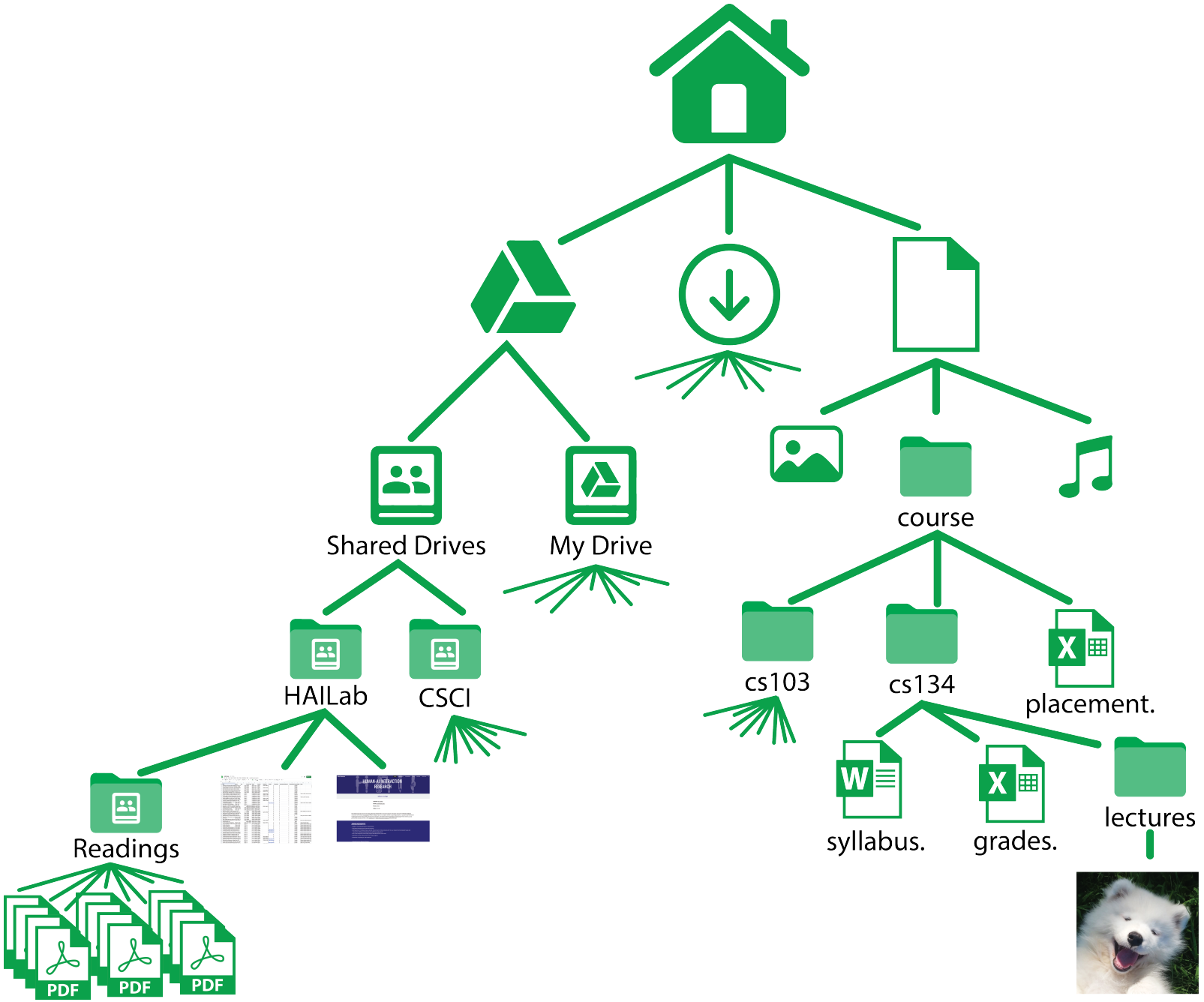



Iterative Approach




Recursive Approach

# Similar: Searching For A File On Our Computer



# Similar: Finding a Word in a Dictionary