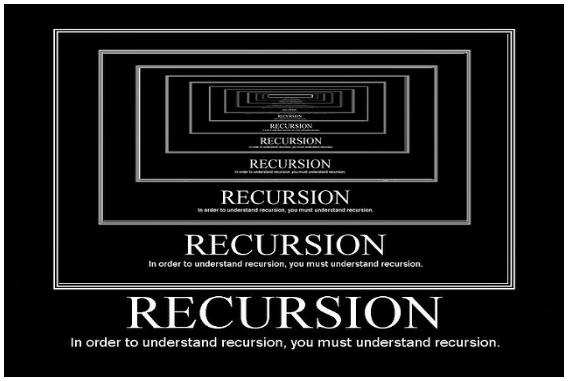
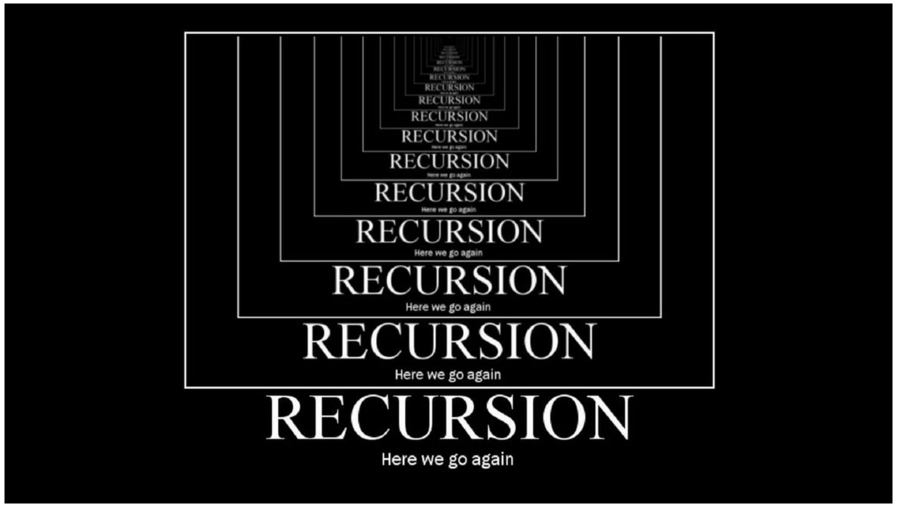
Introduction to Recursion

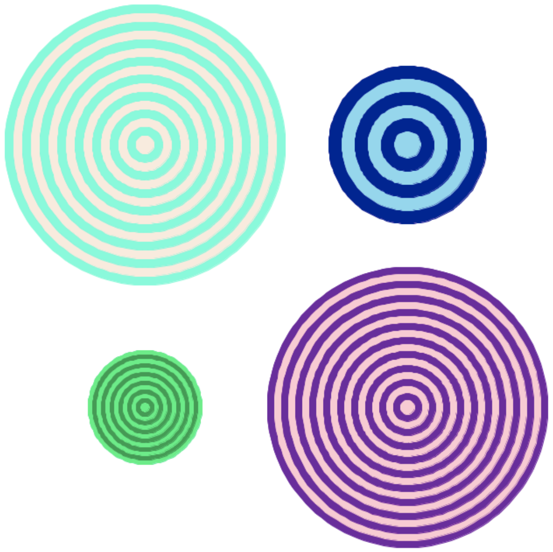

RECURSION
In order to understand recursion, you must understand recursion.

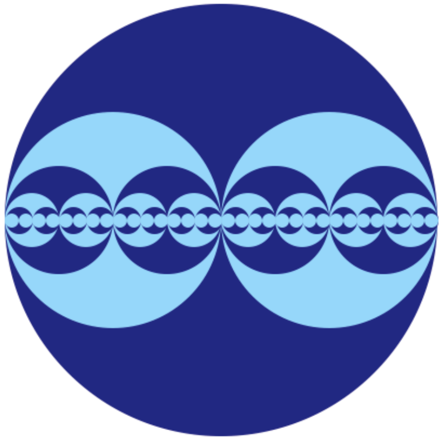
CS111 Computer Programming

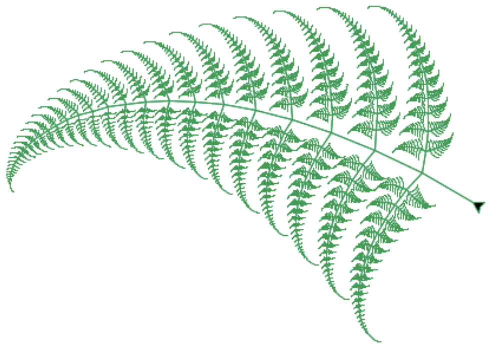
Department of Computer Science
Wellesley College

Recursive Patterns

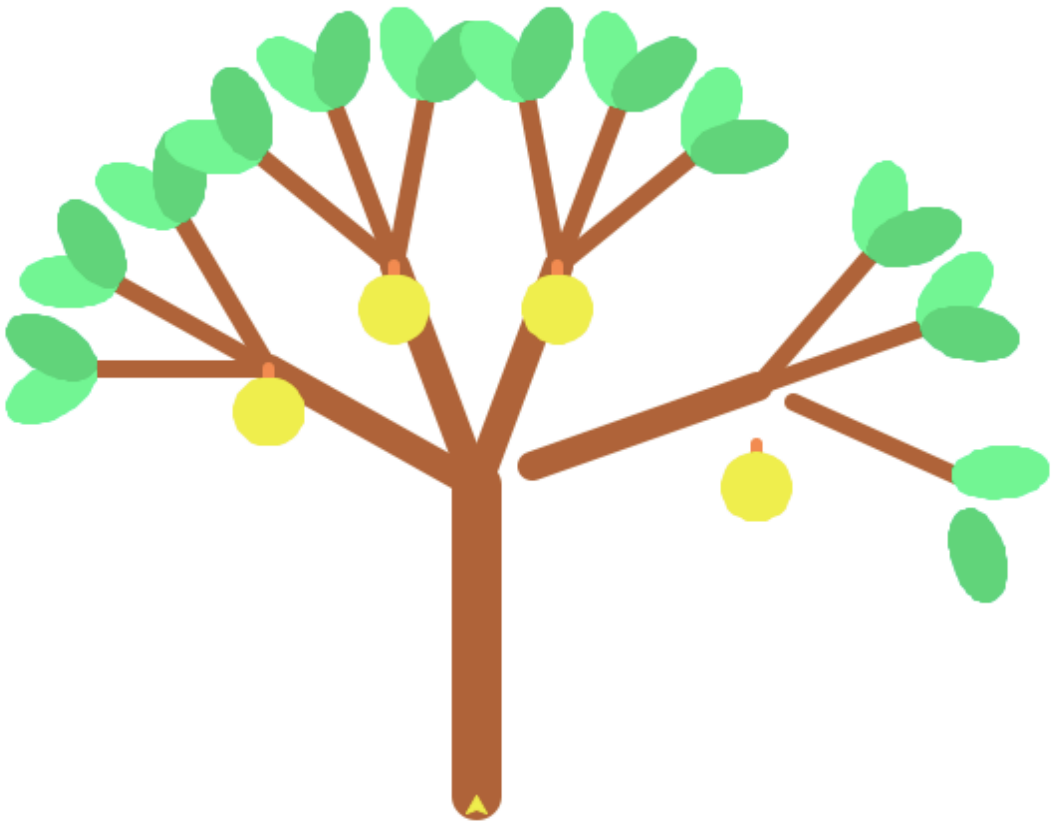




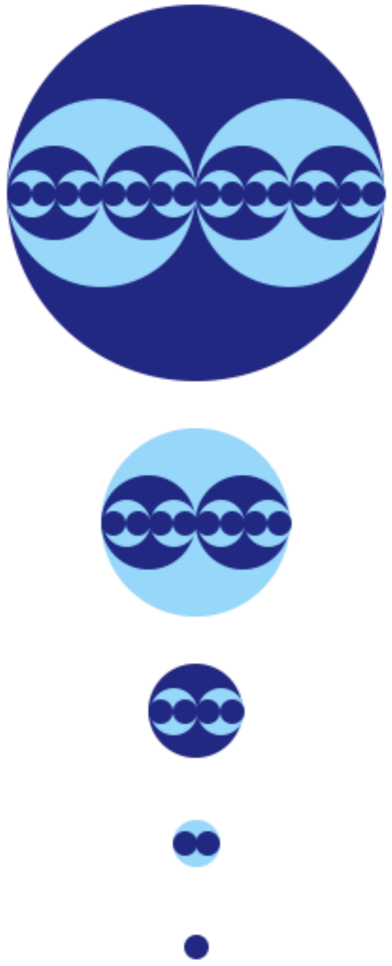




Reminder: Gluing Functions Together



tree → branch → twig → leaf



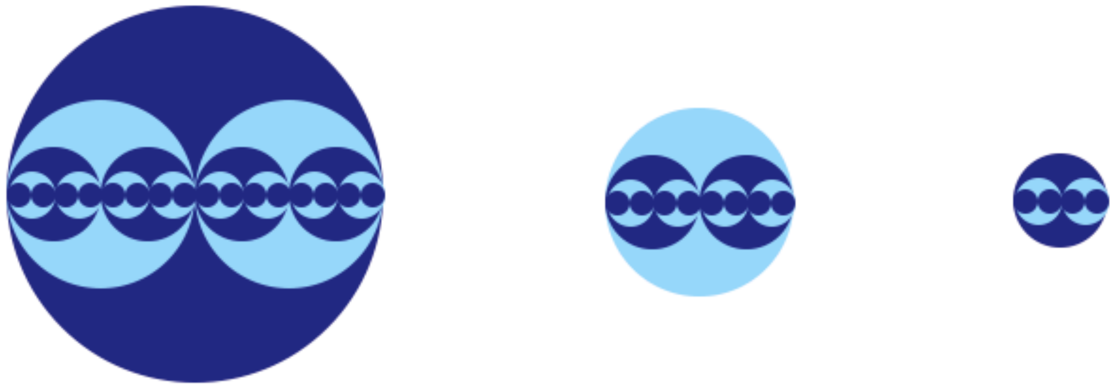
bubbles → bubbles → ...

What is Recursion?

Concepts in this slide:
Recursion is an instance of solving a problem by sub-division.

Gluing functions together...

...where the sub-problems involve the problem itself!

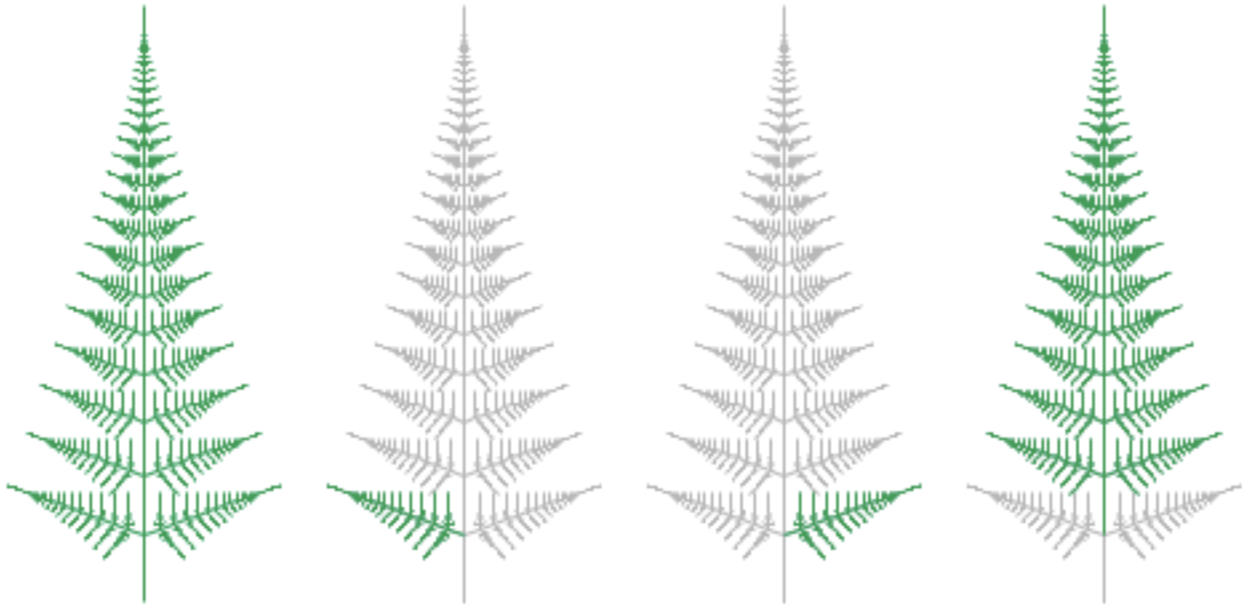


With recursion, the solution to a problem depends on solutions to **smaller** instances of the **same** problem

A recursive function is a function that invokes itself.

Self-Containing Patterns

Fractals, found in nature and mathematics, are examples of patterns that contain smaller copies of themselves.



This fern image is made of three smaller ferns.

Changing Parameters

In recursion, the **parameters** to the function often change, so the smaller patterns aren't quite the same as the original.

5	4	3	2	1
4	3	2	1	
3	2	1		
2	1			
1				

Each of these number sequences contains the next, which starts from a smaller number.

Review: functions calling other functions

(in anticipation of writing functions that call themselves)

Which would work? Why/why not?

```
def print2(s):
    print(s)
    print(s)

def print4(s):
    print2(s)
    print2(s)

print4('okay')
```

```
def print4(s):
    print2(s)
    print2(s)

def print2(s):
    print(s)
    print(s)

print4('okay')
```

```
def print4(s):
    print2(s)
    print2(s)

print4('okay')

def print2(s):
    print(s)
    print(s)
```

Our first recursive function: countDown

Concepts in this slide: In recursion, the smaller problems are the same as the problem we're trying to solve.

Let's write a function that prints the integers from **n** down to 1 (**without using loops**):

```
def countDown(n):
    '''Prints integers from n down to 1'''
```

```
In [ ]: countDown(5)
5
4
3
2
1
```

```
print(5)
countDown(4)
```

```
In [ ]: countDown(4)
4
3
2
1
```

```
print(4)
countDown(3)
```

Decomposition

We can think of the countDown(5) as composed of the print(5) statement and the invocation of countDown(4). And so on.

`countDown`: First Define Base Case

The base case. When the problem is so simple that we can solve it trivially and we need not decompose it into subproblems.

```
def countDown(n):
    '''Prints integers from n down to 1'''
    if n < 1:
        pass # Do nothing
```

Knowing when to stop
At some point, there is no more decomposition, we have come to a point where we'll need to stop. This is similar to the stopping condition for a while loop. The base case is what tells the recursion to stop.

countDown: More Cases

When you're solving a recursive problem, you can start by writing many **elif** cases. Continue until you see a pattern develop.

```
def countDown(n):
    '''Prints integers from n down to 1'''
    if n < 1:
        pass # Do nothing
    elif n == 1:
        print(1)
    elif n == 2:
        print(2)
        print(1)
    elif n == 3:
        print(3)
        print(2)
        print(1)
    elif ...
```

These are the same!

Finding the pattern

As you write **elif** cases, you'll eventually see the same code appearing again and again. That's the code that can be made into a recursive function call.

We are repeating print statements in elif clauses

countDown: More Cases

You can simplify the pattern by using a recursive call to the same function with different parameters (which will send you into a different `elif` case).

```python
def countDown(n):
    '''Prints integers from n down to 1'''
    if n < 1:
        pass # Do nothing
    elif n == 1:
        print(1)
    elif n == 2:
        print(2)
        print(1)
    elif n == 3:
        print(3)
        countDown(2)
    elif ...
```

Now it's recursive – we replaced print(2) and print(1) with countdown(2)

`countDown`: More Cases

Finally, try to generalize your code so that it uses the parameter in an `else` case that works for all other values.

```
def countDown(n):
    '''Prints integers from n down to 1'''
    if n < 1:
        pass # Do nothing
    elif n == 1:
        print(1)
    elif n == 2:
        print(2)
        print(1)
    elif n == 3:
        print(3)
        countDown(2)
    else: # Works for any n >= 1
        print(n)
        countDown(n-1)
```

Don't need these cases any more

countDown: Recursive Case

Concepts in this slide: The recursive case makes the problem smaller. The base case can be omitted.

The recursive case. One we understand how to write a recursive **else** case, we can eliminate unnecessary **elif** cases. Sometimes we can leave out the base case too, if it doesn't do anything (shown by countDownImplicit).

```
def countDown(n):
    '''Prints integers from
    n down to 1'''
    if n < 1:
        pass # Do nothing
    else:
        print(n)
        countDown(n-1)
```

```
def countDownImplicit(n):
    '''Prints integers from
    n down to 1'''
    if n > 0:
        print(n)
        countDownImplicit(n-1)
```

To notice:

- We got rid of several **elif** cases at once
- The recursive step does two things:
  a) performs an action that contributes to the solution
  b) Invokes the function with "smaller" parameters
- It is possible to omit the base case when it does nothing

Structure of Recursion

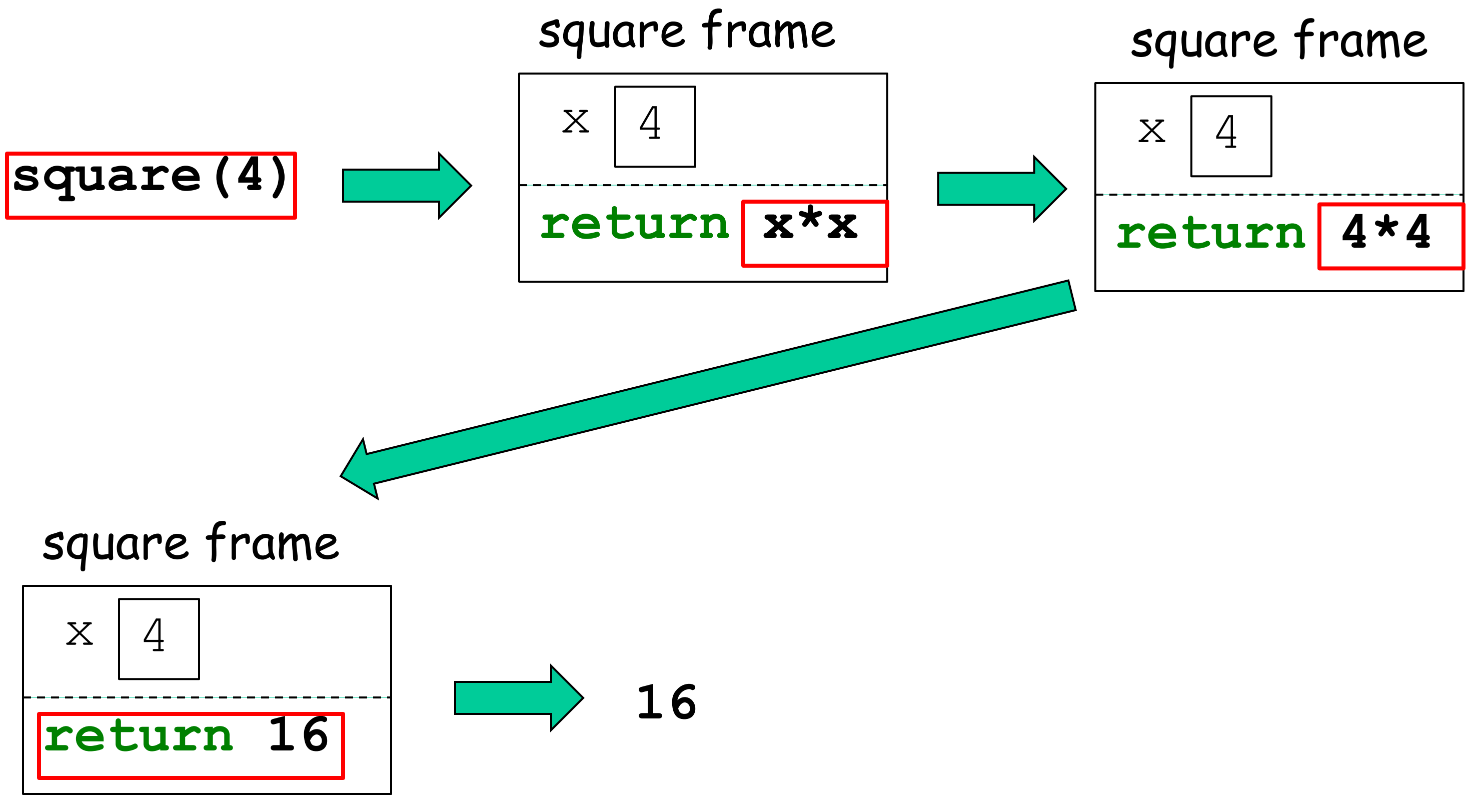
All recursive functions must have two types of cases:

- **BASE case**: a simple case where the solution is obvious.
  In this case the function does **not** invoke itself, since there is no need to decompose the problem into subproblems. *Sometimes, we can leave out the base case, if it doesn't do anything.*

- **RECURSIVE case**: a case where the problem
  - is decomposed into subproblems
  - at least one of the subproblems is solved by **invoking the function being defined**, i.e., the function is invoked in its own body.
  - You should assume the recursive function works correctly for **the smaller subproblems** (this is known as "wishful thinking")

Review: function call frames

Recursion isn't magic. It works because of the frame model for functions we introduced back in Lecture 04.

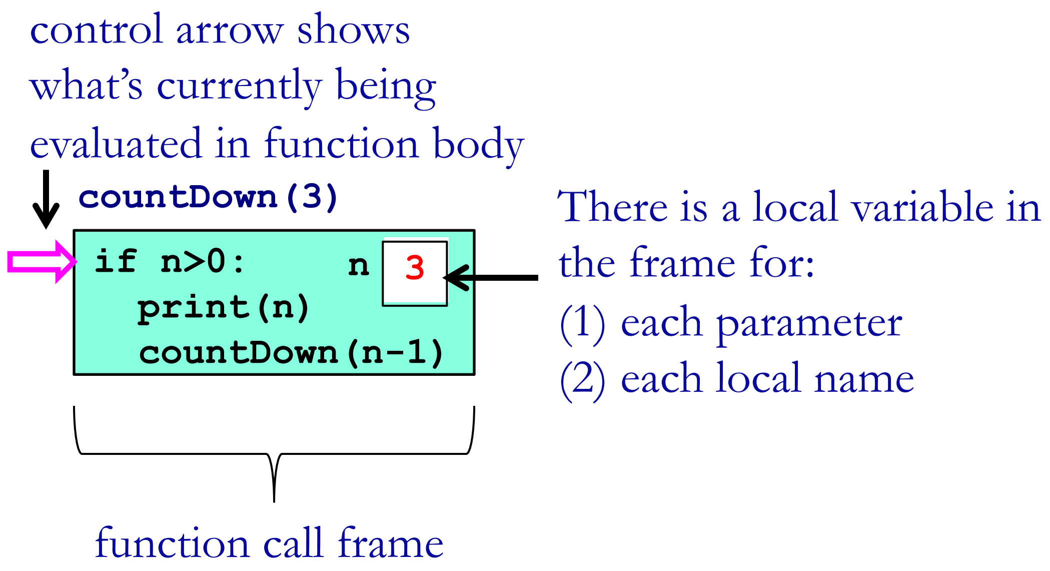


We'll now use this model to explain some recursion examples.

Invocation of `countDown(3)`
Anatomy of function call frames

```
def countDown(n):
    '''Prints integers from n down to 1'''
    if n>0:
        print(n)
        countDown(n-1)
```

```
In [4]: countDown(3)
```

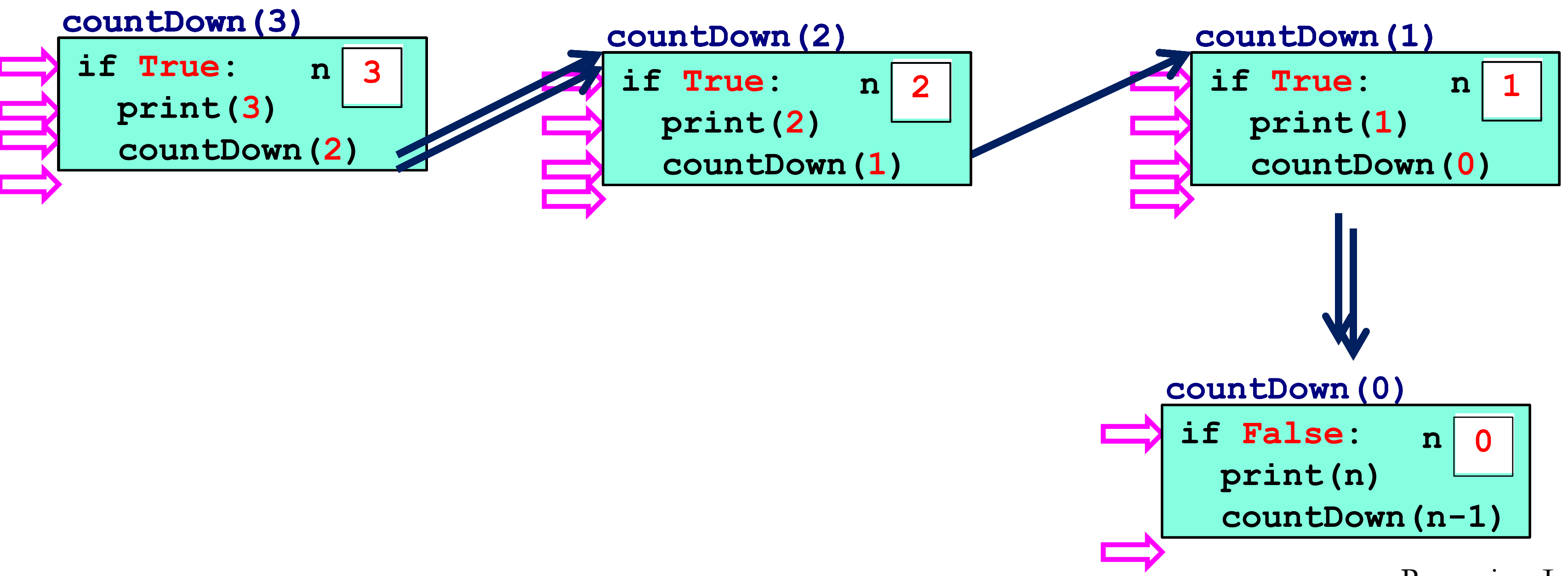

control arrow shows what's currently being evaluated in function body
countDown(3)
if n>0:
print(n)
countDown(n-1)
n
3
There is a local variable in the frame for:
(1) each parameter
(2) each local name
function call frame

Invocation of `countDown(3)`

Draw the diagram of function call frames

```
def countDown(n):
    '''Prints integers from n down to 1'''
    if n>0:
        print(n)
        countDown(n-1)
```

```
In [4]: countDown(3)
3
2
1
```



Recursion GOTCHA! #1: subproblem not smaller

```
def countDown(n):

    if n < 1:   # Base case
        pass   # Do nothing
    else: # Recursive case
        print n
        countDown(n)
```

The problem that you are solving recursively must get smaller each time you recur, i.e., you must get closer to the base case.

Otherwise, the recursion will not terminate -- a so-called **infinite recursion**.

Recursion GOTCHA! #2: missing base case

```
def countDown(n):
    print n
    countDown(n-1)
```

The recursion must eventually reach a base case in order to end.
If it doesn't, that's another way to get an **infinite recursion**.

"Maximum recursion depth exceeded"

In practice, the infinite recursion examples **will** terminate when Python runs out of resources for creating function call frames, leading to a **maximum recursion depth exceeded** error message:

```
In [2]: countDown(3)
3
2
1
0
-1
-2
-3
...
RuntimeError: maximum recursion depth exceeded
while calling a Python object
```

What does this function do?

it's *your* turn

```
def mystery(n):
    if n < 1:
        pass
    else:
        mystery(n - 1)
        print(n)
```

What does mystery(3) print?

countDownUp

it's your turn

Let's write a function that prints the integers from **n** down to 1 and then from 1 up to **n**:

```
def countDownUp(n):
    '''Prints integers from n down
    to 1 and then from 1 up to n
    '''
```

```
In [ ]: countDownUp(4)
4
3
2
1
1
2
3
4
```

Fill in the BLANK

countDownUp – Base Case

When is the problem so simple that we needn't decompose it into subproblems?
What code do we want to execute in this case?

```
def countDownUp(n):
    '''Prints integers from n down
    to 1 and then from 1 up to n
    '''
    if n < 1: # base case
        pass # do nothing
    else: # recursive case
```

```
In [ ]: countDownUp(4)
4
3
2
1
1
2
3
4
```

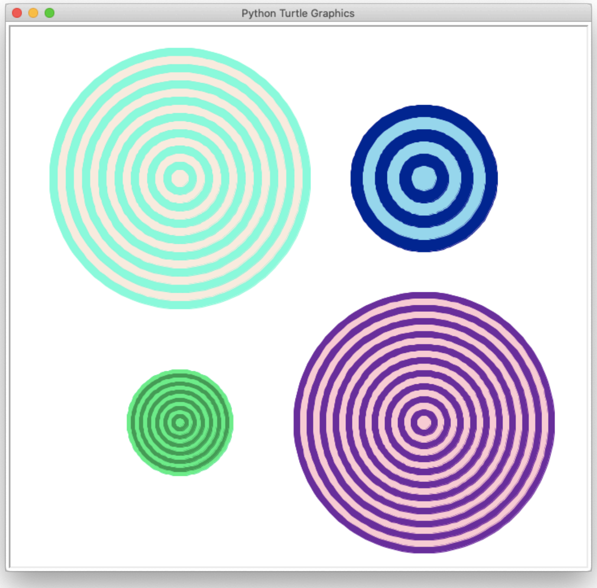
countDownUp – Recursive Case

How can we decompose the problem into subproblems so that one of the subproblems can be solved using **countDownUp**?

Target Practice (concentric circles)

it's your turn

Let's draw **turtle** "targets" using recursion:



We can use the **drawDot** function from **turtleBeads** to draw a circle.

Previously, we used loops to write **concentricCircles**.

Now we'll solve the same problem with recursion.

```
teleport(-150, 150)
drawTarget(160, 10, 'Aquamarine', 'AntiqueWhite')
teleport(150, 150)
drawTarget(90, 15, 'navyblue', 'skyblue')
teleport(-150, -150)
drawTarget(65, 5, 'springgreen2', 'springgreen4')
teleport(150, -150)
drawTarget(160, 8, 'purple4', 'pink')
```

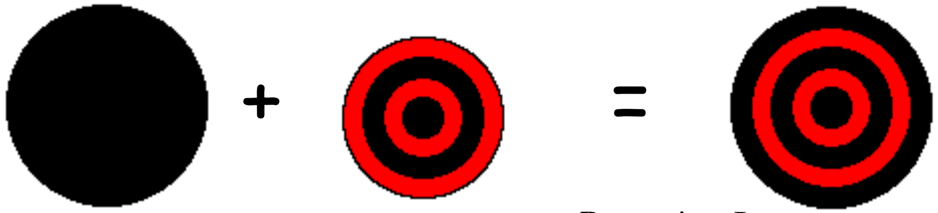
drawTarget: base case?

it's *your* turn

```
def drawTarget(radius, thickness, color1, color2):
    '''Draws a bullseye target with the given radius with
    alternating colors, color1 and color2, where color1 is
    the outermost color. thickness is the width of each "band”
    in the ring. thickness is also the radius of the smallest
    circle that gets drawn.'''
```

drawTarget: recursive case?

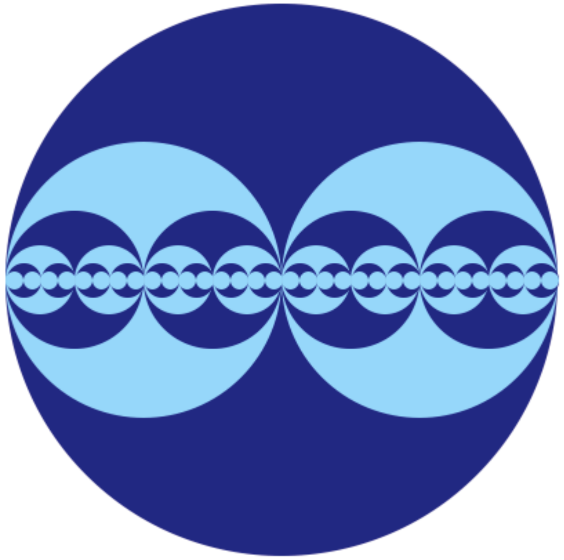
Hint: how can we decompose the problem into two subproblems such that one of them involves drawing a target?



Define bubbles

EXTRA it's your turn

```
bubbles(400, 12, 'MidnightBlue', 'LightSkyBlue')
```

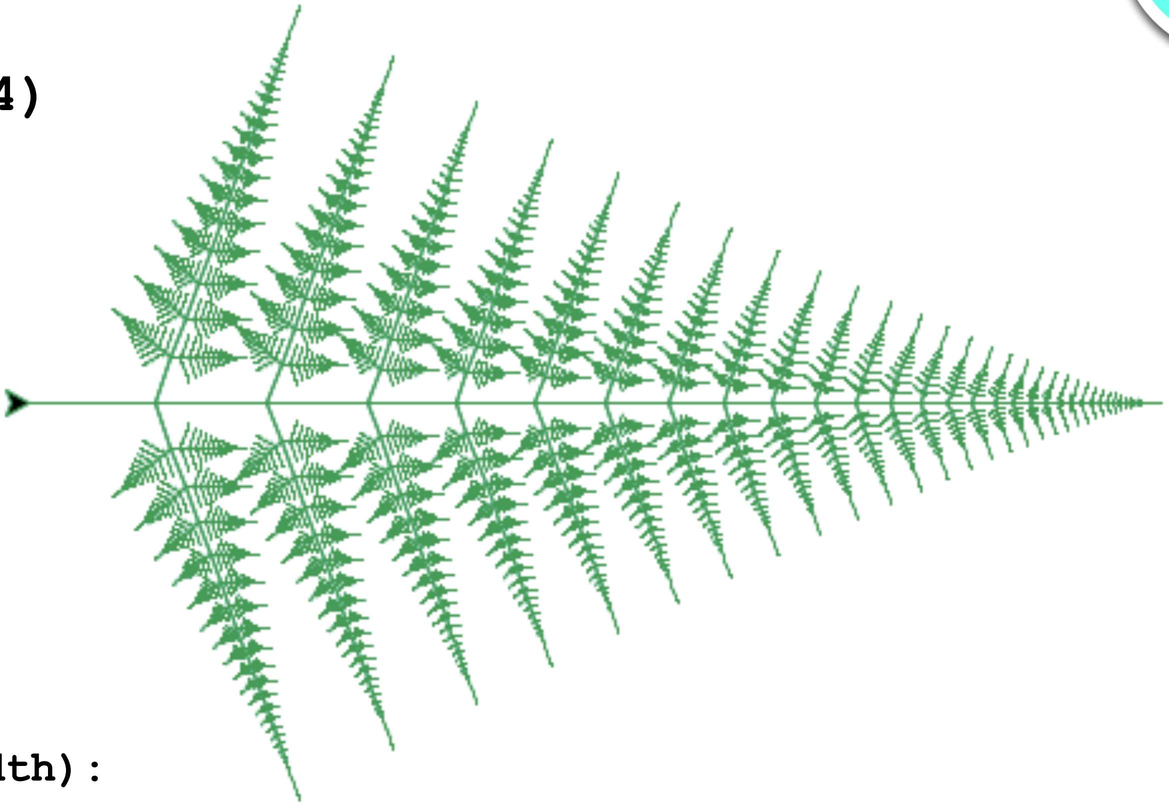


```
def bubbles(radius, minRadius, color1, color2):
    '''
    Draws a circle with two half-sized circles inside it, and two
    half-sized circles inside those, etc. unless the size is smaller
    than the given minimum, in which case nothing is drawn. The outer
    circle is colored color1, while the inner circles are color2, and
    their inner circles are color1 again, and so on.
    Hint: use the turtleBeads drawDot function for faster and smoother
    circles.'''
```

EXTRA it's your turn

Define fern

```
fern(400, 300, 4)
```



```
def fern(length, width):
    '''
    Draws a fern which is built from a single straight line and three
    smaller ferns: one continuing forward, and two more at 70-degree
    angles to either side. The length of the ferns on each side is
    half the initial width, while their widths are 1/7 of the initial
    length. The line drawn is 1/8 of the initial length. Ferns where
    the length is less than 2 are not drawn.
    '''
```