Introduction to Recursion
Recursion

- Recursion is the name given to a process that is composed of (smaller versions) of the itself
Recursive Approach

- **REDUCE** the problem to smaller subproblem(s) (smaller version(s) of itself)
- **DELEGATE** the smaller problems to the recursion fairy *(formally known as induction hypothesis)* and assume they're solved correctly
- **COMBINE** the solution(s) of the smaller subproblems to reach/return the solution
Palindromes

EVE
CIVIC
MADAM
AVID DIVA
STEP ON NO PETS
STRESSED DESSERTS
ABLE WAS I ERE I SAW ELBA
LIVED ON DECAF FACED NO DEVIL
Iterative Approach

\[
\text{reverse}(s)
\]
Iterative Approach

\[ s \]

reverse(s)
Iterative Approach

\[ s \rightarrow \text{reverse}(s) \]
Iterative Approach
Recursive Approach

- REDUCE it smaller version of the same problem
  - Check if $s' = s[1:-1]$ is a palindrome
Recursive Approach

• **DELEGATE** the smaller problems to the recursion fairy (*formally known as induction hypothesis*) and assume they're solved correctly.
Recursive Approach

• **COMBINE** the solution(s) of the smaller subproblems to reach/return the solution

• return True if \( \text{palindrome}(s') \) is True and \( s[0] \) is same as \( s[-1] \)
Summing Numbers in a List
Iterative Approach

3, 4, 20, 12, 2, 20

sum

3
Iterative Approach

3, 4, 20, 12, 2, 20

sum
Iterative Approach

3, 4, 20, 12, 2, 20

sum = 61
Recursive Approach

- **REDUCE** it smaller version of the same problem
  - `sum numList[1:]`

```
3, 4, 20, 12, 2, 20
```

```
sum(numList[1:])
```
Recursive Approach

• **DELEGATE** the smaller problems to the recursion fairy (formally known as induction hypothesis) and assume they're solved correctly

```
3, 4, 20, 12, 2, 20
```

```
sum(numList[1:])
```
Recursive Approach

- **COMBINE** the solution(s) of the smaller subproblems to reach/return the solution
- return numList[0] + sum(numList[1:])

```
3, 4, 20, 12, 2, 20
```

```
3 + sum(numList[1:])
```
Base Case(s): 
the buck stops here

- Reached when the problem is sufficiently small that its trivial to solve directly
- Palindrome: empty string and all strings of length 1 are trivially palindromes
- Summing a list of numbers: the sum is trivially zero when the list is empty
Recursive Algorithm

- **Base case:** Solving problem directly.
- **Recursive case:**
  - **REDUCE** the problem to smaller subproblem(s) (smaller version(s) of itself)
  - **DELEGATE** the smaller problems to the recursion fairy *(formally known as induction hypothesis)* and assume they're solved correctly
  - **COMBINE** the solution(s) of the smaller subproblems to reach/return the solution
Recursive Functions

A recursive function is a function that calls itself
countDown

- Let’s write a recursive function that prints integers from $n$ down to 1 (without using any loops)
- Recursive case. (REDUCE/ DELEGATE/ COMBINE):
  Can think of $\text{countDown}(5)$ as $\text{print}(5)$ followed by $\text{countDown}(4)$

<table>
<thead>
<tr>
<th>In[1] countDown(5)</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>In[2] countDown(4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>
**countDown: Base Case**

- **Base case.** When the problem is so simple that we can solve it without decomposing any further.

```python
def countDown(n):
    '''Prints ints from n down to 1'''
    if n < 1:
        pass # do nothing
```

```
In[1] countDown(0)
```

Nothing is printed.
countDown: Recursive Case

- Recursive step does two things:
  - performs an action that contributes to the solution
  - Invokes the function on a smaller subproblem

```python
def countDown(n):
    '''Prints ints from n down to 1'''
    if n < 1:
        pass  # do nothing
    else:
        print(n)
        countDown(n-1)
```
countDown: Recursive Case

- Recursive step does two things:
  - performs an action that contributes to the solution
  - Invokes the function on a smaller subproblem

```python
def countDown_implicit_base_case(n):
    '''Prints ints from n down to 1'''
    if n > 0:
        print(n)
        countDown(n-1)
```
Structure of Recursion

- All recursive functions must have two types of cases
  - **Base case.** A case where problem is so simple, its solution can be returned directly
  - **Recursive case.** A case where the problem is
    - Decomposed into smaller subproblems, where at least of the subproblems is solved by invoking the function it
    - Trust the *recursion fairy* (aka *mathematical induction*) that smaller subproblems are solved correctly
Recursion GOTCHAs!
GOTCHA #1

- **Problem** that you are solving recursively is **not getting smaller**, that is, not getting closer to the base case --- *infinite recursion*!

- **Does not reach the base case**

```python
def countDownGotcha(n):
    '''Prints ints from n down to 1'''
    if n < 1:
        pass  # do nothing
    else:
        print(n)
        countDownGotcha(n)
```
GOTCHA #2

• Missing base case/ unreachable base case---another way to cause *infinite recursion*!

```python
def printHalvesGotcha(n):
    if n > 0:
        print(n)
        printHalvesGotcha(n/2)
```

*Always true!*
"Maximum recursion depth exceeded"

• In practice, the infinite recursion examples will terminate when Python runs out of resources for creating function call frames, leads to a "maximum recursion depth exceeded" error message
Review: Function Frame Model

A model to understand what happens we call a function
Review:
Function Frame Model

- Consider a simple function `square`
- What happens when \texttt{square(5)} is invoked?

```python
def square(x):
    return x*x
```
Review:
Function Frame Model

```python
>>> square(5)
```
Review:
Function Frame Model

```python
>>> square(5)
5
```

```python
def square(x):
    return x * x
```

```python
def square(x):
    return x * x
```
Review:
Function Frame Model

```python
>>> square(5)
5
```

```
def square(x):
    return x * x
```
Review:
Function Frame Model

```python
>>> square(5)
25
```

```
def square(x):
    return x * x
```

```python
square(5)  # returns 25
```
Summary:
Function Frame Model

• When we return from a function frame "control flow" goes back to where the function call was made

• The function frame (the state of the local variables inside it) are destroyed after the return

```
>>> square(5) + 4
25
```

Return value replaces the function call
Summary: Function Frame Model

- When we return from a function frame "control flow" goes back to where the function call was made.
- The function frame (the state of the local variables inside it) are destroyed after the return.
- If a function does not have an explicit return statement, then we return `None` when after all statements in the function body are executed.

```
>>> square(5) + 4
25
```

Return value replaces the function call.

```
square(5)
+------------------+
| x | 5 |
+------------------+
| return | 25 |
+------------------+
```
Review:
Function Frame Model
Review:
Function Frame Model

- How about functions that call other functions?
How about functions that call other functions?

```python
def sumSquare(a, b):
    return square(a) + square(b)
```
Review:
Function Frame Model

• How about functions that call other functions?

```python
def sumSquare(a, b):
    return square(a) + square(b)
```

• What happens when we call `sumSquare(5, 3)`?
def sumSquare(a, b):
    return square(a) + square(b)

>>> sumSquare(5,3)
def sumSquare(a, b):
    return square(a) + square(b)

>>> sumSquare(5,3)
def sumSquare(a, b):
    return square(a) + square(b)

>>> sumSquare(5, 3)
def sumSquare(a, b):
    return square(a) + square(b)

>>> sumSquare(5, 3)

sumSquare(5, 3)

return square(a) + square(b)
def sumSquare(a, b):
    return square(a) + square(b)

>>> sumSquare(5,3)
def sumSquare(a, b):
    return square(a) + square(b)

>>> sumSquare(5, 3)
25

square(5)

return x * x
def sumSquare(a, b):
    return square(a) + square(b)

>>> sumSquare(5,3)

25
```python
def sumSquare(a, b):
    return square(a) + square(b)

>>> sumSquare(5, 3)

25 + 9 = 34
```

**Note:**

- **square(5)**:
  - `x = 5`
  - `return x * x`
  - `25`

- **square(3)**:
  - `x = 3`
  - `return x * x`
  - `9`
def sumSquare(a, b):
    return square(a) + square(b)

>>> sumSquare(5, 3)
34
Function Frame Model to Understand countDown
def countDown(n):
    '''Prints ints from n down to 1'''
    if n < 1:
        pass # do nothing
    else:
        print(n)
        countDown(n-1)
def countDown(n):
    '''Prints ints from n down to 1'''
    if n < 1:
        pass # do nothing
    else:
        print(n)
        countDown(n-1)

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

In[2] countDown(4)
    4
    3
    2
    1
```python
>>> countDown(3)

n 3

if n < 1:
    pass # do nothing
else:
    print(n)
    countDown(n-1)
```
```python
countDown(3)

n 3

if n < 1:
    pass # do nothing
else:
    print(n)
    countDown(n-1)

>>> countDown(3)
```
```python
countDown(3)

n 3

if n < 1:
    pass  # do nothing
else:
    print(n)
    countDown(n-1)

>>> countDown(3)
3
```
```python
def countDown(n):
    if n < 1:
        pass  # do nothing
    else:
        print(n)
        countDown(n-1)

>>> countDown(3)
3
```
countDown(3)

n  3

if n < 1:
    pass # do nothing
else:
    print(n)
    countDown(n-1)

countDown(2)

n  2

if n < 1:
    pass # do nothing
else:
    print(n)
    countDown(n-1)

>>> countDown(3)
3
```python
countDown(3)

if n < 1:
    pass  # do nothing
else:
    print(n)
    countDown(n-1)

countDown(2)

if n < 1:
    pass  # do nothing
else:
    print(n)
    countDown(n-1)

>>> countDown(3)
3
```
```python
countDown(3)

n 3

if n < 1:
    pass # do nothing
else:
    print(n)
    countDown(n-1)

countDown(2)

n 2

if n < 1:
    pass # do nothing
else:
    print(n)
    countDown(n-1)

>>> countDown(3)
3
2
```
```python
countDown(3)

n = 3
if n < 1:
    pass # do nothing
else:
    print(n)
    countDown(n-1)

>>> countDown(3)
3
2

countDown(2)

n = 2
if n < 1:
    pass # do nothing
else:
    print(n)
    countDown(n-1)
```
```python
def countDown(n):
    if n < 1:
        pass  # do nothing
    else:
        print(n)
        countDown(n-1)
```

```python
>>> countDown(3)
3
2
```
```python
def countDown(n):
    if n < 1:
        pass  # do nothing
    else:
        print(n)
        countDown(n-1)

>>> countDown(3)
3
2
```
def countDown(n):
    if n < 1:
        pass  # do nothing
    else:
        print(n)
        countDown(n-1)

countDown(3)
```python
def countDown(n):
    if n < 1:
        pass  # do nothing
    else:
        print(n)
        countDown(n-1)

countDown(3)
```

Output:
```
3
2
1
```
```python
def countDown(n):
    if n < 1:
        pass  # do nothing
    else:
        print(n)
        countDown(n-1)
```
```python
countDown(3)

n = 3

if n < 1:
    pass  # do nothing
else:
    print(n)
    countDown(n-1)

countDown(2)

n = 2

if n < 1:
    pass  # do nothing
else:
    print(n)
    countDown(n-1)

countDown(1)

n = 1

if n < 1:
    pass  # do nothing
else:
    print(n)
    countDown(n-1)

countDown(0)

n = 0

if n < 1:
    pass  # do nothing
else:
    print(n)
    countDown(n-1)

Base case reached!

>>> countDown(3)
3
2
1
countDown(3)

n 3
if n < 1:
    pass # do nothing
else:
    print(n)
    countDown(n-1)

Base case reached!

countDown(2)

n 2
if n < 1:
    pass # do nothing
else:
    print(n)
    countDown(n-1)

countDown(1)

n 1
if n < 1:
    pass # do nothing
else:
    print(n)
    countDown(n-1)

countDown(0)

n 0
if n < 1:
    pass # do nothing
else:
    print(n)
    countDown(n-1)

Implicit return

>>> countDown(3)
3
2
1
```python
def countDown(n):
    if n < 1:
        pass  # do nothing
    else:
        print(n)
        countDown(n-1)

>>> countDown(3)
3
2
1
```

Base case reached!
```python
def countDown(n):
    if n < 1:
        pass  # do nothing
    else:
        print(n)
        countDown(n-1)

>>> countDown(3)
3
2
1
Base case reached!
```

```python
countDown(3)

n = 3

if n < 1:
    pass  # do nothing
else:
    print(n)
    countDown(n-1)

countDown(2)

n = 2

if n < 1:
    pass  # do nothing
else:
    print(n)
    countDown(n-1)

countDown(1)

n = 1

if n < 1:
    pass  # do nothing
else:
    print(n)
    countDown(n-1)

countDown(0)

n = 0

if n < 1:
    pass  # do nothing
else:
    print(n)
    countDown(n-1)

```

Base case reached!

```
```python
def countDown(n):
    if n < 1:
        pass  # do nothing
    else:
        print(n)
        countDown(n-1)
```

```
>>> countDown(3)
3
2
1
```

Base case reached!
```python
def countDown(n):
    if n < 1:
        pass  # do nothing
    else:
        print(n)
        countDown(n-1)

>>> countDown(3)
3
2
1
```

```python
def countDown(n):
    if n < 1:
        pass  # do nothing
    else:
        print(n)
        countDown(n-1)
```

>>> countDown(3)
3
2
1

Base case reached!
```python
countDown(3)

n = 3
if n < 1:
    pass # do nothing
else:
    print(n)
    countDown(n-1)

countDown(2)

n = 2
if n < 1:
    pass # do nothing
else:
    print(n)
    countDown(n-1)

countDown(1)

n = 1
if n < 1:
    pass # do nothing
else:
    print(n)
    countDown(n-1)

countDown(0)

n = 0
if n < 1:
    pass # do nothing
else:
    print(n)
    countDown(n-1)

Base case reached!

>>> countDown(3)
3
2
1
```

Implicit return

Implicit return

Implicit return
```python
def countDown(n):
    if n < 1:
        pass  # do nothing
    else:
        print(n)
        countDown(n-1)

>>> countDown(3)
3
2
1
Base case reached!
```
```python
def countDown(n):
    if n < 1:
        pass  # do nothing
    else:
        print(n)
        countDown(n-1)

countDown(3)

>>> countDown(3)
3
2
1
```

Base case reached!
```python
def countDown(n):
    if n < 1:
        pass  # do nothing
    else:
        print(n)
        countDown(n-1)
```

>>> countDown(3)
3
2
1
Base case reached!
```python
countDown(3)

n 3
if n < 1:
    pass # do nothing
else:
    print(n)
    countDown(n-1)

countDown(2)

n 2
if n < 1:
    pass # do nothing
else:
    print(n)
    countDown(n-1)

countDown(1)

n 1
if n < 1:
    pass # do nothing
else:
    print(n)
    countDown(n-1)

countDown(0)

n 0
if n < 1:
    pass # do nothing
else:
    print(n)
    countDown(n-1)

Base case reached!

>>> countDown(3)
3
2
1
```
More Recursion: countUp
countUp(n)
countUp(n)

- Recursive function that prints integers from 1 up to n \((\text{without using any loops})\)
countUp(n)

- Recursive function that prints integers from 1 up to n (without using any loops)
Function Frame Model to Understand countUp
countUp(3)

```
if n < 1:
    pass # do nothing
else:
    countUp(n-1)
    print(n)
```

```python
>>> countUp(3)
```
countUp(3)

if n < 1:
    pass # do nothing
else:
    countUp(n-1)
    print(n)

>>> countUp(3)
countUp(3)

n 3

if n < 1:
    pass # do nothing
else:
    countUp(n-1)
    print(n)

countUp(2)

n 2

if n < 1:
    pass # do nothing
else:
    countUp(n-1)
    print(n)

>>> countUp(3)
```python
def countUp(n):
    if n < 1:
        pass  # do nothing
    else:
        countUp(n-1)
        print(n)

>>> countUp(3)
```
```python
countUp(3)

n = 3
if n < 1:
    pass # do nothing
else:
    countUp(n-1)
    print(n)

countUp(2)

n = 2
if n < 1:
    pass # do nothing
else:
    countUp(n-1)
    print(n)

countUp(1)

n = 1
if n < 1:
    pass # do nothing
else:
    countUp(n-1)
    print(n)

>>> countUp(3)
```
countUp(3)

```
if n < 1:
    pass  # do nothing
else:
    countUp(n-1)
    print(n)
```

countUp(2)

```
if n < 1:
    pass  # do nothing
else:
    countUp(n-1)
    print(n)
```

countUp(1)

```
if n < 1:
    pass  # do nothing
else:
    countUp(n-1)
    print(n)
```

```
>>> countUp(3)
```

```python
>>> countUp(3)
3
2
1
0
```
```python
def countUp(n):
    if n < 1:
        pass  # do nothing
    else:
        countUp(n-1)
        print(n)

>>> countUp(3)
```

Base case reached!

```python
>>> countUp(0)
```
```python
countUp(3)

n 3
if n < 1:
    pass # do nothing
else:
    countUp(n-1)
    print(n)

countUp(2)

n 2
if n < 1:
    pass # do nothing
else:
    countUp(n-1)
    print(n)

countUp(1)

n 1
if n < 1:
    pass # do nothing
else:
    countUp(n-1)
    print(n)

countUp(0)

n 0
if n < 1:
    pass # do nothing
else:
    countUp(n-1)
    print(n)

Base case reached!
Implicit return

>>> countUp(3)
```
countUp(3)

n 3

if n < 1:
    pass # do nothing
else:
    countUp(n-1)
    print(n)

countUp(2)

n 2

if n < 1:
    pass # do nothing
else:
    countUp(n-1)
    print(n)

countUp(1)

n 1

if n < 1:
    pass # do nothing
else:
    countUp(n-1)
    print(n)

countUp(0)

n 0

if n < 1:
    pass # do nothing
else:
    countUp(n-1)
    print(n)

>>> countUp(3)

Base case reached!
countUp(3)

n 3

if n < 1:
    pass # do nothing
else:
    countUp(n-1)
    print(n)

countUp(2)

n 2

if n < 1:
    pass # do nothing
else:
    countUp(n-1)
    print(n)

countUp(1)

n 1

if n < 1:
    pass # do nothing
else:
    countUp(n-1)
    print(n)

countUp(0)

n 0

if n < 1:
    pass # do nothing
else:
    countUp(n-1)
    print(n)

Base case reached!

>>> countUp(3)
```
countUp(3)

n 3
if n < 1:
    pass # do nothing
else:
    countUp(n-1)
    print(n)

countUp(2)

n 2
if n < 1:
    pass # do nothing
else:
    countUp(n-1)
    print(n)

countUp(1)

n 1
if n < 1:
    pass # do nothing
else:
    countUp(n-1)
    print(n)

n 0
if n < 1:
    pass # do nothing
else:
    countUp(n-1)
    print(n)

>>> countUp(3)
```

Base case reached!
```python
def countUp(n):
    if n < 1:
        pass  # do nothing
    else:
        countUp(n-1)
        print(n)

# Test cases
print(countUp(3))  # Output: 1
print(countUp(2))  # Output: 2
print(countUp(1))  # Output: 1
print(countUp(0))  # Output: 0
```

Base case reached!
```python
def countUp(n):
    if n < 1:
        pass  # do nothing
    else:
        countUp(n-1)
        print(n)
```

```
>>> countUp(3)
1
```

Base case reached!

Implicit return
countUp(3)

```
if n < 1:
    pass # do nothing
else:
    countUp(n-1)
    print(n)
```

countUp(2)

```
if n < 1:
    pass # do nothing
else:
    countUp(n-1)
    print(n)
```

Base case reached!

countUp(1)

```
if n < 1:
    pass # do nothing
else:
    countUp(n-1)
    print(n)
```

countUp(0)

```
if n < 1:
    pass # do nothing
else:
    countUp(n-1)
    print(n)
```

Implicit return

```python
>>> countUp(3)
1
```
```python
def countUp(n):
    if n < 1:
        pass  # do nothing
    else:
        countUp(n-1)
        print(n)

>>> countUp(3)
1
```

Base case reached!

Implicit return
def countUp(n):
    if n < 1:
        pass  # do nothing
    else:
        countUp(n-1)
        print(n)

>>> countUp(3)
1
2

Base case reached!
```
countUp(3)

n 3
if n < 1:
    pass # do nothing
else:
    countUp(n-1)
    print(n)

countUp(2)

n 2
if n < 1:
    pass # do nothing
else:
    countUp(n-1)
    print(n)

countUp(1)

n 1
if n < 1:
    pass # do nothing
else:
    countUp(n-1)
    print(n)

countUp(0)

n 0
if n < 1:
    pass # do nothing
else:
    countUp(n-1)
    print(n)

>>> countUp(3)
1
2
3
```

Base case reached!
Implicit return
Implicit return
countUp(3)

```
if n < 1:
    pass  # do nothing
else:
    countUp(n-1)
    print(n)
```

countUp(2)

```
if n < 1:
    pass  # do nothing
else:
    countUp(n-1)
    print(n)
```

CountUp(1)

```
if n < 1:
    pass  # do nothing
else:
    countUp(n-1)
    print(n)
```

CountUp(0)

```
if n < 1:
    pass  # do nothing
else:
    countUp(n-1)
    print(n)
```

Base case reached!

```
>>> countUp(3)
1
2
```

Implicit return
`countUp(3)`

```python
if n < 1:
    pass  # do nothing
else:
    countUp(n-1)
    print(n)
```

**Base case reached!**

```python
>>> countUp(3)
1
2
```

```
countUp(2)

if n < 1:
    pass  # do nothing
else:
    countUp(n-1)
    print(n)
```

```
countUp(1)

if n < 1:
    pass  # do nothing
else:
    countUp(n-1)
    print(n)
```

```
countUp(0)

if n < 1:
    pass  # do nothing
else:
    countUp(n-1)
    print(n)
```

**Implicit return**
```python
def countUp(n):
    if n < 1:
        pass  # do nothing
    else:
        countUp(n-1)
        print(n)

countUp(3)
```

```
>>> countUp(3)
1
2
```

```
Base case reached!
```

```
Implicit return
```

```
countUp(2)
```

```
countUp(1)
```

```
countUp(0)
```

```
>>> countUp(3)
1
2
```

```
Base case reached!
```

```
Implicit return
```
countUp(3)

n 3
if n < 1:
    pass # do nothing
else:
    countUp(n-1)
    print(n)

countUp(2)

n 2
if n < 1:
    pass # do nothing
else:
    countUp(n-1)
    print(n)

countUp(1)

n 1
if n < 1:
    pass # do nothing
else:
    countUp(n-1)
    print(n)

countUp(0)

n 0
if n < 1:
    pass # do nothing
else:
    countUp(n-1)
    print(n)

>>> countUp(3)
1
2
3

Base case reached!

Implicit return
```python
countUp(3)

n 3
if n < 1:
    pass  # do nothing
else:
    countUp(n-1)
print(n)

countUp(2)

n 2
if n < 1:
    pass  # do nothing
else:
    countUp(n-1)
print(n)

countUp(1)

n 1
if n < 1:
    pass  # do nothing
else:
    countUp(n-1)
print(n)

countUp(0)

n 0
if n < 1:
    pass  # do nothing
else:
    countUp(n-1)
print(n)

>>> countUp(3)
1
2
3
Base case reached!
```
```python
countUp(3)
if n < 1:
    pass  # do nothing
else:
    countUp(n-1)
    print(n)

countUp(2)
if n < 1:
    pass  # do nothing
else:
    countUp(n-1)
    print(n)

countUp(1)
if n < 1:
    pass  # do nothing
else:
    countUp(n-1)
    print(n)

countUp(0)
if n < 1:
    pass  # do nothing
else:
    countUp(n-1)
    print(n)

Base case reached!
```
def countUp(n):
    if n < 1:
        pass # do nothing
    else:
        countUp(n-1)
        print(n)

countUp(3)

>>> countUp(3)
1
2
3

Base case reached!
More Recursion: countDownUp
countDownUp(n)
countDownUp(n)

- Recursive function that prints integers from \( n \) down to 1 and back up from 1 up to \( n \) \((\text{without using any loops})\)
countDownUp(n)

- Recursive function that prints integers from \( n \) down to 1 and back up from 1 up to \( n \) (without using any loops)

```python
In[1] countDownUp(5)
5
4
3
2
1
1
1
2
3
4
5
```
**countDownUp(n)**

- Recursive function that prints integers from $n$ down to 1 and back up from 1 up to $n$ (*without using any loops*)

<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>
Recursive Patterns
triangle(n, c)

>>> triangle(10, '*')

************
************
************
************
************
************
*****
***
*
triangleAlt(n, c1, c2)

```python
>>> triangleAlt(10, '@', '#')
```

```
@@@@@@@@@@@@
###########
@@@@@@@@@@
###########
@@@@@@@@@@
############
@@@@@@@@@@
###
@@@@@@@@@
#
```
Function Frame Model:
triangleAlt(3, '*', '^')
```python
triangleAlt(3, '*', '^')

n 3  c1 '*'  c2 '^'

if n < 1:
    pass # do nothing
else:
    print(n * c1)
    triangleAlt(n-1, c2, c1)

>>> triangleAlt(3, '*', '^')
```

```python
n 3  c1 '*'  c2 '^'

if n < 1:
    pass # do nothing
else:
    print(n * c1)
    triangleAlt(n-1, c2, c1)

>>> triangleAlt(3, '*', '^')
```
triangleAlt(3, '*', '^')

if n < 1:
    pass  # do nothing
else:
    print(n * c1)
    triangleAlt(n-1, c2, c1)

>>> triangleAlt(3, '*', '^')
triangleAlt(3, '*', '^')

```
if n < 1:
    pass # do nothing
else:
    print(n * c1)
triangleAlt(n-1, c2, c1)
```

```python
>>> triangleAlt(3, '*', '^')
***
```
triangleAlt(3, '*', '^')

```
if n < 1:
    pass # do nothing
else:
    print(n * c1)
    triangleAlt(n-1, c2, c1)
```
triangleAlt(3, '*', '^')

n 3  c1 '*'  c2 '^'
if n < 1:
    pass # do nothing
else:
    print(n * c1)
    triangleAlt(n-1, c2, c1)

triangleAlt(2, '*', '^')

n 2  c1 '^'  c2 '*'
if n < 1:
    pass # do nothing
else:
    print(n * c1)
    triangleAlt(n-1, c2, c1)

>>> triangleAlt(3, '*', '^')
***
^ ^
```python
triangleAlt(3, '*', '^')

if n < 1:
    pass # do nothing
else:
    print(n * c1)
    triangleAlt(n-1, c2, c1)

>>> triangleAlt(3, '*', '^')
***
^^
```
triangleAlt(3, '*', '^')

if n < 1:
    pass # do nothing
else:
    print(n * c1)
    triangleAlt(n-1, c2, c1)

>>> triangleAlt(3, '*', '^')
***
^^
*

triangleAlt(2, '*', '^')

if n < 1:
    pass # do nothing
else:
    print(n * c1)
    triangleAlt(n-1, c2, c1)

triangleAlt(1, '*', '^')

if n < 1:
    pass # do nothing
else:
    print(n * c1)
    triangleAlt(n-1, c2, c1)
Class Exercise

It's your Turn!
triangleDownUp(n, c1, c2)

```python
>>> triangleDownUp(8, '@', '#')
```

```
@@@@@@@@@
######
@@@@@@@
####
@@
#
#  
@@
###
####
@@@#
#   
@@@
#   
@@@
```
Acknowledgments

These slides have been adapted from:

- http://cs111.wellesley.edu/spring19 and