Recursion
Class outline:

- Recursive functions
- Recursion in environment diagrams
- Mutual recursion
- Recursion vs. iteration
Recursive functions

A function is **recursive** if the body of that function calls itself, either directly or indirectly.

Recursive functions often operate on increasingly smaller instances of a problem.

Circle Limit, by M.C. Escher
Summing digits

\[2 + 0 + 2 + 1 = 5\]
Summing digits

2 + 0 + 2 + 1 = 5

Fun fact: The sum of the digits in a multiple of 9 is also divisible by 9.

9 * 82 = 738

7 + 3 + 8 = 18
The problems within the problem

The sum of the digits of 6 is simply 6. Generally: the sum of any one-digit non-negative number is that number.
The problems within the problem

The sum of the digits of 6 is simply 6. Generally: the sum of any one-digit non-negative number is that number.

The sum of the digits of 2021 is:

2 0 2 1

Sum of these digits + This digit

Generally: the sum of a number is the sum of the first digits (number // 10), plus the last digit (number % 10).
Summing digits without a loop

```python
def sum_digits(n):
    """Return the sum of the digits of positive integer n.
    >>> sum_digits(6)
    6
    >>> sum_digits(2021)
    5
    """
```
def sum_digits(n):
    """Return the sum of the digits of positive integer n.
    >>> sum_digits(6)
    6
    >>> sum_digits(2021)
    5
    ""
    if n < 10:
        return n
    else:
        all_but_last = n // 10
        last = n % 10
        return sum_digits(all_but_last) + last
Anatomy of a recursive function

- **Base case**: Evaluated without a recursive call (the smallest subproblem).
- **Recursive case**: Evaluated with a recursive call (breaking down the problem further)
- **Conditional statement** to decide if it's a base case

```python
def sum_digits(n):
    if n < 10:
        return n
    else:
        all_but_last = n // 10
        last = n % 10
        return sum_digits(all_but_last) + last
```
Anatomy of a recursive function

- **Base case**: Evaluated without a recursive call (the smallest subproblem).
- **Recursive case**: Evaluated with a recursive call (breaking down the problem further)
- **Conditional statement** to decide if it's a base case

```python
def sum_digits(n):
    if n < 10:  # BASE CASE
        return n
    else:
        all_but_last = n // 10
        last = n % 10
        return sum_digits(all_but_last) + last
```
Anatomy of a recursive function

- **Base case**: Evaluated without a recursive call (the smallest subproblem).
- **Recursive case**: Evaluated with a recursive call (breaking down the problem further)
- **Conditional statement** to decide if it's a base case

```python
def sum_digits(n):
    if n < 10:  # BASE CASE
        return n
    else:       # RECURSIVE CASE
        all_but_last = n // 10
        last = n % 10
        return sum_digits(all_but_last) + last
```
Visualizing recursion
Recursive factorial

The factorial of a number is defined as:

\[
 n! = \begin{cases} 
 1 & \text{if } n = 0 \\
 n \cdot (n - 1)! & \text{otherwise}
\end{cases}
\]

```python
def fact(n):
    """
    >>> fact(0)
    1
    >>> fact(4)
    24
    """
```
Recursive factorial

The factorial of a number is defined as:

\[ n! = \begin{cases} 
1 & \text{if } n = 0 \\
 n \cdot (n - 1)! & \text{otherwise}
\end{cases} \]

def fact(n):
    """
    >>> fact(0)
    1
    >>> fact(4)
    24
    """
    if n == 0:
        return 1
    else:
        return n * fact(n-1)
Recursive call visualization

def fact(n):
    if n == 0:
        return 1
    else:
        return n * fact(n-1)

fact(3)
fact(3)
Recursive call visualization

def fact(n):
    if n == 0:
        return 1
    else:
        return n * fact(n-1)

fact(3)
Recursive call visualization

def fact(n):
    if n == 0:
        return 1
    else:
        return n * fact(n-1)

fact(3)
def fact(n):
    if n == 0:
        return 1
    else:
        return n * fact(n-1)

fact(3)
fact(3) → fact(2) → fact(1) → fact(0)
def fact(n):
    if n == 0:
        return 1
    else:
        return n * fact(n-1)

fact(3)
fact(1)

1

fact(0)
def fact(n):
    if n == 0:
        return 1
    else:
        return n * fact(n-1)

fact(3)
def fact(n):
    if n == 0:
        return 1
    else:
        return n * fact(n-1)

fact(3)
Recursive call visualization

```python
def fact(n):
    if n == 0:
        return 1
    else:
        return n * fact(n-1)

fact(3)
```
fact(3)

fact(2)

fact(1)

fact(0)
Recursion in environment diagrams

```python
def fact(n):
    if n == 0:
        return 1
    else:
        return n * fact(n-1)

fact(3)

• The same function `fact` is called multiple times
• Different frames keep track of the different arguments in each call
• What `n` evaluates to depends upon the current environment
• Each call to `fact` solves a simpler problem than the last: smaller `n`
```
Global frame
    fact → func fact[parent=Global]

f1:

Return
value______
f2:

Return
value______

f3:

Return
value______

f4:

______
Recursion in environment diagrams

```python
def fact(n):
    if n == 0:
        return 1
    else:
        return n * fact(n-1)

fact(3)
```

- The same function `fact` is called multiple times
- Different frames keep track of the different arguments in each call
- What `n` evaluates to depends upon the current environment
- Each call to `fact` solves a simpler problem than the last: smaller `n`

Global frame

```
fact → func fact[parent=Global]
```

f1: fact[parent=Global]

```
n 3

| Return | 6 |
```
f2: fact[parent=Global]
   n 2
   Return 2
   value

f3: fact[parent=Global]
   n 1
   Return 1
   value

f4: fact[parent=Global]
   n 0
   Return 1
   value
Verifying recursive functions
Falling dominoes

If a million dominoes are equally spaced out and we tip the first one, will they all fall?

1. Verify that one domino will fall, if tipped
2. Assume that any given domino falling over will tip the next one over
3. Verify that tipping the first domino tips over the next one
Falling dominoes

If a million dominoes are equally spaced out and we tip the first one, will they all fall?

1. Verify that one domino will fall, if tipped
2. Assume that any given domino falling over will tip the next one over
3. Verify that tipping the first domino tips over the next one
The recursive leap of faith

```python
def fact(n):
    """Returns the factorial of N."""
    if n == 0:
        return 1
    else:
        return n * fact(n-1)
```

Is `fact` implemented correctly?

1. Verify the base case
2. Treat `fact` as a functional abstraction!
3. Assume that `fact(n-1)` is correct (← the leap!)
4. Verify that `fact(n)` is correct
The recursive elf's promise

Imagine we're trying to compute 5!

We ask ourselves, "If I somehow knew how to compute 4!, could I compute 5!?"

Credit: FuschiaKnight, r/compsci
The recursive elf's promise

Imagine we're trying to compute $5!$

We ask ourselves, "If I somehow knew how to compute $4!$, could I compute $5!$?"

Yep, $5! = 5 \times 4!$

Credit: FuschiaKnight, r/compsci
The recursive elf's promise

Imagine we're trying to compute 5!

We ask ourselves, "If I somehow knew how to compute 4!, could I compute 5!?"

Yep, 5! = 5 * 4!

♀ The `fact()` function promises, "hey friend, tell you what, while you're working hard on 5!, I'll compute 4! for you, and you can finish it off!"

Credit: FuschiaKnight, r/compsci
Mutual recursion
The Luhn algorithm

Used to verify that a credit card numbers is valid.

1. From the rightmost digit, which is the check digit, moving left, double the value of every second digit; if product of this doubling operation is greater than 9 (e.g., 7 * 2 = 14), then sum the digits of that product (e.g., 14: 1 + 4 = 5)

2. Take the sum of all the digits

<table>
<thead>
<tr>
<th>Original</th>
<th>1</th>
<th>3</th>
<th>8</th>
<th>7</th>
<th>4</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processed</td>
<td>[Gray]</td>
<td>[Gray]</td>
<td>[Gray]</td>
<td>[Gray]</td>
<td>[Gray]</td>
<td>[Gray]</td>
</tr>
</tbody>
</table>
The Luhn algorithm

Used to verify that a credit card numbers is valid.

1. From the rightmost digit, which is the check digit, moving left, double the value of every second digit; if product of this doubling operation is greater than 9 (e.g., 7 * 2 = 14), then sum the digits of that product (e.g., 14: 1 + 4 = 5)

2. Take the sum of all the digits

<table>
<thead>
<tr>
<th>Original</th>
<th>1</th>
<th>3</th>
<th>8</th>
<th>7</th>
<th>4</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processed</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3</td>
</tr>
</tbody>
</table>
The Luhn algorithm

Used to verify that a credit card numbers is valid.

1. From the rightmost digit, which is the check digit, moving left, double the value of every second digit; if product of this doubling operation is greater than 9 (e.g., 7 * 2 = 14), then sum the digits of that product (e.g., 14: 1 + 4 = 5)

2. Take the sum of all the digits

<table>
<thead>
<tr>
<th>Original</th>
<th>1</th>
<th>3</th>
<th>8</th>
<th>7</th>
<th>4</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processed</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>8</td>
<td>3</td>
</tr>
</tbody>
</table>
The Luhn algorithm

Used to verify that a credit card numbers is valid.

1. From the rightmost digit, which is the check digit, moving left, double the value of every second digit; if product of this doubling operation is greater than 9 (e.g., 7 * 2 = 14), then sum the digits of that product (e.g., 14: 1 + 4 = 5)

2. Take the sum of all the digits

<table>
<thead>
<tr>
<th>Original</th>
<th>1</th>
<th>3</th>
<th>8</th>
<th>7</th>
<th>4</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processed</td>
<td></td>
<td></td>
<td></td>
<td>7</td>
<td>8</td>
<td>3</td>
</tr>
</tbody>
</table>
The Luhn algorithm

Used to verify that a credit card numbers is valid.

1. From the rightmost digit, which is the check digit, moving left, double the value of every second digit; if product of this doubling operation is greater than 9 (e.g., 7 × 2 = 14), then sum the digits of that product (e.g., 14: 1 + 4 = 5)

2. Take the sum of all the digits

<table>
<thead>
<tr>
<th>Original</th>
<th>1</th>
<th>3</th>
<th>8</th>
<th>7</th>
<th>4</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processed</td>
<td></td>
<td></td>
<td>1+6=7</td>
<td></td>
<td>8</td>
<td>3</td>
</tr>
</tbody>
</table>
The Luhn algorithm

Used to verify that a credit card numbers is valid.

1. From the rightmost digit, which is the check digit, moving left, double the value of every second digit; if product of this doubling operation is greater than 9 (e.g., 7 * 2 = 14), then sum the digits of that product (e.g., 14: 1 + 4 = 5)

2. Take the sum of all the digits

<table>
<thead>
<tr>
<th>Original</th>
<th>1</th>
<th>3</th>
<th>8</th>
<th>7</th>
<th>4</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processed</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td></td>
<td>7</td>
<td></td>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>
The Luhn algorithm

Used to verify that a credit card numbers is valid.

1. From the rightmost digit, which is the check digit, moving left, double the value of every second digit; if product of this doubling operation is greater than 9 (e.g., 7 \* 2 = 14), then sum the digits of that product (e.g., 14: 1 + 4 = 5)

2. Take the sum of all the digits

<table>
<thead>
<tr>
<th>Original</th>
<th>1</th>
<th>3</th>
<th>8</th>
<th>7</th>
<th>4</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processed</td>
<td>2</td>
<td>3</td>
<td>1+6=7</td>
<td>7</td>
<td>8</td>
<td>3</td>
</tr>
</tbody>
</table>
The Luhn algorithm

Used to verify that a credit card numbers is valid.

1. From the rightmost digit, which is the check digit, moving left, double the value of every second digit; if product of this doubling operation is greater than 9 (e.g., 7 * 2 = 14), then sum the digits of that product (e.g., 14: 1 + 4 = 5)

2. Take the sum of all the digits

<table>
<thead>
<tr>
<th>Original</th>
<th>1</th>
<th>3</th>
<th>8</th>
<th>7</th>
<th>4</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processed</td>
<td>2</td>
<td>3</td>
<td>1+6=7</td>
<td>7</td>
<td>8</td>
<td>3</td>
</tr>
</tbody>
</table>

The Luhn sum of a valid credit card number is a multiple of 10
Calculating the Luhn sum

Let's start with...

```python
def sum_digits(n):
    if n < 10:
        return n
    else:
        last = n % 10
        all_but_last = n // 10
        return last + sum_digits(all_but_last)

def luhn_sum(n):
    '''Returns the Luhn sum for the positive number N.
    >>> luhn_sum(2)
    2
    >>> luhn_sum(32)
    8
    >>> luhn_sum(5105105105105100)
    20
    '''
```

def luhn_sum(n):
    if n < 10:
        return n
    else:
        last = n % 10
        all_but_last = n // 10
        return last + luhn_sum_double(all_but_last)

def luhn_sum_double(n):
    last = n % 10
    all_but_last = n // 10
    luhn_digit = sum_digits(last * 2)
    if n < 10:
        return luhn_digit
    else:
        return luhn_digit + luhn_sum(all_but_last)
Recursion and Iteration
Recursion vs. iteration

<table>
<thead>
<tr>
<th>Using recursion:</th>
<th>Using iteration:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>def</strong> fact(n):</td>
<td><strong>def</strong> fact(n):</td>
</tr>
<tr>
<td>if n == 0:</td>
<td>total = 1</td>
</tr>
<tr>
<td>return 1</td>
<td>k = 1</td>
</tr>
<tr>
<td>else:</td>
<td>while k &lt;= n:</td>
</tr>
<tr>
<td>return n * fact(n-1)</td>
<td>total *= k</td>
</tr>
<tr>
<td></td>
<td>k += 1</td>
</tr>
<tr>
<td></td>
<td>return total</td>
</tr>
</tbody>
</table>

Math: \( n! = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot (n - 1)! & \text{otherwise} \end{cases} \)

\( n! = \prod_{k=1}^{n} k \)

Names: **fact**, **n** **fact**, **n**, **total**, **k**
Converting recursion to iteration

Can be tricky: Iteration is a special case of recursion.

Figure out what state must be maintained by the iterative function.

```python
def sum_digits(n):
    if n < 10:
        return n
    else:
        all_but_last = n // 10
        last = n % 10
        return sum_digits(all_but_last) + last
```
Converting recursion to iteration

Can be tricky: Iteration is a special case of recursion.

Figure out what state must be maintained by the iterative function.

```python
def sum_digits(n):
    if n < 10:
        return n
    else:
        all_but_last = n // 10
        last = n % 10
        return sum_digits(all_but_last) + last
```

```python
def sum_digits(n):
    digit_sum = 0
    while n >= 10:
        last = n % 10
        n = n // 10
        digit_sum += last
    return digit_sum
```
Converting iteration to recursion

More formulaic: Iteration is a special case of recursion. The state of an iteration can be passed as arguments.

def sum_digits(n):
    digit_sum = 0
    while n >= 10:
        last = n % 10
        n = n // 10
        digit_sum += last
    return digit_sum
Converting iteration to recursion

More formulaic: Iteration is a special case of recursion.
The state of an iteration can be passed as arguments.

def sum_digits(n):
    digit_sum = 0
    while n >= 10:
        last = n % 10
        n = n // 10
        digit_sum += last
    return digit_sum

def sum_digits(n, digit_sum):
    if n == 0:
        return digit_sum
    else:
        last = n % 10
        all_but_last = n // 10
        return sum_digits(all_but_last, digit_sum + last)