Recursion Examples
Recursion (review)
Recursion (review)

Scenario: You are waiting in line for a concert. You can't see the front of the line, but you want to know your place in the line.

The person at the front, knows they are at the front!
You ask the person in front of you: “what is your place in the line?”
When the person in front of you figures it out and tells you, add one to that answer.

Base case
Recursive call
Use the solution to the smaller problem
Iteration vs. Recursion

- Iteration and recursion are somewhat related.
- Converting iteration to recursion is formulaic, but converting recursion to iteration can be more tricky.

**Iterative**

```python
def fact_iter(n):
    total, k = 1, 1
    while k <= n:
        total, k = total \* k, k + 1
    return total
```

\[ n! = \prod_{k=1}^{n} k \]

Names: n, total, k, fact_iter

**Recursive**

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

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

Names: n, fact
Sum Digits

Let’s implement a recursive function to sum all the digits of `n`. Assume `n` is positive.

```python
def sum_digits(n):
    """Calculates the sum of the digits `n`.
    >>> sum_digits(8)
    8
    >>> sum_digits(18)
    9
    >>> sum_digits(2018)
    11
    ""
    "*** YOUR CODE HERE ***"
```

1. One or more **base cases**
2. One or more **recursive calls** with simpler arguments.
3. **Using the recursive call** to solve our larger problem.
def sum_digits(n):
    """Calculates the sum of the digits n
    >>> sum_digits(8)
    8
    >>> sum_digits(18)
    9
    >>> sum_digits(2018)
    11
    """

    if n < 10:
        return n
    else:
        all_but_last, last = n // 10, n % 10
        return sum_digits(all_but_last) + last
Order of Recursive Calls
Cascade

Goal: Print out a cascading tree of a positive integer \( n \).

>>> cascade(486)
486
48
4
48
486

>>> cascade(48)
48
4
48

>>> cascade(4)
4

Ideas:
- If \( n \) is a single digit, just print it out!
- Otherwise, let \( \text{cascade}(n \div 10) \) take care of the middle and \( \text{print}(n) \) around it

```python
def cascade(n):
    if n < 10:
        print(n)
    else:
        print(n)
cascade(n \div 10)
print(n)
```
The Cascade Function

```python
def cascade(n):
    if n < 10:
        print(n)
    else:
        print(n)
        cascade(n // 10)
    print(n)

cascade(123)
```

Each cascade frame is from a different call to `cascade`. Until the `Return value` appears, that call has not completed.

Any statement can appear before or after the recursive call.
Two Implementations of Cascade

- If two implementations are equally clear, then shorter is usually better.
- In this case, the longer implementation might be more clear.
- When learning to write recursive functions, put the base case/s first.
Fibonacci
## Fibonacci Sequence

<table>
<thead>
<tr>
<th>n</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>...</th>
<th>30</th>
</tr>
</thead>
<tbody>
<tr>
<td>fib(n)</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>8</td>
<td>13</td>
<td>21</td>
<td>...</td>
<td>832040</td>
</tr>
</tbody>
</table>

+ + + + + + + + +

![Fibonacci Diagram](image_url)
Fibonacci's rabbits

fib(1) == 1
fib(2) == 1
fib(3) == 2
fib(4) == 3
fib(5) == 5
fib(6) == 8

Fibonacci's rabbits
Fibonacci

Goal: Return the \( n \)th Fibonacci number.

<table>
<thead>
<tr>
<th>( n )</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>...</th>
<th>30</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{fib}(n) )</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>8</td>
<td>13</td>
<td>21</td>
<td>...</td>
<td>832040</td>
</tr>
</tbody>
</table>

Ideas:
- The first two Fibonacci numbers are known; if we ask for the 0th or 1st Fibonacci number, we know it immediately.
- Otherwise, we sum up the previous two Fibonacci numbers.

```python
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n - 2) + fib(n - 1)
```
fib(n): a tree-recursive process

Fibonacci Call Tree
Broken Fibonacci

```python
def broken_fib(n):
    if n == 0:
        return 0
    # Missing base case!
    else:
        return broken_fib(n - 2) + broken_fib(n - 1)
```

```python
>>> broken_fib(5)
Traceback (most recent call last):
  ...  
RecursionError: maximum recursion depth exceeded in comparison
```
Broken $\text{fib}(n)$

$\text{fib}(5)$

$\text{fib}(3)$

$\text{fib}(1)$

$\text{fib}(-1)$

$\text{fib}(-3)$

Never computed!
Counting Partitions
Count Partitions

**Goal**: Count the number of ways to give out \( n \) (> 0) pieces of chocolate if nobody can have more than \( m \) (> 0) pieces.

"How many different ways can I give out 6 pieces of chocolate if nobody can have more than 4 pieces?"

```python
>>> count_part(6, 4)
9
```

<table>
<thead>
<tr>
<th>Largest Piece</th>
<th>( 2 + 4 = 6 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( 1 + 1 + 4 = 6 )</td>
<td></td>
</tr>
<tr>
<td>( 3 + 3 = 6 )</td>
<td></td>
</tr>
<tr>
<td>( 1 + 2 + 3 = 6 )</td>
<td></td>
</tr>
<tr>
<td>( 1 + 1 + 1 + 3 = 6 )</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Largest Piece</th>
<th>( 2 + 2 + 2 = 6 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( 1 + 1 + 2 + 2 = 6 )</td>
<td></td>
</tr>
<tr>
<td>( 1 + 1 + 1 + 1 + 2 = 6 )</td>
<td></td>
</tr>
<tr>
<td>( 1 + 1 + 1 + 1 + 1 + 1 = 6 )</td>
<td></td>
</tr>
</tbody>
</table>
Count Partitions

2 + 4
1 + 1 + 4

3 + 3
1 + 2 + 3
1 + 1 + 1 + 3

2 + 2 + 2
1 + 1 + 2 + 2
1 + 1 + 1 + 1 + 2

1 + 1 + 1 + 1 + 1 + 2
1 + 1 + 1 + 1 + 1 + 1
Count Partitions

Ideas:
Find simpler instances of the problem

Explore two possibilities:
- Use a 4
- Don’t use a 4

Solve two simpler problems:
- \texttt{count\_part(2, 4)}
- \texttt{count\_part(6, 3)}

Sum up the results of these smaller problems!
Count Partitions

Ideas:
Find simpler instances of the problem
Explore two possibilities:
● Use a 4
● Don’t use a 4

Solve two simpler problems:
● count_part(2, 4)
● count_part(6, 3)

Sum up the results of these smaller problems!

```python
1 def count_part(n, m):
2     if
3         with_m = count_part(n-m, m)
4         wo_m = count_part(n, m-1)
5     return with_m + wo_m
```
Count Partitions

How do we know we’re done?

- If $n$ is negative, then we cannot get to a valid partition.
- If $n$ is 0, then we have arrived at a valid partition.
- If the largest piece we can use is 0, then we cannot get to a valid partition.

Diagram:

- Use a 4
  - (6, 4)
- Don’t use a 4
  - (6, 3)
- Use a 2
  - (2, 2)
- Don’t use a 2
  - (2, 1)
- Use a 3
  - (2, 3)
  - Don’t use a 3
  - (6, 3)
- Use a 4
  - (2, 4)
- Don’t use a 4
  - (6, 4)
- (-2, 4)
- (0, 2)
- (1, 1)
- (2, 0)
- (0, 1)
- (1, 0)
- 0

4 + 2 = 6

4 + 4 + ... ≠ 6

n = 0

n = -1

n = 2
Count Partitions

Ideas:
Explore two possibilities:
- Use a 4
- Don’t use a 4
Solve two simpler problems:
- `count_part(2, 4)`
- `count_part(6, 3)`

Sum up the results of these smaller problems!

How do we know we’re done?
- If `n` is 0, then we have arrived at a valid partition
- If `n` is negative, then we cannot get to a valid partition
- If the largest piece we can use is 0, then we cannot get to a valid partition

```python
def count_part(n, m):
    if n == 0:
        return 1
    elif n < 0:
        return 0
    elif m == 0:
        return 0
    else:
        with_m = count_part(n-m, m)
        wo_m = count_part(n, m-1)
        return with_m + wo_m
```
Takeaways

● Tree recursion allows you to explore different possibilities

● Oftentimes, the recursive calls for tree recursion represent different choices
  ○ One such choice is “do I use this value, or do I try another?”

● Sometimes it is easier to start with the recursive cases, and see which base cases those lead you to
If Time - Speeding Up Recursion
(Teaser for the ~Future~)
Back to Fib

```
fib(5)
fib(3)
  fib(1)
  fib(2)
    fib(0)
    fib(1)
    fib(2)
      fib(0)
      fib(1)
```
Basic Idea to Improve:

```python
def better_fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    elif already_called(better_fib(n)):
        return stored_value
    else:
        store & return better_fib(n - 2) + better_fib(n - 1)
```
Summary

● **Recursion** has three main components
  ○ **Base case/s**: The simplest form of the problem
  ○ **Recursive call/s**: Smaller version of the problem
  ○ Use the solution to the smaller version of the problem to arrive at the solution to the original problem

● When working with recursion, use *functional abstraction*: assume the recursive call gives the correct result

● **Tree recursion** makes multiple recursive calls and explores different choices

● Use doctests and your own examples to help you figure out the simplest forms and how to make the problem smaller