What is recursion?

- defining things in simpler terms of themselves
- a problem solving approach where a task is divided into simpler and simpler versions of the original task
- a function that calls itself

```python
def countdown(n):
    '''
    for i in range(n, 0, -1):
        print(i)
        print('Blastoff!')
    return None
```

```python
def countdown(n):
    '''
    while n > 0:
        print(n)
        n = n - 1
        print('Blastoff!')
    return None
```

Recursive function:

```python
def countdown(n):
    '''
    if n == 0:
        print('Blastoff!')
        # base case
    else:
        print(n)
        countdown(n-1)  # recursive call
    return None
```
Recursive function:

```python
def countdown(n):
    >>> countdown(4)
    """
    if n == 0:
        print('Blastoff!')
    else:
        print(n)
        countdown(n-1)
    return None
```

Why recursion?

- elegant approach to problem solving for problems with a recursive structure

Recall: Factorials have a recursive definition

\[
\begin{align*}
0! &= 1 \\
n! &= n \times (n-1)!
\end{align*}
\]

** underlying algorithm is clear **
Recall: Factorials have a recursive definition

0! = 1
n! = n * (n-1)!
    = n * (n-1) * (n-2)!
    = n * (n-1) * (n-2) * (n-3)! ...

---

def factR(n):
    '''
    not as elegant ''
    result = 1
    for i in range(1, n+1):
        result *= i
    return result
Draw trunk
Recursively draw a tree on the right
Recursively draw a tree on the left
Stop when the tree is sufficiently small

```
def tree(n):
    '''(int) \rightarrow None
    if n >= 0:
        forward(n)
        Draw a fractal tree with trunk and branches length n.
        Returns None.
        # go right
        right(30)
        >>> tree(100)
        # go left
        left(60)
        tree(n-15)
        >>> tree(15)
        # back to start
        right(30)
        back(n)
    else:
        return None
```

Define a function stars such that:

```python
tree(n):
    if n >= 0:
        forward(n)
        Draw a fractal tree with trunk and branches length n.
        Returns None.
        # go right
        right(30)
        >>> tree(100)
        # go left
        left(60)
        tree(n-15)
        >>> tree(15)
        # back to start
        right(30)
        back(n)
    else:
        return None
```

```
def main():
    # set up and call tree
    tree(n-15)
    reset()
    speed(0)
    # back to start
    seth(90)
    right(30)
    n = int(input('branch length? '))
    back(n)
    tree()
    return None
```

```
def stars(n):
    if n == 1:
        print('*')
    else:
        stars(n-1)
        print(' ' * (n-1), end='')
        stars(n-1)
```

```python
def main():
    # set up and call tree
    tree(n-15)
    reset()
    speed(0)
    # back to start
    seth(90)
    right(30)
    n = int(input('branch length? '))
    back(n)
    tree()
    return None
```

Do you see a pattern -> recursive solution.
What is the base case?
What is the recursive rule?
Recursion

- a function that calls itself

Why?
✓ an elegant way to solve problems with a recursive structure

Why not?
✓ may be prohibitively expensive

Recursive function
def stars(n):
    """
    """
    if n == 1:
        print('*')
    else:
        stars(n-1)
    return None

def stars(n):
    """
    """
    if n == 1:
        print('*', end=' ')
    else:
        stars(n-1)
        print(n*'***', end=' ')
        stars(n-1)
    return None

✓ Recursion

- Python tuples
- Binary search
- Automated testing
Finding an item in a sequence

def isIn(seq, n):
    """(sequence, item) -> boolean
    Search for item n in a sequence, seq. Return True if n is a member, else False.
    """
    for item in seq:
        if item == n:
            return True
    else:
        return False

Python tuples

Tuples are immutable, heterogeneous sequences of references to any object.

For example,
intseq = (10, 20, 30, 40, 50)
genseq = (10, 20.0, 'a', True)
nestedseq = (10, 20, ('a', 'b'), True)
Python tuples

For example,
>>> intseq = (10, 20, 30, 40, 50)
>>> genseq = (10, 20.0, 'a', True)
>>> nestedseq = (10, 20, ('a', 'b'), True)
>>> shortseq = (99,)

>>> intseq[0] >>> len(nestedseq)
>>> genseq[1:4] >>> len(nestedseq[2])
>>> nestedseq[2] >>> intseq + nestedseq
>>> nestedseq[2][1] >>> 'a' in nestedseq

Tuples are immutable, heterogeneous sequences of references to any object.

For example,

For example,

>>> a = nestedseq[2][0] >>> x = 10,000
>>> b = nestedseq[2][1] >>> x
>>> (a, b) = ('c', 'd') >>> type(x)
??
>>> (a, b) = (b, a) >>> a, b = b, a
??

Finding an item in a sequence

def isIn(seq, n):
    """(sequence, item) -> boolean"
    Search for item n in a sequence, seq. Return True if n is a member, else False.
    >>> isIn('aeiou', 'i')
    True
    >>> isIn([10, 20, 30, 40, 50, 60, 70, 80, 90], 100)
    False
    ***
    for item in seq:
        if item == n:
            return True
    return False

Binary Search

Efficient search technique, as long as the list is already sorted.

Each step divides the remaining data into equal parts and discards one.

Go to list mid-point and compare to target.

If mid-point is the target, then done.
Otherwise, keep the part of the list where \( n \) could be, and search that. (Discard the rest.)

For example,

\[
\text{nums} = (1, 3, 4, 6, 8, 9, 11) \\
\text{target value: } x = 4 \\
\text{compare } x \text{ to } 6; x \text{ is smaller, so repeat with } (1, 3, 4) \\
\text{compare } x \text{ to } 3; x \text{ is larger, so repeat with } (4) \\
\text{compare } x \text{ to } 4; x == 4, \text{ so the position is returned}
\]

<table>
<thead>
<tr>
<th>Even or odd</th>
<th>( O(1) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary search</td>
<td>( O(\log n) )</td>
</tr>
<tr>
<td>Sequential search</td>
<td>( O(n) )</td>
</tr>
<tr>
<td>“Long” multiplication</td>
<td>( O(n^2) )</td>
</tr>
<tr>
<td>Check password combinations</td>
<td>( O(2^n) )</td>
</tr>
</tbody>
</table>

**Sequences**

- **Sequential search** \( O(n) \)
- **Binary search** \( O(\log n) \)
  - When \( n \) is 150 ...
  - When \( n \) is around 1000 ...

```python
def ctemp_to_fhemp(ctemp):
    '''(number) -> float
    return fahrenheit temp (fhemp) equivalent to input celsius temp (ctemp)
    >>> ctemp_to_fhemp(100)
    212.0
    >>> ctemp_to_fhemp(0)
    32.0
    >>> ctemp_to_fhemp(30)
    86.0
    >>> ctemp_to_fhemp(21.1)
    69.98
    '''
    fhemp = ctemp * 9/5 + 32
    return fhemp
```

**Automated Testing**

```python
def main():
    ctemp_to_fhemp(100)
    ctemp_to_fhemp(0)
    ctemp_to_fhemp(30)
    ctemp_to_fhemp(21.1)
```

```output
212.0
32.0
86.0
69.98
```
automated testing!

import doctest

>>> doctest.testmod()
TestResults(failed=0, attempted=4)
**Good bugs**

```python
while pin > 0:
    next2 = pin % 100
    pin = pin // 100
    code = ''
    code = next2 + code
return code
```