Fractals are a great example of recursion in action
def study_recursion(lecture):
    if lecture > 9:
        return False
    else:
        knowledge = study_recursion(lecture + 1)
        if knowledge:
            return True
        else:
            return False
What will this code return?

def study_recursion(lecture):
    if lecture > 9:
        return False
    else:
        knowledge = study_recursion(lecture + 1)
        if knowledge:
            return True
        else:
            return False

A. True
B. False
C. None
D. 0
E. it won’t (infinite loop)
What will this code return?

def study_recursion(lecture):
    if lecture > 9:
        return False
    else:
        knowledge = study_recursion(lecture + 1)
        if knowledge:
            return True
        else:
            return False

A. True
B. False
C. None
D. 0
E. it won’t (infinite loop)
Recursive Humor

Finding the Largest Element in a List

• my_max(values)
  • input: a *non-empty* list of numbers
  • returns: the largest element in the list

• examples:
  >>> my_max([5, 8, 10, 2])
  10
  >>> my_max([30, 2, 18])
  30
How can we code this?

• my_max(values)
  • input: a non-empty list of numbers
  • returns: the largest element in the list

• examples:
  >>> my_max([5, 8, 10, 2])
  10
  >>> my_max([30, 2, 18])
  30
How can we code this?

• Use Recursion

• Use Reduce (will talk about later)
What is signature and some test cases?

• my_max(values)
  • input: a *non-empty* list of numbers
  • returns: the largest element in the list

• examples:

  >>> my_max([5, 8, 10, 2])
  10

  >>> my_max([30, 2, 18])
  30
What is signature and some test cases?

- **my_max(values)**
  - input: a *non-empty* list of numbers
  - returns: the largest element in the list

- examples:
  >>> my_max([5, 8, 10, 2])
  10
  >>> my_max([30, 2, 18])
  30

```python
def my_max(values):
    '''returns the largest element in a non-empty list'''

def test_my_max():
    assert my_max([-1])==-1
    assert my_max([0, 1, -1])==1
```
Design Questions for `my_max()`

(base case) When can I determine the largest element in a list without needing to look at a smaller list?

(recursive case) How could I use the largest element in a smaller list to determine the largest element in the entire list?

list1 = [30, 2, 18]          list2 = [5, 12, 25, 2]

largest element = 18          largest element = 25

my_max(list1) → ??          my_max(list2) → ??
Design Questions for \texttt{my\_max()}

\textbf{base case}  When can I determine the largest element in a list without needing to look at a smaller list? \hspace{1cm} \text{when there's only one element}

\textbf{recursive case}  How could I use the largest element in a smaller list to determine the largest element in the entire list?

\begin{align*}
\text{list1} &= [30, 2, 18] & \text{list2} &= [5, 12, 25, 2] \\
\text{largest element} &= 18 & \text{largest element} &= 25 \\
\text{my\_max(list1)} &\rightarrow \textbf{30} & \text{my\_max(list2)} &\rightarrow \textbf{25}
\end{align*}

1. compare the first element to largest element in the rest of the list
2. return the larger of the two

\textit{Let the recursive call handle the rest of the list!}
Recursively Finding the Largest Element in a List

def my_max(values):
    """ returns the largest element in a list
    input: values is a *non-empty* list
    """

    if
        # base case

    else:
        # recursive case
def my_max(values):
    """ returns the largest element in a list
    input: values is a *non-empty* list
    """
    if len(values) == 1:    # base case
        return values[0]
    else:                   # recursive case
        max_in_rest = my_max(values[1:]
        if values[0] > max_in_rest:
            return values[0]
        else:
            return max_in_rest
How many times will \texttt{my\_max()} be called?

def my_max(values):
    if len(values) == 1:   # base case
        return values[0]
    else:                # recursive case
        max_in_rest = my_max(values[1:])
        if values[0] > max_in_rest:
            return values[0]
        else:
            return max_in_rest

print(my_max([5, 30, 10, 8]))
How many times will `my_max()` be called?

def my_max(values):
    if len(values) == 1:  # base case
        return values[0]
    else:  # recursive case
        max_in_rest = my_max(values[1:])
        if values[0] > max_in_rest:
            return values[0]
        else:
            return max_in_rest

print(my_max([5, 30, 10, 8]))

A. 1
B. 3
C. 4
D. 5
E. 6
How recursion works...

def my_max(values):
    if len(values) == 1:
        return values[0]
    else:
        max_in_rest = my_max(values[1:])
        if values[0] > max_in_rest:
            return values[0]
        else:
            return max_in_rest

my_max([0, 1, 2, 3])
  └── my_max([1, 2, 3])
      └── my_max([2, 3])
          └── my_max([3])

number of calls for a list of length 4 = 4
number of calls for a list of length n = n  ← grows reasonably.
double the number elements => twice as many calls, linear growth
What's wrong (if anything) with this alternative?

def my_max(values):
    """ returns the largest element in a list
    input: values is a *non-empty* list
    """
    if len(values) == 1:
        return values[0]
    else:
        # max_in_rest = my_max(values[1:])
        if values[0] > my_max(values[1:]):
            return values[0]
        else:
            return my_max(values[1:])
What's wrong (if anything) with this alternative?

def my_max(values):
    """ returns the largest element in a list
    input: values is a *non-empty* list
    """
    if len(values) == 1:
        return values[0]
    else:
        # max_in_rest = my_max(values[1:])
        if values[0] > my_max(values[1:]):
            return values[0]
        else:
            return my_max(values[1:])

Clicker Quiz:
Does this function produce the same results as the alternative?
A) Yes
B) No
C) I don’t know
def my_max(values):
    """ returns the largest element in a list
    input: values is a *non-empty* list
    """
    if len(values) == 1:
        return values[0]
    else:
        # max_in_rest = my_max(values[1:])
        if values[0] > my_max(values[1:])
            return values[0]
        else:
            return my_max(values[1:])

Clicker Quiz:
Does this function produce the same results as the alternative?
A) Yes
B) No
C) I don’t know
What's wrong (if anything) with this alternative?

def my_max(values):
    """ returns the largest element in a list
    input: values is a *non-empty* list
    """
    if len(values) == 1:
        return values[0]
    else:
        # max_in_rest = my_max(values[1:])
        if values[0] > my_max(values[1:]):
            return values[0]
        else:
            return my_max(values[1:])

Clicker Quiz:
Is the alternative function as efficient? (Hint: Try to determine worst case input)
A) Always
B) Sometimes
C) Never
D) I Don’t Know
What's wrong (if anything) with this alternative?

def my_max(values):
    """ returns the largest element in a list
    input: values is a *non-empty* list
    """
    if len(values) == 1:
        return values[0]
    else:
        # max_in_rest = my_max(values[1:])
        if values[0] > my_max(values[1:]):
            return values[0]
        else:
            return my_max(values[1:])

Clicker Quiz:
Is the alternative function as efficient?
A) Always  
B) Sometimes  => Consider my_max([0,1,2,3])
C) Never  
D) I Don’t Know
def my_max(values):
    if len(values) == 1:
        return values[0]
    else:
        if values[0] > my_max(values[1:]):
            return values[0]
        else:
            return my_max(values[1:])

Max number of calls for a list of length 4 = 15
Max number of calls for a list of length n = $2^n - 1$ ← gets big fast!!!

Increasing length by one => twice as many calls. Exponential growth!
Efficient solutions are desirable

Here the first solution made a linear number of calls for an input of length (n), whereas the second made an exponential number calls to itself for an input of length (n)

Last class we created a power function that used

\[ b^n = b*b^{(n-1)} \]

as it’s recursive step. What if we created a new version of power that uses

\[ b^n = [b^{(n/2)}]^2, \text{ if } n \text{ even} \]
\[ b^n = b*b^{(n-1)}, \text{ if } n \text{ odd} \]

Which do you think will be more efficient?
What is the output of this program?

def myst(s):
    if len(s) <= 1:
        return s
    else:
        return s[-1] + myst(s[:-1]) + s[-1]

print(myst('bar'))

A. rabar
B. rabbar
C. barab
D. barrab
E. none of these
What is the output of this program?

```python
def myst(s):
    if len(s) <= 1:
        return s
    else:
        return s[-1] + myst(s[:-1]) + s[-1]

print(myst('bar'))
```

A. rabar
B. rabbar
C. barab
D. barrab
E. none of these
How recursion works...

```python
def myst(s):
    if len(s) <= 1:
        return s
    else:
        return s[-1] + myst(s[:-1]) + s[-1]
```

```
myst('bar')

'r' + myst('ba') + 'r'

'r' + 'a' + myst('b') + 'a' + 'r'

'r' + 'a' + 'b' + 'a' + 'r'
```
How recursion works...

```python
def myst(s):
    if len(s) <= 1:
        return s
    else:
        return s[-1] + myst(s[:-1]) + s[-1]
```

```
myst('bar')

'\text{r}' + \text{myst('ba')} + 'r'

'\text{r}' + 'a' + 'b' + 'a' + 'r'
```
How recursion works...

```python
def myst(s):
    if len(s) <= 1:
        return s
    else:
        return s[-1] + myst(s[:-1]) + s[-1]
```

myst('bar')

' r ' +  'aba'  +  ' r '
How recursion works...

```python
def myst(s):
    if len(s) <= 1:
        return s
    else:
        return s[-1] + myst(s[:-1]) + s[-1]
```

```
myst('bar')
```

```
result: 'rabar'
```
A Recursive Palindrome Checker

• A palindrome is a string that reads the same forward and backward.
  • examples: "radar", "mom", "abcddcba"

• Let's write a function that determines if a string is a palindrome:
  >>> is_pal('radar')
  True
  >>> is_pal('abccda')
  False

• We need more than one base case. What are they?

• How should we reduce the problem in the recursive call?
A Recursive Palindrome Checker

- A palindrome is a string that reads the same forward and backward.
  - examples: "radar", "mom", "abcddcba"

- Let's write a function that determines if a string is a palindrome:
  ```python
  >>> is_pal('radar')
  True
  >>> is_pal('abccda')
  False
  ```

- We need a signature

- We need test cases
A Recursive Palindrome Checker

• A palindrome is a string that reads the same forward and backward.
  • examples: "radar", "mom", "abcddcba"

• Let's write a function that determines if a string is a palindrome:
  >>> is_pal('radar')
  True
  >>> is_pal('abccda')
  False

• We need more than one base case. What are they?
  • empty string
  • single character
  • outer characters don't match

• How should we reduce the problem in the recursive call?
A Recursive Palindrome Checker

def is_pal(s):
    """ returns True if s is a palindrome and False otherwise. 
    input s: a string containing only letters (no spaces, punctuation, etc.) 
    """
def is_pal(s):
    """returs True if s is a palindrome and False otherwise.
    input s: a string containing only letters (no spaces, punctuation, etc.)
    """
    if len(s) <= 1:  # empty string or one letter
        return True
    elif s[0] != s[-1]:
        return False
    else:
        is_pal_rest = is_pal(s[1:-1])
        return is_pal_rest
A Recursive Palindrome Checker
(with temporary printlns for debugging)

def is_pal(s):
    """ returns True if s is a palindrome and False otherwise.
    input s: a string containing only letters (no spaces, punctuation, etc.) """

    print('beginning call for', s)
    if len(s) <= 1:   # empty string or one letter
        print('call for', s, 'returns True')
        return True
    elif s[0] != s[-1]:
        print('call for', s, 'returns False')
        return False
    else:
        is_pal_rest = is_pal(s[1:-1])
        print('call for', s, 'returns', is_pal_rest)
        return is_pal_rest
DONT CURSE.
RECURSE!
More Recursive Design

based in part on notes from the CS-for-All curriculum
developed at Harvey Mudd College
Practicing Design

• replace(s, old, new)
  • inputs: a string s
    two characters, old and new
  • returns: a version of s in which all occurrences of old
    are replaced by new

• examples:

  >>> replace('boston', 'o', 'e')
  'besten'

  >>> replace('banana', 'a', 'o')
  'bonono'

  >>> replace('mama', 'm', 'd')
  'dada'
Practicing Design

• replace(s, old, new)
  • inputs: a string s
    two characters, old and new
  • returns: a version of s in which all occurrences of old
    are replaced by new

• Signature?

• Test Cases?

'boston'

'besten'
Design Questions for replace()

(base case) When do I know that I can stop trying to replace characters in a string \( s \)?

(recursive case) How could I use the "replaced" version of a smaller string to get the "replaced" version of \( s \)?

\[ s_1 = 'a\_\_\_\_\_' \quad s_2 = 'r\_\_\_\_\_' \]

\[
\text{replace}(s_1, 'a', 'o') \\
\text{replace}(s_2, 'e', 'i')
\]

If you knew the "replaced" version of the covered portion, how would you form the "replaced" version of the entire string \( s_1 \)?

If you knew the "replaced" version of the covered portion, how would you form the "replaced" version of the entire string \( s_2 \)?
Design Questions for replace()

(base case) When do I know that I can stop trying to replace characters in a string s?

(recursive case) How could I use the "replaced" version of a smaller string to get the "replaced" version of s?

Let the recursive call handle the covered portion!

\[
s_1 = 'a' \_ \_ 'l\_\_w\_\_a\_\_y''
\]
\[
s_2 = 'r' \_ \_ 'e\_\_u\_r\_\_c\_\_e\_\_s''
\]

replace(s1, 'a', 'o')

If you knew the "replaced" version of the covered portion, how would you form the "replaced" version of the entire string s1?

'o' + replace(...) 

replace(s2, 'e', 'i')

If you knew the "replaced" version of the covered portion, how would you form the "replaced" version of the entire string s2?

'r' + replace(...) 

Don't forget to do your one step!
Complete This Function Together!

def replace(s, old, new):
    """ returns a version of the string s in which all occurrences of old are replaced by new """
    if s == '':  # why not "not (old in s)"?
        return _____
    else:
        # make the recursive call first
        # and store its return value
        repl_rest = replace(______, old, new)

        # do your one step!
        if
            return
        else:
            return
def replace(s, old, new):
    """ returns a version of the string s
    in which all occurrences of old
    are replaced by new
    """

    if s == '':
        return s
    else:
        # make the recursive call first
        # and store its return value
        repl_rest = replace(s[1:], old, new)

        # do your one step!
        if s[0] == old:
            return new + repl_rest  # replace s[0]
        else:
            return s[0] + repl_rest  # leave it
Removing Vowels From a String

- `remove_vowels(s)` - removes the vowels from the string `s`, returning its "vowel-less" version!

  ```python
  >>> remove_vowels('recursive')
  'rcrsv'
  >>> remove_vowels('vowel')
  'vwl'
  ```

- Can we take the usual approach to processing a string recursively? Yes!
  - delegate `s[1:]` to the recursive call
  - we're responsible for handling `s[0]`

- What are the possible cases for our part (`s[0]`)?
  - does what we do with our part depend on its value? Yes!
    - if `s[0]` is a vowel...
    - if `s[0]` isn't a vowel...
Consider Concrete Cases

remove_vowels('after')
  • what is its solution?
  • what is the next smaller subproblem?
  • what is the solution to that subproblem?
  • how can we use the solution to the subproblem?
    What is our one step?

remove_vowels('recurse')
  • what is its solution?
  • what is the next smaller subproblem?
  • what is the solution to that subproblem?
  • how can we use the solution to the subproblem?
    What is our one step?
Consider Concrete Cases

remove_vowels('after')
  • what is its solution? 'ftr'
  • what is the next smaller subproblem? remove_vowels('fter')
  • what is the solution to that subproblem? 'ftr'
  • how can we use the solution to the subproblem? What is our one step? just return the subproblem's solution!

remove_vowels('recurse')
  • what is its solution? 'rcrs'
  • what is the next smaller subproblem? remove_vowels('ecurse')
  • what is the solution to that subproblem? 'crs'
  • how can we use the solution to the subproblem? What is our one step? 'r' + 'crs'.....Now write the function!
def remove_vowels(s):
    """ returns the "vowel-less" version of s
    input s: an arbitrary string
    """
def remove_vowels(s):
    """ returns the "vowel-less" version of s
    input s: an arbitrary string
    """
    if s == '':
        return ''
    else:
        # make the recursive call first
        # and store its return value
        rem_rest = remove_vowels(s[1:])

        # do our one step!
        if s[0] in 'aeiou':  # ok use of in
            return rem_rest
        else:
            return s[0] + rem_rest
Recursion vs Iteration

• Any function that can be written recursively can also be written iteratively.
Recursion

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

For loop

```python
def fac(n):
    result = 1
    for x in range(1, n+1):
        result *= x
    return result
```

While loop

```python
def fac(n):
    result = 1
    while n > 0:
        result *= n
        n = n - 1
    return result
```
Fibonacci

For another example, let’s look at the Fibonacci sequence.

The mathematical definition is generally written recursively.

The sequence:

\[ \{0, 1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots \} \]

The rule:

\[ x_0 = 0 \]
\[ x_1 = 1 \]
\[ x_n = x_{n-1} + x_{n-2} \quad \text{for } n \text{ in } 2, 3, 4, \ldots \]
def iterative_fib(n):
    if n == 0:
        return 0
    val_one = 1  # init value of f(n-1)
    val_two = 0  # init value of f(n-2)
    for i in range(1, n):
        temp_val = val_one + val_two
        val_two = val_one
        val_one = temp_val
    return val_one
def recursive_fib(n):
    if n == 0:
        return 0
    if n == 1:
        return 1
    return recursive_fib(n-2) + recursive_fib(n-1)
Fibonacci

- Let’s do a comparison of the recursive and iterative code

<table>
<thead>
<tr>
<th>N</th>
<th>Recursive Runtime (μs)</th>
<th>Iterative Runtime (μs)</th>
<th>Recursive Function Calls</th>
<th>Iterative Loops</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>6.2</td>
<td>6.7</td>
<td>15</td>
<td>4</td>
</tr>
<tr>
<td>10</td>
<td>17.3</td>
<td>7.0</td>
<td>177</td>
<td>9</td>
</tr>
<tr>
<td>20</td>
<td>11178</td>
<td>7.5</td>
<td>21891</td>
<td>19</td>
</tr>
<tr>
<td>40</td>
<td>&gt; 1 min</td>
<td>8.1</td>
<td>NA</td>
<td>39</td>
</tr>
</tbody>
</table>
So…. is iteration better?

- In general, yes, iteration is a better solution than recursion for many methods
- **However,**
  - Recursion is a way of thinking of problems that is in line with mathematical reasoning e.g. the Fibonacci sequence is better represented by its recursive form than an iterative form
  - Certain data structures like graphs and trees are easier to interact with recursively. Iterative methods would require code to be **complex** and **opaque**. This is something we as programmers want to avoid.
When not to implement recursion...