

Lecture 08

More Recursion!



Fractals are a great example of recursion in action

Lecture 08 Goals

```
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)

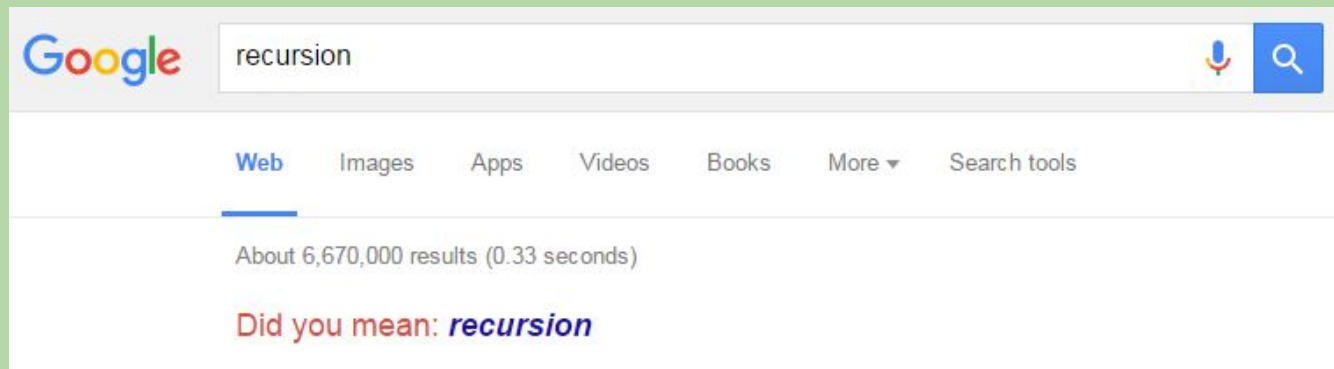
What will this code return?

```
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

- From *The Hacker's Dictionary*:
recursion. *noun.* See **recursion.**



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
```

```
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]`



largest element = 18

`my_max(list1) → ??`

`list2 = [5, 12, 25, 2]`



largest element = 25

`my_max(list2) → ??`

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? when there's only one element

(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]`



largest element = 18

`my_max(list1) → 30`

`list2 = [5, 12, 25, 2]`



largest element = 25

`my_max(list2) → 25`

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

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
```

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 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 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 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

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

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:])
```

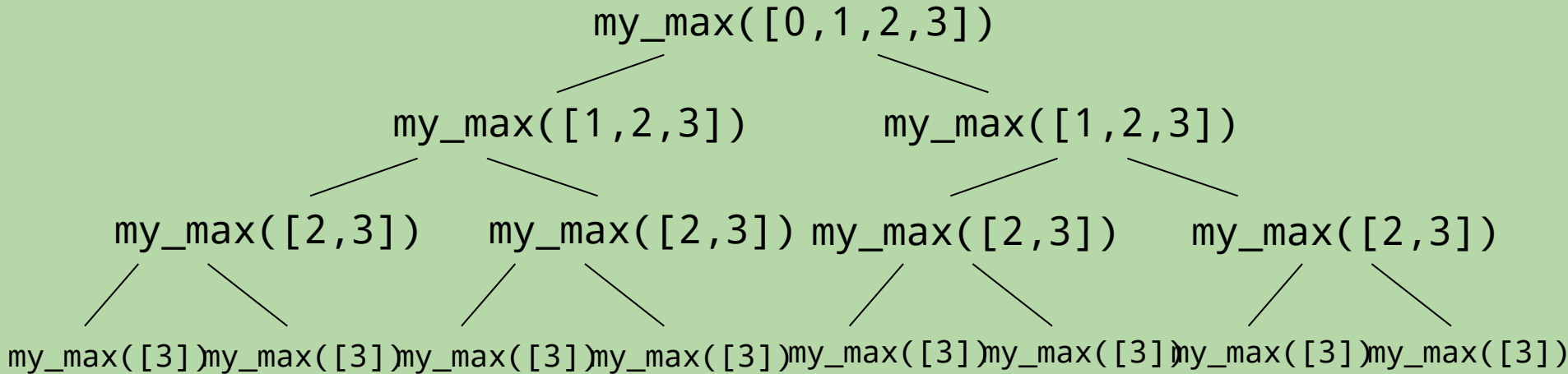
Clicker Quiz:

Is the alternative function as *efficient*?

- A) Always
- B) Sometimes => Consider my_max([0,1,2,3])
- C) Never
- D) I Don't Know

How recursion works...

```
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?

```
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...

```
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...

```
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' + 'b' + 'a' + 'r'

The diagram illustrates the recursive process of the function `myst('bar')`. It shows the function call `myst('bar')` at the top, which is expanded into `'r' + myst('ba') + 'r'`. This expression is further expanded into `'r' + 'a' + 'b' + 'a' + 'r'`, where the character `'b'` is highlighted in red. Brackets are used to show the mapping between the recursive calls and the final string construction.

How recursion works...

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

myst('bar')

'r' + 'aba' + 'r'

The diagram illustrates the result of the recursive call myst('bar'). A large curly brace is positioned above the expression 'r' + 'aba' + 'r'. The word 'aba' is highlighted in red, indicating it is the result of the recursive call myst('bar'[:-1]).

How recursion works...

```
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:

```
>>> 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.)  
    """
```

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.)  
    """  
    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)
```



**DON'T
CURSE.
RECURSE!**

More Recursive Design

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

Practicing Design

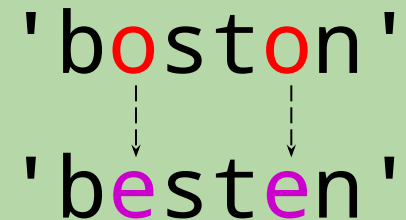
- `replace(s, old, new)`
 - inputs: a string `s`
two characters, `old` and `new`
 - returns: a version of `s` in which all occurrences of `old` are replaced by `new`
- examples:

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



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

```
>>> replace('mama', 'm', 'd')
'dada'
```

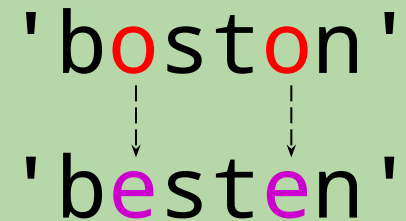


```
'boston'
'besten'
```

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?



'boston'
↓ ↓
'besten'


The diagram illustrates the replacement of the character 'o' in the string 'boston' with the character 'e' to produce the string 'besten'. Dashed arrows point from the 'o' characters in 'boston' to the 'e' characters in 'besten'.


- Test Cases?

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`?

`s1 = 'a' `

`s2 = 'r' `

`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`?

`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`?

Design Questions for `replace()`

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

when the old character doesn't appear in `s`

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

`s1 = 'a' []`

`s2 = 'r' []`

`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(...)`

Let the recursive call handle the covered portion!

Don't forget to do your one step!

`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(...)`

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
```

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 == '':  
        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!

```
>>> 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!

remove_vowels()

```
def remove_vowels(s):  
    """ returns the "vowel-less" version of s  
        input s: an arbitrary string  
    """
```


remove_vowels()

```
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
```



**KEEP
CALM
AND
CODE
ON**

**and go to
section
hours!**

Recursion vs Iteration

- Any function that can be written recursively can also be written iteratively.

Remember this slide?

recursion

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

for loop

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

map

```
def fac(n):  
    return reduce(lambda x,y : x*y, \  
        range(1, max(1, n)))
```

More on this later

while loop

```
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:

$$\{0, 1, 1, 2, 3, 5, 8, 13, 21, 34, \dots\}$$

The rule:

$$x_0 = 0$$

$$x_1 = 1$$

$$x_n = x_{n-1} + x_{n-2} \quad \text{for } n \text{ in } 2, 3, 4, \dots$$

Fibonacci

```
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
```

Fibonacci

```
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

N	Recursive Runtime (μ s)	Iterative Runtime (μ s)	Recursive Function Calls	Iterative Loops
5	6.2	6.7	15	4
10	17.3	7.0	177	9
20	11178	7.5	21891	19
40	> 1 min	8.1	NA	39

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...

