Practicing Recursive Design

Computer Science 111
Boston University

Vahid Azadeh-Ranjbar, Ph.D.

Recall: Designing a Recursive Function

1. Start by programming the base case(s).
   - *What instance(s) of this problem can I solve directly (without looking at anything smaller)?*
2. Find the recursive substructure.
   - *How could I use the solution to **any smaller version** of the problem to solve the overall problem?*
   1. Make a recursive call!
   2. Trace your function before designing the rest of the function after recursive call.
   3. Do one step and build your solution on the result of the recursive call
      - use ***concrete cases*** to figure out what you need to do

Recall: Designing a Recursive Function

Make a recursive call!

It is highly recommended to follow these steps:

1. Call the function
2. Play around the input of the function to make the problem smaller (**converging to the base case**)
3. Assign the function to a variable called "**storage variable**". We will use this **storage variable** in the next step "Do one step"

From the Pre-Lecture Video and Quiz

- `mymax(vals)`
  - input: a *non-empty* list
  - returns: the largest element in the list
- examples:

```
>>> mymax([5, 8, 10, 2])
10
>>> mymax([30, 2, 18])
30
```

Design Questions for `mymax()`

(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

Option (1)

```
 if len(vals) == 1:      # base case
        return vals[0]
    else:
        ...
```

Option (2)

```
if len(vals) == 1:       # base case
        return vals
    else:
        ...
```

Which one???

Option (3)

```
if len(vals) == 1:       # base case
        return vals[:1]
    else:
        ...
```

Design Questions for `mymax()`

(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

Option (1)

```
 if len(vals) == 1:      # base case
        return vals[0]
    else:
        ...
```

Option (2)

```
if len(vals) == 1:       # base case
        return vals
    else:
        ...
```

Which one???

Only Option (1)

Option (3)

```
if len(vals) == 1:       # base case
        return vals[:1]
    else:
        ...
```

Design Questions for `mymax()`

(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?

```
if len(vals) == 1:            # base case
        return vals[0]
    else:                      # recursive case
        max_rest = mymax(vals[1:])
```

3- Store in a variable

1- Call the function

2- Play around the parameter

Design Questions for `mymax()`

(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?

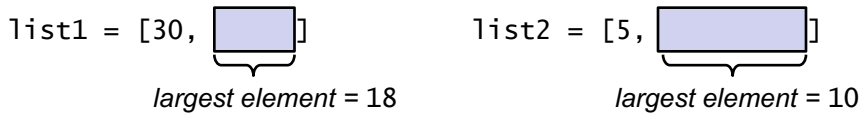
```
if len(vals) == 1:            # base case
        return vals[0]
    else:                      # recursive case
        max_rest = mymax(vals[1:])
```


- Now, its time to stop programming and trace the function to make sure that it converges to the base case.

Design Questions for `mymax()`

(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?



`mymax(list1)` → **30** `mymax(list2)` → **10**

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 mymax(vals):
    """ returns the largest element in a list
        input: values is a *non-empty* list
    """
    if len(vals) == 1:          # base case
        return vals[0]
    else:                       # recursive case
        max_rest = mymax(vals[1:])
        if vals[0] > max_rest:
            return vals[0]
        else:
            return max_rest

print(mymax([5, 30, 10, 8]))
```

How many times will `max_rest` be returned?

```
def mymax(vals):
    if len(vals) == 1:          # base case
        return vals[0]
    else:                       # recursive case
        max_rest = mymax(vals[1:])
        if vals[0] > max_rest:
            return vals[0]
        else:
            return max_rest     # how many times?

print(mymax([5, 30, 10, 8]))
```

A. 0

B. 1

C. 2

D. 3

E. 4

How recursion works...

```
mymax([5, 30, 10, 8])
  vals = [5, 30, 10, 8]
```

```
def mymax(vals):
   if len(vals) == 1:
      return vals[0]
   else:
      max_rest = mymax(vals[1:])
      if vals[0] > max_rest:
          return vals[0]
      else:
          return max_rest
```

How recursion works...

```
mymax([5, 30, 10, 8])
  vals = [5, 30, 10, 8]
  max_rest = mymax([30, 10, 8])
```

```
def mymax(vals):
   if len(vals) == 1:
      return vals[0]
   else:
      max_rest = mymax(vals[1:])
      if vals[0] > max_rest:
          return vals[0]
      else:
          return max_rest
```

How recursion works...

```
mymax([5, 30, 10, 8])
  vals = [5, 30, 10, 8]
  max_rest = mymax([30, 10, 8])
```

```
mymax([30, 10, 8])
  vals = [30, 10, 8]
```


```
def mymax(vals):
   if len(vals) == 1:
      return vals[0]
   else:
      max_rest = mymax(vals[1:])
      if vals[0] > max_rest:
          return vals[0]
      else:
          return max_rest
```

How recursion works...

```
def mymax(vals):
    if len(vals) == 1:
        return vals[0]
    else:
        max_rest = mymax(vals[1:])
        if vals[0] > max_rest:
            return vals[0]
        else:
            return max_rest
```

```
mymax([5, 30, 10, 8])
  vals = [5, 30, 10, 8]
  max_rest = mymax([30, 10, 8])
```

```
mymax([30, 10, 8])
  vals = [30, 10, 8]
  max_rest = mymax([10, 8])
```

How recursion works...

```
def mymax(vals):
    if len(vals) == 1:
        return vals[0]
    else:
        max_rest = mymax(vals[1:])
        if vals[0] > max_rest:
            return vals[0]
        else:
            return max_rest
```

```
mymax([5, 30, 10, 8])
  vals = [5, 30, 10, 8]
  max_rest = mymax([30, 10, 8])
```

```
mymax([30, 10, 8])
  vals = [30, 10, 8]
  max_rest = mymax([10, 8])
```

```
mymax([10, 8])
  vals = [10, 8]
```

How recursion works...

```
def mymax(vals):
    if len(vals) == 1:
        return vals[0]
    else:
        max_rest = mymax(vals[1:])
        if vals[0] > max_rest:
            return vals[0]
        else:
            return max_rest
```

```
mymax([5, 30, 10, 8])
  vals = [5, 30, 10, 8]
  max_rest = mymax([30, 10, 8])
```

```
mymax([30, 10, 8])
  vals = [30, 10, 8]
  max_rest = mymax([10, 8])
```

```
mymax([10, 8])
  vals = [10, 8]
  max_rest = mymax([8])
```

How recursion works...

```
def mymax(vals):
    if len(vals) == 1:
        return vals[0]
    else:
        max_rest = mymax(vals[1:])
        if vals[0] > max_rest:
            return vals[0]
        else:
            return max_rest
```

```
mymax([5, 30, 10, 8])
  vals = [5, 30, 10, 8]
  max_rest = mymax([30, 10, 8])
```

```
mymax([30, 10, 8])
  vals = [30, 10, 8]
  max_rest = mymax([10, 8])
```

```
mymax([10, 8])
  vals = [10, 8]
  max_rest = mymax([8])
```

```
mymax([8])
  vals = [8]
```

How recursion works...

```
def mymax(vals):
    if len(vals) == 1:
        return vals[0]
    else:
        max_rest = mymax(vals[1:])
        if vals[0] > max_rest:
            return vals[0]
        else:
            return max_rest
```

```
mymax([5, 30, 10, 8])
  vals = [5, 30, 10, 8]
  max_rest = mymax([30, 10, 8])
```

```
mymax([30, 10, 8])
  vals = [30, 10, 8]
  max_rest = mymax([10, 8])
```

```
mymax([10, 8])
  vals = [10, 8]
  max_rest = mymax([8])
```

```
mymax([8])
  vals = [8]
  base case!
  return vals[0] = 8
```

4 different stack frames, each with its own set of variables!

How recursion works...

```
def mymax(vals):
    if len(vals) == 1:
        return vals[0]
    else:
        max_rest = mymax(vals[1:])
        if vals[0] > max_rest:
            return vals[0]
        else:
            return max_rest
```

```
mymax([5, 30, 10, 8])
  vals = [5, 30, 10, 8]
  max_rest = mymax([30, 10, 8])
```

```
mymax([30, 10, 8])
  vals = [30, 10, 8]
  max_rest = mymax([10, 8])
```

```
mymax([10, 8])
  vals = [10, 8]
  max_rest = mymax([8])
```

```
mymax([8])
  vals = [8]
  base case!
  return vals[0] = 8
```

How recursion works...

```
def mymax(vals):
    if len(vals) == 1:
        return vals[0]
    else:
        max_rest = mymax(vals[1:])
        if vals[0] > max_rest:
            return vals[0]
        else:
            return max_rest
```

```
mymax([5, 30, 10, 8])
  vals = [5, 30, 10, 8]
  max_rest = mymax([30, 10, 8])
```

```
mymax([30, 10, 8])
  vals = [30, 10, 8]
  max_rest = mymax([10, 8])
```

```
mymax([10, 8])
  vals = [10, 8]
  max_rest =      8
```

```
mymax([8])
  vals = [8]
  base case!
  return vals[0] = 8
```

The final result gets built up ***on the way back*** from the base case!

How recursion works...

```
def mymax(vals):
    if len(vals) == 1:
        return vals[0]
    else:
        max_rest = mymax(vals[1:])
        if vals[0] > max_rest:
            return vals[0]
        else:
            return max_rest
```

```
mymax([5, 30, 10, 8])
  vals = [5, 30, 10, 8]
  max_rest = mymax([30, 10, 8])
```

```
mymax([30, 10, 8])
  vals = [30, 10, 8]
  max_rest = mymax([10, 8])
```

```
mymax([10, 8])
  vals = [10, 8]   vals[0] -> 10
  max_rest = 8
```

The final result gets built up ***on the way back*** from the base case!

How recursion works...

```
def mymax(vals):
    if len(vals) == 1:
        return vals[0]
    else:
        max_rest = mymax(vals[1:])
        if vals[0] > max_rest:
            return vals[0]
        else:
            return max_rest
```

```
mymax([5, 30, 10, 8])
  vals = [5, 30, 10, 8]
  max_rest = mymax([30, 10, 8])
```

```
mymax([30, 10, 8])
  vals = [30, 10, 8]
  max_rest = mymax([10, 8])
```

```
mymax([10, 8])
  vals = [10, 8]   vals[0] -> 10
  max_rest = 8
  return vals[0] = 10
```

The final result gets built up ***on the way back*** from the base case!

How recursion works...

```
def mymax(vals):
    if len(vals) == 1:
        return vals[0]
    else:
        max_rest = mymax(vals[1:])
        if vals[0] > max_rest:
            return vals[0]
        else:
            return max_rest
```

```
mymax([5, 30, 10, 8])
  vals = [5, 30, 10, 8]
  max_rest = mymax([30, 10, 8])
```


```
mymax([30, 10, 8])
  vals = [30, 10, 8]
  max_rest =      10
```


```
mymax([10, 8])
  vals = [10, 8]   vals[0] -> 10
  max_rest = 8
  return vals[0] = 10
```

The final result gets built up ***on the way back*** from the base case!

How recursion works...

```
mymax([5, 30, 10, 8])
  vals = [5, 30, 10, 8]
  max_rest = mymax([30, 10, 8])
```

```
mymax([30, 10, 8])
  vals = [30, 10, 8]   vals[0] -> 30
  max_rest = 10
```

```
def mymax(vals):
   if len(vals) == 1:
      return vals[0]
   else:
      max_rest = mymax(vals[1:])
      if vals[0] > max_rest:
         return vals[0]
      else:
         return max_rest
```

The final result gets built up ***on the way back*** from the base case!

How recursion works...

```
mymax([5, 30, 10, 8])
  vals = [5, 30, 10, 8]
  max_rest = mymax([30, 10, 8])
```

```
mymax([30, 10, 8])
  vals = [30, 10, 8]   vals[0] -> 30
  max_rest = 10
  return vals[0] = 30
```

```
def mymax(vals):
   if len(vals) == 1:
      return vals[0]
   else:
      max_rest = mymax(vals[1:])
      if vals[0] > max_rest:
         return vals[0]
      else:
         return max_rest
```

The final result gets built up ***on the way back*** from the base case!

How recursion works...

```
def mymax(vals):
    if len(vals) == 1:
        return vals[0]
    else:
        max_rest = mymax(vals[1:])
        if vals[0] > max_rest:
            return vals[0]
        else:
            return max_rest
```

```
mymax([5, 30, 10, 8])
  vals = [5, 30, 10, 8]
  max_rest =      30
```

```
mymax([30, 10, 8])
  vals = [30, 10, 8]   vals[0] -> 30
  max_rest = 10
  return vals[0] = 30
```

The final result gets built up ***on the way back*** from the base case!

How recursion works...

```
def mymax(vals):
    if len(vals) == 1:
        return vals[0]
    else:
        max_rest = mymax(vals[1:])
        if vals[0] > max_rest:
            return vals[0]
        else:
            return max_rest
```

```
mymax([5, 30, 10, 8])
  vals = [5, 30, 10, 8] vals[0] -> 5
  max_rest = 30
  return max_rest = 30
```

The final result gets built up ***on the way back*** from the base case!

How recursion works...

```
def mymax(vals):
    if len(vals) == 1:
        return vals[0]
    else:
        max_rest = mymax(vals[1:])
        if vals[0] > max_rest:
            return vals[0]
        else:
            return max_rest
```

```
mymax([5, 30, 10, 8])
  vals = [5, 30, 10, 8]  vals[0] -> 5
  max_rest = 30
  return max_rest = 30
```

final result: **30**

How recursion works...

```
def mymax(vals):
    if len(vals) == 1:
        return vals[0]
    else:
        max_rest = mymax(vals[1:])
        if vals[0] > max_rest:
            return vals[0]
        else:
            return max_rest
```

```
mymax([5, 30, 10, 8])
  vals = [5, 30, 10, 8]  vals[0] -> 5
  max_rest = 30
  return max_rest = 30
```

```
mymax([30, 10, 8])
  vals = [30, 10, 8]   vals[0] -> 30
  max_rest = 10
  return vals[0] = 30
```

```
mymax([10, 8])
  vals = [10, 8]   vals[0] -> 10
  max_rest = 8
  return vals[0] = 10
```

```
mymax([8])
  vals = [8]
  base case!
  return vals[0] = 8
```

How many times will max_rest be returned?

```
def mymax(vals):
    if len(vals) == 1:          # base case
        return vals[0]
    else:                       # recursive case
        max_rest = mymax(vals[1:])
        if vals[0] > max_rest:
            return vals[0]
        else:
            return max_rest     # how many times?

print(mymax([5, 30, 10, 8]))
```

A. 0

B. 1

C. 2

D. 3

E. 4

Recursively Replacing Characters in a String

- `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'
```

old new

'boston'
↓ ↓
'besten'

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

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

Design Questions for `replace()`

(base case) When can I determine the "replaced" version of s *without* looking at a smaller string? when s is empty

(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')
= 'o' + solution of
        covered portion
```

```
replace(s2, 'e', 'i')
= 'r' + solution of
        covered portion
```

Let the recursive call handle the covered portion!

Design Questions for `replace()`

(base case) When can I determine the "replaced" version of s *without* looking at a smaller string? when s is empty

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

`s1 = 'always'`

`s2 = 'recurse!'`

```
replace(s1, 'a', 'o')
= 'o' + solution of
        covered portion
```

```
replace(s2, 'e', 'i')
= 'r' + solution of
        covered portion
```

Let the recursive call handle the covered portion!

Design Questions for `replace()`

(base case) When can I determine the "replaced" version of s *without* looking at a smaller string? when s is empty

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

`s1 = 'always'`

```
replace(s1, 'a', 'o')
= 'o' + solution of
        covered portion
= 'o' + 'lwoys'
= 'olwoys'
```

`s2 = 'recurse!'`

```
replace(s2, 'e', 'i')
= 'r' + solution of
        covered portion
= 'r' + 'icursi!'
= 'ricursi!'
```

Let the recursive call handle the covered portion!

Complete This Function Together!

```
def replace(s, old, new):
    if s == '':
        return _____
    else:
        # recursive call handles rest of string
        repl_rest = replace(______, _____, ____)

        # do your one step!
        if _________________:
            return _________________
        else:
            return _________________
```

Complete This Function Together!

```
def replace(s, old, new):
    if s == '':
        return ''
    else:
        # recursive call handles rest of string
        repl_rest = replace(s[1:], old, new)

        # do your one step!
        if ________________:
            return ________________
        else:
            return ________________
```

Complete This Function Together!

```
def replace(s, old, new):
    if s == '':
        return ''
    else:
        # recursive call handles rest of string
        repl_rest = replace(s[1:], old, new)

        # do your one step!
        if s[0] == old:
            return _______??_______
        else:
            return ________________
```

Which concrete case is this?

`replace('always', 'a', 'o')`

`return 'o' +` solution to rest of string

`replace('recurse!', 'e', 'i')`

`return 'r' +` solution to rest of string

Complete This Function Together!

```
def replace(s, old, new):
    if s == '':
        return ''
    else:
        # recursive call handles rest of string
        repl_rest = replace(s[1:], old, new)

        # do your one step!
        if s[0] == old:
            return new + repl_rest
        else:
            return _________________
```

Use the first concrete case:

replace('always', 'a', 'o')

```
    return 'o' + solution to rest of string
           new + repl_rest
```

Complete This Function Together!

```
def replace(s, old, new):
    if s == '':
        return ''
    else:
        # recursive call handles rest of string
        repl_rest = replace(s[1:], old, new)

        # do your one step!
        if s[0] == old:
            return new + repl_rest
        else:
            return _______??_______
```

Now use the second case:

replace('recurse!', 'e', 'i')

```
    return 'r' + solution to rest of string
```

Complete This Function Together!

```
def replace(s, old, new):
    if s == '':
        return ''
    else:
        # recursive call handles rest of string
        repl_rest = replace(s[1:], old, new)

        # do your one step!
        if s[0] == old:
            return new + repl_rest
        else:
            return s[0] + repl_rest
```

Now use the second case:

replace('recurse!', 'e', 'i')

```
    return 'r' + solution to rest of string
           s[0] + repl_rest
```

Final Version!

```
def replace(s, old, new):
    if s == '':
        return ''
    else:
        # recursive call handles rest of string
        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
```

Add Temporary `prints` for Debugging

```
def replace(s, old, new):
    print('starting call for', s)
    if s == '':
        print('base case returns empty string')
        return ''
    else:
        repl_rest = replace(s[1:], old, new)

        if s[0] == old:
            print('call for', s, '->', new + repl_rest)
            return new + repl_rest
        else:
            print('call for', s, '->', s[0] + repl_rest)
            return s[0] + repl_rest
```

Returning to mymax…

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

An Alternative Version…

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

An Alternative Version…

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

An Alternative Version…

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

What's Wrong (If Anything) With This Alternative?

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

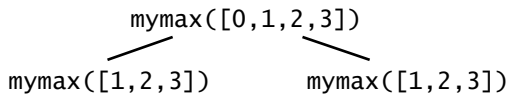
How recursion works...

```
def mymax(vals):
    if len(vals) == 1:
        return vals[0]
    else:
        if vals[0] > mymax(vals[1:]):
            return vals[0]
        else:
            return mymax(vals[1:])
```

```
mymax([0,1,2,3])
```

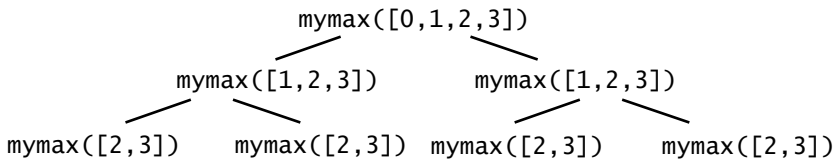
How recursion works...

```
def mymax(vals):
    if len(vals) == 1:
        return vals[0]
    else:
        if vals[0] > mymax(vals[1:]):
            return vals[0]
        else:
            return mymax(vals[1:])
```



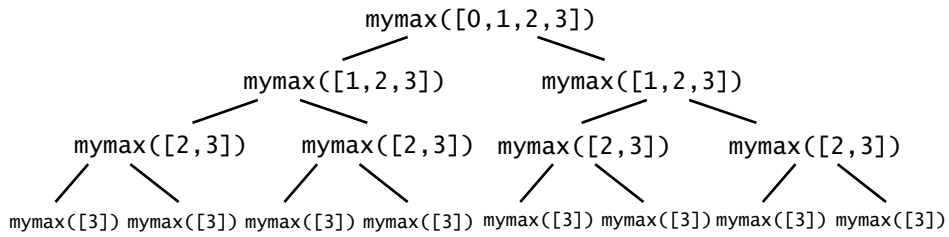
How recursion works...

```
def mymax(vals):
    if len(vals) == 1:
        return vals[0]
    else:
        if vals[0] > mymax(vals[1:]):
            return vals[0]
        else:
            return mymax(vals[1:])
```



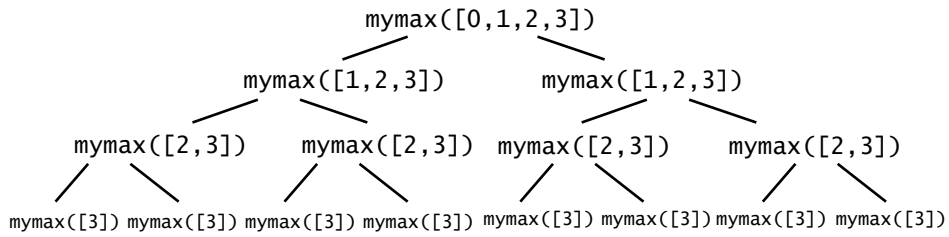
How recursion works...

```
def mymax(vals):
    if len(vals) == 1:
        return vals[0]
    else:
        if vals[0] > mymax(vals[1:]):
            return vals[0]
        else:
            return mymax(vals[1:])
```



How recursion works...

```
def mymax(vals):
    if len(vals) == 1:
        return vals[0]
    else:
        if vals[0] > mymax(vals[1:]):
            return vals[0]
        else:
            return mymax(vals[1:])
```

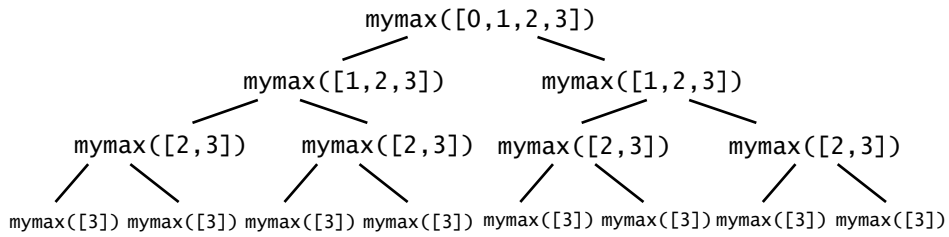


number of calls for a list of length 4 = 15

number of calls for a list of length n = ???

How recursion works...

```
def mymax(vals):
    if len(vals) == 1:
        return vals[0]
    else:
        if vals[0] > mymax(vals[1:]):
            return vals[0]
        else:
            return mymax(vals[1:])
```



number of calls for a list of length 4 = 15

number of calls for a list of length n = $2^n - 1$ ← gets big fast!!!
exponential growth