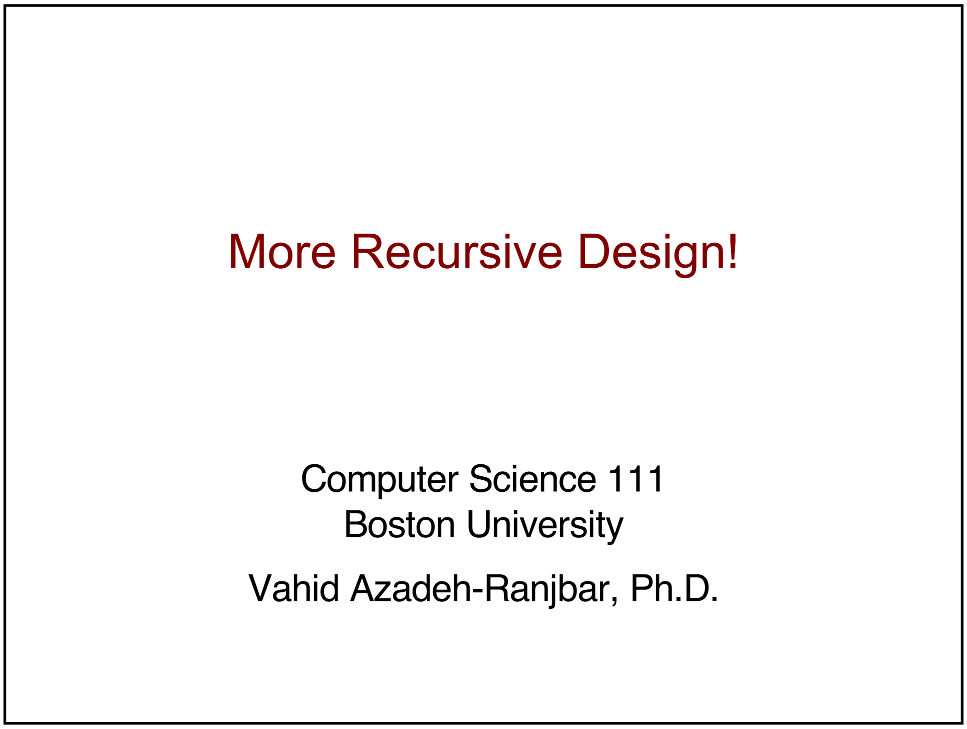
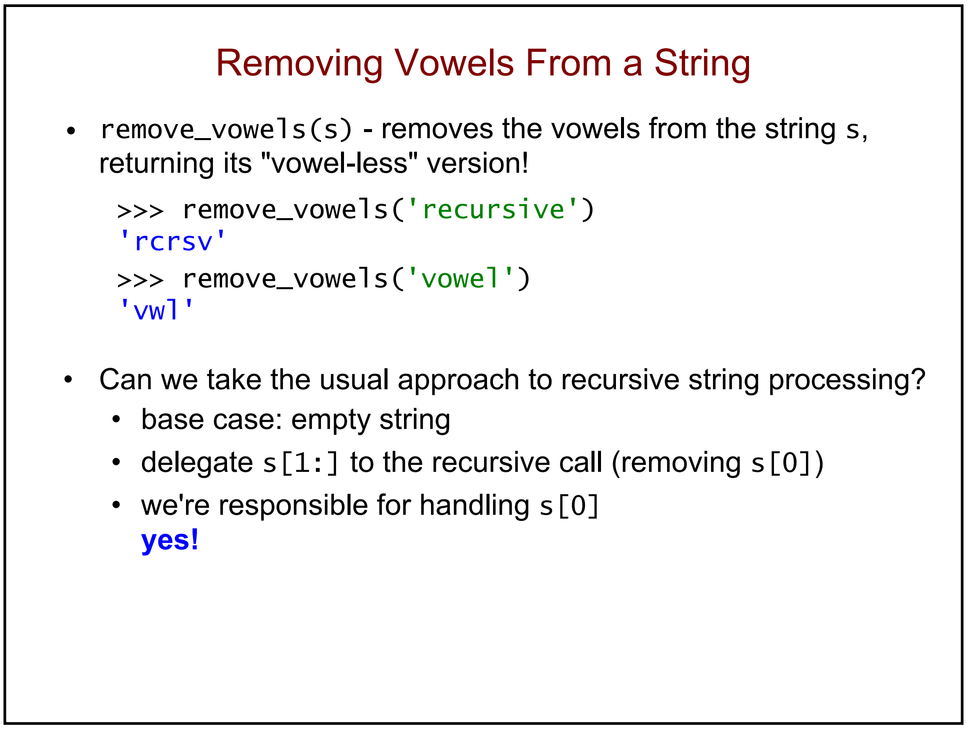


More Recursive Design!

Computer Science 111
Boston University

Vahid Azadeh-Ranjbar, Ph.D.

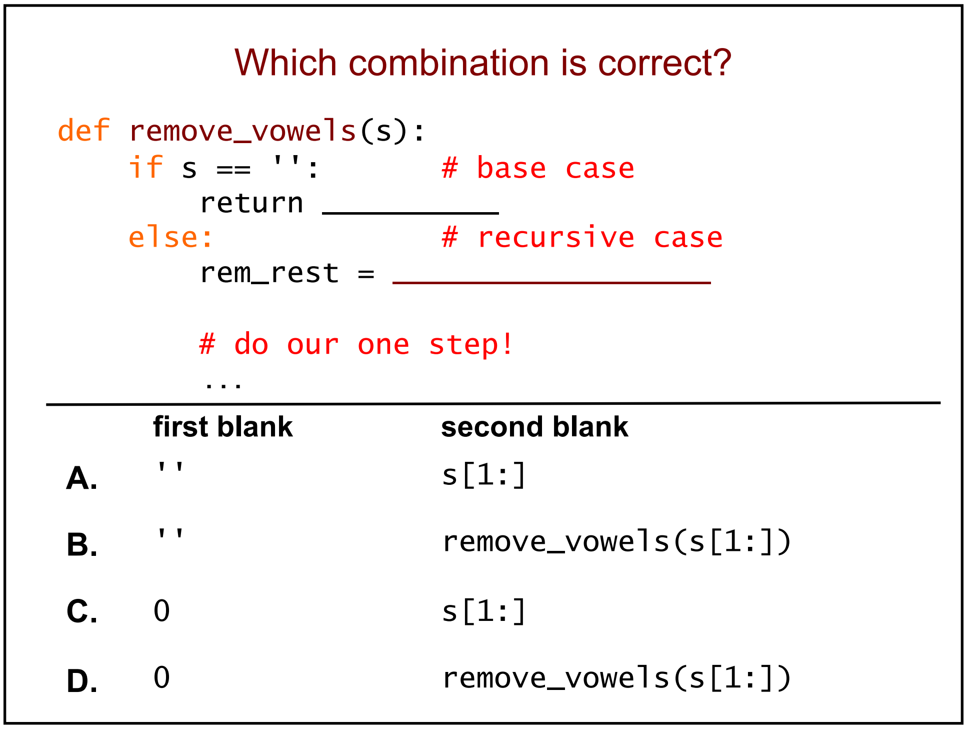


Removing Vowels From a String

- `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 recursive string processing?
  - base case: empty string
  - delegate `s[1:]` to the recursive call (removing `s[0]`)
  - we're responsible for handling `s[0]`
    yes!

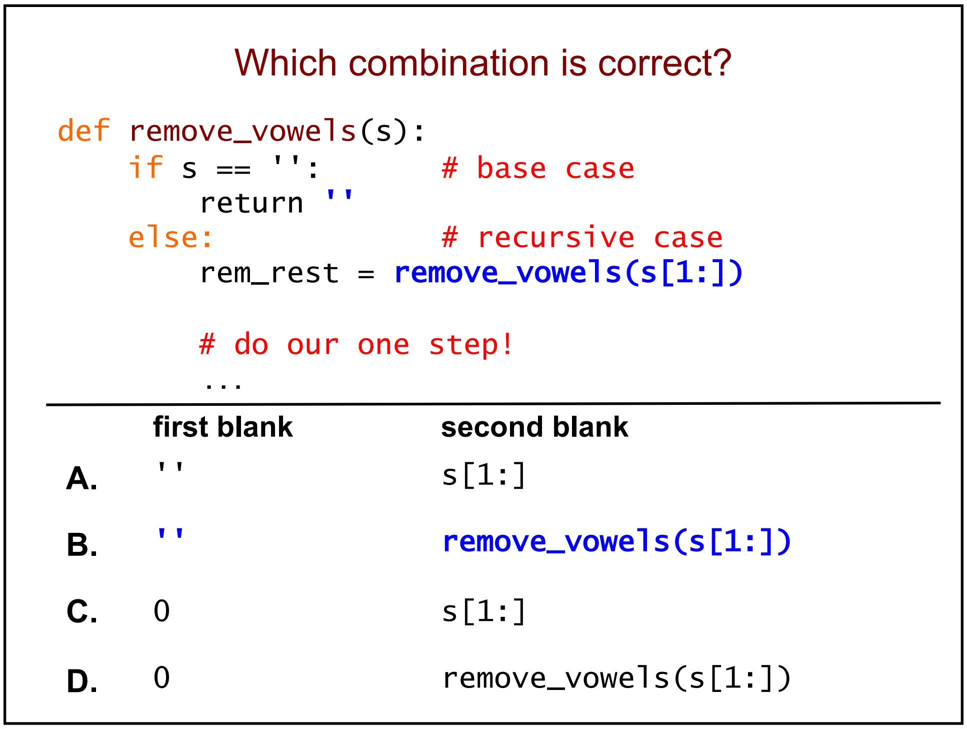


Which combination is correct?

```
def remove_vowels(s):
    if s == '':         # base case
        return __________
    else:               # recursive case
        rem_rest = __________________

        # do our one step!
        ...
```

	first blank	second blank
A.	`''`	`s[1:]`
B.	`''`	`remove_vowels(s[1:])`
C.	`0`	`s[1:]`
D.	`0`	`remove_vowels(s[1:])`

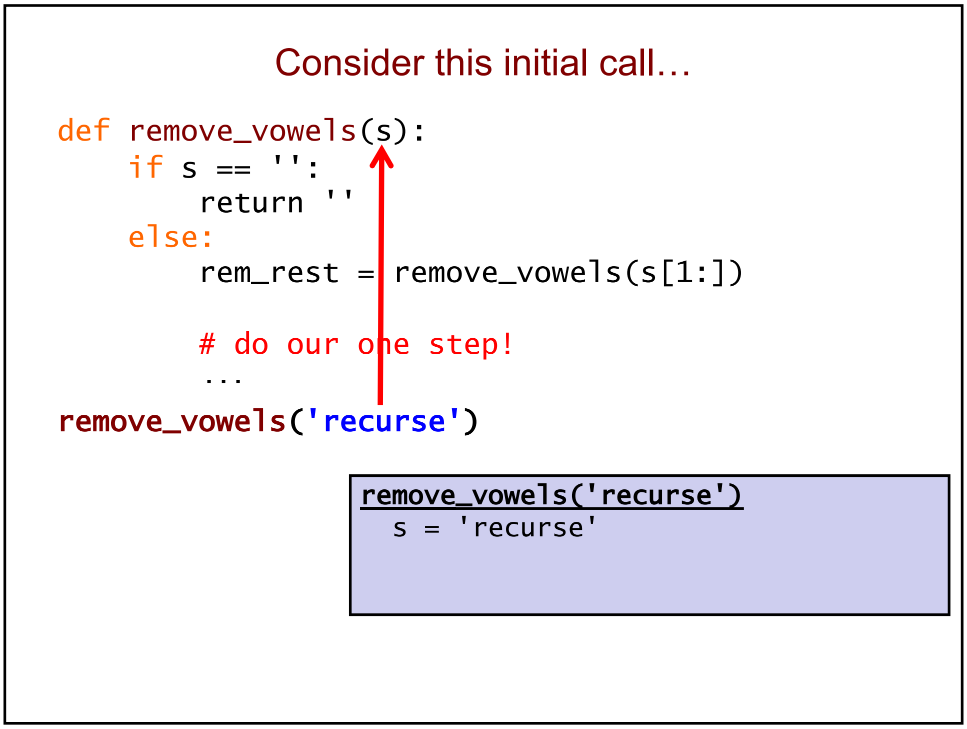


Which combination is correct?

```
def remove_vowels(s):
    if s == '':         # base case
        return ''
    else:               # recursive case
        rem_rest = remove_vowels(s[1:])

        # do our one step!
        ...
```

	first blank	second blank
A.	`''`	`s[1:]`
B.	`''`	`remove_vowels(s[1:])`
C.	`0`	`s[1:]`
D.	`0`	`remove_vowels(s[1:])`

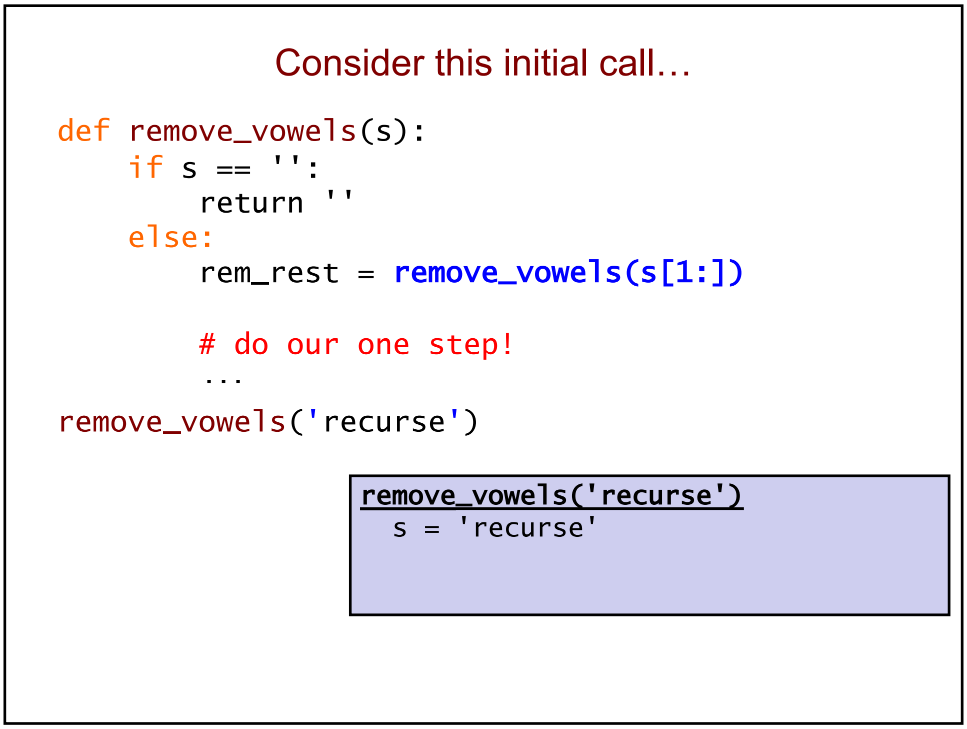


Consider this initial call…

```
def remove_vowels(s):
    if s == '':
        return ''
    else:
        rem_rest = remove_vowels(s[1:])

        # do our one step!
        ...
remove_vowels('recurse')
```

```
remove_vowels('recurse')
  s = 'recurse'
```

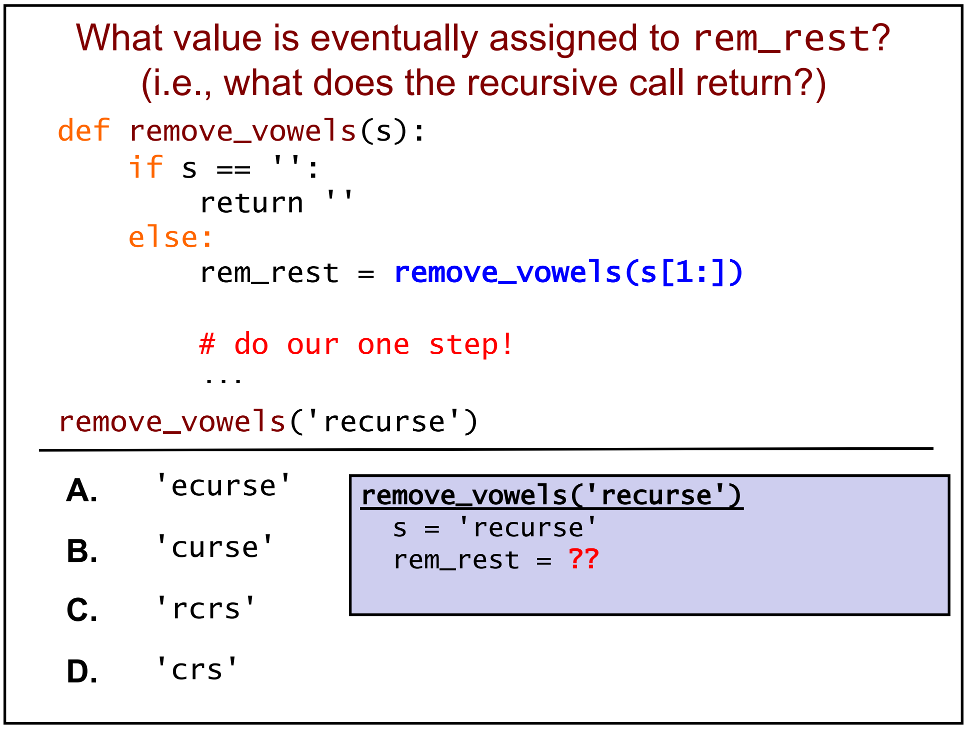


Consider this initial call…

```
def remove_vowels(s):
    if s == '':
        return ''
    else:
        rem_rest = remove_vowels(s[1:])

        # do our one step!
        ...
remove_vowels('recurse')
```

```
remove_vowels('recurse')
  s = 'recurse'
```



What value is eventually assigned to `rem_rest`?
(i.e., what does the recursive call return?)

```
def remove_vowels(s):
    if s == '':
        return ''
    else:
        rem_rest = remove_vowels(s[1:])

        # do our one step!
        ...
remove_vowels('recurse')
```

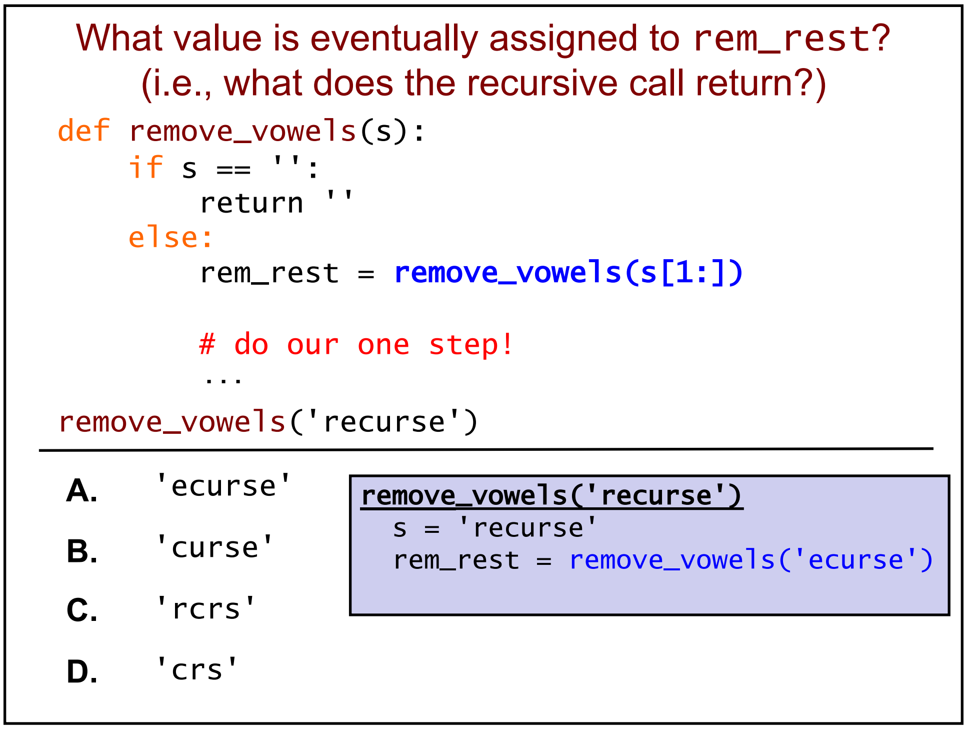
A. 'ecurse'

B. 'curse'

C. 'rcrs'

D. 'crs'

```
remove_vowels('recurse')
  s = 'recurse'
  rem_rest = ??
```



What value is eventually assigned to `rem_rest`?
(i.e., what does the recursive call return?)

```
def remove_vowels(s):
    if s == '':
        return ''
    else:
        rem_rest = remove_vowels(s[1:])

        # do our one step!
        ...
remove_vowels('recurse')
```

A. 'ecurse'

B. 'curse'

C. 'rcrs'

D. 'crs'

```
remove_vowels('recurse')
  s = 'recurse'
  rem_rest = remove_vowels('ecurse')
```

What value is eventually assigned to rem_rest? (i.e., what does the recursive call return?)

```
def remove_vowels(s):
    if s == '':
        return ''
    else:
        rem_rest = remove_vowels(s[1:])

        # do our one step!
        ...
remove_vowels('recurse')
```

A. 'ecurse'

B. 'curse'

C. 'rcrs'

D. **'crs'**

```
remove_vowels('recurse')
  s = 'recurse'
  rem_rest = remove_vowels('ecurse')
           = 'crs'
```

Applying the String-Processing Template

```
def remove_vowels(s):
    if s == '':
        return ''
    else:
        rem_rest = remove_vowels(s[1:])

        # do our one step!
        if s[0] in 'aeiou':
            ...
```

- In our one step, we take care of `s[0]`.
  - we build the solution to the larger problem on the solution to the smaller problem (in this case, `rem_rest`)
  - does what we do depend on the value of `s[0]`? yes!
    - if `s[0]` is a vowel...
    - if `s[0]` isn't a vowel...

Consider Concrete Cases

`remove_vowels('after')      # s[0] is a vowel`

- what is its ultimate 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! (`'ftr'`)

`remove_vowels('recurse')    # s[0] is not a vowel`

- what is its ultimate 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 lets write the rest of the function!

remove_vowels()

```
def remove_vowels(s):
    if s == '':
        return ''
    else:
        rem_rest = remove_vowels(s[1:])

        # do our one step!
        if s[0] in 'aeiou':
            ...
```

`remove_vowels('after')      # s[0] is a vowel`

- what is its solution? `'ftr'`
- what is the next smaller subproblem? `remove_vowels('fter')`
- what is the solution to that subproblem? `'ftr'`

`remove_vowels('recurse')    # s[0] is not a vowel`

- what is its solution? `'rcrs'`
- what is the next smaller subproblem? `remove_vowels('ecurse')`
- what is the solution to that subproblem? `'crs'`

remove_vowels()

```
def remove_vowels(s):
    """ returns the "vowel-less" version of s
        input s: an arbitrary string
    """
    if s == '':
        return ''
    else:
        rem_rest = remove_vowels(s[1:])

        # do our one step!
        if s[0] in 'aeiou':
            return rem_rest
        else:
            return s[0] + rem_rest
```

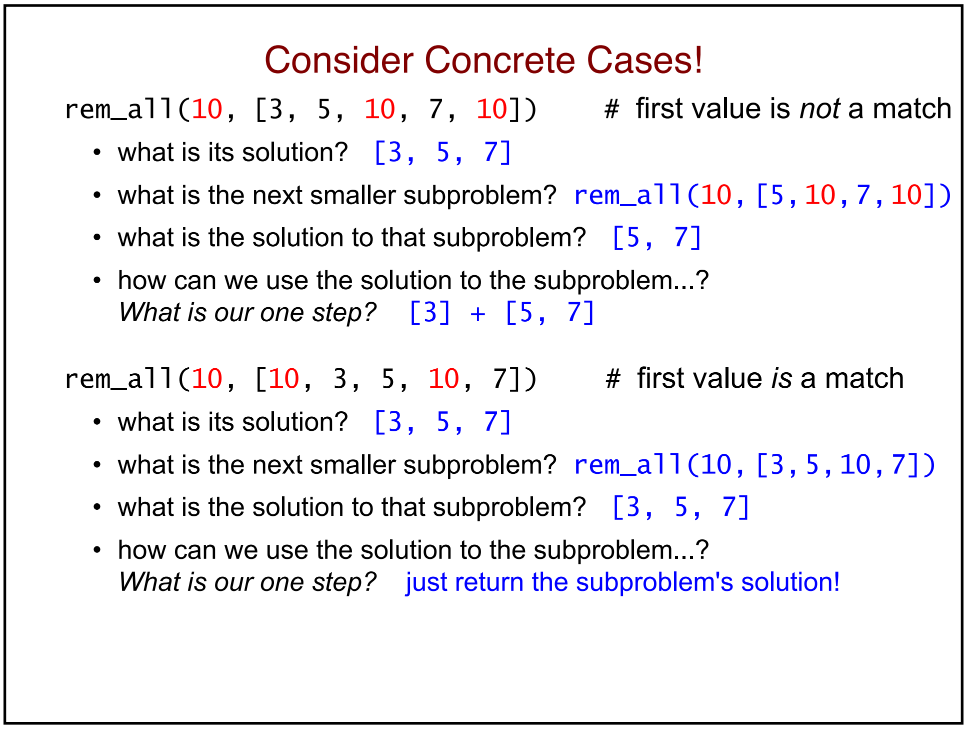
More Recursive Design! rem_all()

- `rem_all(elem, values)`
  - inputs: an arbitrary value (`elem`) and a list (`values`)
  - returns: a version of `values` in which *all* occurrences of `elem` in `values` (if any) are removed

```
>>> rem_all(10, [3, 5, 10, 7, 10])
[3, 5, 7]
```

- Can we take the usual approach to processing a list recursively?
  - base case: empty list
  - delegate `values[1:]` to the recursive call
  - we're responsible for handling `values[0]` **yes!**
- What are the possible cases for our part (`values[0]`)?
  - does what we do with our part depend on its value? **yes!**

    whether `values[0]` matches the value being removed



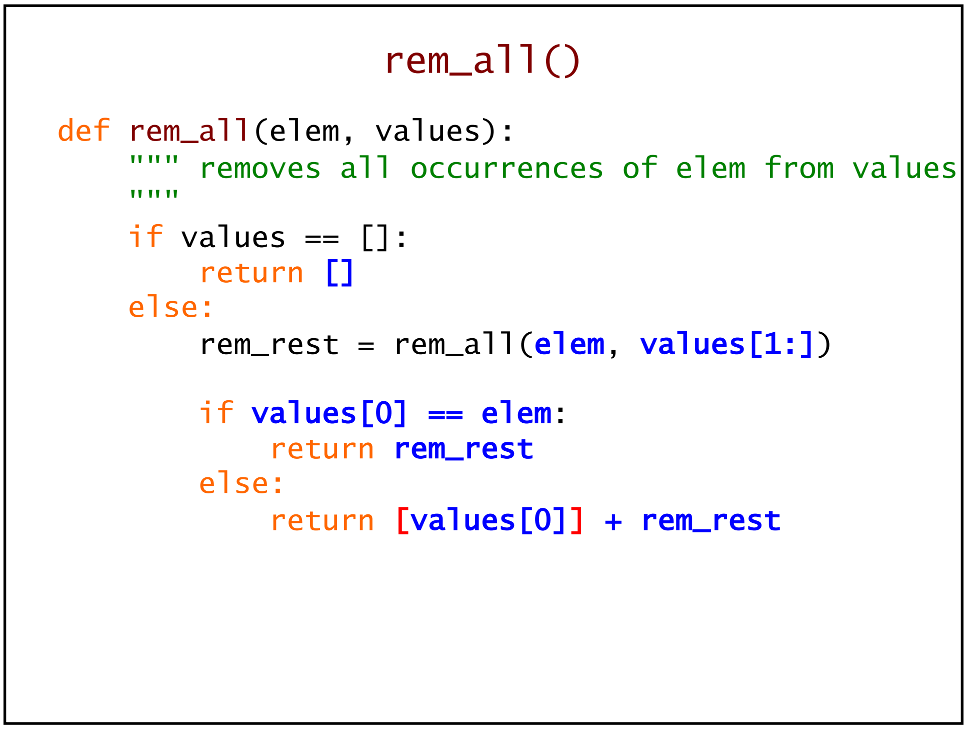
Consider Concrete Cases!

`rem_all(10, [3, 5, 10, 7, 10])` # first value is *not* a match

- what is its solution? `[3, 5, 7]`
- what is the next smaller subproblem? `rem_all(10,[5,10,7,10])`
- what is the solution to that subproblem? `[5, 7]`
- how can we use the solution to the subproblem...?
  What is our one step? `[3] + [5, 7]`

`rem_all(10, [10, 3, 5, 10, 7])` # first value *is* a match

- what is its solution? `[3, 5, 7]`
- what is the next smaller subproblem? `rem_all(10,[3,5,10,7])`
- what is the solution to that subproblem? `[3, 5, 7]`
- how can we use the solution to the subproblem...?
  What is our one step? just return the subproblem's solution!

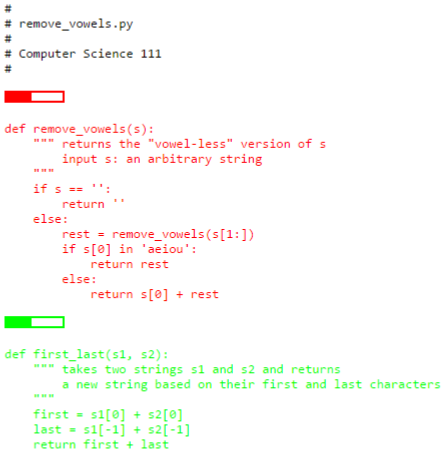


rem_all()

```
def rem_all(elem, values):
    """ removes all occurrences of elem from values
    """
    if values == []:
        return []
    else:
        rem_rest = rem_all(elem, values[1:])

        if values[0] == elem:
            return rem_rest
        else:
            return [values[0]] + rem_rest
```

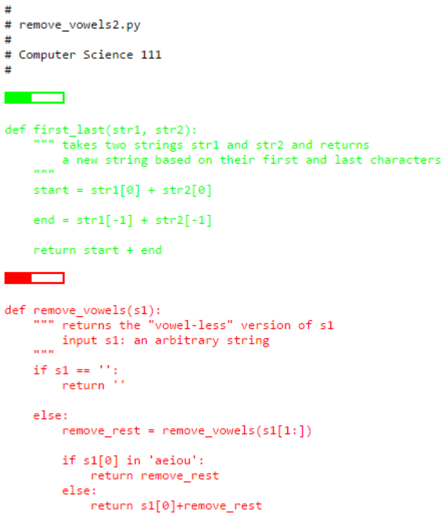
PSA: Follow the Collaboration Policies!



```
#
# remove_vowels.py
#
# Computer Science 111
#

def remove_vowels(s):
    """ returns the "vowel-less" version of s
        input s: an arbitrary string
    """
    if s == '':
        return ''
    else:
        rest = remove_vowels(s[1:])
        if s[0] in 'aeiou':
            return rest
        else:
            return s[0] + rest

def first_last(s1, s2):
    """ takes two strings s1 and s2 and returns
        a new string based on their first and last characters
    """
    first = s1[0] + s2[0]
    last = s1[-1] + s2[-1]
    return first + last
```



```
#
# remove_vowels2.py
#
# Computer Science 111
#

def first_last(str1, str2):
    """ takes two strings str1 and str2 and returns
        a new string based on their first and last characters
    """
    start = str1[0] + str2[0]

    end = str1[-1] + str2[-1]

    return start + end

def remove_vowels(s1):
    """ returns the "vowel-less" version of s1
        input s1: an arbitrary string
    """
    if s1 == '':
        return ''

    else:
        remove_rest = remove_vowels(s1[1:])

        if s1[0] in 'aeiou':
            return remove_rest
        else:
            return s1[0]+remove_rest
```

Don't look at someone else's code!
Don't show your code to someone else!

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
```

- Can we take the usual approach to processing it recursively? No!
  - base case: empty list
  - delegate `s[1:]` to the recursive call
  - we're responsible for handling `s[0]`

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?

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

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

- 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?

  use a slice that omits both the first and last characters

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:              # recursive case
        is_pal_rest = ____________________

        # do our one step!
```

Try to complete it at home!!!