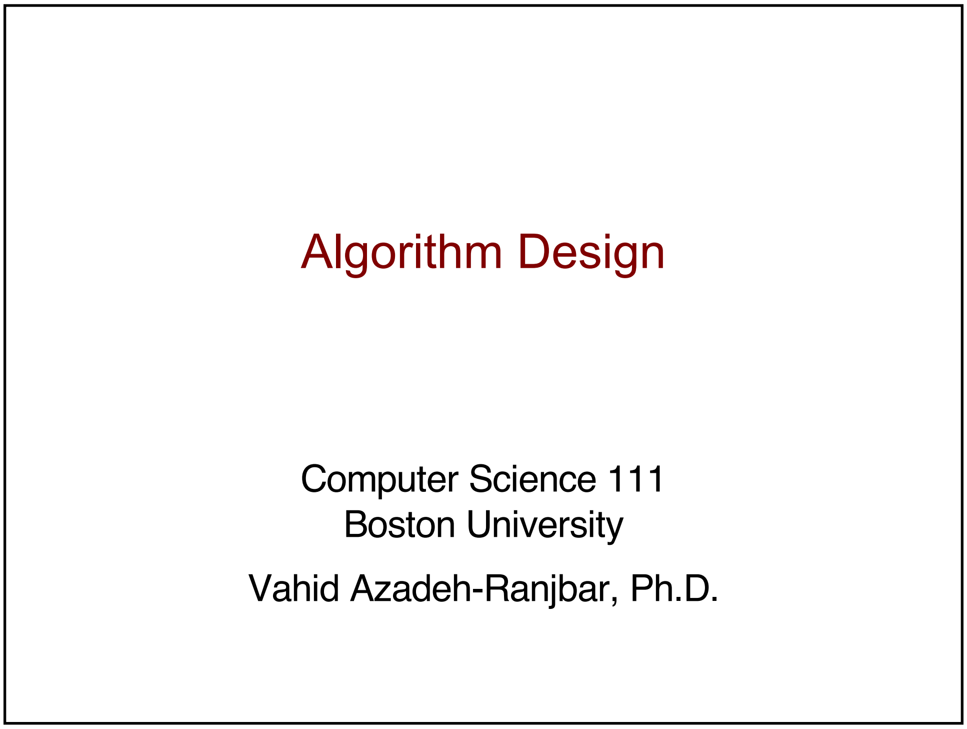
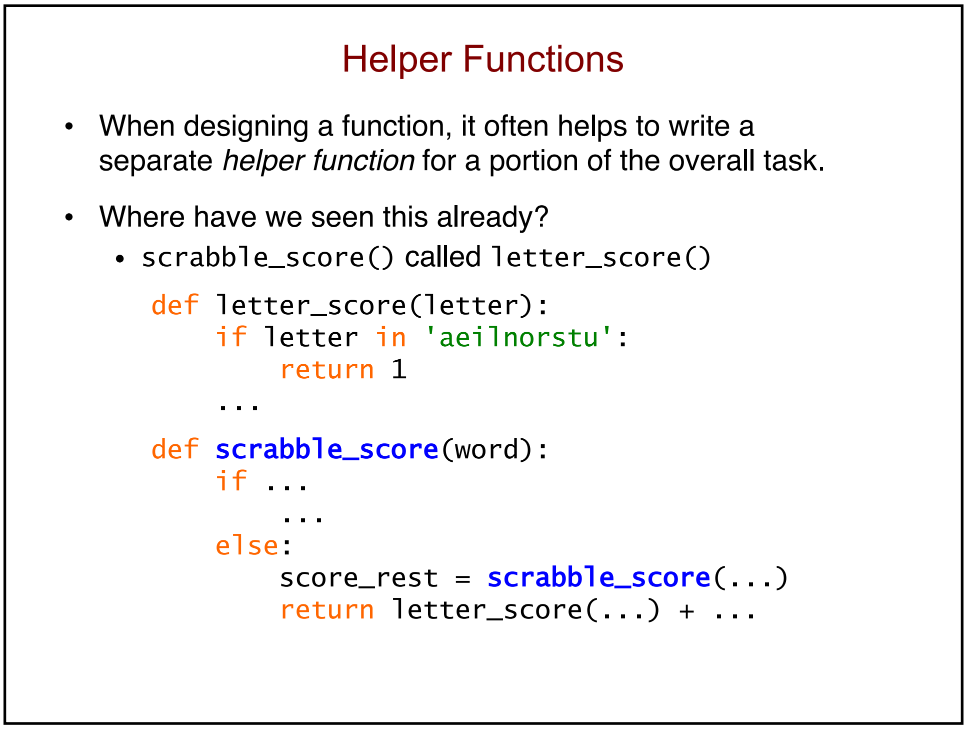


# Algorithm Design

Computer Science 111
Boston University

Vahid Azadeh-Ranjbar, Ph.D.



## Helper Functions

- When designing a function, it often helps to write a separate *helper function* for a portion of the overall task.
- Where have we seen this already?
  - `scrabble_score()` called `letter_score()`

```
def letter_score(letter):
    if letter in 'aeilnorstu':
        return 1
    ...

def scrabble_score(word):
    if ...
        ...
    else:
        score_rest = scrabble_score(...)
        return letter_score(...) + ...
```

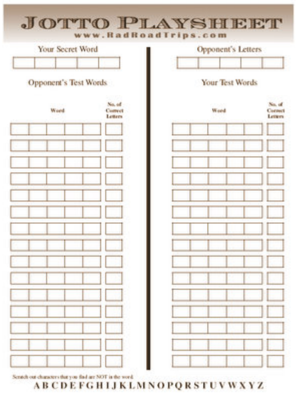
## Helper Functions

- When designing a function, it often helps to write a separate *helper function* for a portion of the overall task.
- Where have we seen this already?
  - scrabble_score() called letter_score()

```
def letter_score(letter):
    if letter in 'aeilnorstu':
        return 1
    ...
def scrabble_score(word):
    if ...
        ...
    else:
        score_rest = scrabble_score(...)
        return letter_score(...) + ...
```

  - other places as well!

## In PS 3: Jotto Score



- jscore(s1, s2)
  - returns the number of characters in s1 that are shared by s2
  - the positions and the order of the characters do *not* matter
  - repeated letters are counted multiple times
- Examples:
  - jscore('diner', 'syrup') → 1
  - jscore('always', 'bananas') → 3
  - jscore('always', 'walking') → 3

## What will this call return?

```
jscore('recursion', 'explorations')
```

A. 4
B. 5
C. 6
D. 7
E. none of the above

## What will this call return?

```
jscore('recursion', 'explorations')
```

A. 4
B. 5
C. **6**
D. 7
E. none of the above

## Jotto Score: Consider Concrete Cases

```
jscore('always', 'walking')
```

Can we take the usual approach to recursive string processing?

```
def jscore(s1,s2):
    if ____________:
        return _____
    else:
        j_rest = _____________

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

## Jotto Score: Consider Concrete Cases

```
jscore('always', 'walking')
```

Can we take the usual approach to recursive string processing?

```
def jscore(s1,s2):
    if they are empty:
        return 0
    else:
        j_rest = jscore(s1[1:], s2[1:])

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

## Jotto Score: Consider Concrete Cases

jscore('~~a~~lways', '~~w~~alking')

- what is its solution? 3
- what is the next smaller subproblem?
  - will jscore('lways', 'alking') work?

    no! – it will miss the shared 'w'

## Jotto Score: Consider Concrete Cases

jscore('always', 'walking')

How about this approach?

```
def jscore(s1,s2):
    if they are empty:
        return 0
    else:
        j_rest = jscore(s1[1:], s2)

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

## Jotto Score: Consider Concrete Cases

`jscore('always', 'walking')`

- what is its solution? 3
- what is the next smaller subproblem?
  - will `jscore('lways', 'alking')` work?

    no! – it will miss the shared `'w'`
  - will `jscore('lways', 'walking')` work?

    no! – it will find another shared `'a'`, but there's only one

## Jotto Score: Consider Concrete Cases

`jscore('always', 'walking')`

- what is its solution? 3
- what is the next smaller subproblem?
  - will `jscore('lways', 'alking')` work?

    no! – it will miss the shared `'w'`
  - will `jscore('lways', 'walking')` work?

    no! – it will find another shared `'a'`, but there's only one
  - what should we do instead?

```
jscore('lways', 'wlking')   # removed one 'a'
                            # from 'walking'
```

Need a ***helper function*** to remove one occurrence of a character from a **string**… Let's look at a similar function for **lists**.

## Look Familiar?

- 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]
```

- rem_**first**(elem, values)
  - inputs:  an arbitrary value (elem) and a list (values)
  - returns: a version of values in which ***only the first*** occurrence of elem in values (if any) is removed

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

## How Can We Adapt 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
```

## What Other Changes Are Needed?

```
def rem_first(elem, values):
    """ removes the first occurrence of elem from
        values
    """
    if values == []:
        return []
    else:
        rem_rest = rem_first(elem, values[1:])

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

## Consider Concrete Cases!

`rem_first(10, [3, 5, 10, 7, 10])`

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

`rem_first(10, [10, 3, 5, 10, 7])`

- what is its solution? **`[3, 5, 10, 7]`**
- what is the next smaller subproblem? `rem_first(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?* we can't easily use it!!
  what could we do instead?

## What Other Changes Are Needed?

```
def rem_first(elem, values):
    """ removes the first occurrence of elem from
        values
    """
    if values == []:
        return []
    else:
        rem_rest = rem_first(elem, values[1:])

        if values[0] == elem:
            return values[1:]
        else:
            return [values[0]] + rem_rest
```

## What Other Changes Are Needed?

```
def rem_first(elem, values):
    """ removes the first occurrence of elem from
        values
    """
    if values == []:
        return []
    else:
        rem_rest = rem_first(elem, values[1:])

        if values[0] == elem:
            return values[1:]
        else:
            return [values[0]] + rem_rest
```

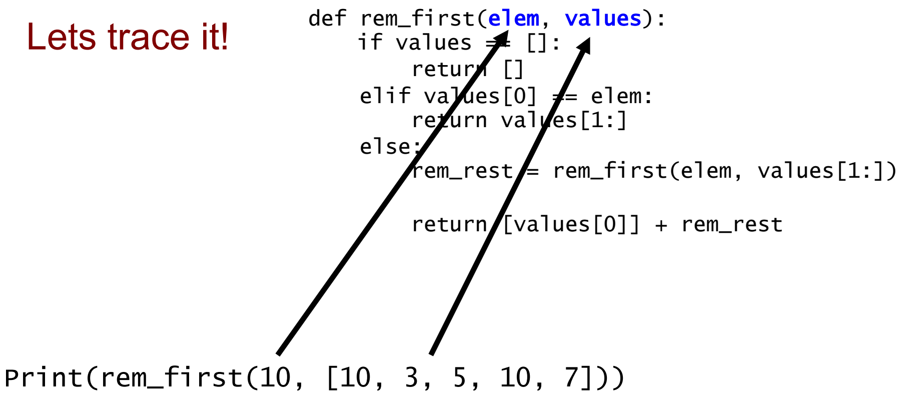
## Done!

```
def rem_first(elem, values):
    """ removes the first occurrence of elem from
        values
    """
    if values == []:
        return []
    elif values[0] == elem:     # now a base case!
        return values[1:]
    else:
        rem_rest = rem_first(elem, values[1:])

        return [values[0]] + rem_rest
```

Lets trace the function with some test calls.

## Lets trace it!

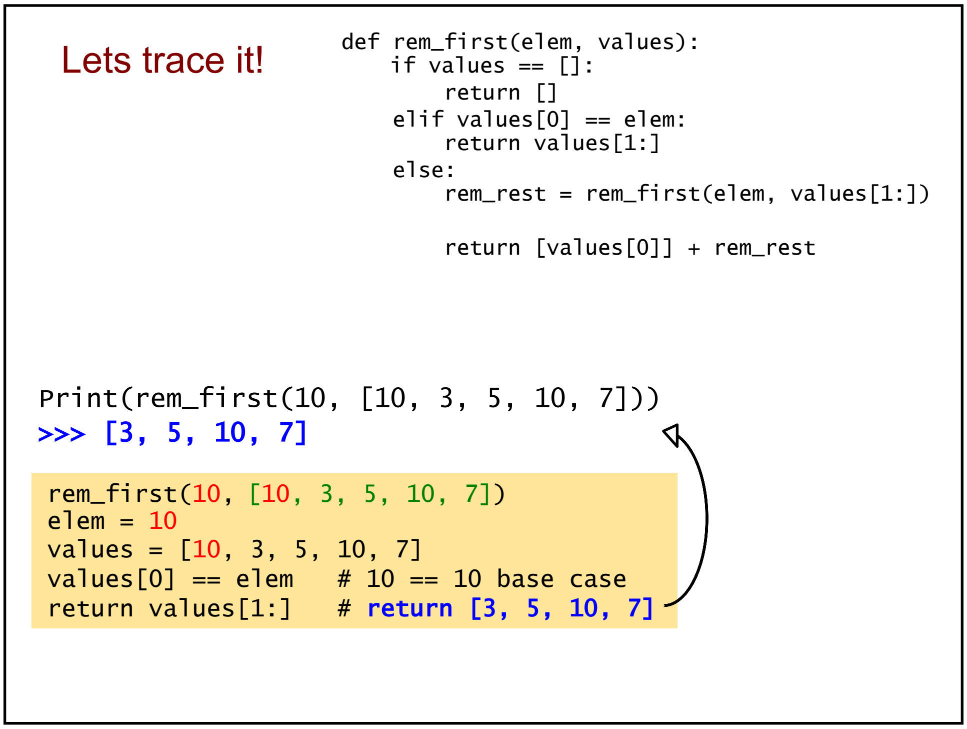


```
def rem_first(elem, values):
    if values == []:
        return []
    elif values[0] == elem:
        return values[1:]
    else:
        rem_rest = rem_first(elem, values[1:])

        return [values[0]] + rem_rest
```

```
Print(rem_first(10, [10, 3, 5, 10, 7]))
```

```
rem_first(10, [10, 3, 5, 10, 7])
elem = 10
values = [10, 3, 5, 10, 7]
```



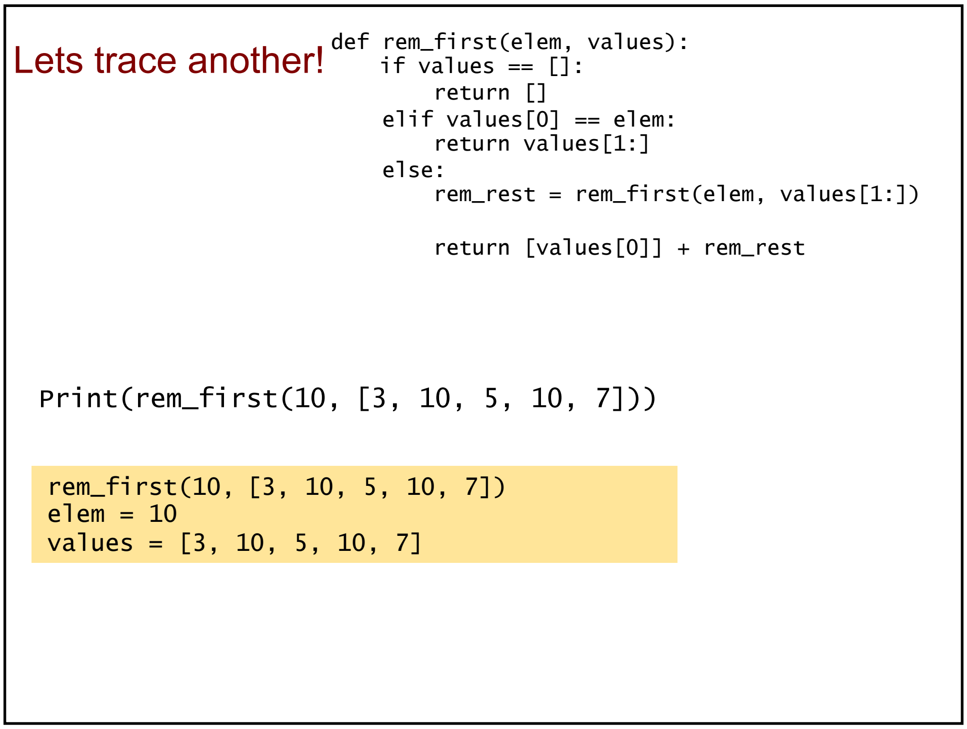
## Lets trace it!

```
def rem_first(elem, values):
    if values == []:
        return []
    elif values[0] == elem:
        return values[1:]
    else:
        rem_rest = rem_first(elem, values[1:])

        return [values[0]] + rem_rest
```

```
Print(rem_first(10, [10, 3, 5, 10, 7]))
>>> [3, 5, 10, 7]
```

```
rem_first(10, [10, 3, 5, 10, 7])
elem = 10
values = [10, 3, 5, 10, 7]
values[0] == elem   # 10 == 10 base case
return values[1:]   # return [3, 5, 10, 7]
```



## Lets trace another!

```
def rem_first(elem, values):
    if values == []:
        return []
    elif values[0] == elem:
        return values[1:]
    else:
        rem_rest = rem_first(elem, values[1:])

        return [values[0]] + rem_rest
```

```
Print(rem_first(10, [3, 10, 5, 10, 7]))
```

```
rem_first(10, [3, 10, 5, 10, 7])
elem = 10
values = [3, 10, 5, 10, 7]
```

## Lets trace another!

```
def rem_first(elem, values):
    if values == []:
        return []
    elif values[0] == elem:
        return values[1:]
    else:
        rem_rest = rem_first(elem, values[1:])

        return [values[0]] + rem_rest
```

```
Print(rem_first(10, [3, 10, 5, 10, 7]))
```

```
rem_first(10, [3, 10, 5, 10, 7])
elem = 10
values = [3, 10, 5, 10, 7]
```

## Lets trace another!

```
def rem_first(elem, values):
    if values == []:
        return []
    elif values[0] == elem:
        return values[1:]
    else:
        rem_rest = rem_first(elem, values[1:])

        return [values[0]] + rem_rest
```

```
Print(rem_first(10, [3, 10, 5, 10, 7]))
```

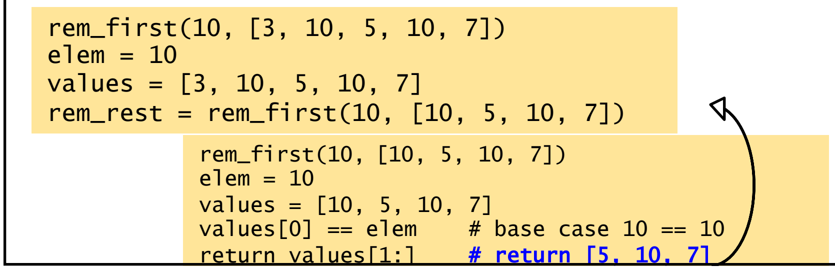
```
rem_first(10, [3, 10, 5, 10, 7])
elem = 10
values = [3, 10, 5, 10, 7]
rem_rest = rem_first(10, [10, 5, 10, 7])
```

## Lets trace another!

```
def rem_first(elem, values):
    if values == []:
        return []
    elif values[0] == elem:
        return values[1:]
    else:
        rem_rest = rem_first(elem, values[1:])

        return [values[0]] + rem_rest
```

```
Print(rem_first(10, [3, 10, 5, 10, 7]))
```



```
rem_first(10, [3, 10, 5, 10, 7])
elem = 10
values = [3, 10, 5, 10, 7]
rem_rest = rem_first(10, [10, 5, 10, 7])
```

```
rem_first(10, [10, 5, 10, 7])
elem = 10
values = [10, 5, 10, 7]
values[0] == elem    # base case 10 == 10
return values[1:]    # return [5, 10, 7]
```

## Lets trace another!

```
def rem_first(elem, values):
    if values == []:
        return []
    elif values[0] == elem:
        return values[1:]
    else:
        rem_rest = rem_first(elem, values[1:])

        return [values[0]] + rem_rest
```

```
Print(rem_first(10, [3, 10, 5, 10, 7]))
```

```
rem_first(10, [3, 10, 5, 10, 7])
elem = 10
values = [3, 10, 5, 10, 7]
rem_rest = [5, 10, 7]
```

## Lets trace another!

```
def rem_first(elem, values):
    if values == []:
        return []
    elif values[0] == elem:
        return values[1:]
    else:
        rem_rest = rem_first(elem, values[1:])

        return [values[0]] + rem_rest
```

```
Print(rem_first(10, [3, 10, 5, 10, 7]))
>>> [3, 5, 10, 7]
```

```
rem_first(10, [3, 10, 5, 10, 7])
elem = 10
values = [3, 10, 5, 10, 7]
rem_rest = [5, 10, 7]
return [values[0]] + rem_rest  # return [3, 5, 10, 7]
```

## Done!

```
def rem_first(elem, values):
    """ removes the first occurrence of elem from
        values
    """
    if values == []:
        return []
    elif values[0] == elem:      # now a base case!
        return values[1:]
    else:
        rem_rest = rem_first(elem, values[1:])

        return [values[0]] + rem_rest
```

For the jscore() problem, modify this to work with strings!

## 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?
  - empty string
  - single character
  - outer characters don't match

## 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!
```

- 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 = is_pal(s[1:-1])

        # do our one step!
```

## Consider Concrete Cases!

`is_pal('radar')`

- what is its solution? `True`
- what is the next smaller subproblem? `is_pal('ada')`
- what is the solution to that subproblem? `True`
- how can we use the solution to the subproblem...?
  *What is our one step?* just return the soln to the subproblem!

`is_pal('modem')`

- what is its solution? `False`
- what is the next smaller subproblem? `is_pal('ode')`
- what is the solution to that subproblem? `False`
- how can we use the solution to the subproblem...?
  *What is our one step?* just return the soln to the subproblem!

## 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 = is_pal(s[1:-1])

          # do our one step!
```

## 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 = is_pal(s[1:-1])

        # do our one step!
        return is_pal_rest
```