Lecture 15

Recursion

Announcements for Today

Prelim 1

- Tonight at 7:30-9pm
  - **A–J** (Uris G01)
  - **K-Z** (Statler Auditorium)
- Graded by noon on Sun
  - Scores will be in CMS
  - In time for drop date
- Make-ups were e-mailed
  - If not, e-mail Jessica NOW

Other Announcements

- Reading: 5.8 – 5.10
- Assignment 3 now graded
  - **Mean** 94, **Median** 99
  - **Time**: 7 hrs, **StdDev**: 3 hrs
  - Unchanged from last year
- Assignment 4 posted Friday
  - Parts 1-3: Can do already
  - Part 4: material from today
  - Due two weeks from today

Recursion

- **Recursive Definition**:

  A definition that is defined in terms of itself
- **Recursive Function**:

  A function that calls itself (directly or indirectly)

- **Recursion**: If you understand the definition, stop; otherwise, see Recursion
- **Infinite Recursion**: See Infinite Recursion

A Mathematical Example: Factorial

- Non-recursive definition:

  $n! = n \times n\text{-}1 \times \ldots \times 2 \times 1$

  $= n\,(n\text{-}1 \times \ldots \times 2 \times 1)$

- Recursive definition:

  $n! = n\,(n\text{-}1)!$ for $n \geq 0$ **Recursive case**

  $0! = 1$ **Base case**

What happens if there is no base case?

Factorial as a Recursive Function

```
def factorial(n):
    """Returns: factorial of n.
    Pre: n ≥ 0 an int"""
    if n == 0:
        return 1

    return n*factorial(n-1)
```

- $n! = n\,(n-1)!$
- $0! = 1$

Base case(s)

Recursive case

What happens if there is no base case?

Example: Fibonnaci Sequence

- Sequence of numbers: 1, 1, 2, 3, 5, 8, 13, ...

  a_0 a_1 a_2 a_3 a_4 a_5 a_6

  - Get the next number by adding previous two
  - What is a_8?

A: $a_8 = 21$
B: $a_8 = 29$
C: $a_8 = 34$
D: None of these.

Example: Fibonnaci Sequence

- Sequence of numbers: 1, 1, 2, 3, 5, 8, 13, ...

  a_0 a_1 a_2 a_3 a_4 a_5 a_6

  - Get the next number by adding previous two
  - What is a_8?

A: $a_8 = 21$
B: $a_8 = 29$
C: $a_8 = 34$ **correct**
D: None of these.

Example: Fibonnaci Sequence

- Sequence of numbers: 1, 1, 2, 3, 5, 8, 13, ...

  a_0 a_1 a_2 a_3 a_4 a_5 a_6

  - Get the next number by adding previous two
  - What is a_8?
- Recursive definition:
  - $a_n = a_{n-1} + a_{n-2}$ **Recursive Case**
  - $a_0 = 1$ **Base Case**
  - $a_1 = 1$ **(another) Base Case**

Why did we need two base cases this time?

Fibonacci as a Recursive Function

```
def fibonacci(n):
    """Returns: Fibonacci no. an
    Precondition: n ≥ 0 an int"""
    if n <= 1:
        return 1

    return (fibonacci(n-1)+
            fibonacci(n-2))
```

Base case(s)

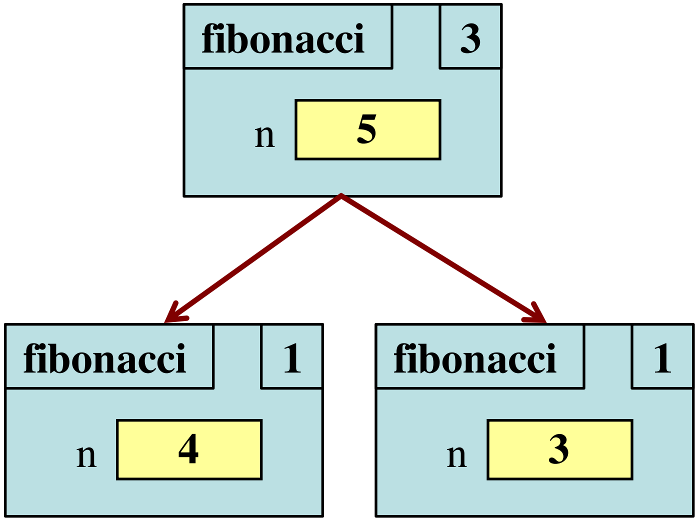
Recursive case

Note difference with base case conditional.

Fibonacci as a Recursive Function

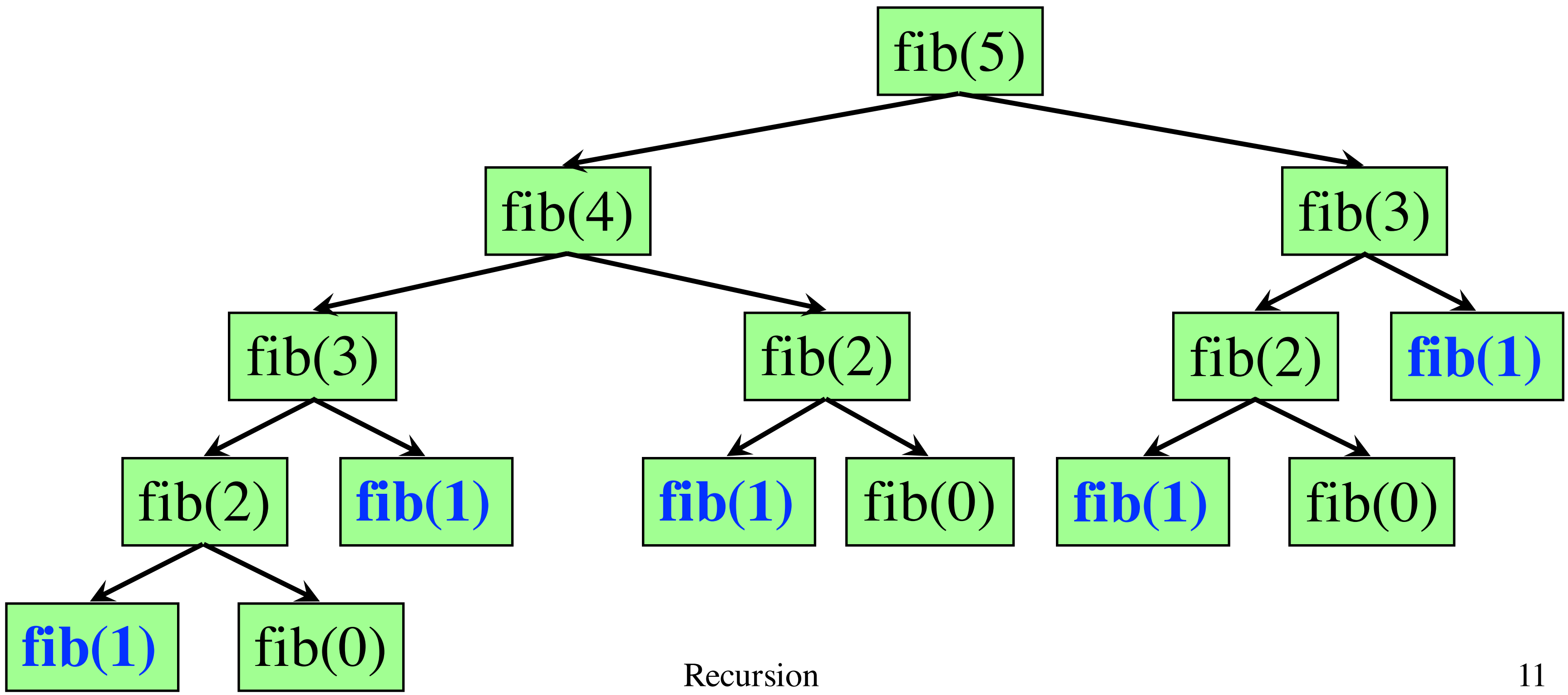
```
def fibonacci(n):
    """Returns: Fibonacci no. a_n
    Precondition: n ≥ 0 an int"""
    if n <= 1:
        return 1

    return (fibonacci(n-1)+
            fibonacci(n-2))
```

- Function that calls itself
  - Each call is new frame
  - Frames require memory
  - ∞ calls = ∞ memory



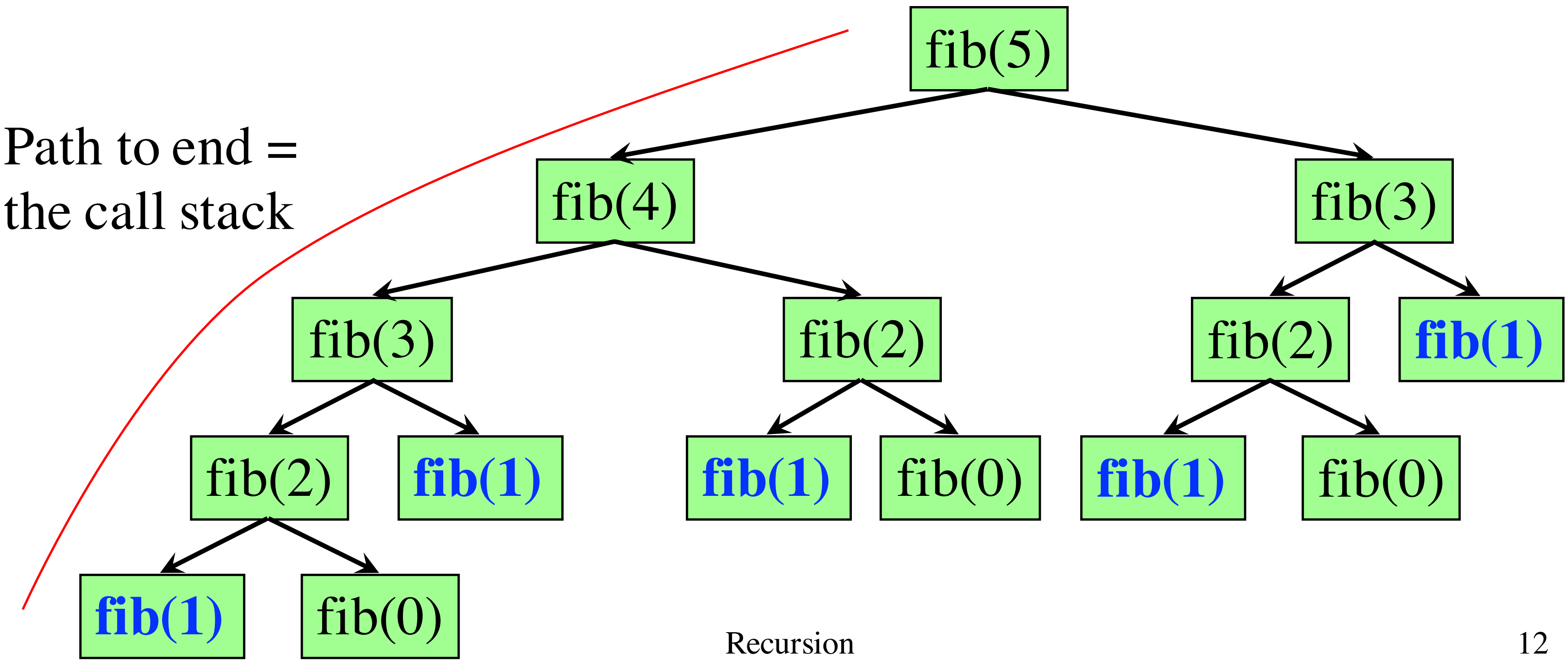
Fibonacci: # of Frames vs. # of Calls

- Fibonacci is very inefficient.
  - fib(n) has a stack that is always $\leq n$
  - But fib(n) makes a lot of redundant calls



Fibonacci: # of Frames vs. # of Calls

- Fibonacci is very inefficient.
  - fib(n) has a stack that is always $\leq n$
  - But fib(n) makes a lot of redundant calls



Two Major Issues with Recursion

- **How are recursive calls executed?**
  - We saw this with the Fibonacci example
  - Use the call frame model of execution
- **How do we understand a recursive function (and how do we create one)?**
  - You cannot trace the program flow to understand what a recursive function does – too complicated
  - You need to rely on the **function specification**

How to Think About Recursive Functions

1. **Have a precise function specification.**
2. **Base case(s):**
   - When the parameter values are as small as possible
   - When the answer is determined with little calculation.
3. **Recursive case(s):**
   - Recursive calls are used.
   - Verify recursive cases with the specification
4. **Termination:**
   - Arguments of calls must somehow get “smaller”
   - Each recursive call must get closer to a base case

Understanding the String Example

```
def num_es(s):
    """Returns: # of 'e's in s"""
    # s is empty
    if s == '':
        return 0

    # s has at least one 'e'
    if s[0] == 'e':
        return 1+num_es(s[1:])

    return num_es(s[1:]))
```

Base case

Recursive case

	0	1	len(s)
s	H	ello World!	

- Break problem into parts

> number of e's in s =
> number of e's in s[0]
> + number of e's in s[1:]

- Solve small part directly

> number of e's in s =
> number of e's in s[1:]
> (+1 if s[0] is an 'e')
> (+0 is s[0] not an 'e')

Understanding the String Example

- **Step 1:** Have a precise specification

```
def num_es(s):
    """Returns: # of 'e's in s"""
    # s is empty
    if s == '':
        return 0

    # return # of 'e's in s[0]+# of 'e's in s[1:]
    if s[0] == 'e':
        return 1+num_es(s[1:])

    return num_es(s[1:]))
```

Base case

Recursive case

"Write" your return statement using the **specification**

- **Step 2:** Check the base case
  - When s is the empty string, 0 is (correctly) returned.

Understanding the String Example

- **Step 3**: Recursive calls make progress toward termination

```
def num_es(s):
    """Returns: # of 'e's in s"""
    # s is empty
    if s == '':
        return 0

    # return # of 'e's in s[0]+# of 'e's in s[1:]
    if s[0] == 'e':
        return 1+num_es(s[1:])

    return num_es(s[1:]))
```

parameter s

argument s[1:]

argument s[1:] is smaller than parameter s, so there is progress toward reaching base case 0

- **Step 4**: Check the recursive case
  - Does it match the specification?

Exercise: Remove Blanks from a String

1. Have a precise specification

```
def deblank(s):
    """Returns: s but with its blanks removed"""
```

2. **Base Case**: the smallest String s is ''.

```
if s == '':
    return s
```

3. **Other Cases**: String s has at least 1 character.

```
return (s[0] with blanks removed) + (s[1:] with blanks removed)
```

Exercise: Remove Blanks from a String

1. Have a precise specification

```
def deblank(s):
    """Returns: s but with its blanks removed"""
```

2. **Base Case**: the smallest String s is ''.

```
if s == '':
    return s
```

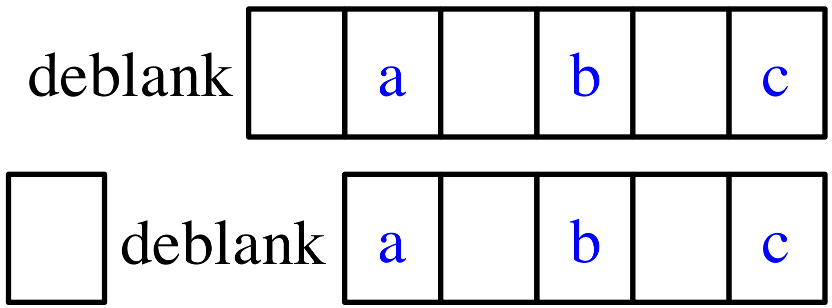
3. **Other Cases**: String s has at least 1 character.

$$\texttt{return } \underbrace{\texttt{(s[0] with blanks removed)}}_{\texttt{('' if s[0] == ' ' else s[0])}} \texttt{ + (s[1:] with blanks removed)}$$

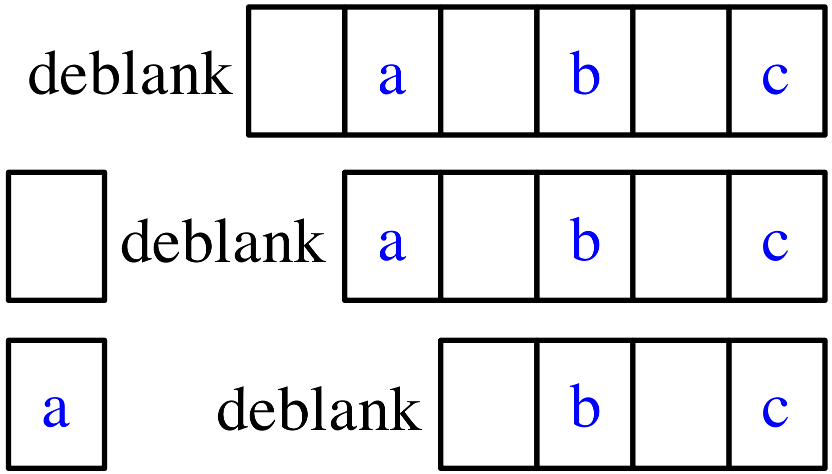
What the Recursion Does

deblank		a		b		c

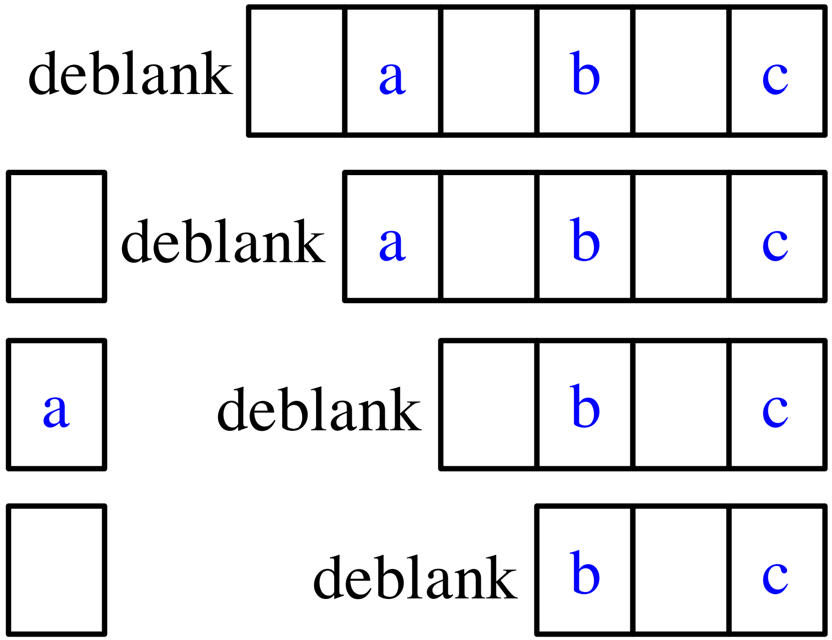
What the Recursion Does



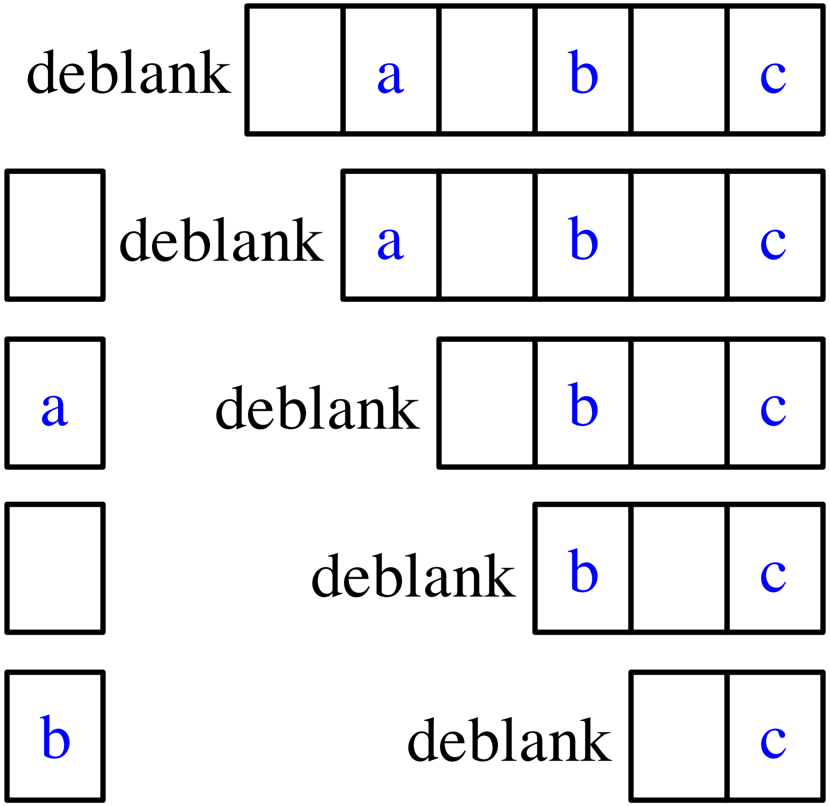
What the Recursion Does



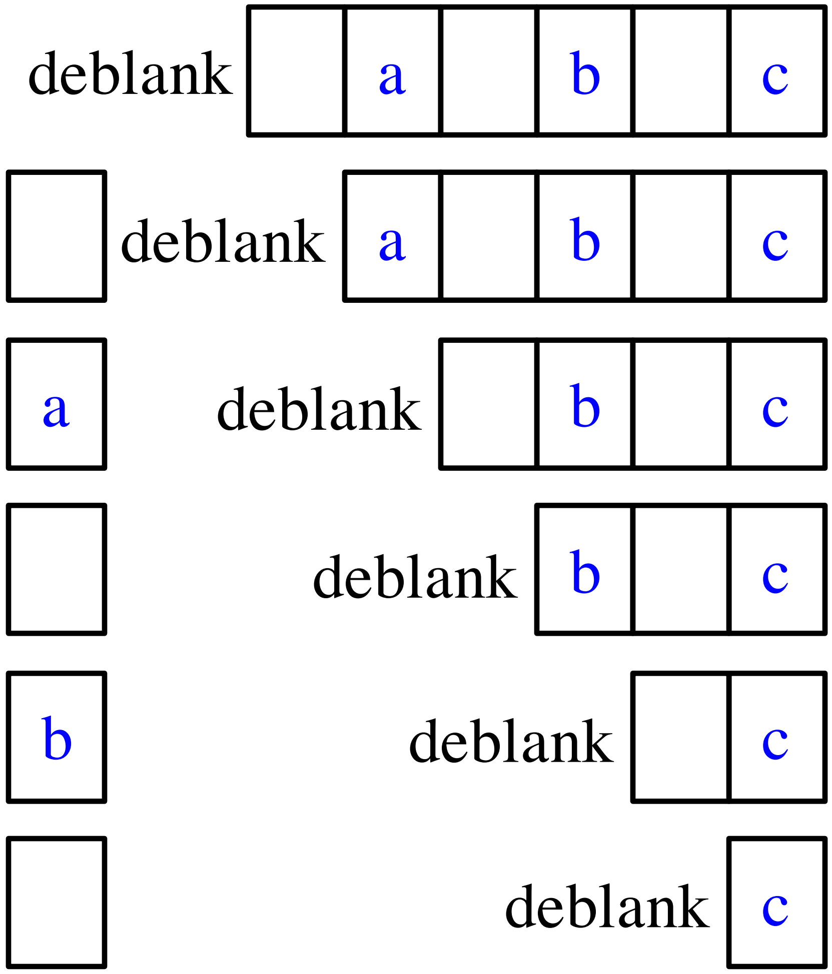
What the Recursion Does



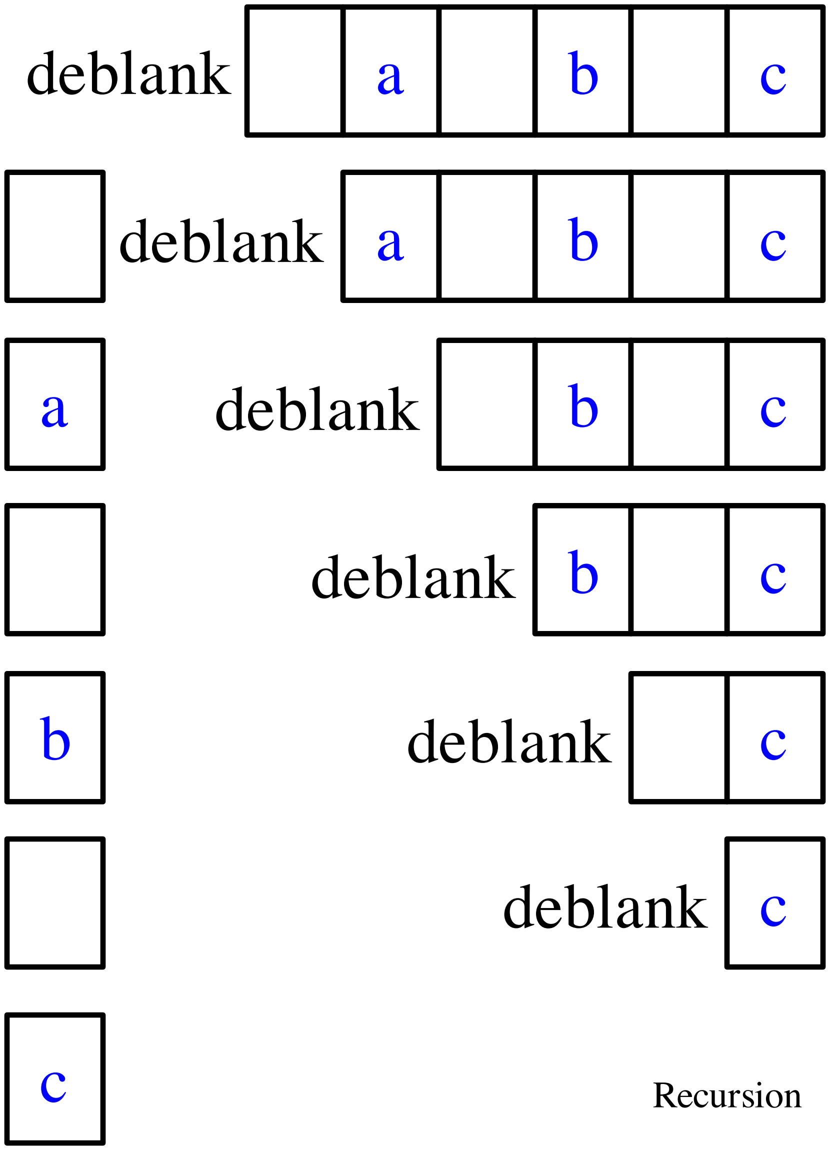
What the Recursion Does



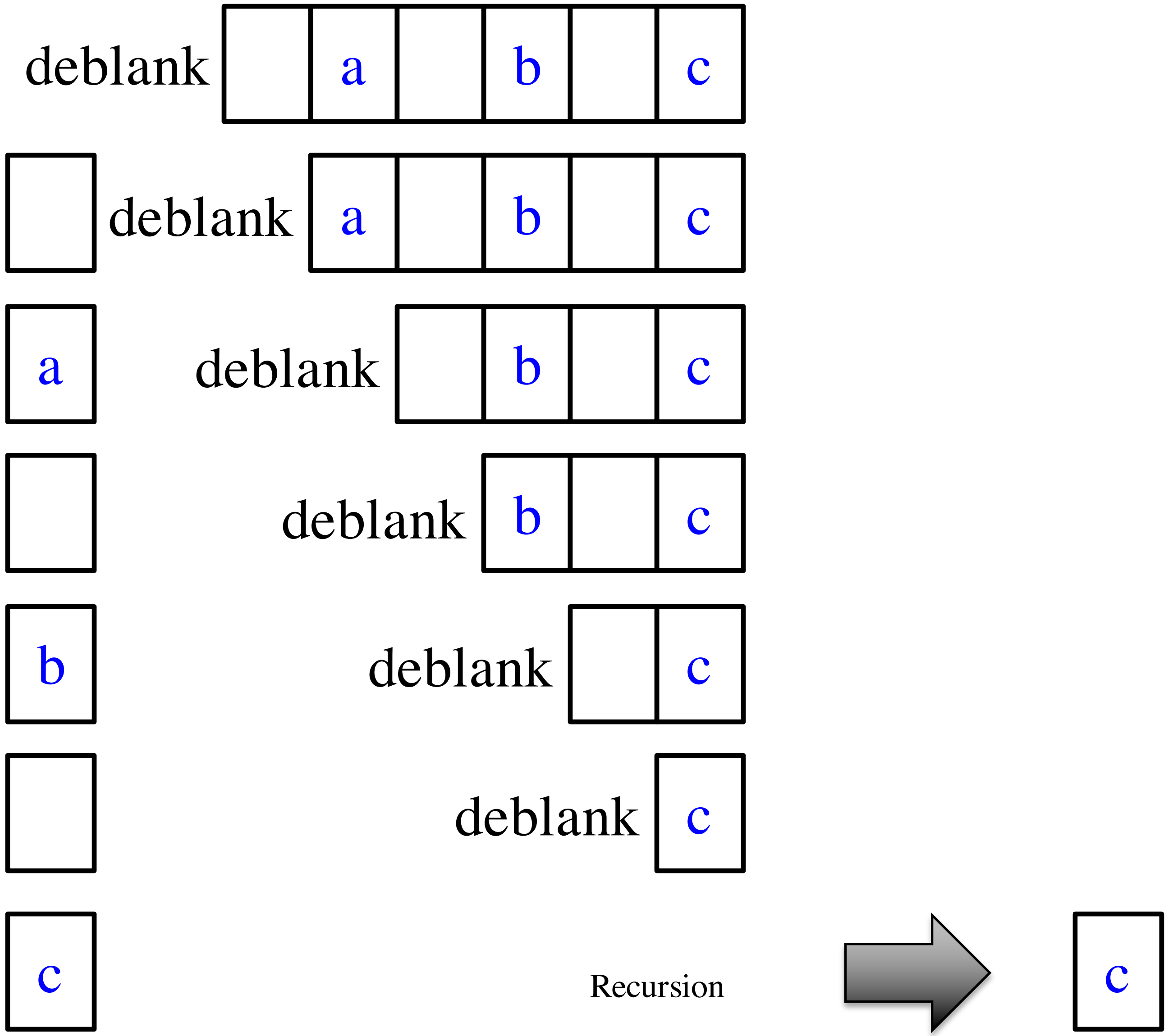
What the Recursion Does



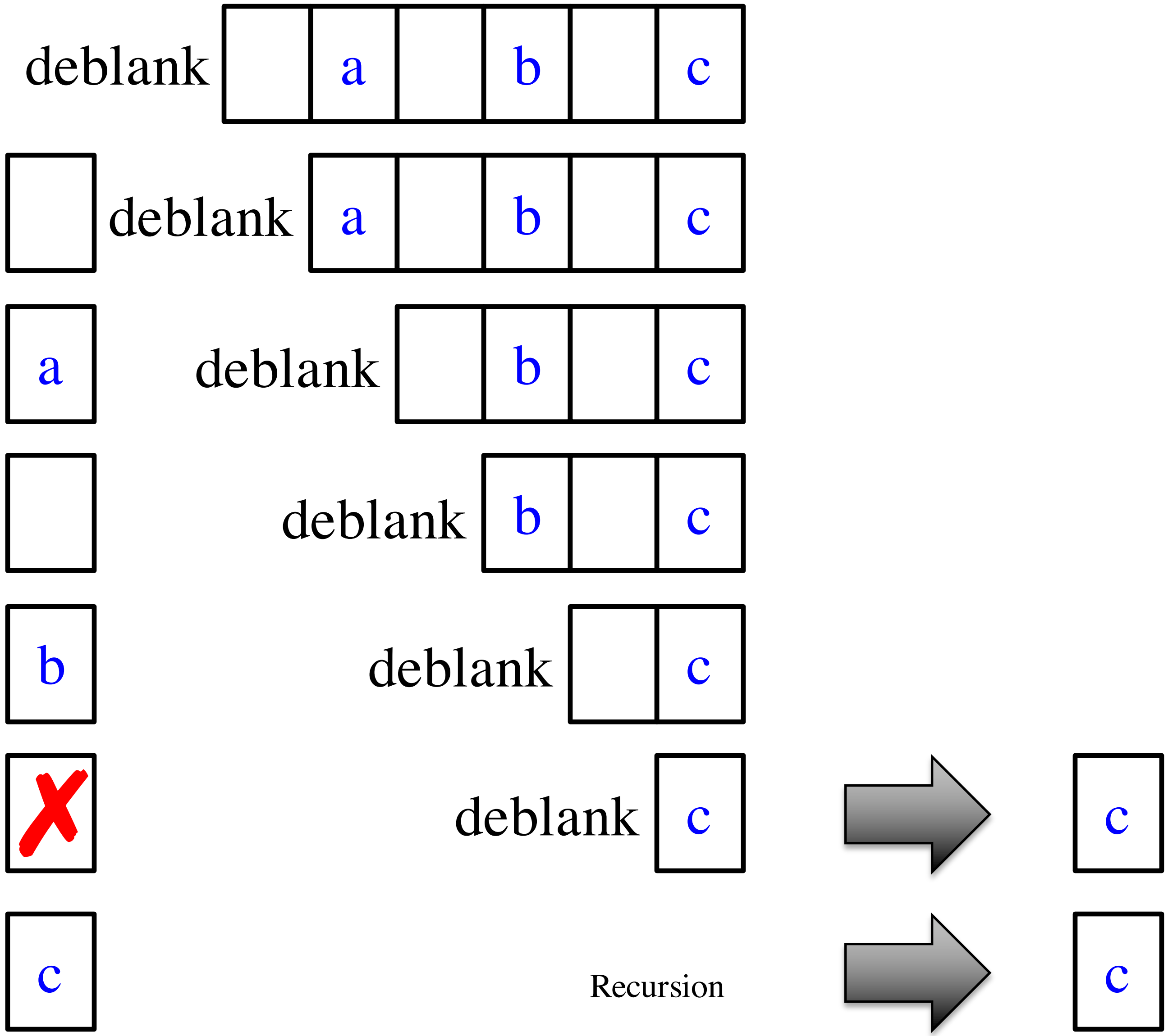
What the Recursion Does



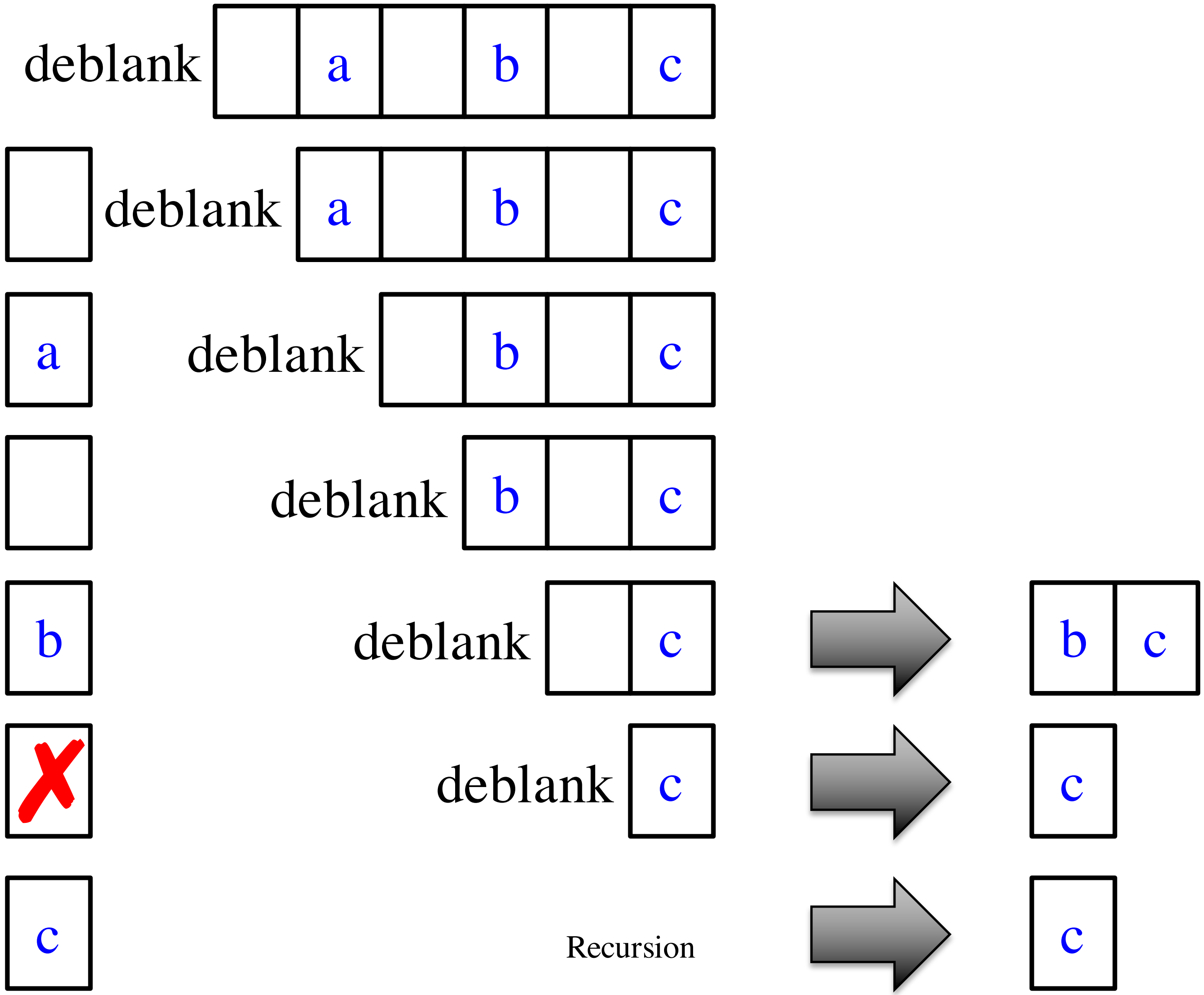
What the Recursion Does



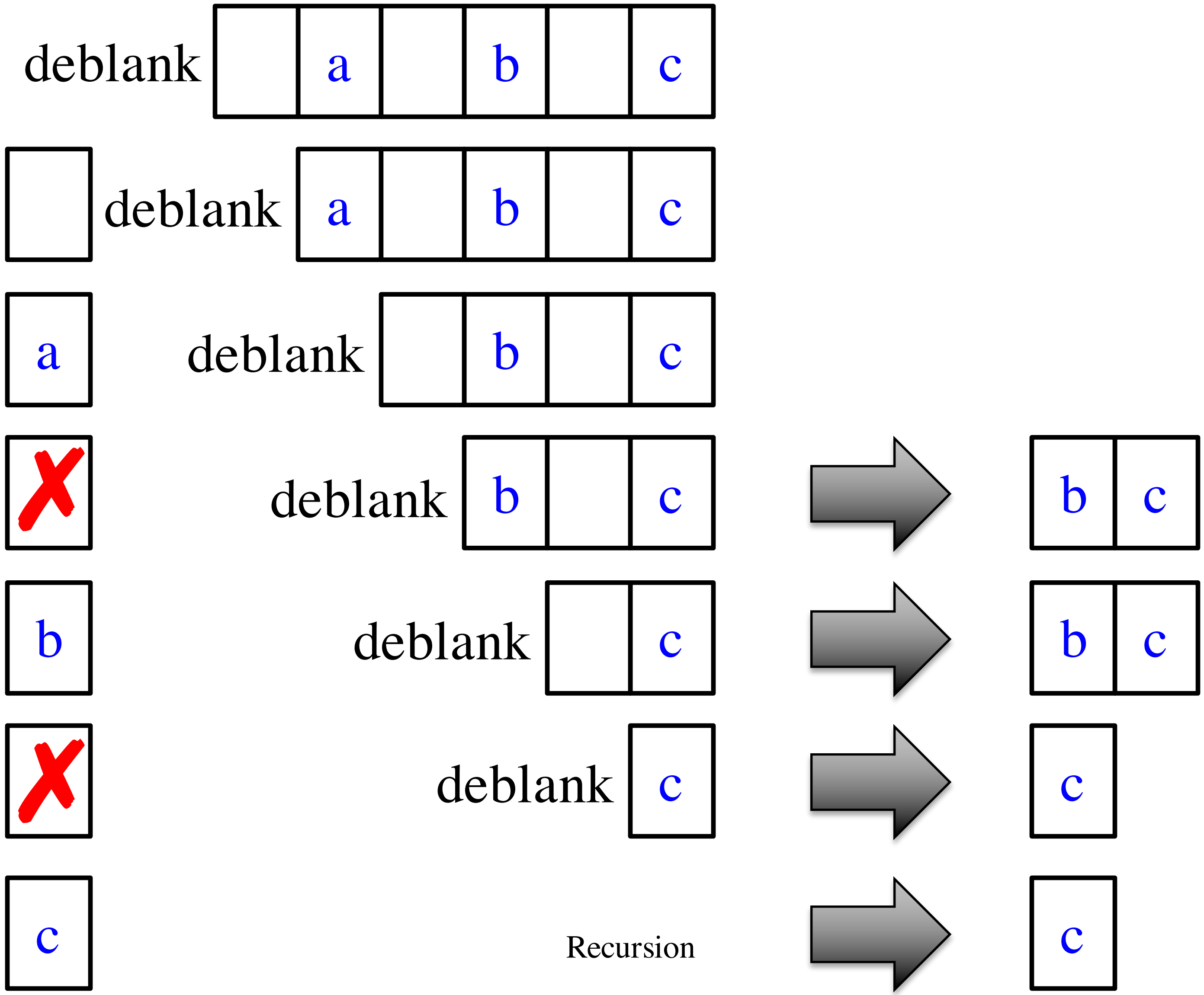
What the Recursion Does



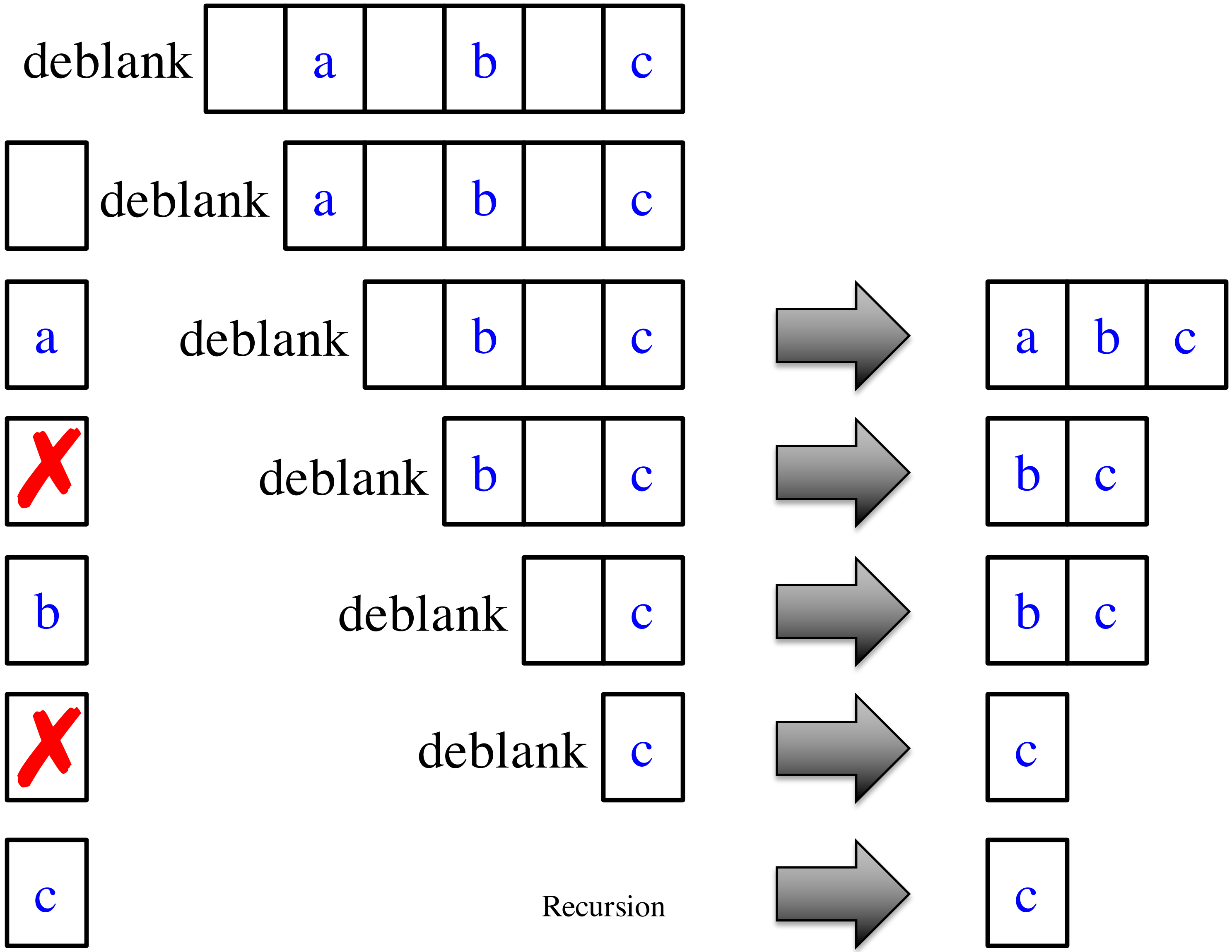
What the Recursion Does



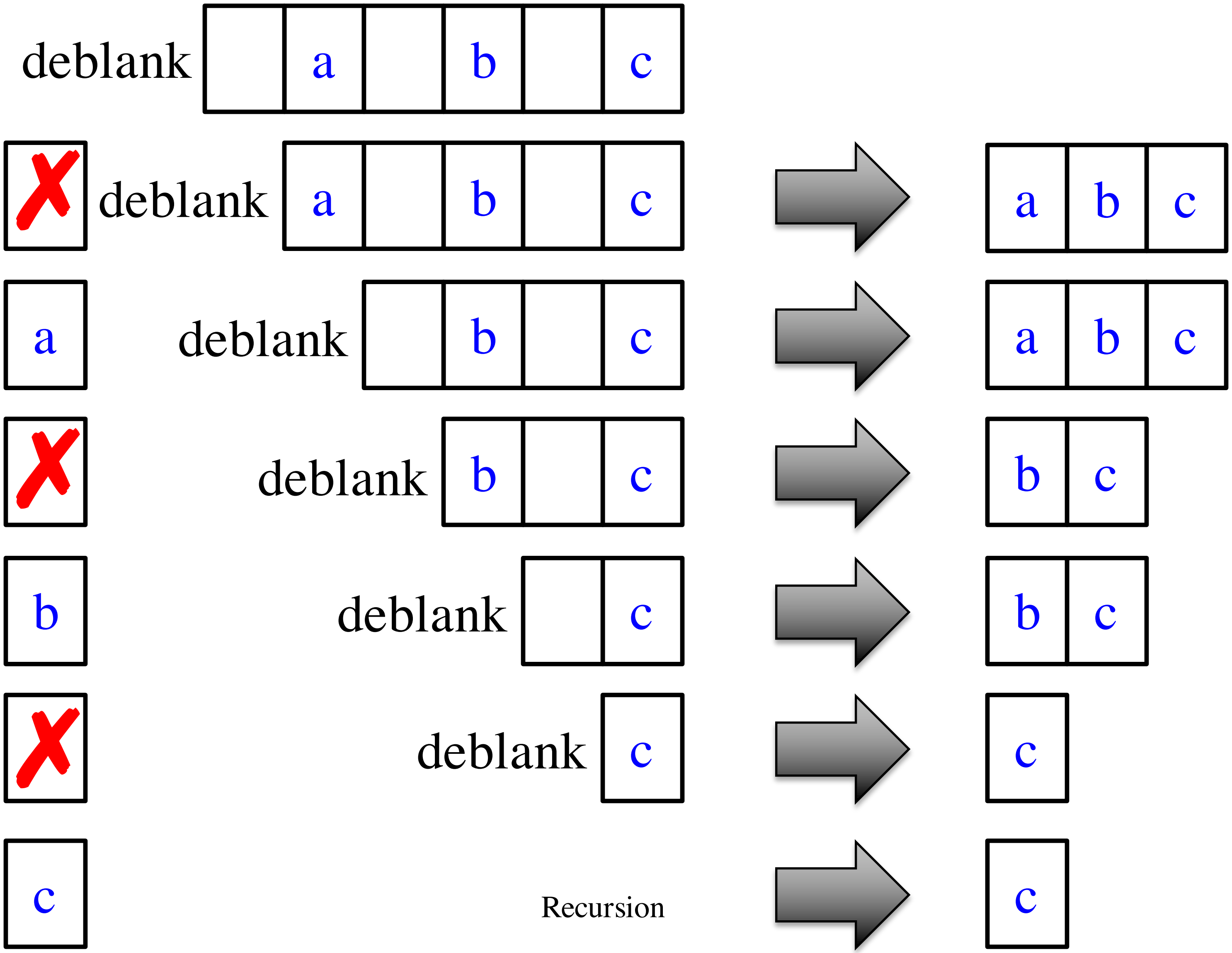
What the Recursion Does



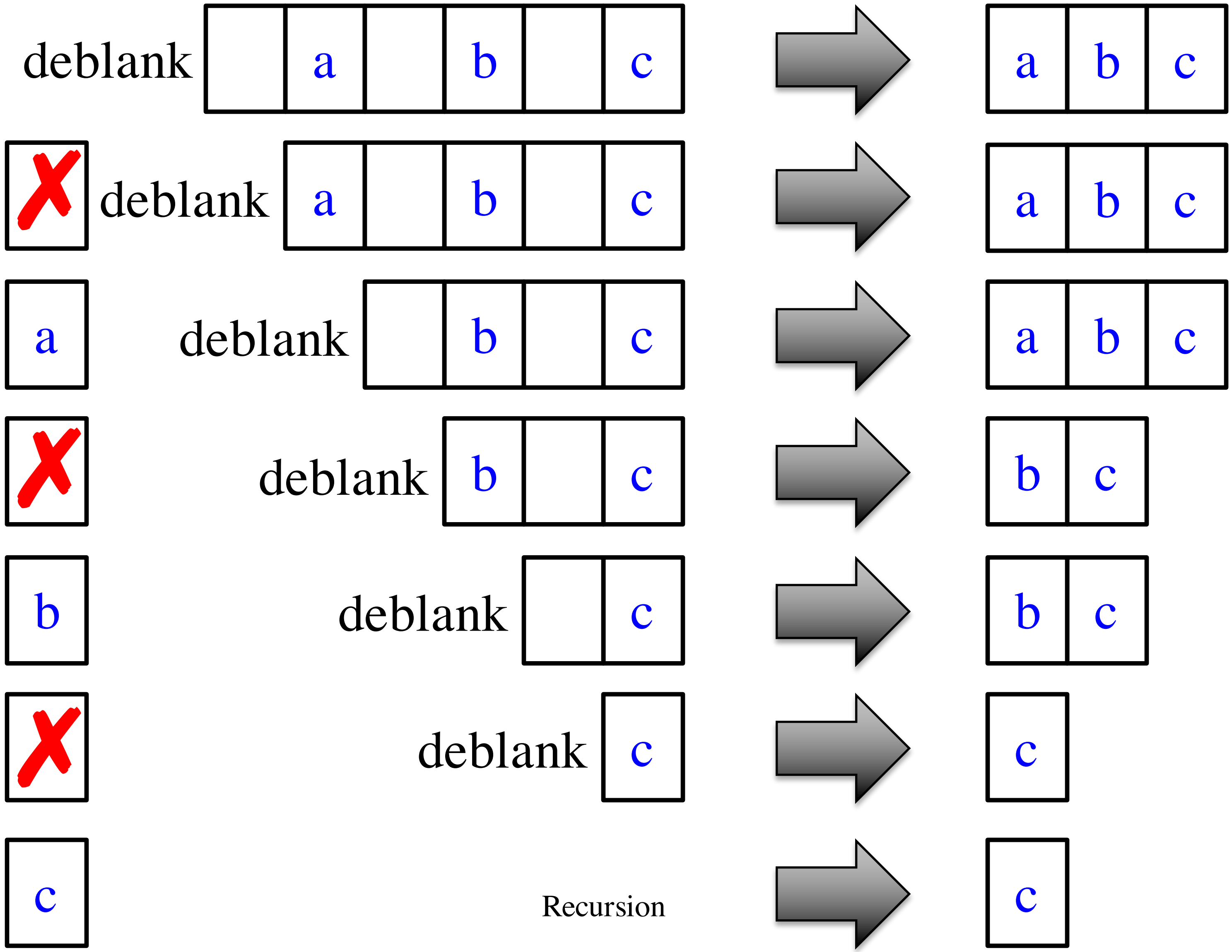
What the Recursion Does



What the Recursion Does



What the Recursion Does



Exercise: Remove Blanks from a String

```
def deblank(s):
    """Returns: s with blanks removed"""
    if s == '':
        return s

    # s is not empty
    if s[0] is a blank:
        return s[1:] with blanks removed

    # s not empty and s[0] not blank
    return (s[0] +
            s[1:] with blanks removed)
```

- Sometimes easier to break up the recursive case
  - Particularly om small part
  - Write recursive case as a sequence of if-statements
- Write code in *pseudocode*
  - Mixture of English and code
  - Similar to top-down design
- Stuff in **red** looks like the function specification!
  - But on a smaller string
  - Replace with deblank(s[1:])

Exercise: Remove Blanks from a String

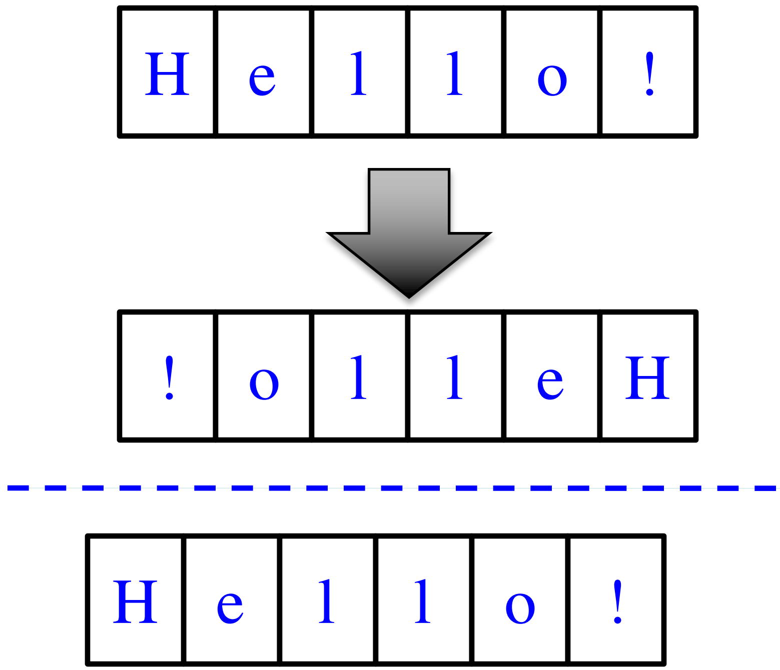
```
def deblank(s):
    """Returns: s with blanks removed"""
    if s == '':
        return s

    # s is not empty
    if s[0] in string.whitespace:
        return deblank(s[1:])

    # s not empty and s[0] not blank
    return (s[0] +
            deblank(s[1:]))
```

- Check the four points:
  1. Precise specification?
  2. **Base case**: correct?
  3. Progress towards termination?
  4. **Recursive case**: correct?

Module string has special constants to simplify detection of whitespace and other characters.

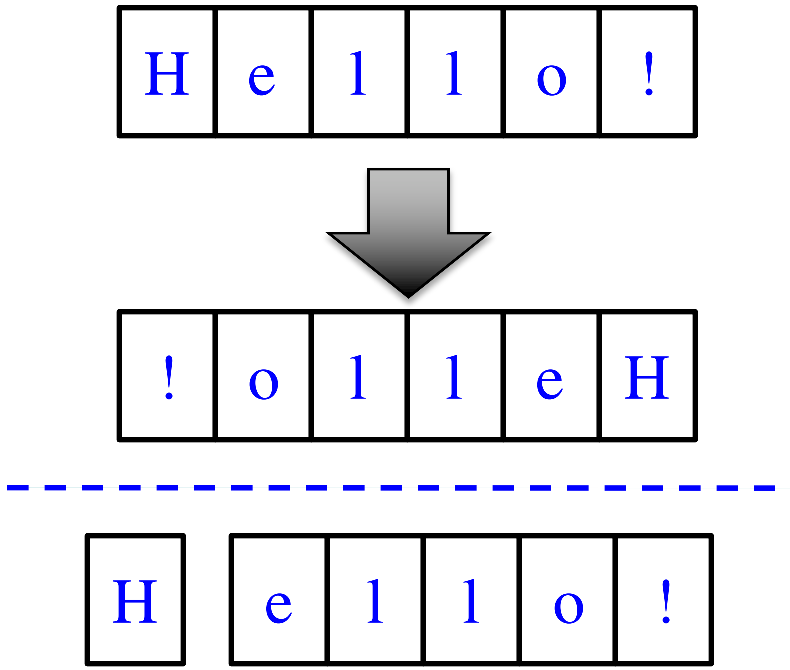
Example: Reversing a String

- **Precise Specification**:
  - Returns: reverse of s
- Solving with recursion
  - Suppose we can reverse a smaller string (e.g. less one character)
  - Can we use that solution to reverse whole string?
- Often easy to understand first without Python
  - Then sit down and code



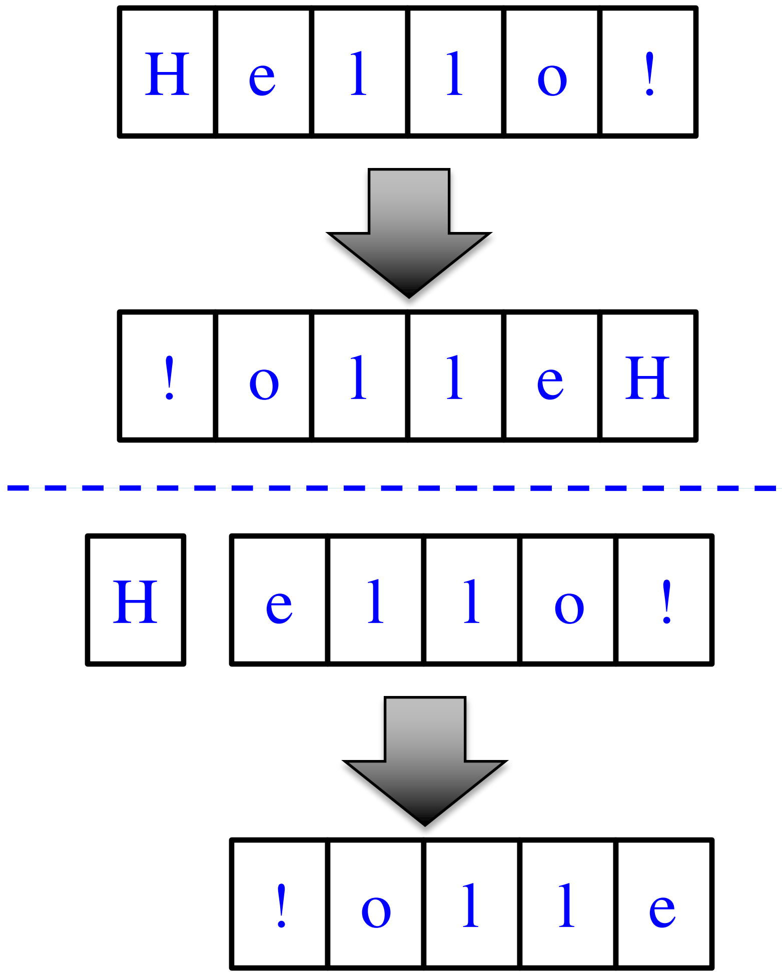
Example: Reversing a String

- **Precise Specification**:
  - Returns: reverse of s
- Solving with recursion
  - Suppose we can reverse a smaller string (e.g. less one character)
  - Can we use that solution to reverse whole string?
- Often easy to understand first without Python
  - Then sit down and code



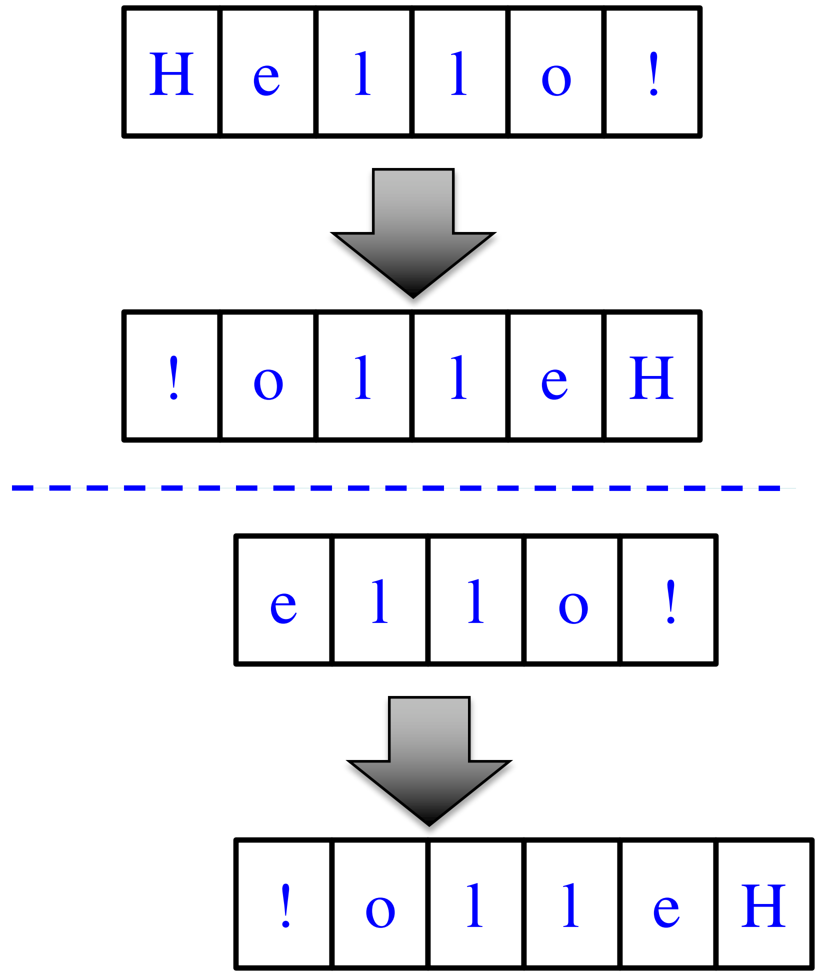
Example: Reversing a String

- **Precise Specification**:
  - Returns: reverse of s
- Solving with recursion
  - Suppose we can reverse a smaller string (e.g. less one character)
  - Can we use that solution to reverse whole string?
- Often easy to understand first without Python
  - Then sit down and code



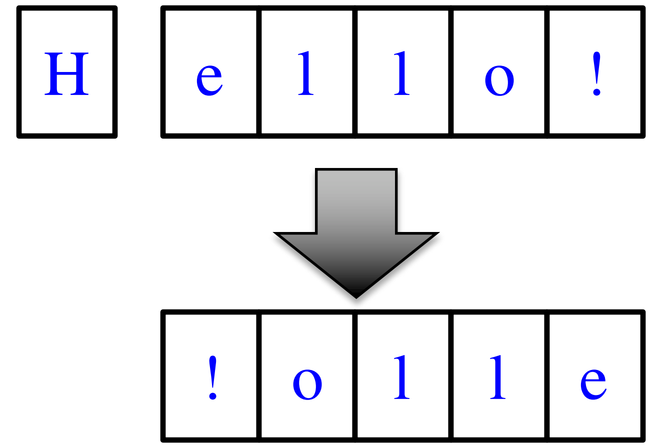
Example: Reversing a String

- **Precise Specification**:
  - Returns: reverse of s
- Solving with recursion
  - Suppose we can reverse a smaller string (e.g. less one character)
  - Can we use that solution to reverse whole string?
- Often easy to understand first without Python
  - Then sit down and code



Example: Reversing a String

```
def reverse(s):
    """Returns: reverse of s

    Precondition: s a string"""
    # s is empty
    if s == '':
        return s

    # s has at least one char
    # (reverse of s[1:])+s[0]
    return reverse(s[1:])+s[0]
```



✔ 1. Precise specification?

✔ 2. Base case: correct?

✔ 3. Recursive case: progress to termination?

✔ 4. Recursive case: correct?

Next Time: Recursion vs. For-Loops