CS 1110:
Introduction to Computing Using Python

Lecture 15

Recursion

[Andersen, Gries, Lee, Marschner, Van Loan, White]

Announcements: Prelim 1

- Graded and released
- **Mean**: 81 out of 104 (78%)
- Can pick up your exam in homework handback room
  - Need Cornell ID
  - Suggest printing your netid on paper
- Do not discuss exam with people taking makeups.
- **Regrade requests**: we will send email to you

Announcements: Assignment 3

- Released.
- **Due**: Thursday, March 30th, 11:59pm
- Recommendation: follow milestone deadlines.
- You MUST acknowledge help from others
  - We run software analyzers to detect similar programs
  - Have had some academic integrity violations so far
- Not a recursion assignment!

Announcement: Lab 8

- Out.
- Not a recursion lab!

Recursion

- **Recursive Definition**:

  A definition that is defined in terms of itself

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?

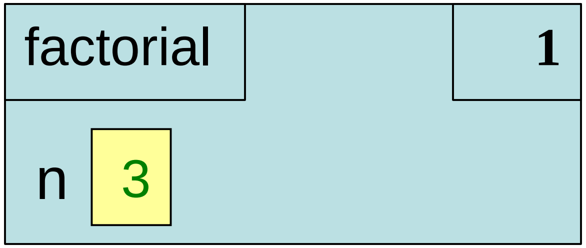
Recursion

- **Recursive Definition**:

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

  A function that calls itself (directly or indirectly)

Recursive Call Frames

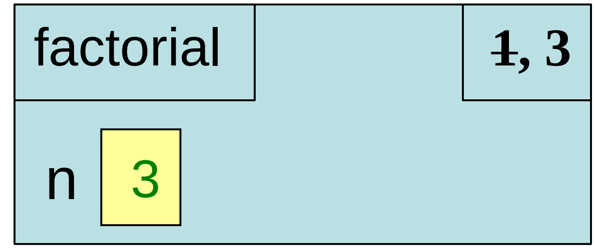



```
def factorial(n):
    """Returns: factorial of n.
    Pre: n ≥ 0 an int"""
    if n == 0:
        return 1

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

Call: factorial(3)

Recursive Call Frames

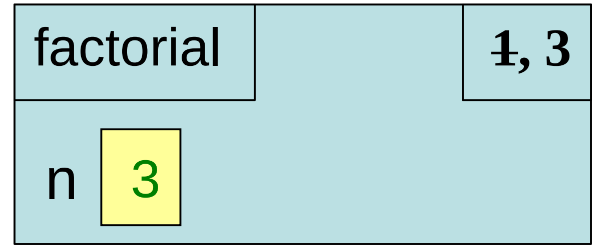
```
def factorial(n):
    """Returns: factorial of n.
    Pre: n ≥ 0 an int"""
1   if n == 0:
2       return 1

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



Call: factorial(3)

Recursion

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

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



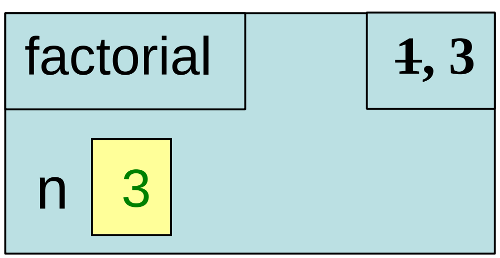
Now what?
Each call is a new frame.

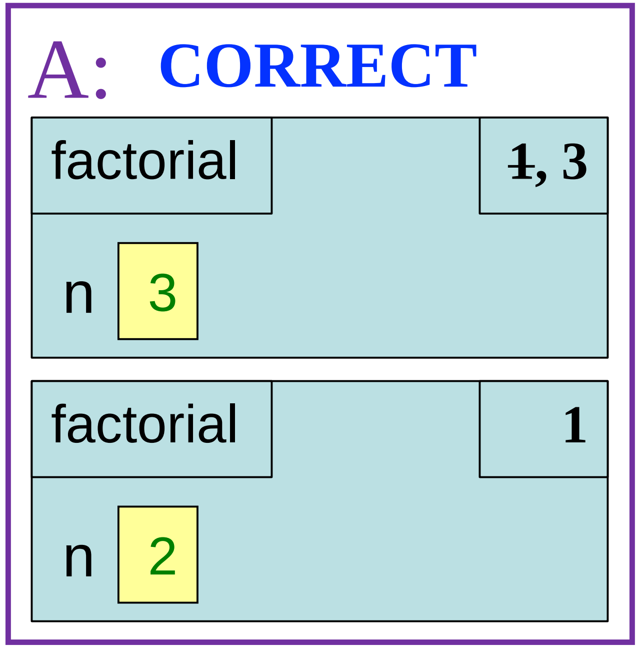
Call: factorial(3)

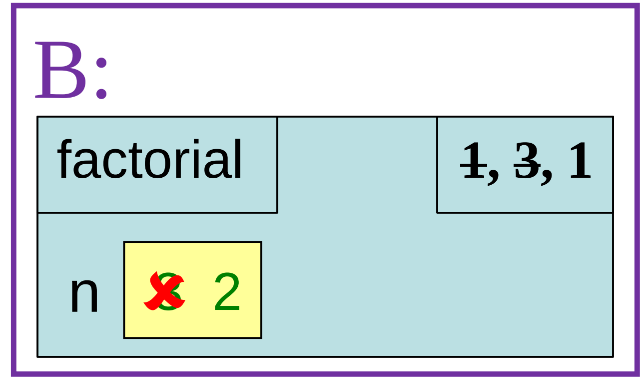
What happens next?

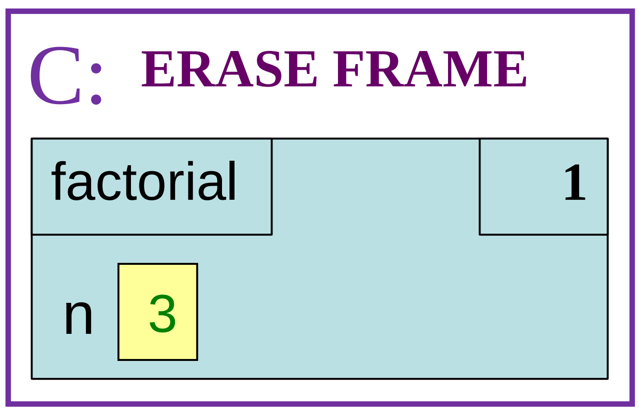
```
  def factorial(n):
      """Returns: factorial of n.
      Pre: n ≥ 0 an int"""
1     if n == 0:
2         return 1

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

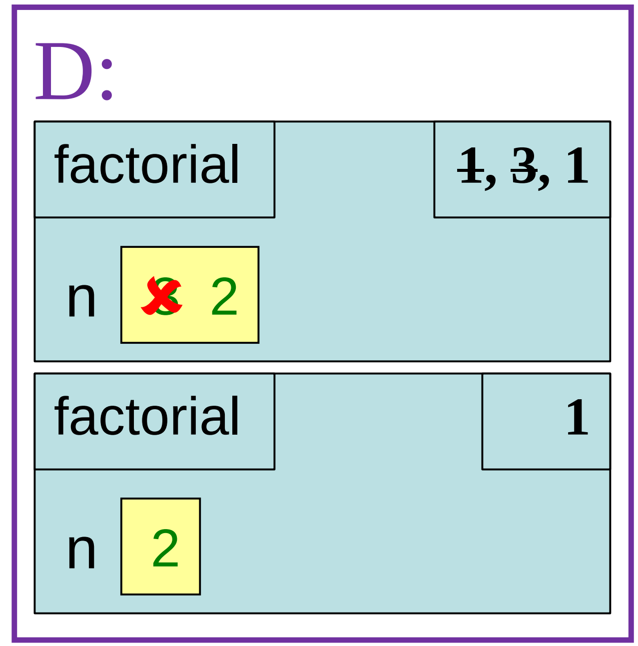
Call: factorial(3)





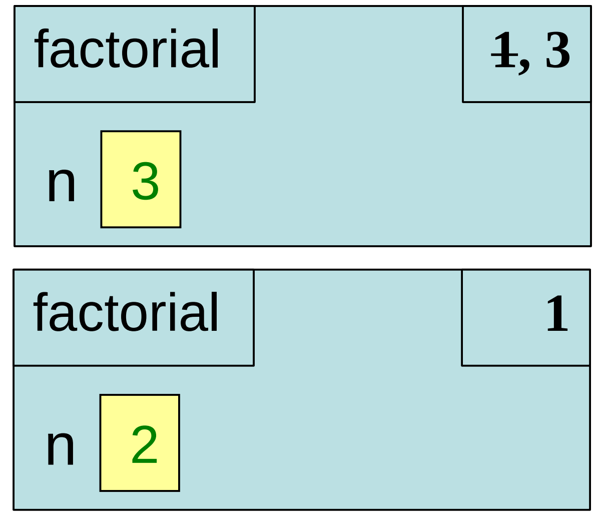






Recursive Call Frames

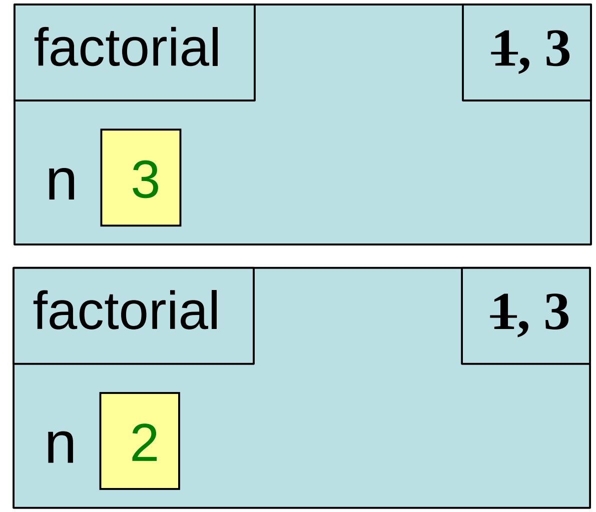
```
def factorial(n):
    """Returns: factorial of n.
    Pre: n ≥ 0 an int"""
    if n == 0:
        return 1

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



Call: factorial(3)

Recursive Call Frames

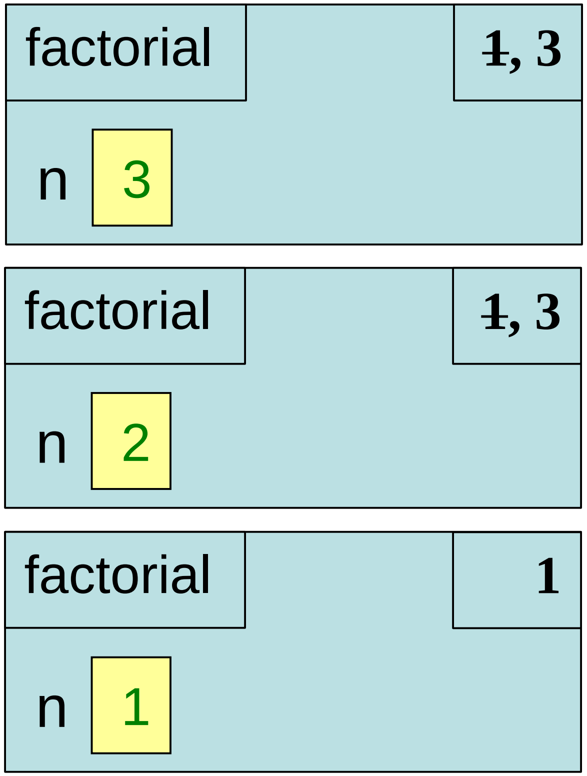
```
def factorial(n):
    """Returns: factorial of n.
    Pre: n ≥ 0 an int"""
    if n == 0:
        return 1

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



Call: factorial(3)

Recursive Call Frames

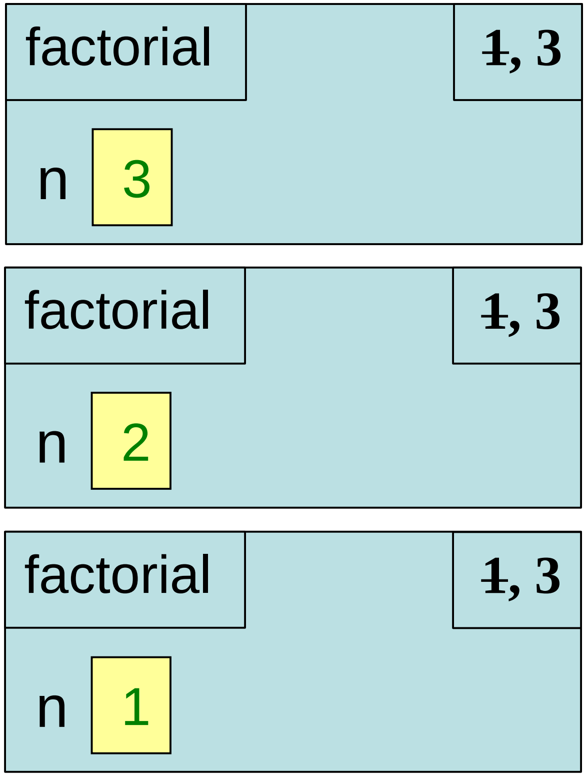
```
  def factorial(n):
      """Returns: factorial of n.
      Pre: n ≥ 0 an int"""
1     if n == 0:
2         return 1

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



Call: factorial(3)

Recursive Call Frames

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

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

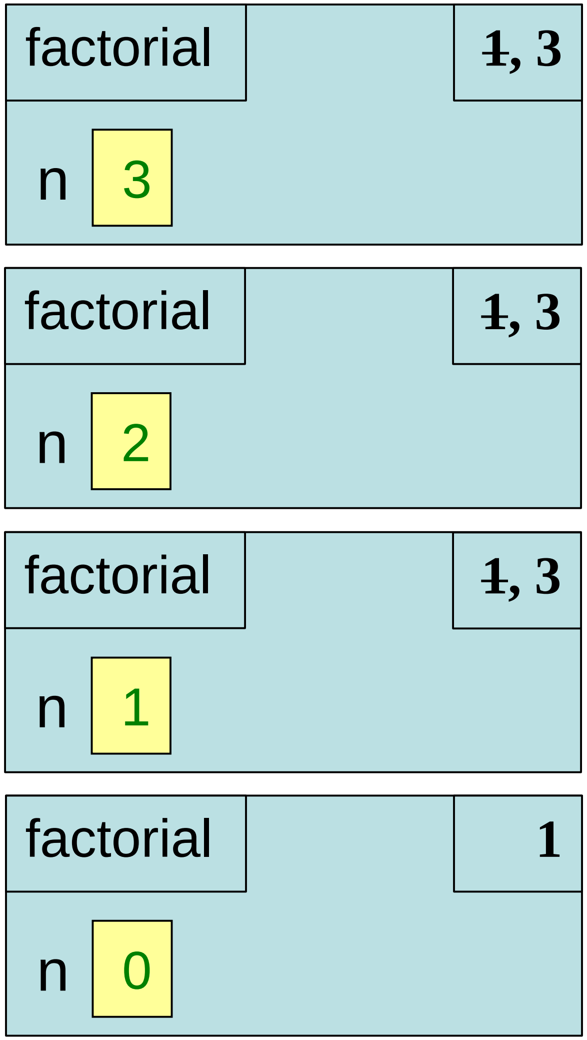


Call: factorial(3)

Recursive Call Frames

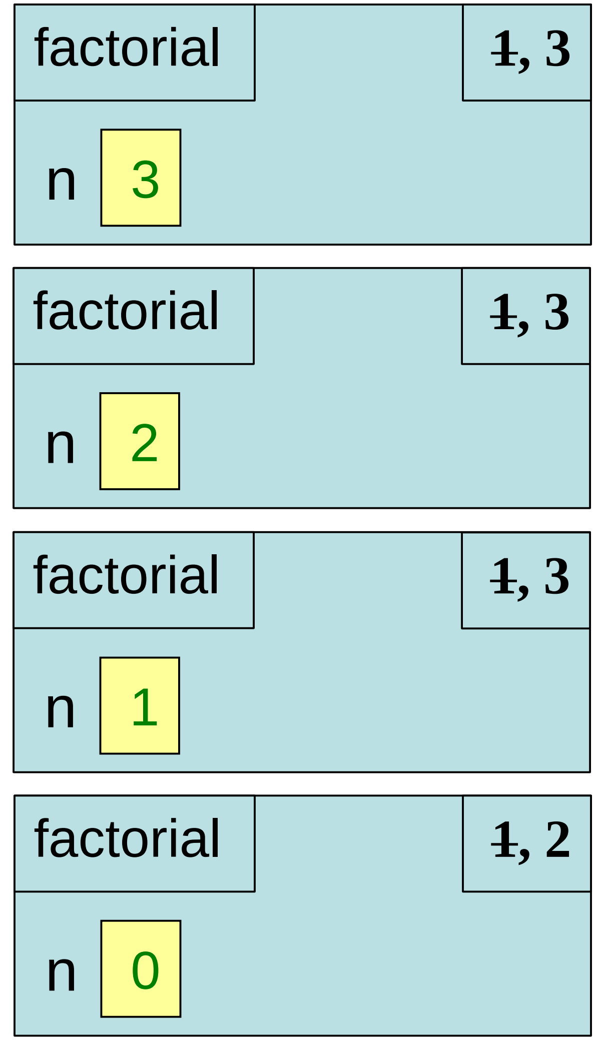
```
def factorial(n):
    """Returns: factorial of n.
    Pre: n ≥ 0 an int"""
1   if n == 0:
2       return 1

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

Call: factorial(3)



Recursive Call Frames

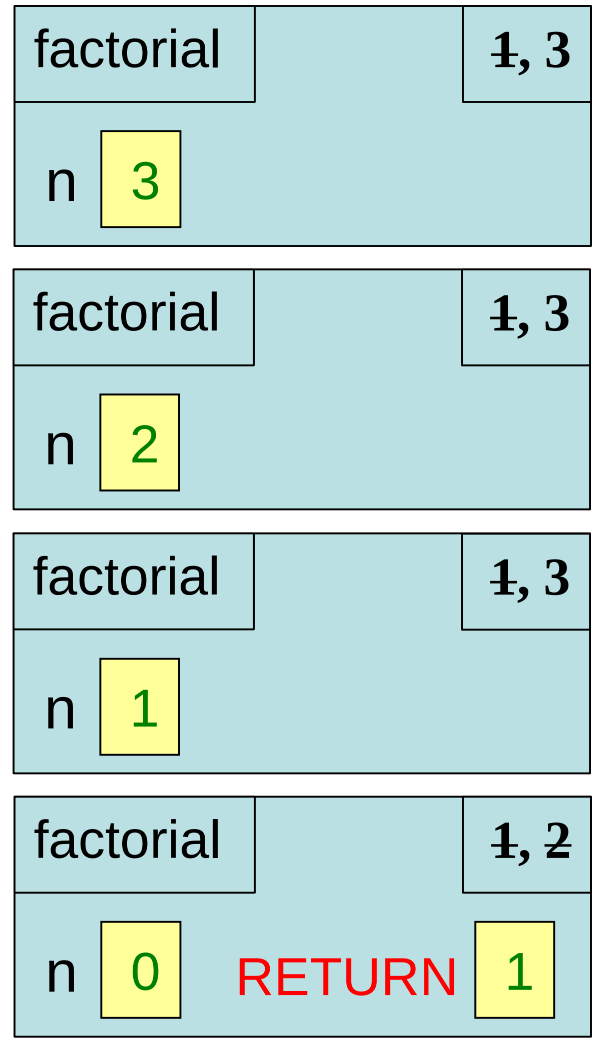
```
    def factorial(n):
        """Returns: factorial of n.
        Pre: n ≥ 0 an int"""
1       if n == 0:
2           return 1

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



Call: factorial(3)

Recursive Call Frames

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

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

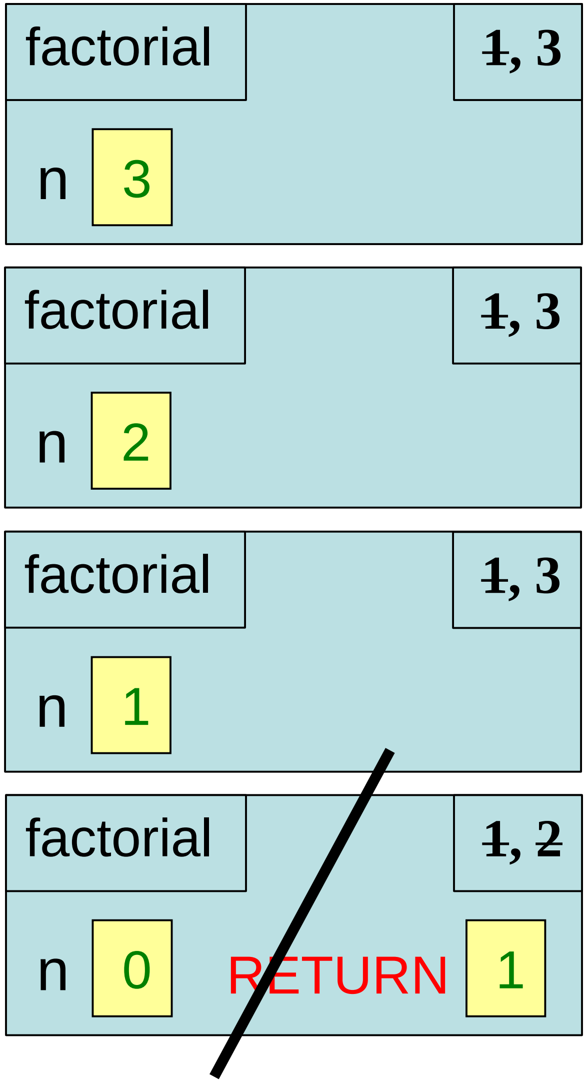


Call: factorial(3)

Recursive Call Frames

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

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

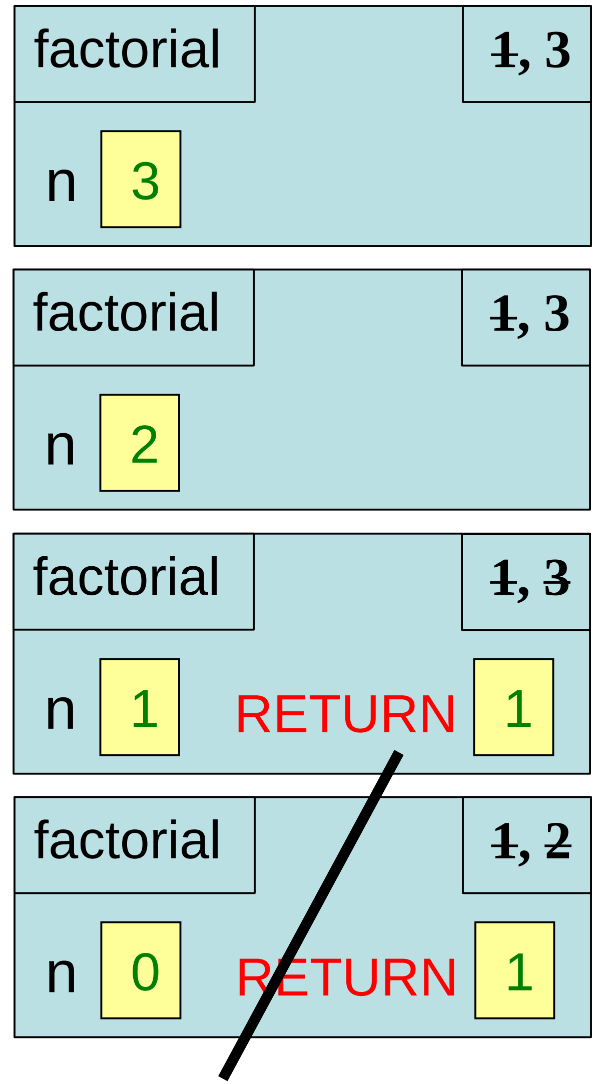
Call: factorial(3)



Recursive Call Frames

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

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

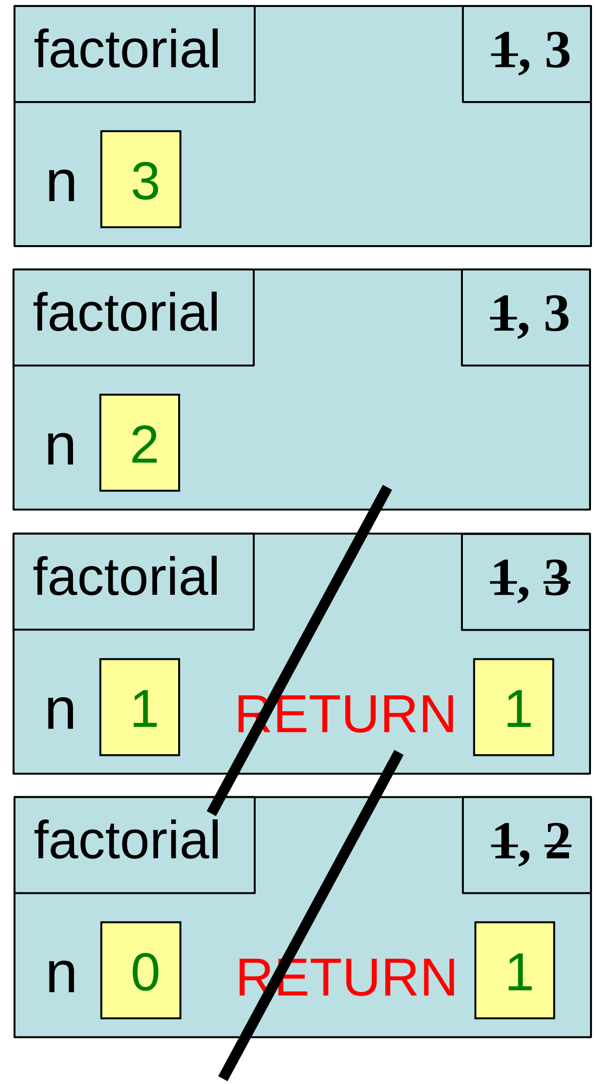
Call: factorial(3)



Recursive Call Frames

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

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

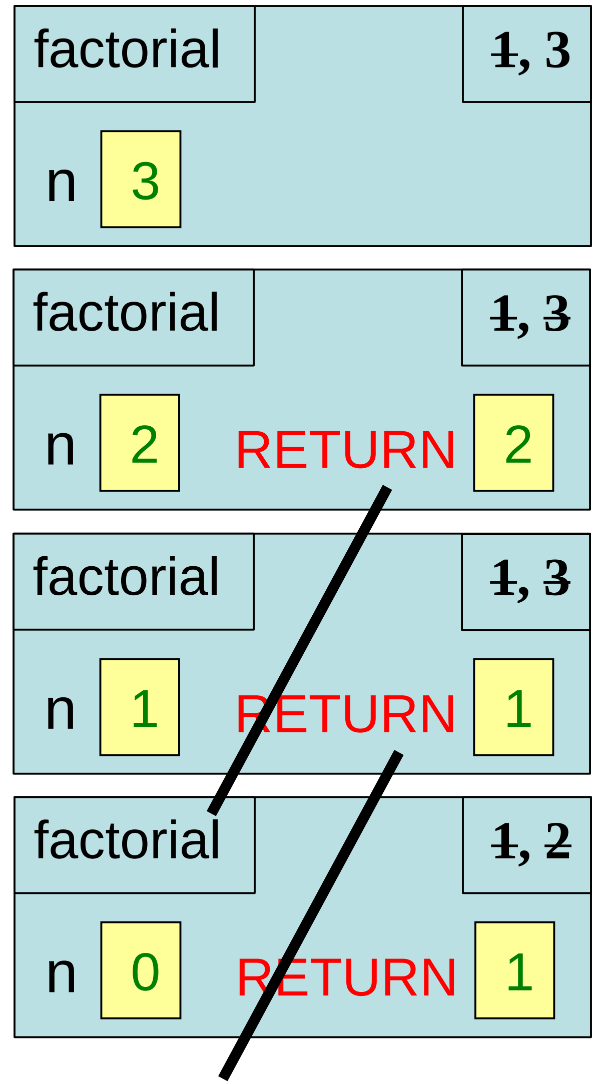
Call: factorial(3)



Recursive Call Frames

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

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

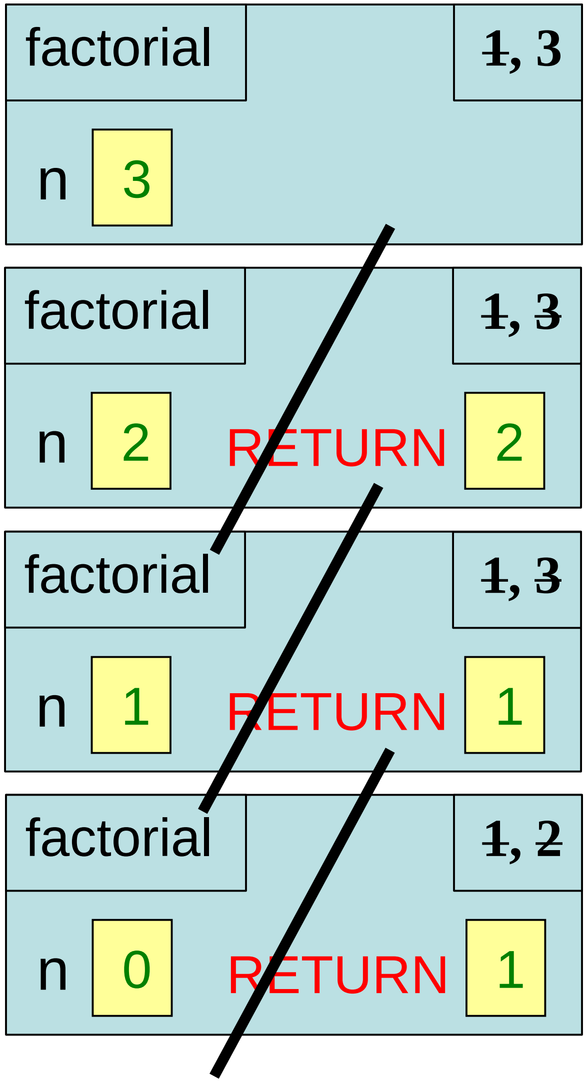
Call: factorial(3)



Recursive Call Frames

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

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

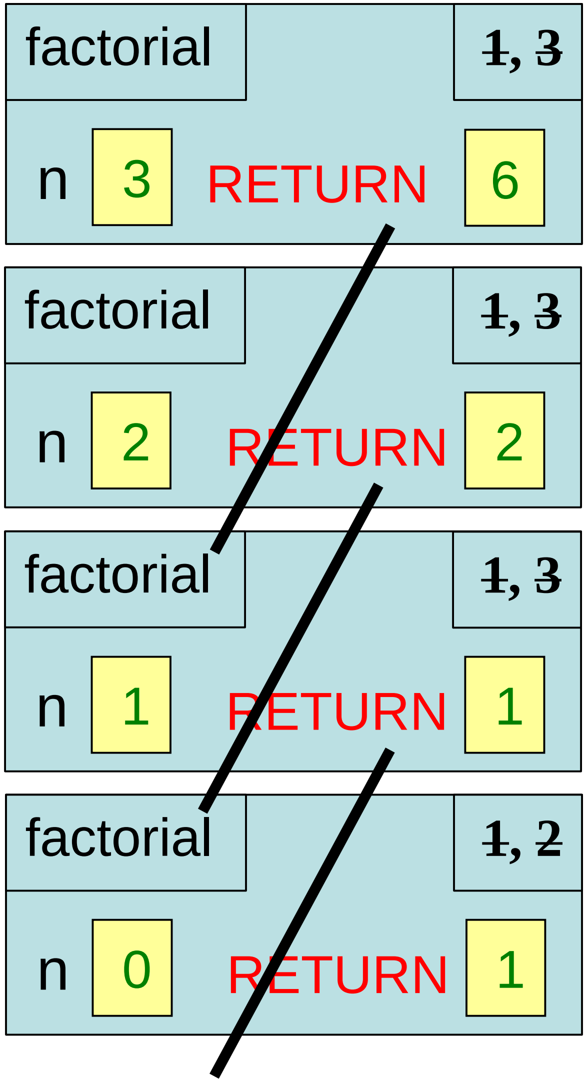
Call: factorial(3)



Recursive Call Frames

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

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

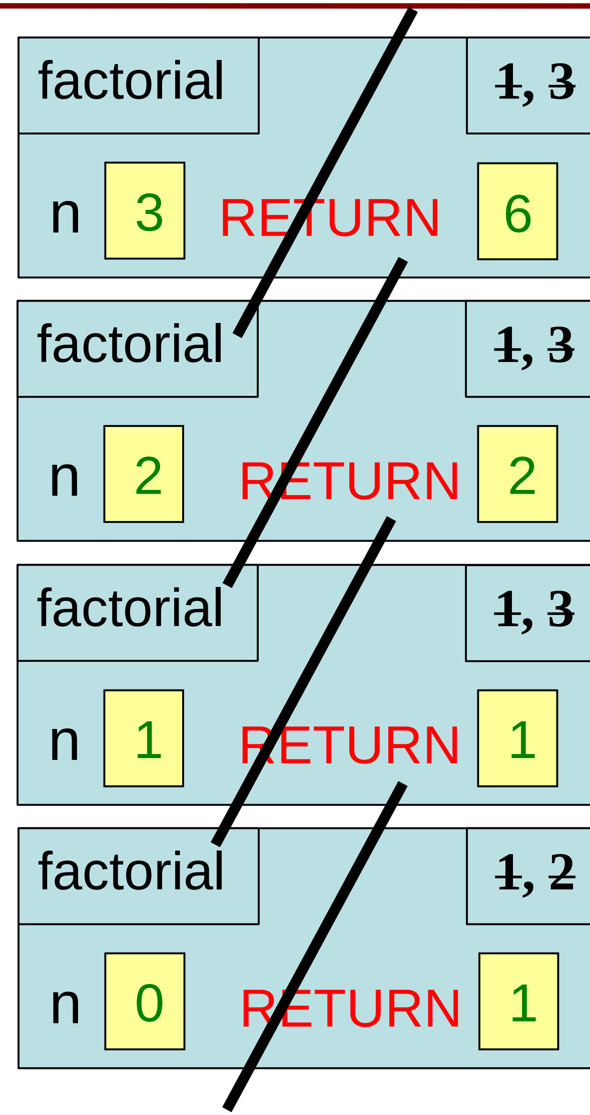
Call: factorial(3)



Recursive Call Frames

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

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

Call: factorial(3)



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:
  - $a_n = a_{n\text{-}1} + a_{n\text{-}2}$ **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. a_n
    Precondition: n ≥ 0 an int"""
    if n <= 1:
        return 1

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

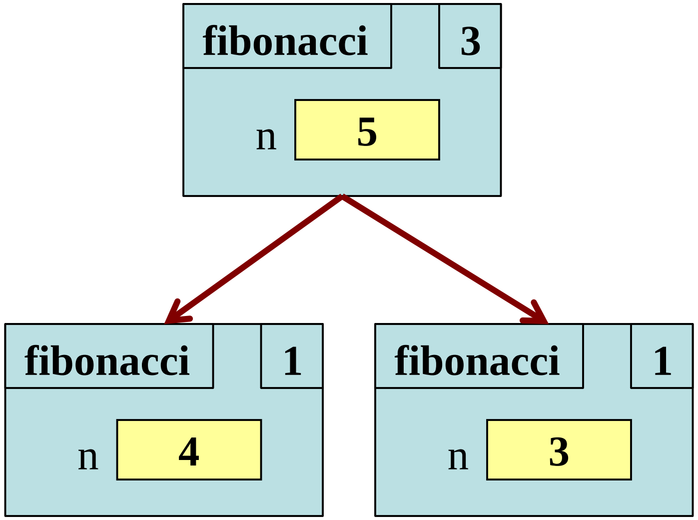
Base case(s)

Recursive case

Handles both base cases in one 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))
```



Recursion vs Iteration

- **Recursion** is ***provably equivalent*** to **iteration**
  - Iteration includes **for-loop** and **while-loop** (later)
  - Anything can do in one, can do in the other
- But some things are easier with recursion
  - And some things are easier with iteration
- Will **not** teach you when to choose recursion
- We just want you to ***understand the technique***

Recursion is best for Divide and Conquer

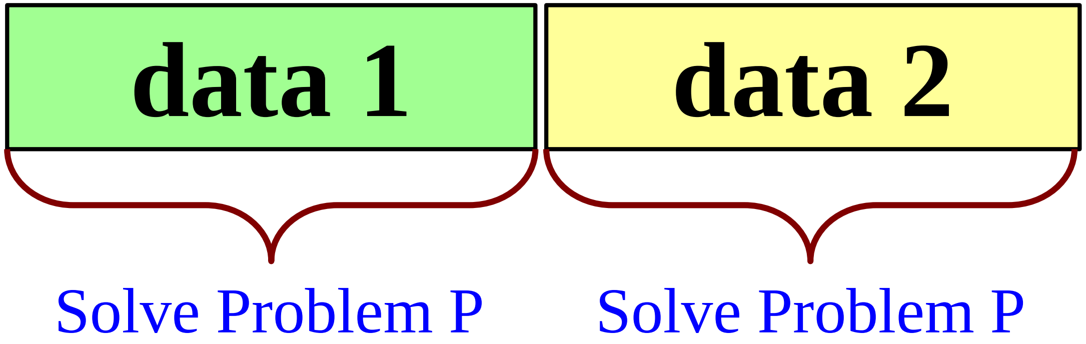
Goal: Solve problem P on a piece of data

data

Recursion is best for Divide and Conquer

Goal: Solve problem P on a piece of data

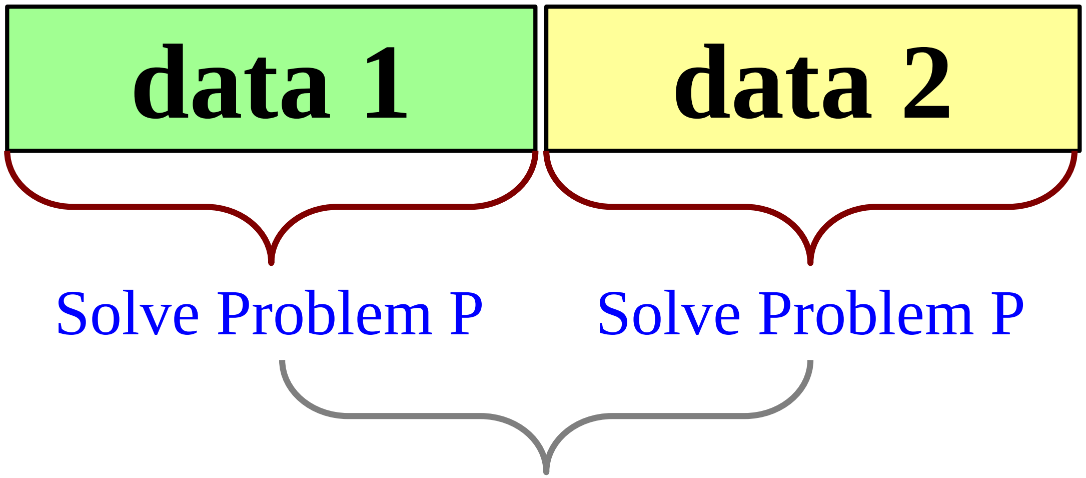
Idea: Split data into two parts and solve problem



Recursion is best for Divide and Conquer

Goal: Solve problem P on a piece of data

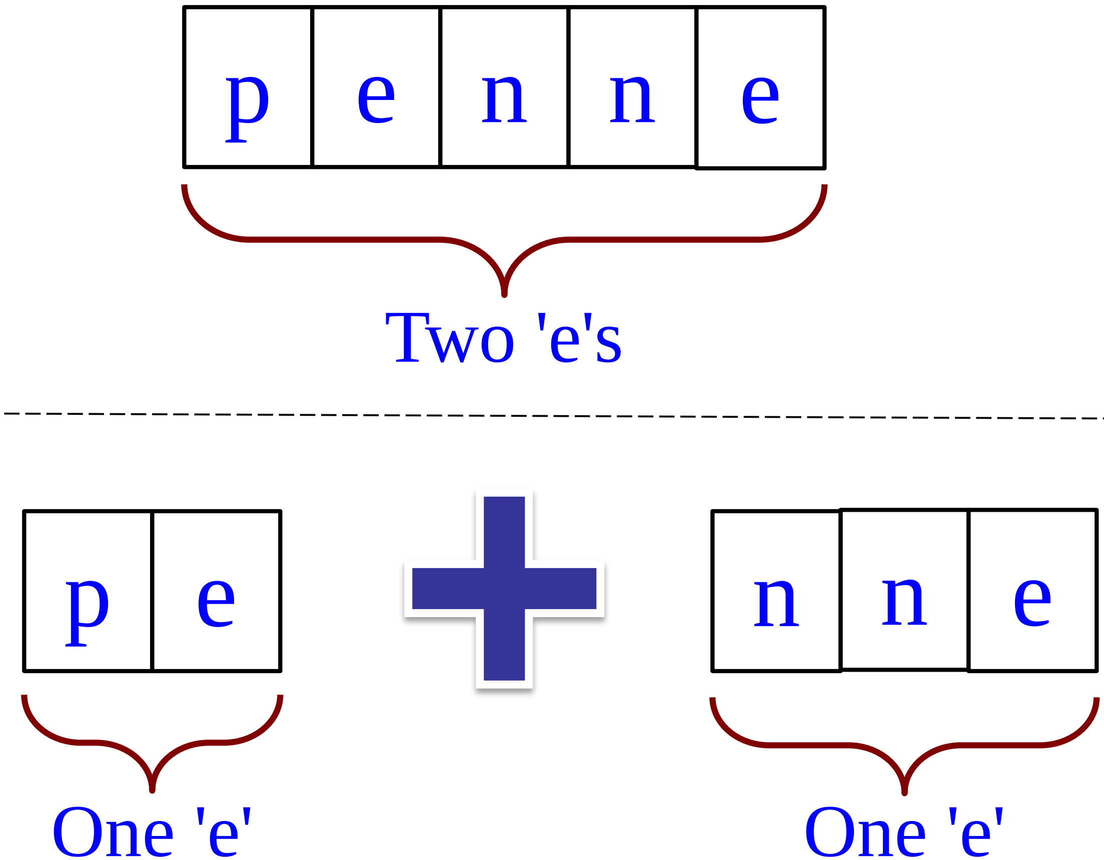
Idea: Split data into two parts and solve problem



Combine Answer!

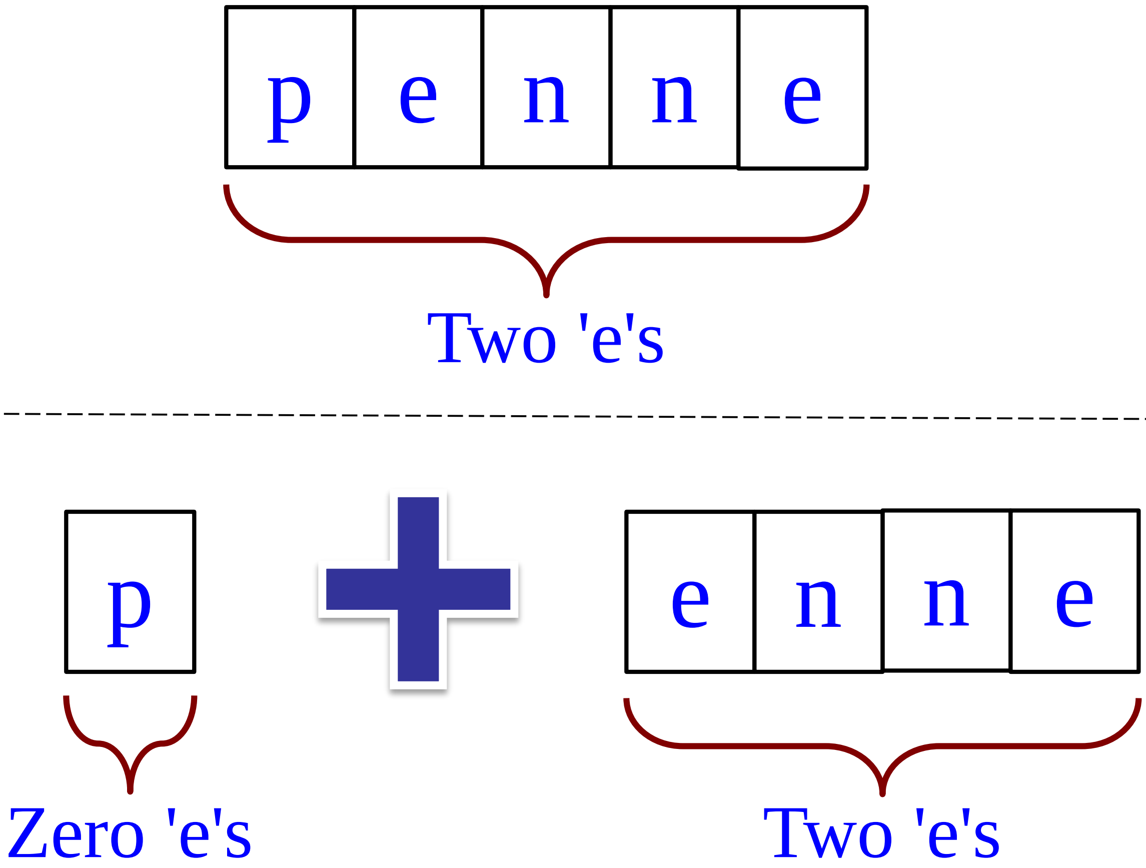
Divide and Conquer Example

Count the number of 'e's in a string:



Divide and Conquer Example

Count the number of 'e's in a string:



Three Steps for Divide and Conquer

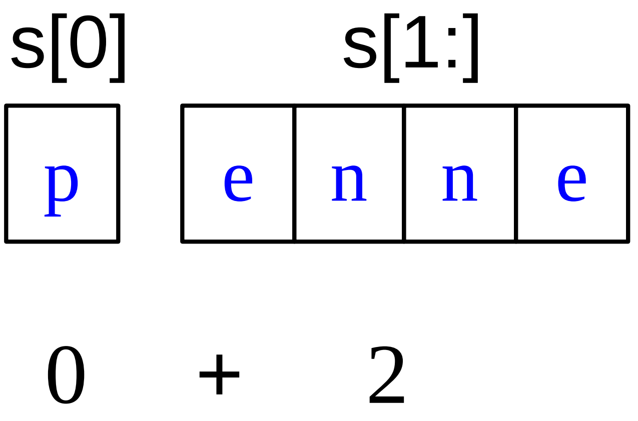
1. Decide what to do on "small" data
   - Some data cannot be broken up
   - Have to compute this answer directly
2. Decide how to break up your data
   - Both "halves" should be smaller than whole
   - Often no wrong way to do this (next lecture)
3. Decide how to combine your answers
   - Assume the smaller answers are correct
   - Combining them should give bigger answer

Divide and Conquer Example

```
def num_es(s):
    """Returns: # of 'e's in s"""
    # 1. Handle small data
    if s == '':
        return 0
    elif len(s) == 1:
        return 1 if s[0] == 'e' else 0

    # 2. Break into two parts
    left = num_es(s[0])
    right = num_es(s[1:])

    # 3. Combine the result
    return left+right
```

"Short-cut" for

```
if s[0] == 'e':
    return 1
else:
    return 0
```



Divide and Conquer Example

```
def num_es(s):
    """Returns: # of 'e's in s"""
    # 1. Handle small data
    if s == '':
        return 0
    elif len(s) == 1:
        return 1 if s[0] == 'e' else 0

    # 2. Break into two parts
    left = num_es(s[0])
    right = num_es(s[1:])

    # 3. Combine the result
    return left+right
```

“Short-cut” for

```
if s[0] == 'e':
    return 1
else:
    return 0
```

s[0] s[1:]

p | e n n e

0 + 2

Divide and Conquer Example

```
def num_es(s):
    """Returns: # of 'e's in s"""
    # 1. Handle small data
    if s == '':
        return 0
    elif len(s) == 1:
        return 1 if s[0] == 'e' else 0

    # 2. Break into two parts
    left = num_es(s[0])
    right = num_es(s[1:])

    # 3. Combine the result
    return left+right
```

"Short-cut" for

```
if s[0] == 'e':
    return 1
else:
    return 0
```

s[0]	s[1:]			
p	e	n	n	e

0 + 2

Divide and Conquer Example

```
def num_es(s):
    """Returns: # of 'e's in s"""
    # 1. Handle small data
    if s == '':
        return 0
    elif len(s) == 1:
        return 1 if s[0] == 'e' else 0

    # 2. Break into two parts
    left = num_es(s[0])
    right = num_es(s[1:])

    # 3. Combine the result
    return left+right
```

"Short-cut" for

```
if s[0] == 'e':
    return 1
else:
    return 0
```

s[0] s[1:]

p		e	n	n	e

0 + 2

Divide and Conquer Example

```
def num_es(s):
    """Returns: # of 'e's in s"""
    # 1. Handle small data
    if s == '':
        return 0
    elif len(s) == 1:
        return 1 if s[0] == 'e' else 0

    # 2. Break into two parts
    left = num_es(s[0])
    right = num_es(s[1:])

    # 3. Combine the result
    return left+right
```

Base Case (step 1)

Recursive Case (steps 2 and 3)

Exercise: Remove Blanks from a String

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

1. Decide what to do on “small” data
   - If it is the **empty string**, nothing to do

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

   - If it is a **single character**, delete it if a blank

```
if s == ' ':    # There is a space here
    return ''  # Empty string
else:
    return s
```

Exercise: Remove Blanks from a String

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

2. Decide how to break it up

```
left = deblank(s[0])      # A string with no blanks
right = deblank(s[1:])   # A string with no blanks
```

3. Decide how to combine the answer

```
return left+right         # String concatenation
```

Putting it All Together

```
def deblank(s):
    """Returns: s w/o blanks"""
    if s == '':
        return s
    elif len(s) == 1:
        return '' if s[0] == ' ' else s

    left = deblank(s[0])
    right = deblank(s[1:])

    return left+right
```

Handle small data

Break up the data

Combine answers

Putting it All Together

```
def deblank(s):
    """Returns: s w/o blanks"""
    if s == '':
        return s
    elif len(s) == 1:
        return '' if s[0] == ' ' else s

    left = deblank(s[0])
    right = deblank(s[1:])

    return left+right
```

Base Case

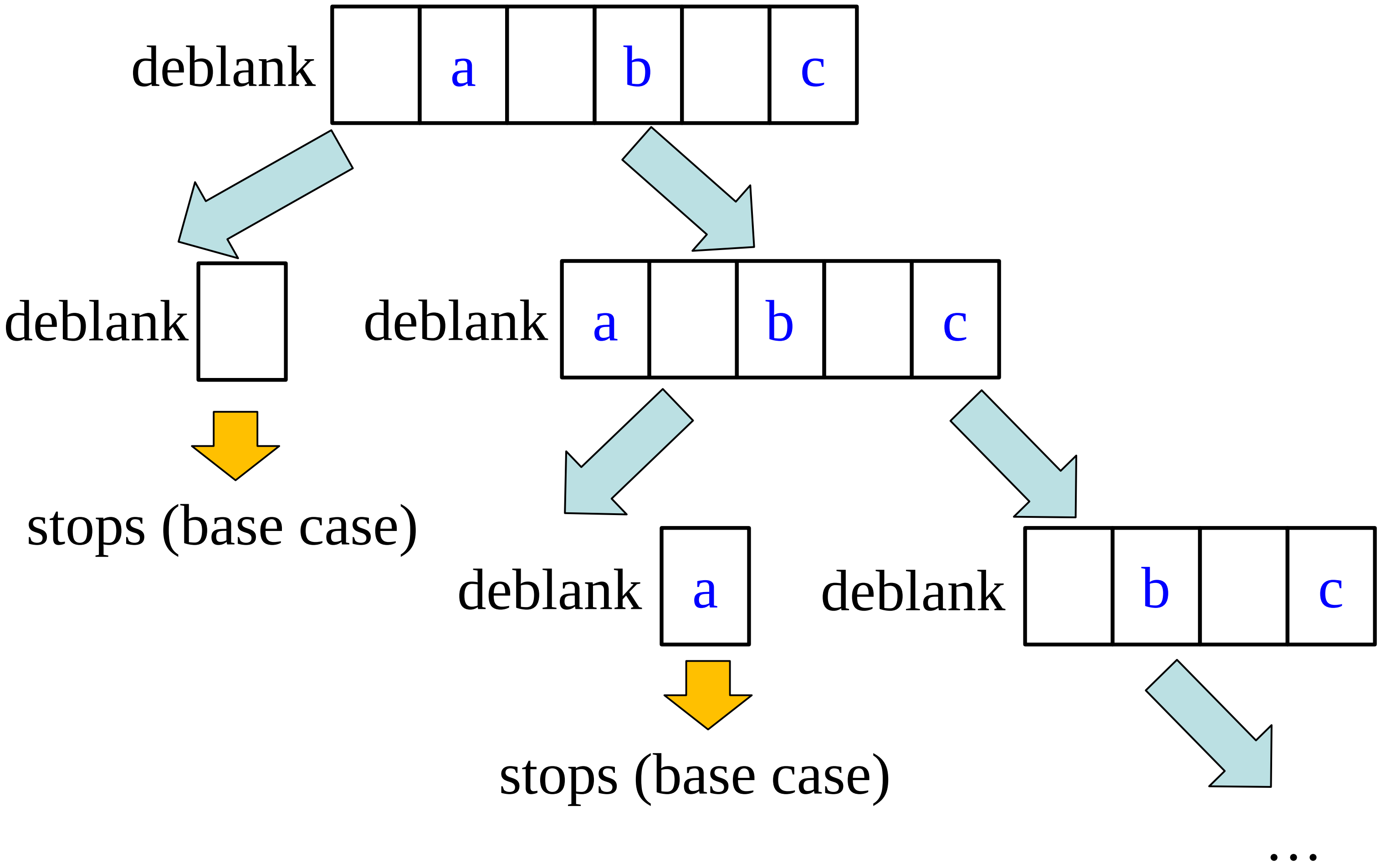
Recursive Case

Following the Recursion

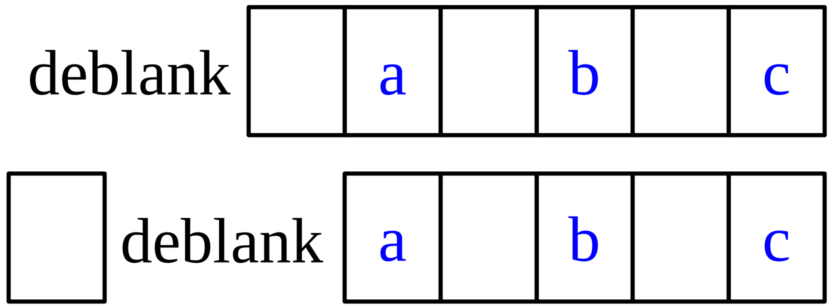
deblank

	a		b		c

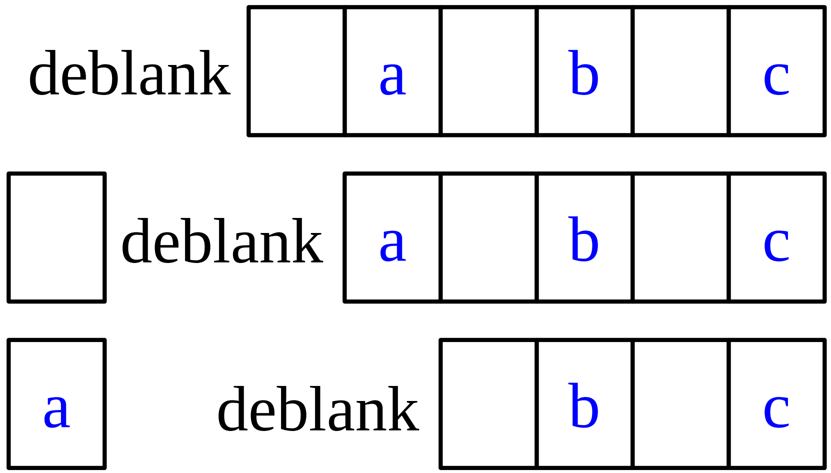
Following the Recursion



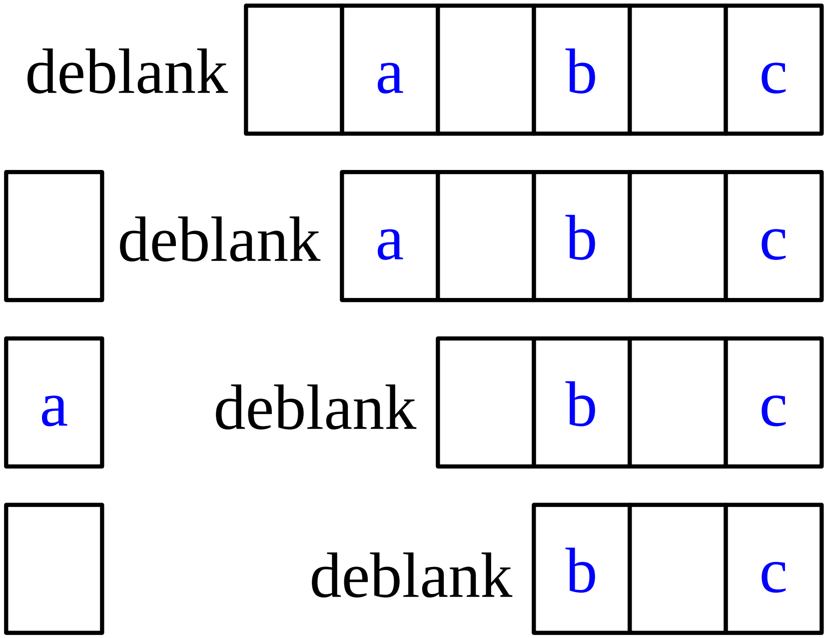
Following the Recursion



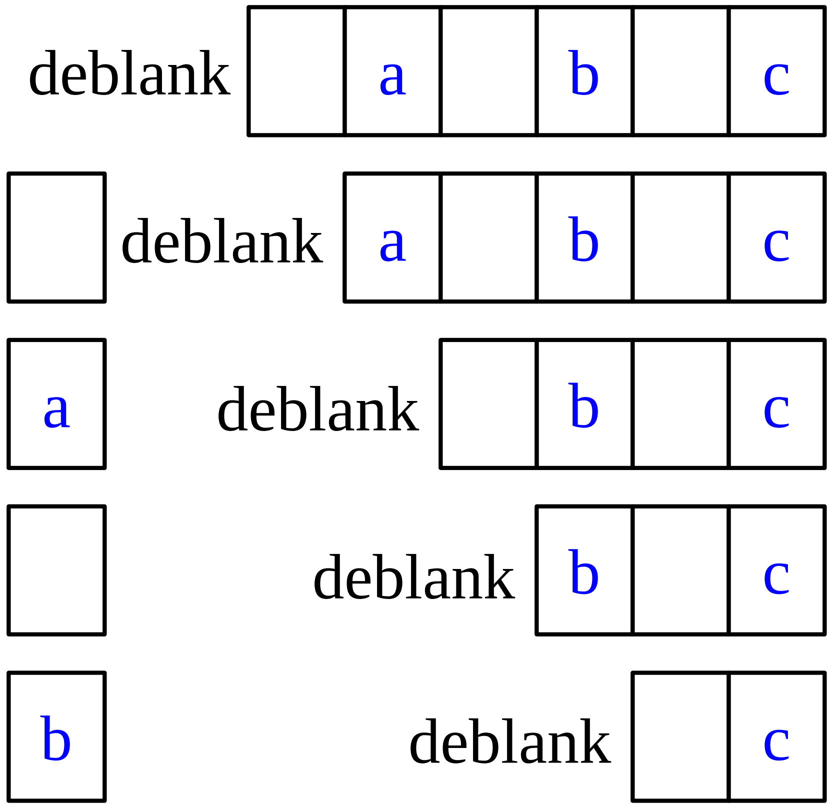
Following the Recursion



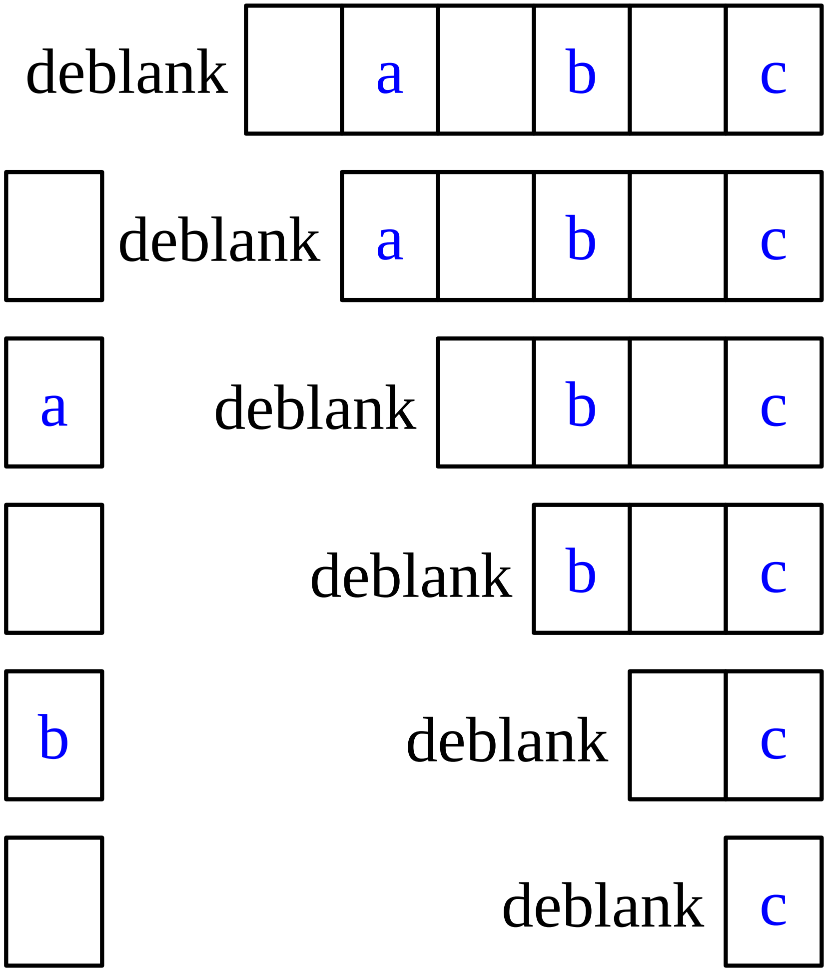
Following the Recursion



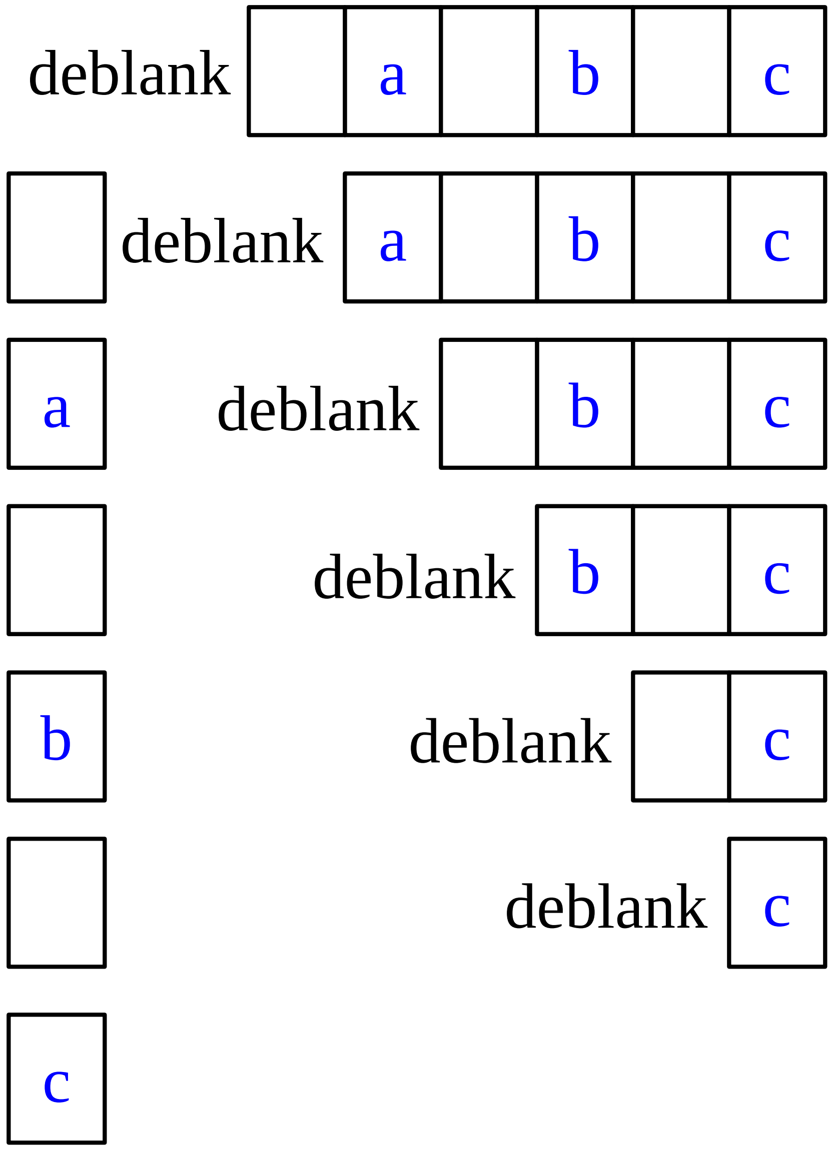
Following the Recursion



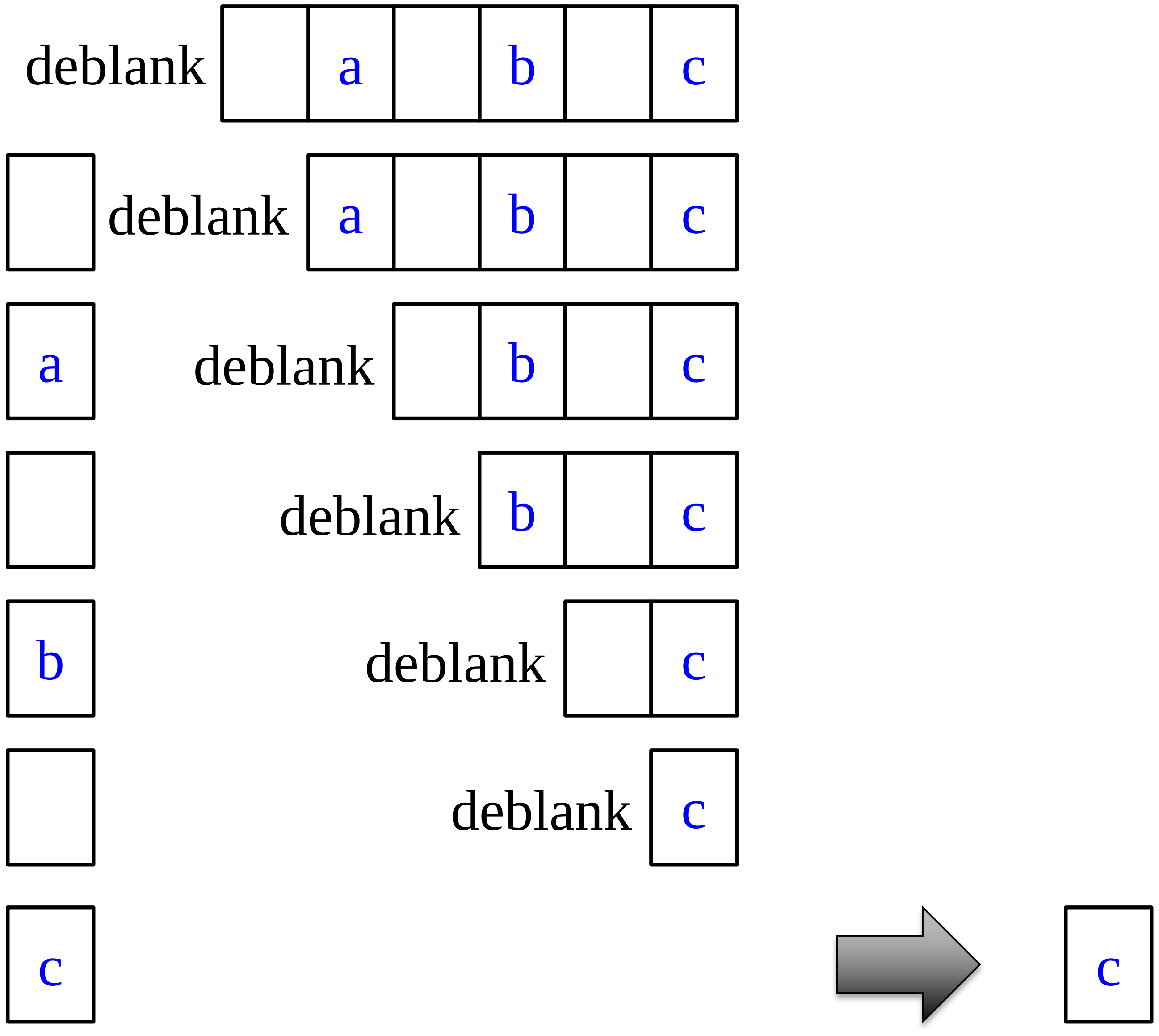
Following the Recursion



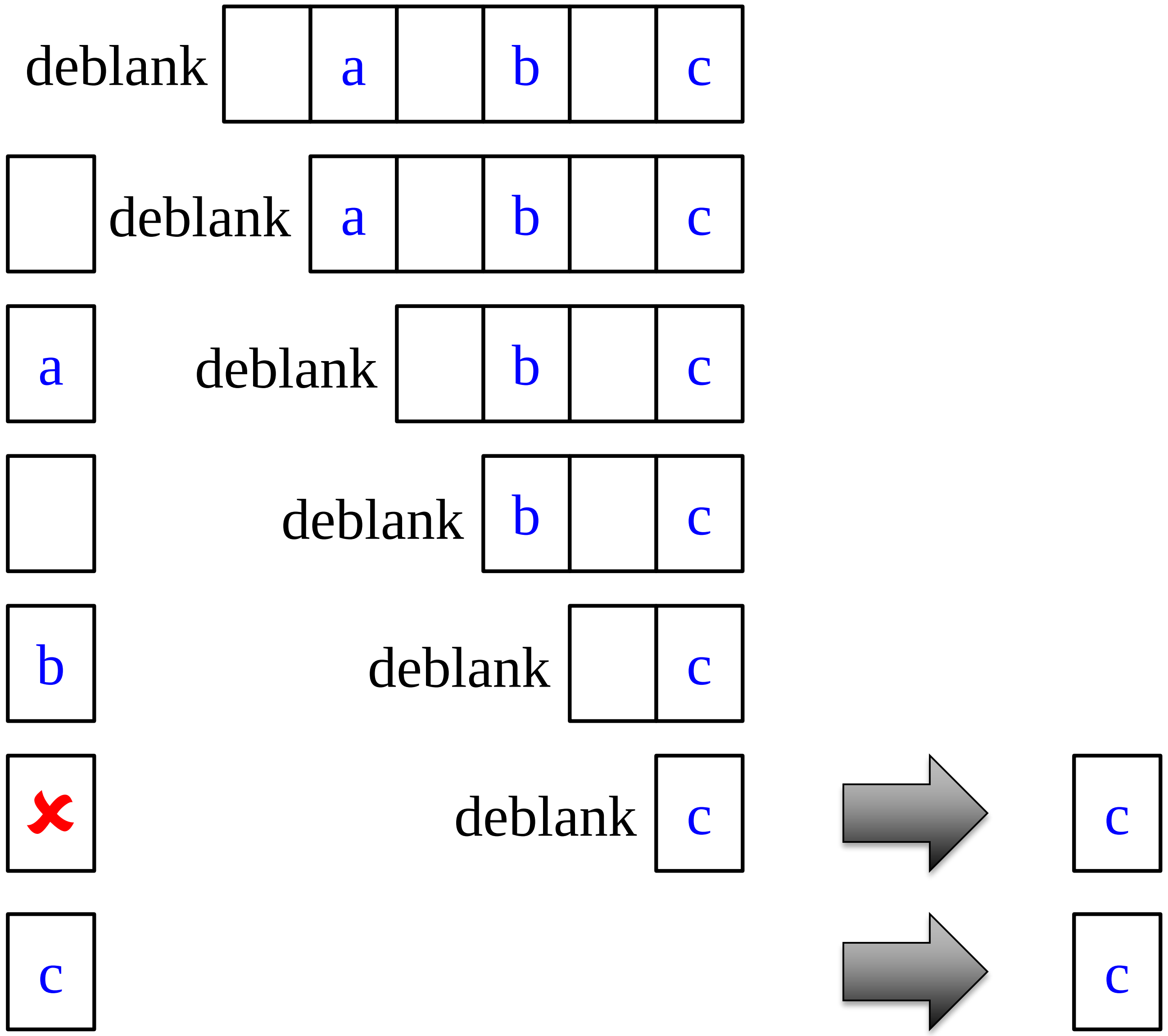
Following the Recursion



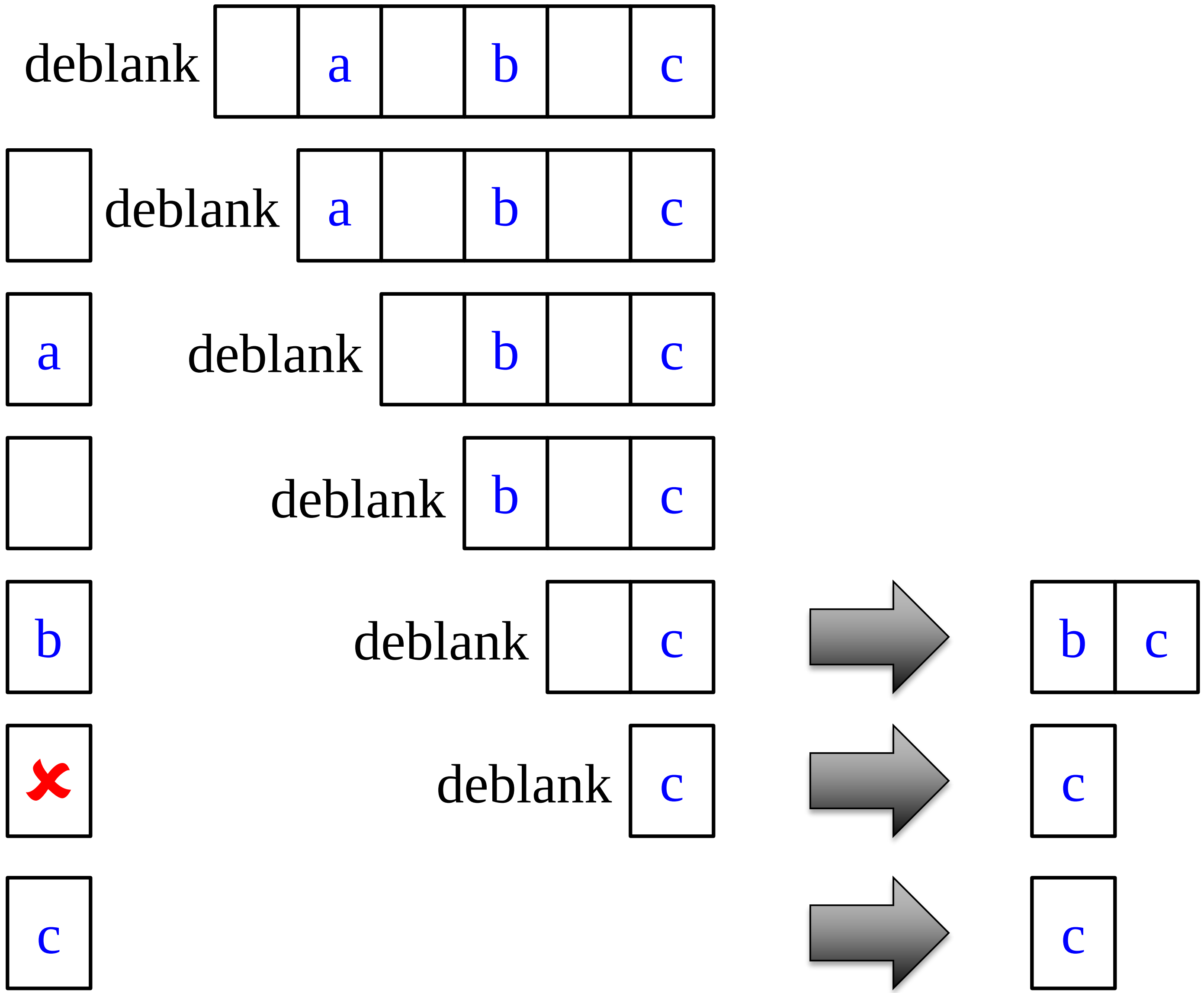
Following the Recursion



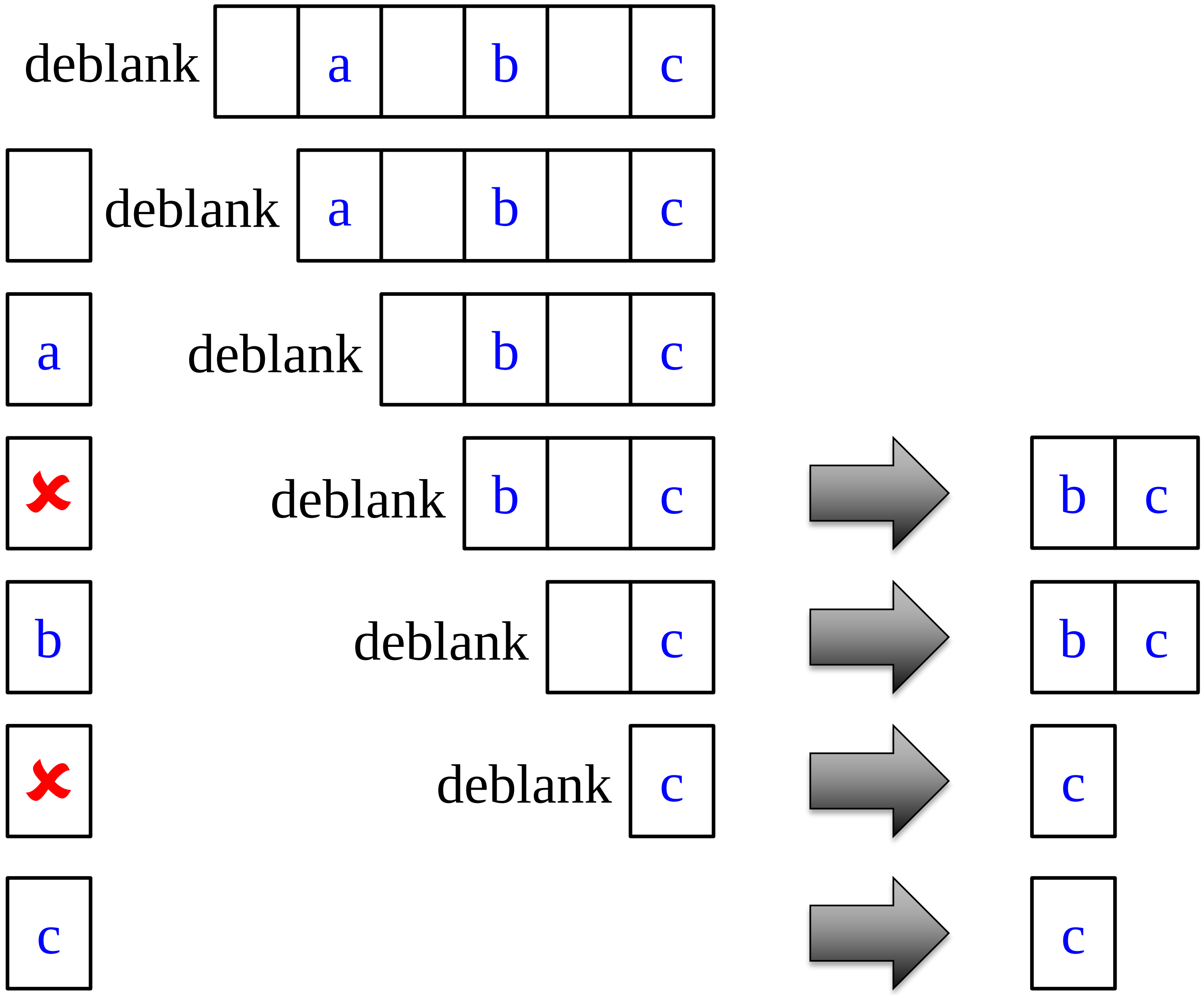
Following the Recursion



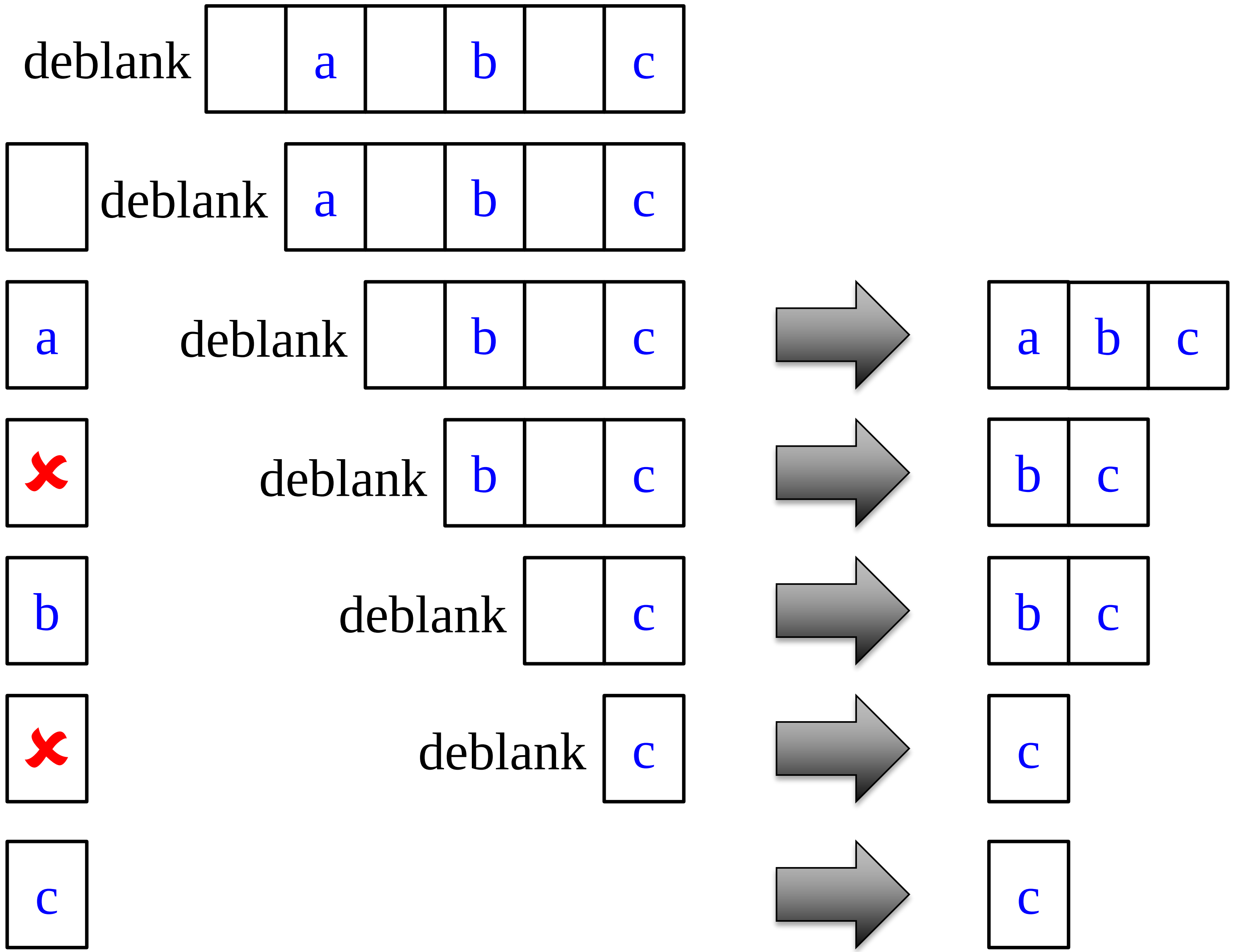
Following the Recursion



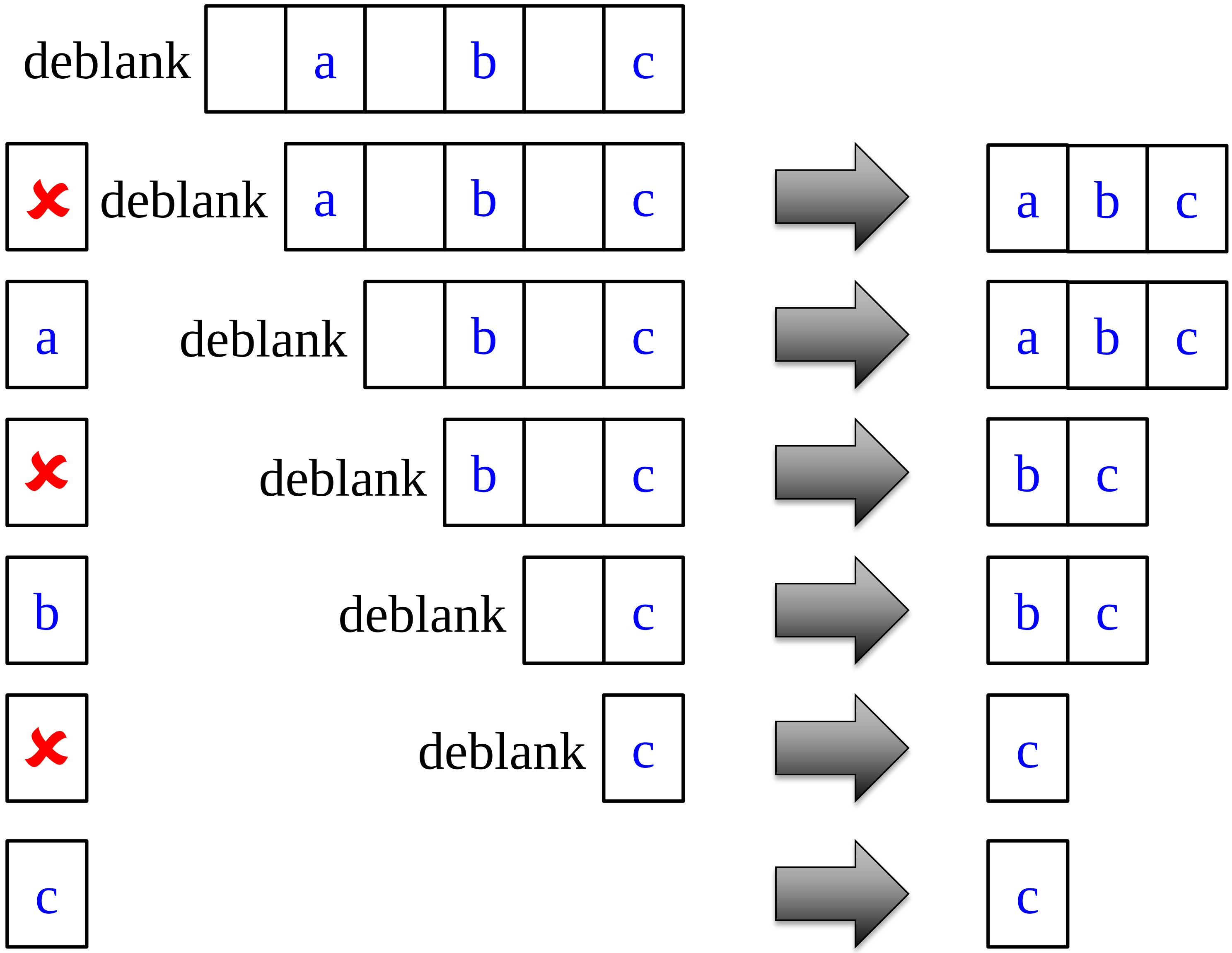
Following the Recursion



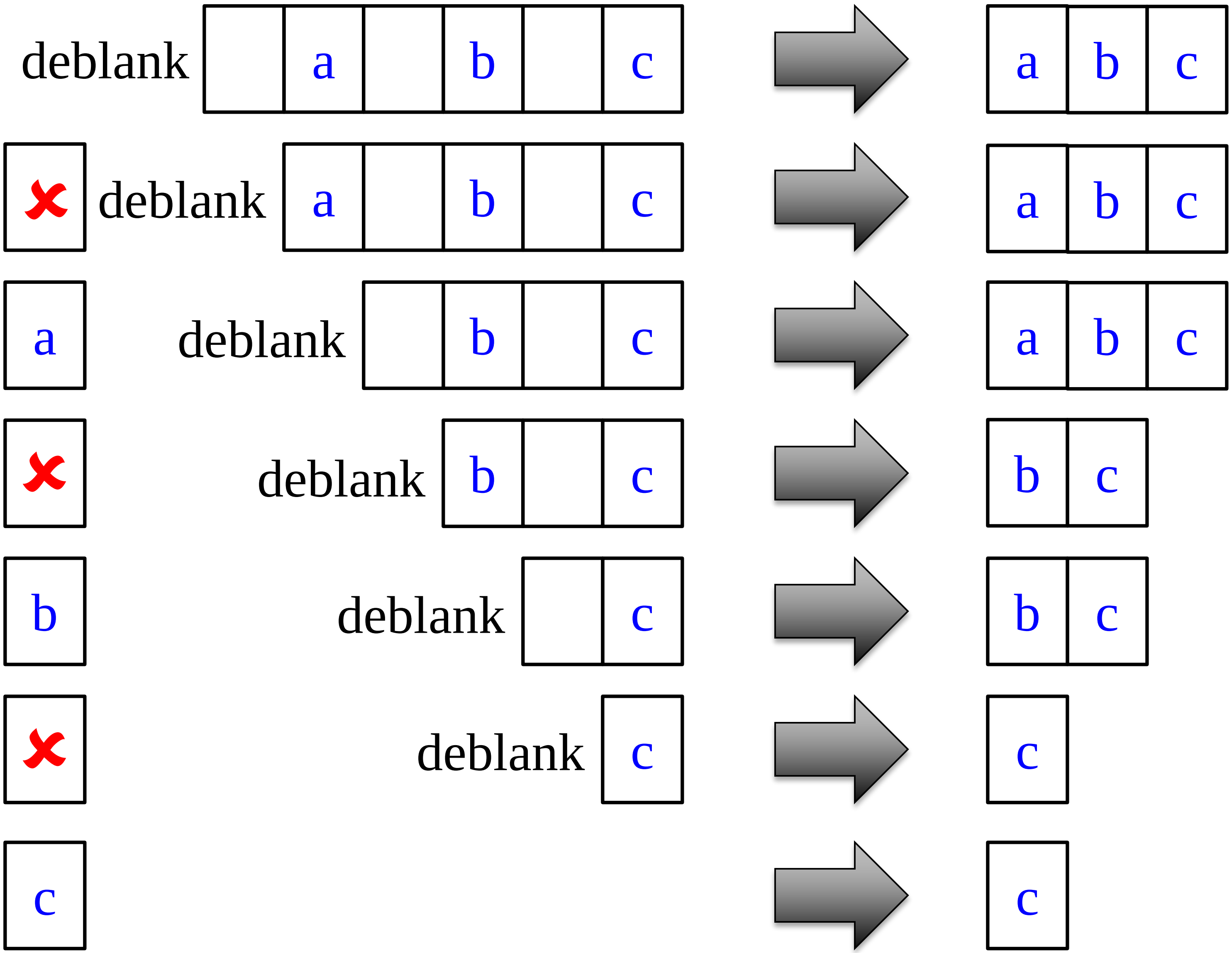
Following the Recursion



Following the Recursion

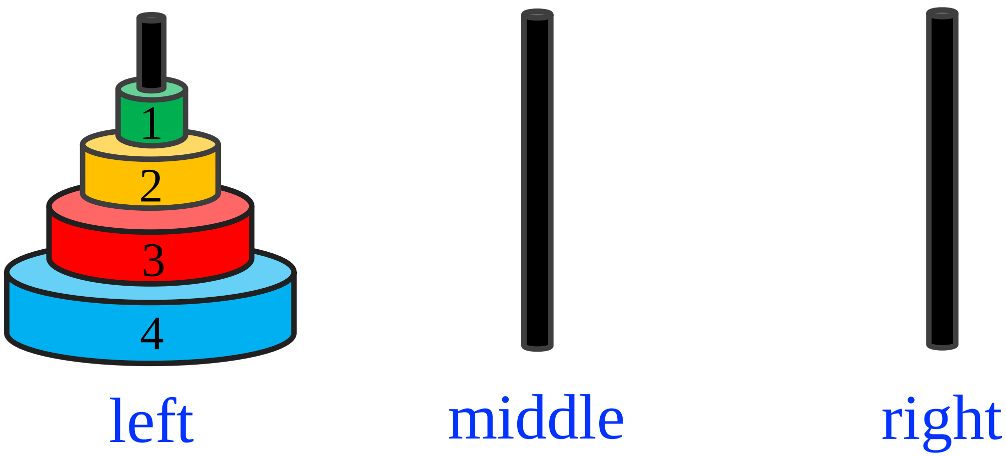


Following the Recursion



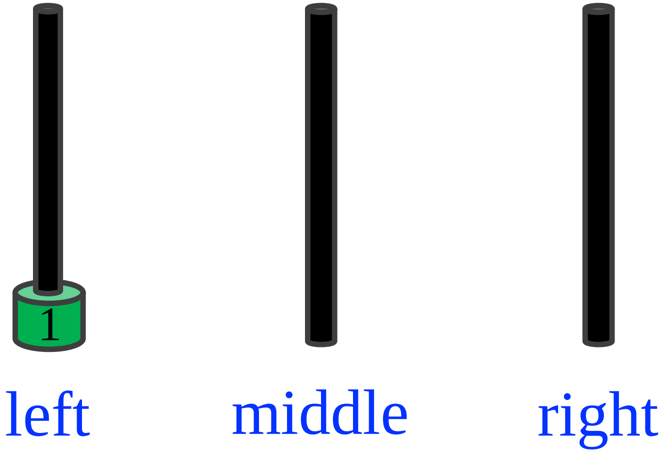
Tower of Hanoi

- Three towers: *left*, *middle*, and *right*
- *n* disks of unique sizes on *left*
- **Goal**: move all disks from *left* to *right*
- Cannot put a larger disk on top of a smaller disk



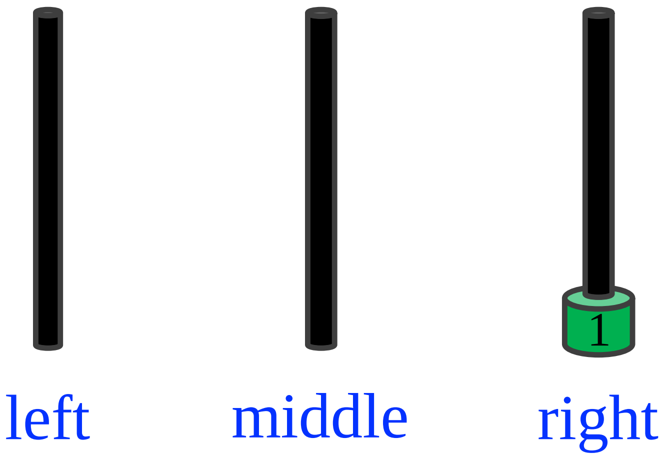
1 Disc

1. Move from *left* to *right*



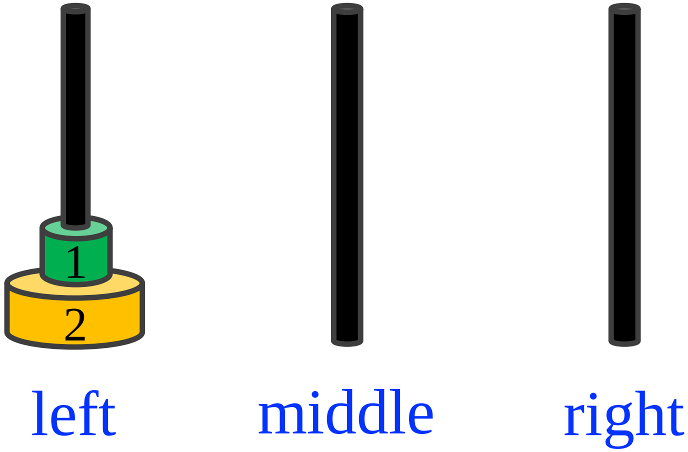
1 Disc

1. Move from *left* to *right*



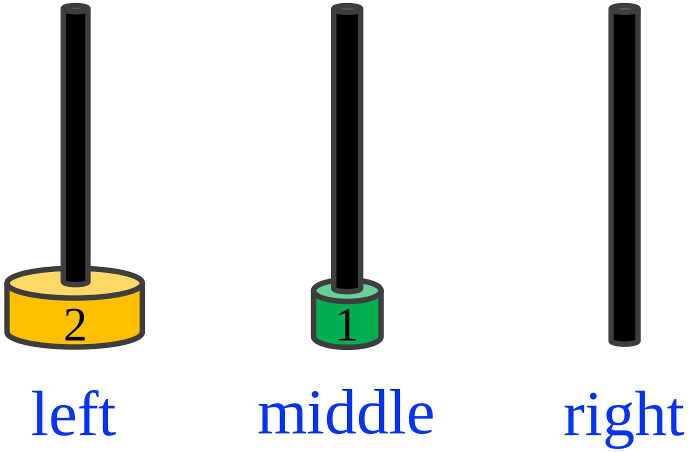
2 Discs

1. Move from *left* to *middle*

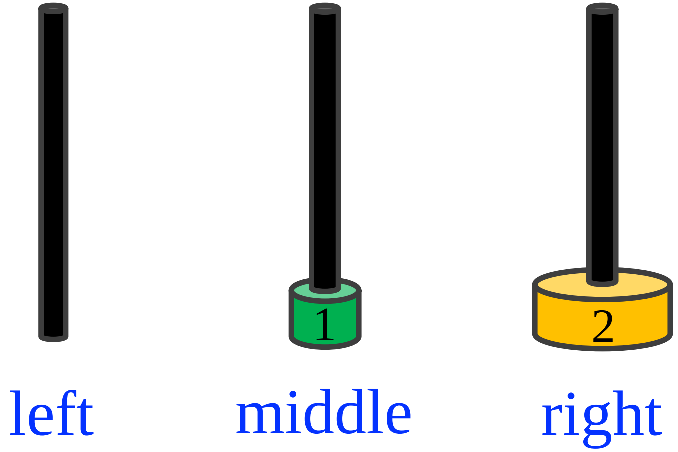


2 Discs

1. Move from *left* to *middle*
2. Move from *left* to *right*

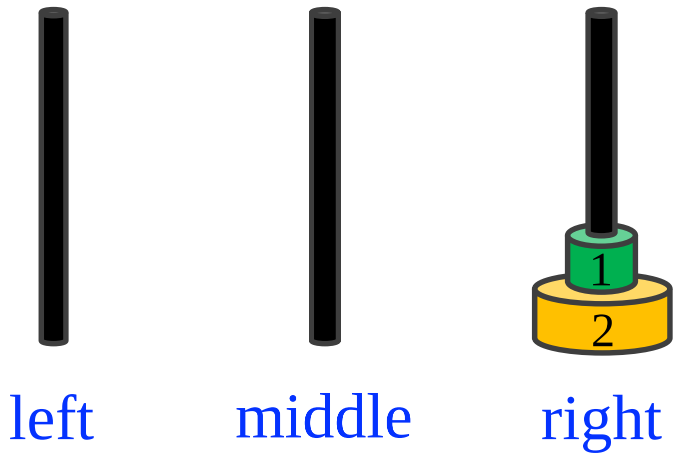


2 Discs



1. Move from *left* to *middle*
2. Move from *left* to *right*
3. Move from *middle* to *right*

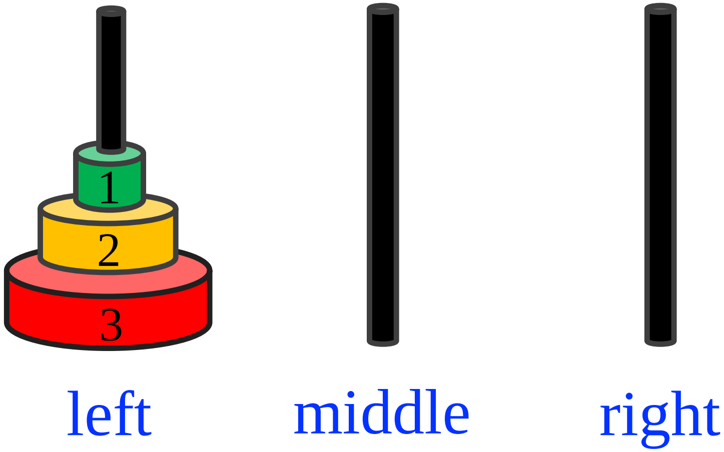
2 Discs



1. Move from *left* to *middle*
2. Move from *left* to *right*
3. Move from *middle* to *right*

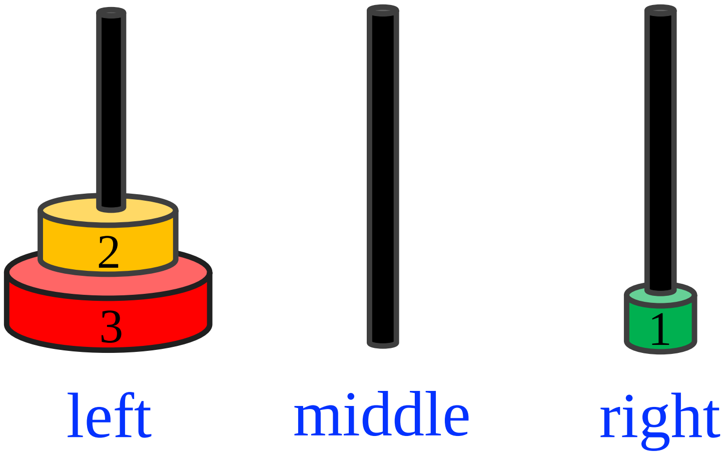
3 Discs

1. Move from *left* to *right*

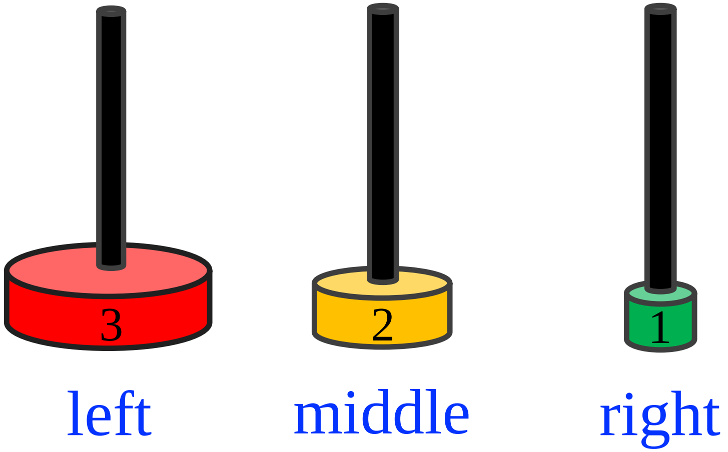


3 Discs

1. Move from *left* to *right*
2. Move from *left* to *middle*

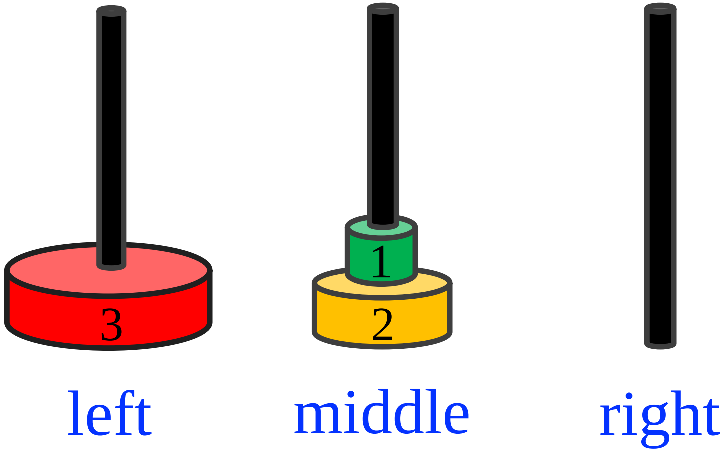


3 Discs



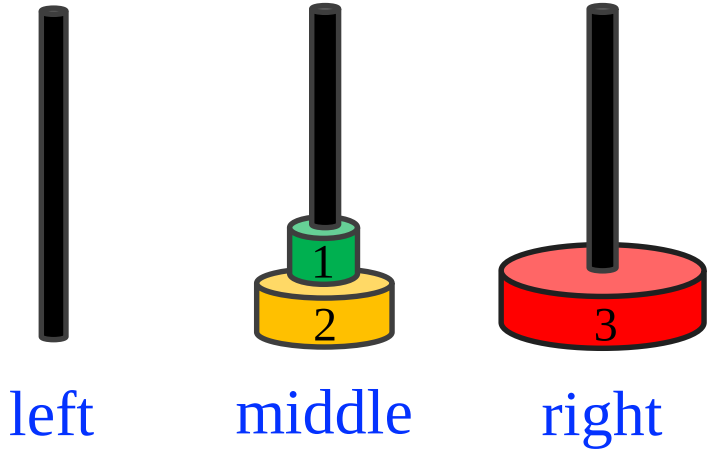
1. Move from *left* to *right*
2. Move from *left* to *middle*
3. Move from *right* to *middle*

3 Discs



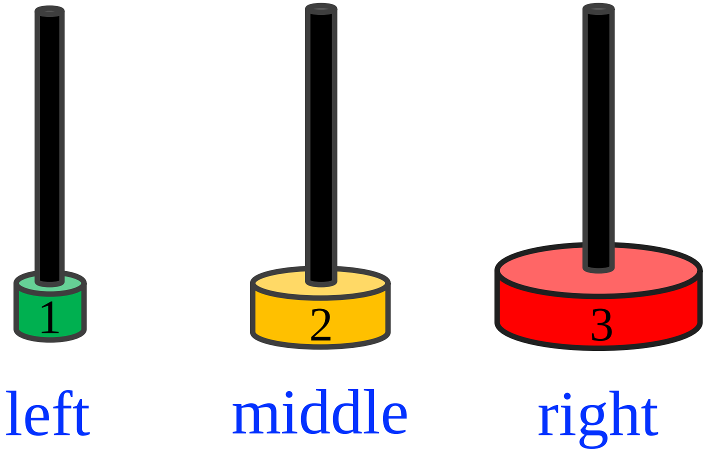
1. Move from *left* to *right*
2. Move from *left* to *middle*
3. Move from *right* to *middle*
4. Move from *left* to *right*

3 Discs



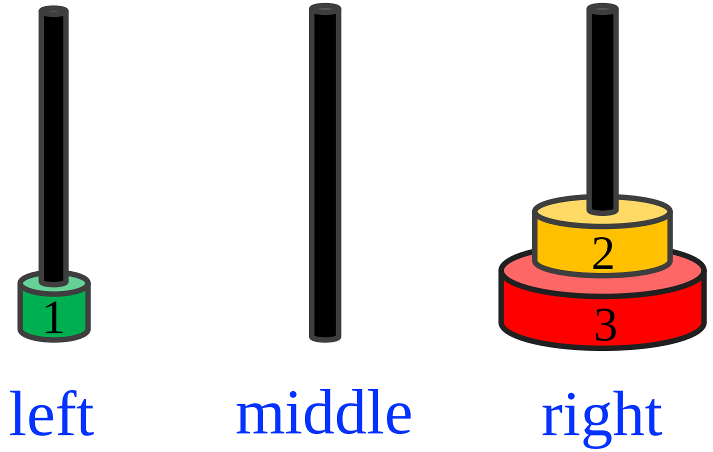
1. Move from *left* to *right*
2. Move from *left* to *middle*
3. Move from *right* to *middle*
4. Move from *left* to *right*
5. Move from *middle* to *left*

3 Discs



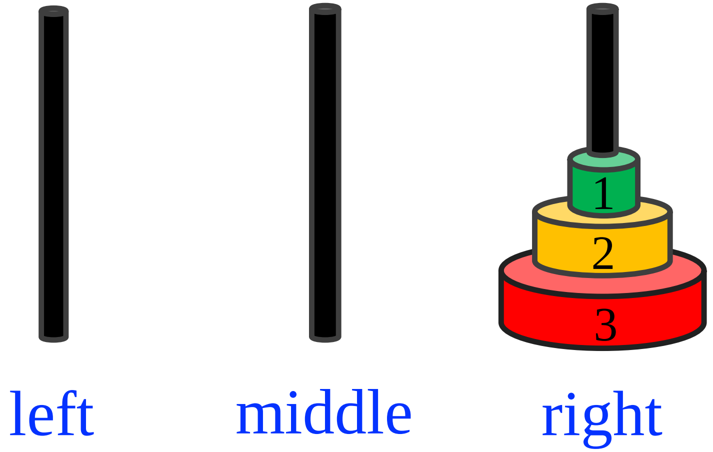
1. Move from *left* to *right*
2. Move from *left* to *middle*
3. Move from *right* to *middle*
4. Move from *left* to *right*
5. Move from *middle* to *left*
6. Move from *middle* to *right*

3 Discs



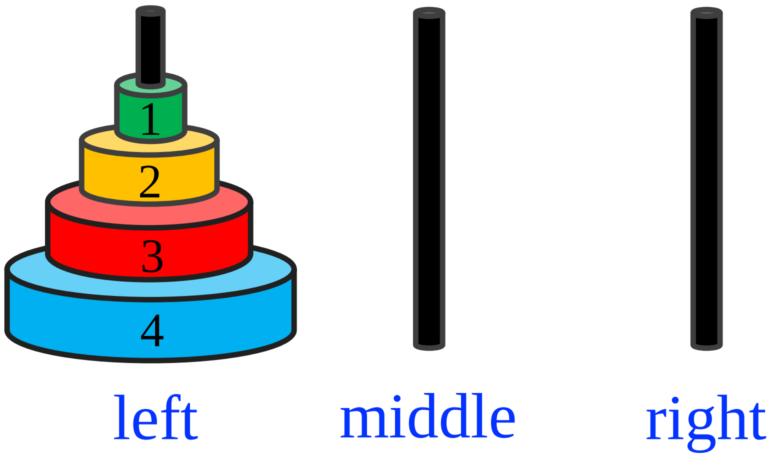
1. Move from *left* to *right*
2. Move from *left* to *middle*
3. Move from *right* to *middle*
4. Move from *left* to *right*
5. Move from *middle* to *left*
6. Move from *middle* to *right*
7. Move from *left* to *right*

3 Discs



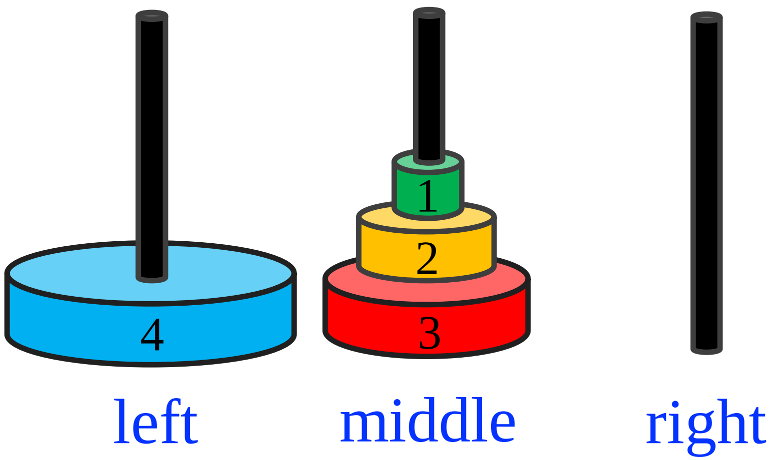
1. Move from *left* to *right*
2. Move from *left* to *middle*
3. Move from *right* to *middle*
4. Move from *left* to *right*
5. Move from *middle* to *left*
6. Move from *middle* to *right*
7. Move from *left* to *right*

4 Discs: High-level Idea

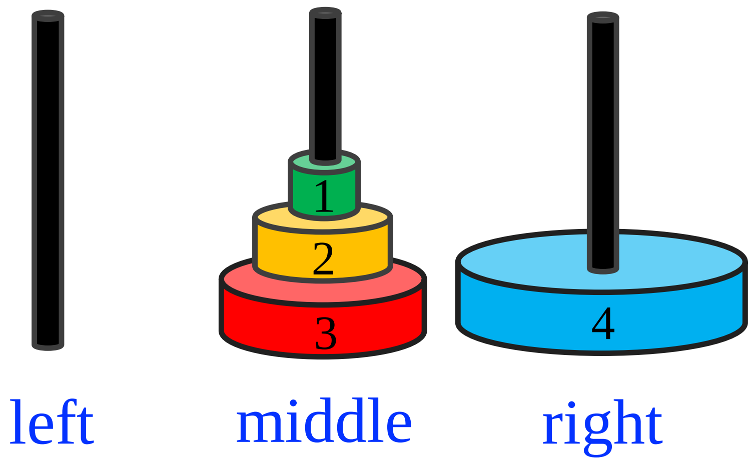


4 Discs: High-level Idea

- **Plan**: move top three disks from *left* to *middle*

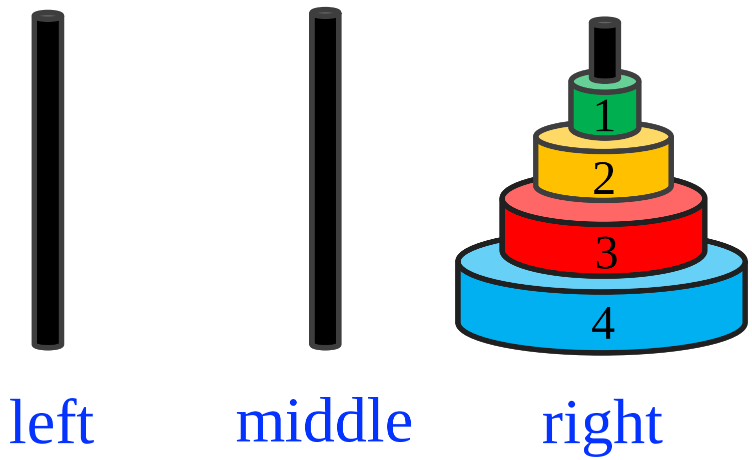


4 Discs: High-level Idea



- **Plan**: move top three disks from *left* to *middle*
- **Move**: largest disk from *left* to *right*

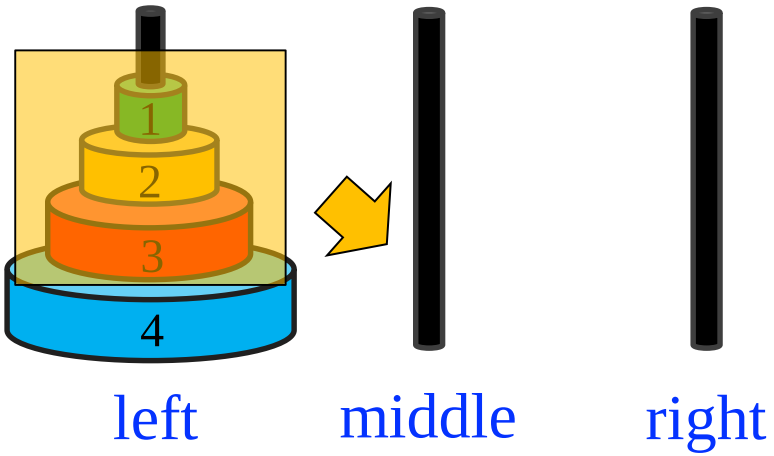
4 Discs: High-level Idea



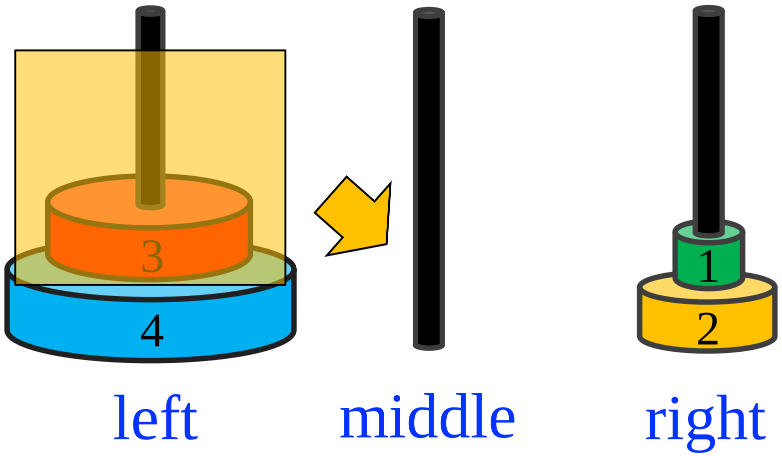
- **Plan**: move top three disks from *left* to *middle*
- **Move**: largest disk from *left* to *right*
- **Plan**: move top three disks from *middle* to *right*

4 Discs: High-level Idea

- **Plan**: move disks 1, 2, and 3 from *left* to *middle*

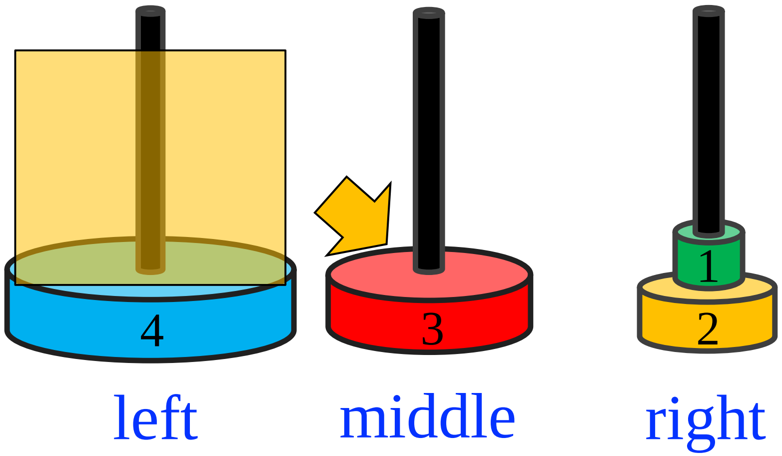


4 Discs: High-level Idea



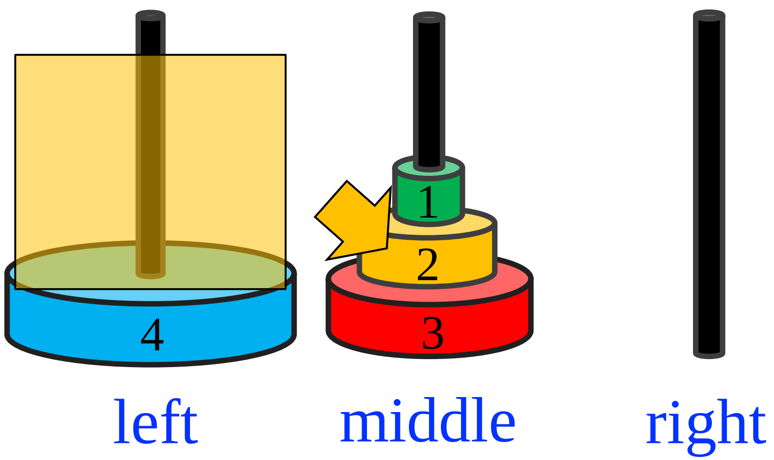
- **Plan**: move disks 1, 2, and 3 from *left* to *middle*
  - **Plan**: move disks 1 and 2 from *left* to *right*

4 Discs: High-level Idea



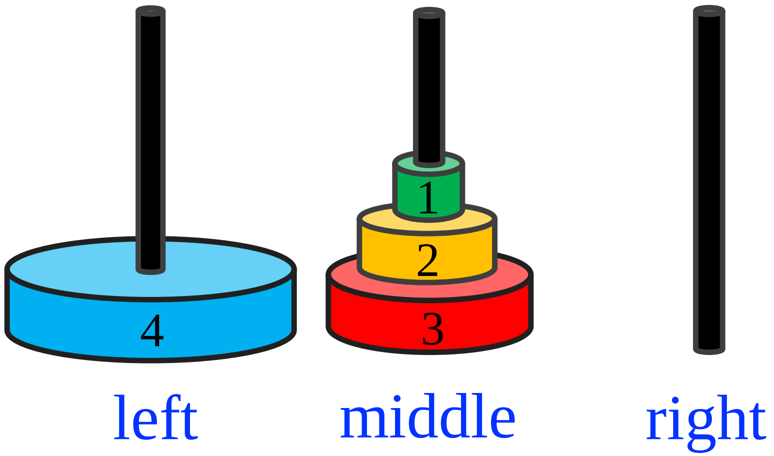
- **Plan**: move disks 1, 2, and 3 from *left* to *middle*
  - **Plan**: move disks 1 and 2 from *left* to *right*
  - **Move**: disk 3 from *left* to *right*

4 Discs: High-level Idea



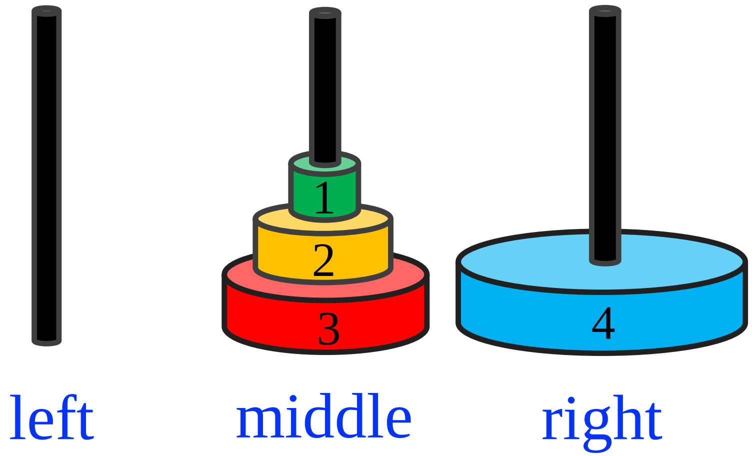
- **Plan**: move disks 1, 2, and 3 from *left* to *middle*
  - **Plan**: move disks 1 and 2 from *left* to *right*
  - **Move**: disk 3 from *left* to *right*
  - **Plan**: move disks 1 and 2 from *right* to *middle*

4 Discs: High-level Idea



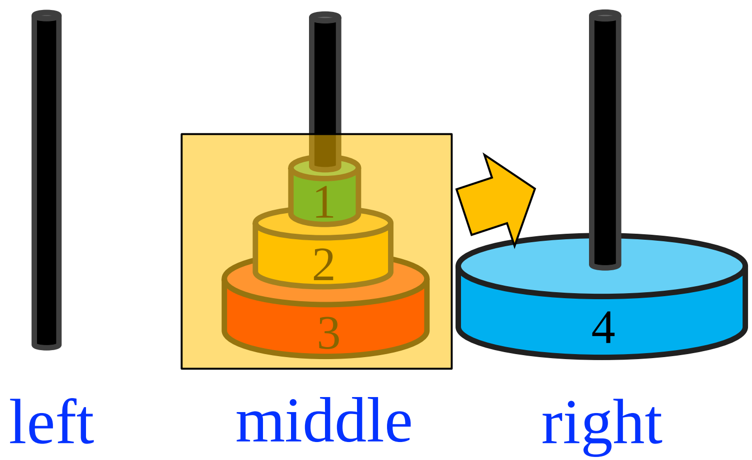
- **Plan**: move disks 1, 2, and 3 from *left* to *middle*
  - **Plan**: move disks 1 and 2 from *left* to *right*
  - **Move**: disk 3 from *left* to *right*
  - **Plan**: move disks 1 and 2 from *right* to *middle*

4 Discs: High-level Idea



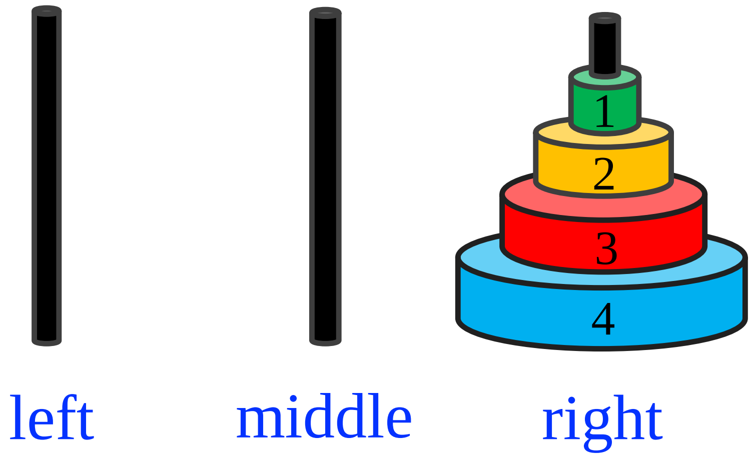
- **Plan**: move disks 1, 2, and 3 from *left* to *middle*
  - **Plan**: move disks 1 and 2 from *left* to *right*
  - **Move**: disk 3 from *left* to *right*
  - **Plan**: move disks 1 and 2 from *right* to *middle*
- **Move**: disk 4 from *left* to *right*

4 Discs: High-level Idea



- **Plan**: move disks 1, 2, and 3 from *left* to *middle*
  - **Plan**: move disks 1 and 2 from *left* to *right*
  - **Move**: disk 3 from *left* to *right*
  - **Plan**: move disks 1 and 2 from *right* to *middle*
- **Move**: disk 4 from *left* to *right*
- **Plan**: move disks 1, 2, and 3 from *middle* to *right*

4 Discs: High-level Idea



- **Plan**: move disks 1, 2, and 3 from *left* to *middle*
  - **Plan**: move disks 1 and 2 from *left* to *right*
  - **Move**: disk 3 from *left* to *right*
  - **Plan**: move disks 1 and 2 from *right* to *middle*
- **Move**: disk 4 from *left* to *right*
- **Plan**: move disks 1, 2, and 3 from *middle* to *right*

Observation: Plans within a Plan

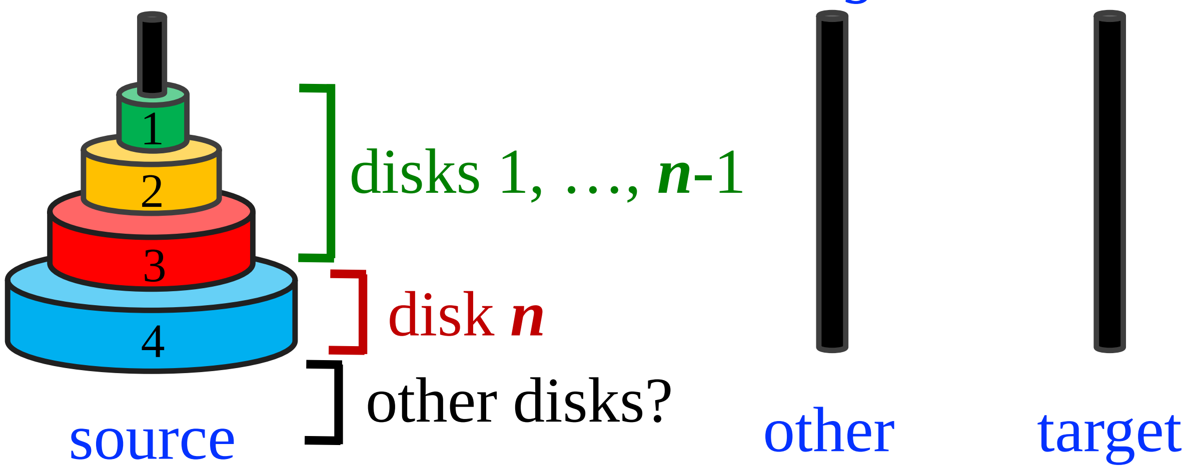
High-level plan

Low-level plan

- **Plan**: move disks 1, 2, and 3 from *left* to *middle*
  - **Plan**: move disks 1 and 2 from *left* to *right*
  - **Move**: disk 3 from *left* to *right*
  - **Plan**: move disks 1 and 2 from *right* to *middle*
- **Move**: disk 4 from *left* to *right*
- **Plan**: move disks 1, 2, and 3 from *middle* to *right*

General Pattern

To move ***n*** disks from *source* to *target:*



(*source*, *other*, and *target* can be any permutation of *left*, *middle* and *right*)

1. **Plan**: move disks 1, …, ***n***-1 from *source* to *other*
2. **Move**: disk ***n*** to from *source* to *target*
3. **Plan**: move disks 1, …, ***n***-1 from *other* to *target*