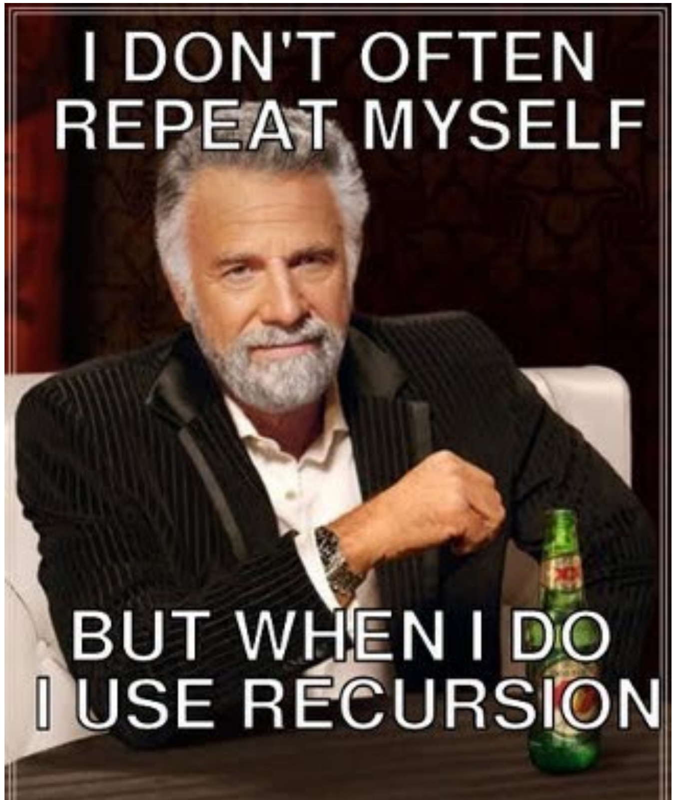
15-112
Fundamentals of Programming

Week 5 - Lecture 2:
Recursion

June 14, 2016



What is recursion?

Recursion:

To understand recursion, you have to first understand recursion.

What is recursion?

Recursion:

To understand recursion, you have to first understand recursion.

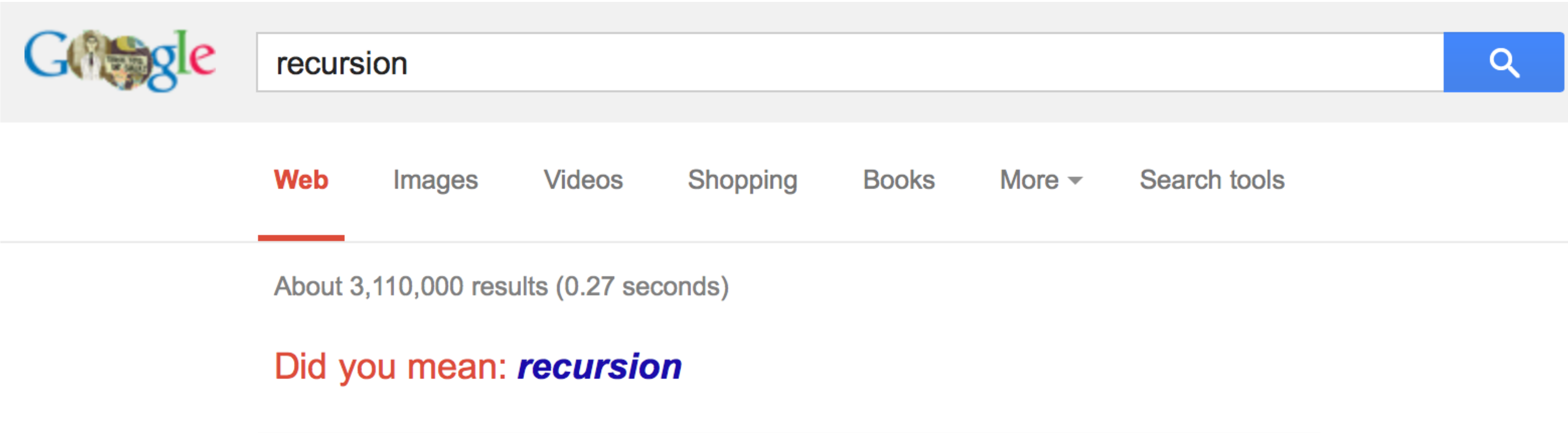
What is recursion?

Recursion:

To understand recursion, you have to first understand recursion.

Not making progress. Let's ask Google.

What is recursion?



Let’s see what my dictionary says.

What is recursion?

recursion (n):

See *recursion*

What is recursion in programming?

We say that a function is recursive if at some point, it calls itself.

```
def test():
    test()
```

Can we do something more meaningful?

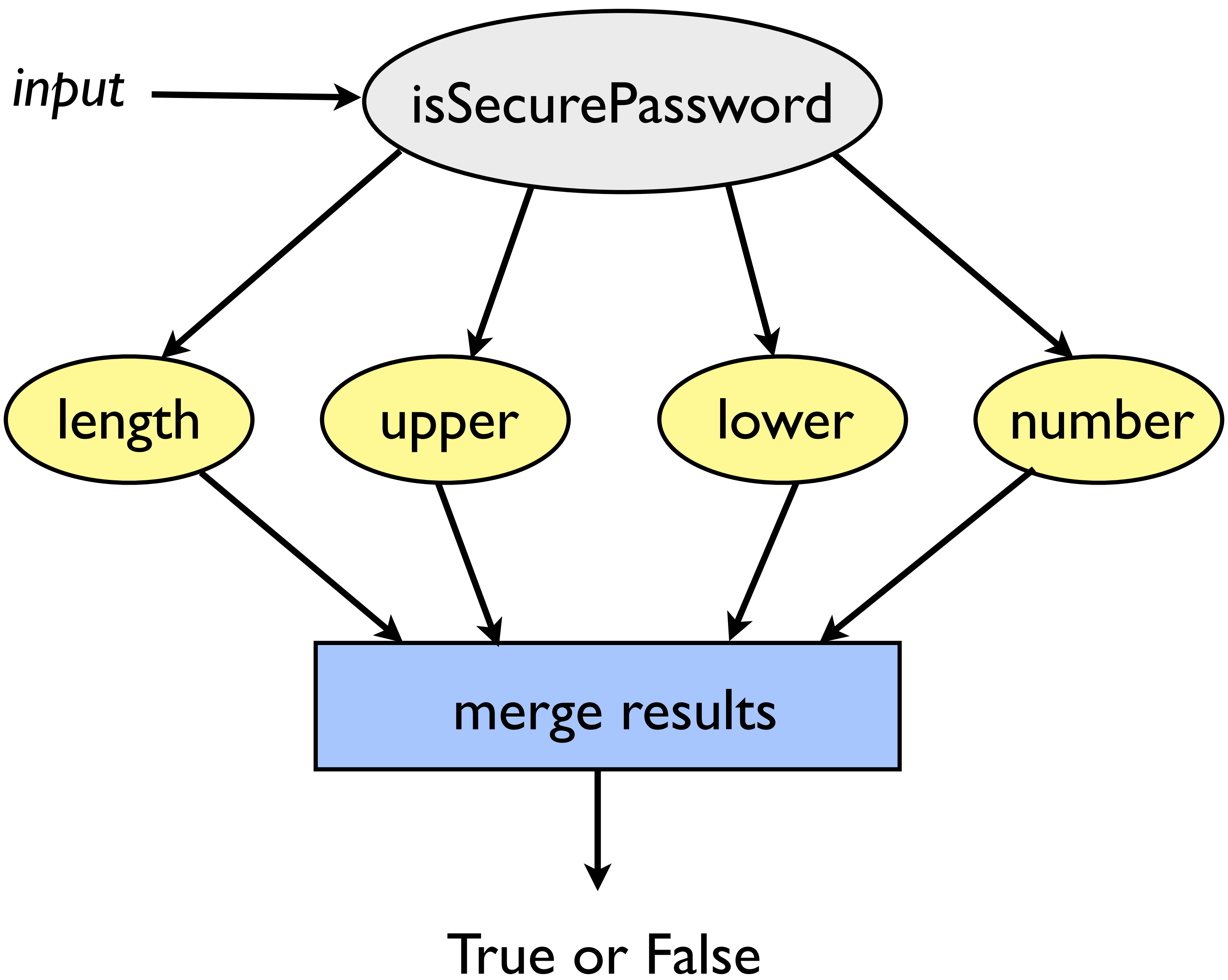
Warning:
Recursion can be weird and counter-intuitive at first!

Motivation: break a problem into smaller parts

Example: Figuring out if a given password is secure.

- Is string length at least 10?
- Does the string contain an upper-case letter?
- Does the string contain a lower-case letter?
- Does the string contain a number?

Motivation: break a problem into smaller parts



Motivation: break a problem into smaller parts

isSecurePassword:

The problem is split into smaller but **<u>different</u>** problems.

Recursion:

The smaller problems are **<u>not</u>** different.

They are smaller versions of the original problem.

Recursion Example: Sorting

Sorting the midterms by name.

Sort:

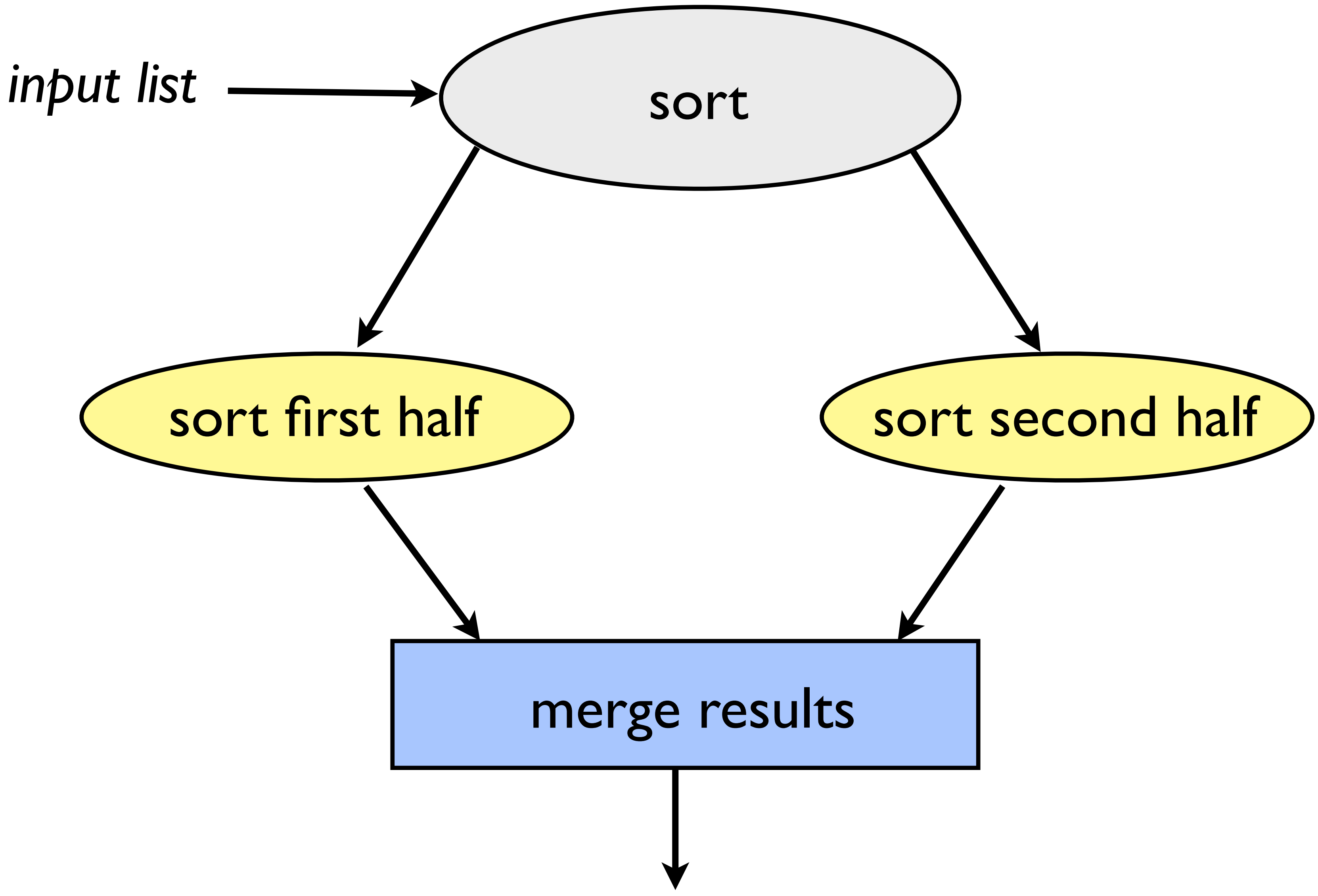
Divide the pile in half.

Sort the first half.

Sort the second half.

Merge the sorted piles.

Recursion Example: Sorting



Recursion Example: Sorting

Sort:

Divide the pile in half.

Sort the first half.

Sort the second half.

Merge the sorted piles.

What if my pile consists of just a single exam?

Recursion Example: Sorting

Sort:

If the pile consists of one element, do nothing.

Else:

Divide the pile in half.

Sort the first half.

Sort the second half.

Merge the sorted piles.

Recursion Example: Sorting

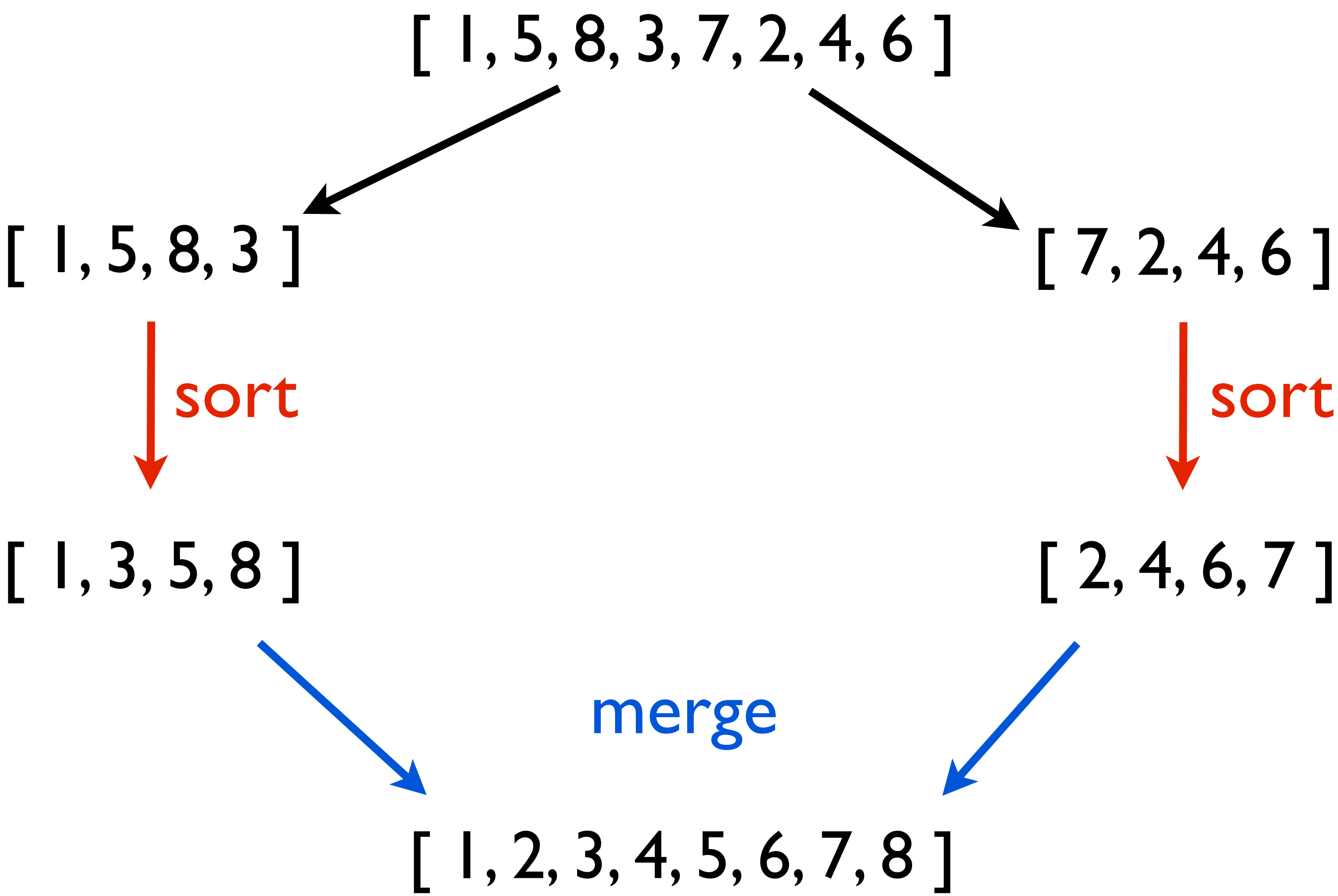
```
def merge(a, b):
    # We have already seen this.


def sort(a):
    if (len(a) <= 1):
        return a
    leftHalf = a[0 : len(a)//2]
    rightHalf = a[len(a)//2 : len(a)]
    return merge(sort(leftHalf),  sort(rightHalf))
```

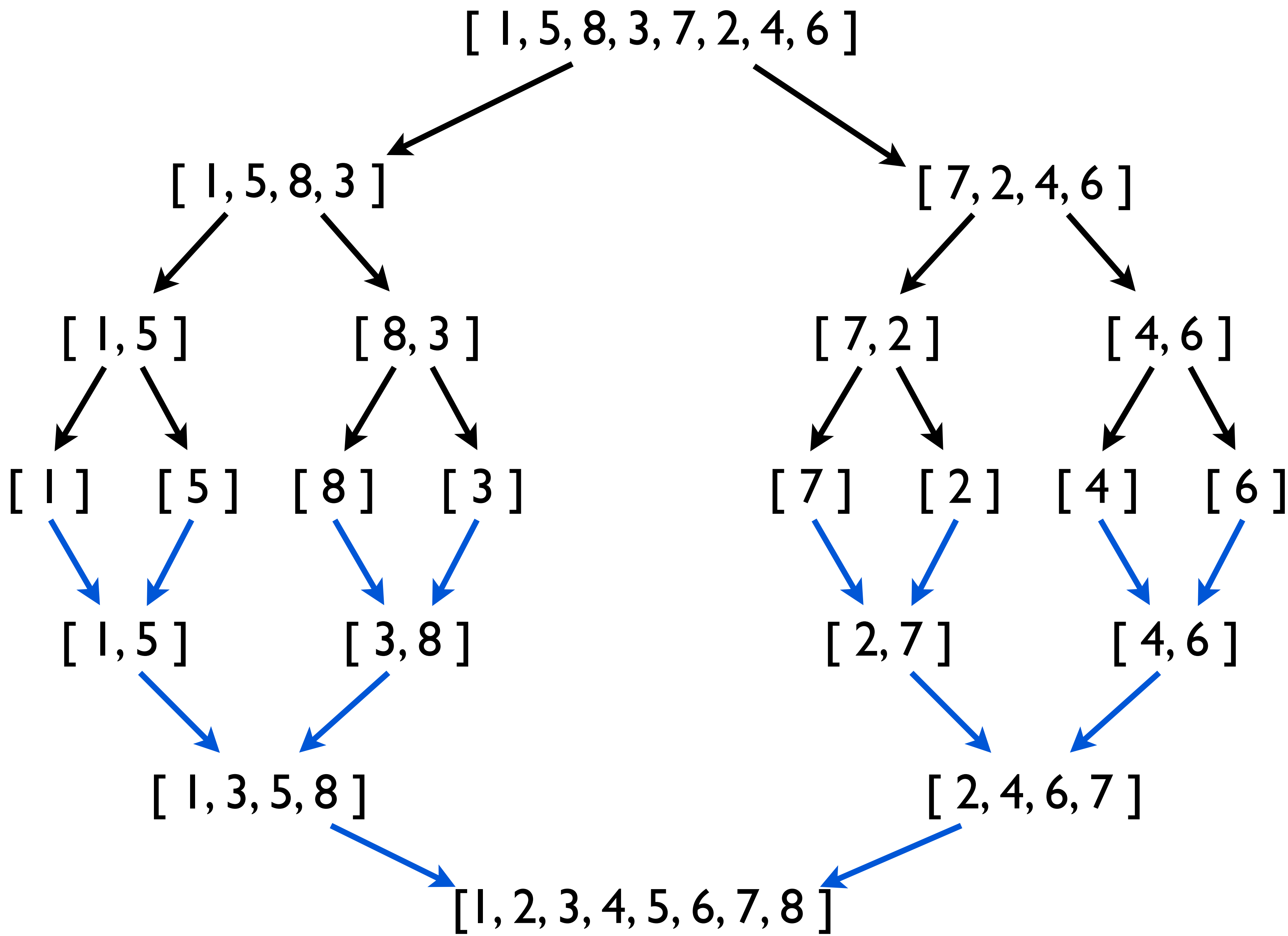
This works!

And it is called merge sort.

Recursion Example: Sorting



Recursion Example: Sorting



To understand how recursion works,
let's look at simpler examples.

Simple Example: Factorial

n factorial is the product of integers from 1 to n.

$1! = 1$

$2! = 2 \times 1$

$3! = 3 \times 2 \times 1$

$4! = 4 \times 3 \times 2 \times 1$

$5! = 5 \times 4 \times 3 \times 2 \times 1$

...

$n! = n \times (n-1) \times (n-2) \times \ldots \times 1$

Simple Example: Factorial

Finding the recursive structure in factorial:

Can we express n! using a smaller factorial ?

$n! = n \times (n - 1) \times (n - 2) \times \ldots \times 1$

Simple Example: Factorial

Finding the recursive structure in factorial:

Can we express n! using a smaller factorial ?

$n! = n \times (n - 1) \times (n - 2) \times \ldots \times 1$

$(n-1)!$

$n! = n \times (n - 1)!$

Simple Example: Factorial

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

“Unwinding” the code when n = 4:

factorial(4)

    4 * factorial(3)

        3 * factorial(2)

            2 * factorial(1)

                1 * factorial(0)

                    0 * factorial(-1)

                    ...

No stopping condition

Simple Example: Factorial

```
def factorial(n):
        if (n == 1): return 1
        else: return n * factorial(n - 1)
```

factorial(4)

4 * factorial(3)

3 * factorial(2)

2 * factorial(1)

1

Simple Example: Factorial

```
def factorial(n):
    if (n == 1): return 1
    else: return n * factorial(n - 1)
```

factorial(4)

4 * factorial(3)

3 * factorial(2)

2 * 1

Simple Example: Factorial

```
def factorial(n):
    if (n == 1): return 1
    else: return n * factorial(n - 1)
```

factorial(4)

4 * factorial(3)

3 * 2

Simple Example: Factorial

```
def factorial(n):
    if (n == 1): return 1
    else: return n * factorial(n - 1)
```

factorial(4) ⟶ evaluates to 24

4 * 6

Simple Example: Factorial

```
def factorial(n):
    if (n == 1): return 1
    else: return n * factorial(n - 1)
```

factorial(4) ⟶ evaluates to 24

4 * 6

Recursive calls make their way *down* to the base case.

The solution is then built *up* from base case.

Simple Example: Factorial

```
def factorial(n):
    if (n == 1): return 1
    else: return n * factorial(n - 1)
```

factorial(4) depth 0

4 * factorial(3) depth 1

3 * factorial(2) depth 2

2 * factorial(1) depth 3

1

Call Stack

Simple Example: Factorial

```
def factorial(n):
    if (n == 1): return 1
    else: return n * factorial(n - 1)
```

factorial(4) — depth 0

4 * factorial(3) — depth 1

3 * factorial(2) — depth 2

2 * factorial(1) — depth 3

1

Independent!

Call Stack

Simple Example: Factorial

```
def factorial(n):
    if (n == 1): return 1
    else: return n * factorial(n - 1)
```

<u>Another way of convincing ourselves it works:</u>

Does factorial(1) work (base case) ? ✓

Does factorial(2) work ? ✓

returns 2*factorial(1)

Does factorial(3) work ? ✓

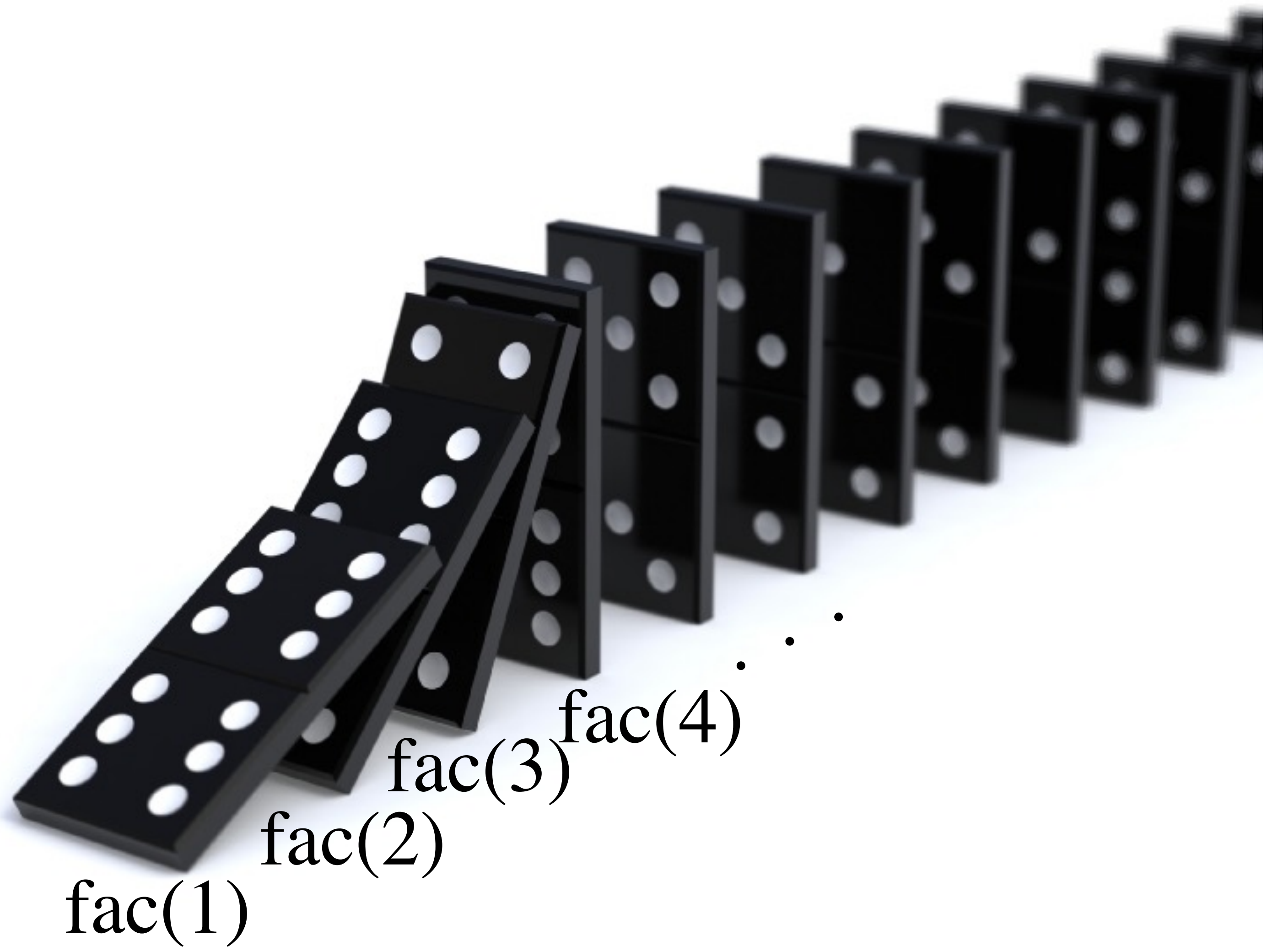
returns 3*factorial(2)

Does factorial(4) work ? ✓

returns 4*factorial(3)

How recursion works

fac(1) —> fac(2) —> fac(3) —> fac(4) —> ...



2 important properties of recursive functions

1. “Base case”

There should be a **base case**
(a case which does not make a recursive call)

2. “Progress”

The recursive call(s) should make **progress** towards the base case.

Simple Example: Factorial

def factorial(n):

    if (n == 1): **return** 1

    else: **return** n * factorial(n-1)

Simple Example: Factorial

```
def factorial(n):
    if (n == 1): return 1
    else: return n * factorial(n-1)
```

Base case

Simple Example: Factorial

```
def factorial(n):
    if (n == 1): return 1
    else: return n * factorial(n-1)
```

Making progress towards base case

Another example: Fibonacci

Fibonacci Sequence: 1 1 2 3 5 8 13 21 ...

```
def fib(n):
    if (n == 0): return 1
    else: return fib(n-1) + fib(n-2)
```

What happens when we call fib(1) ?

Another example: Fibonacci

Fibonacci Sequence: 1 1 2 3 5 8 13 21 ...

def fib(n):

    if (n == 0 **or** n == 1): **return** 1

    else: **return** fib(n-1) + fib(n-2)

Another example: Fibonacci

Fibonacci Sequence: 1 1 2 3 5 8 13 21 ...

```
def fib(n):
    if (n == 0 or n == 1): return 1
    else: return fib(n-1) + fib(n-2)
```

Base case

Another example: Fibonacci

Fibonacci Sequence: 1 1 2 3 5 8 13 21 ...

```
def fib(n):
    if (n == 0 or n == 1): return 1
    else: return fib(n-1) + fib(n-2)
```

Each recursive call makes progress towards the base case
(and doesn't skip it!!!)

Unwinding the code

fib(4)

Unwinding the code

fib(4)

$\downarrow$

fib(3) + fib(2)

Unwinding the code

fib(4)

fib(3) + fib(2)

fib(2) + fib(1)

Unwinding the code

fib(4)

fib(3) + fib(2)

fib(2) + fib(1)

fib(1) + fib(0)

Unwinding the code

fib(4)

fib(3) + fib(2)

fib(2) + fib(1)

1 + 1

Unwinding the code

fib(4)

fib(3) + fib(2)

2 + fib(1)

Unwinding the code

fib(4)

fib(3) + fib(2)

2 + 1

Unwinding the code

$$\mathrm{fib}(4)$$

$$\downarrow$$

$$3 \qquad + \qquad \mathrm{fib}(2)$$

Unwinding the code

fib(4)

3 + fib(2)

fib(1) + fib(0)

Unwinding the code

fib(4)

3 + fib(2)

1 + fib(0)

Unwinding the code

fib(4)

3 + fib(2)

1 + 1

Unwinding the code

fib(4)

↓

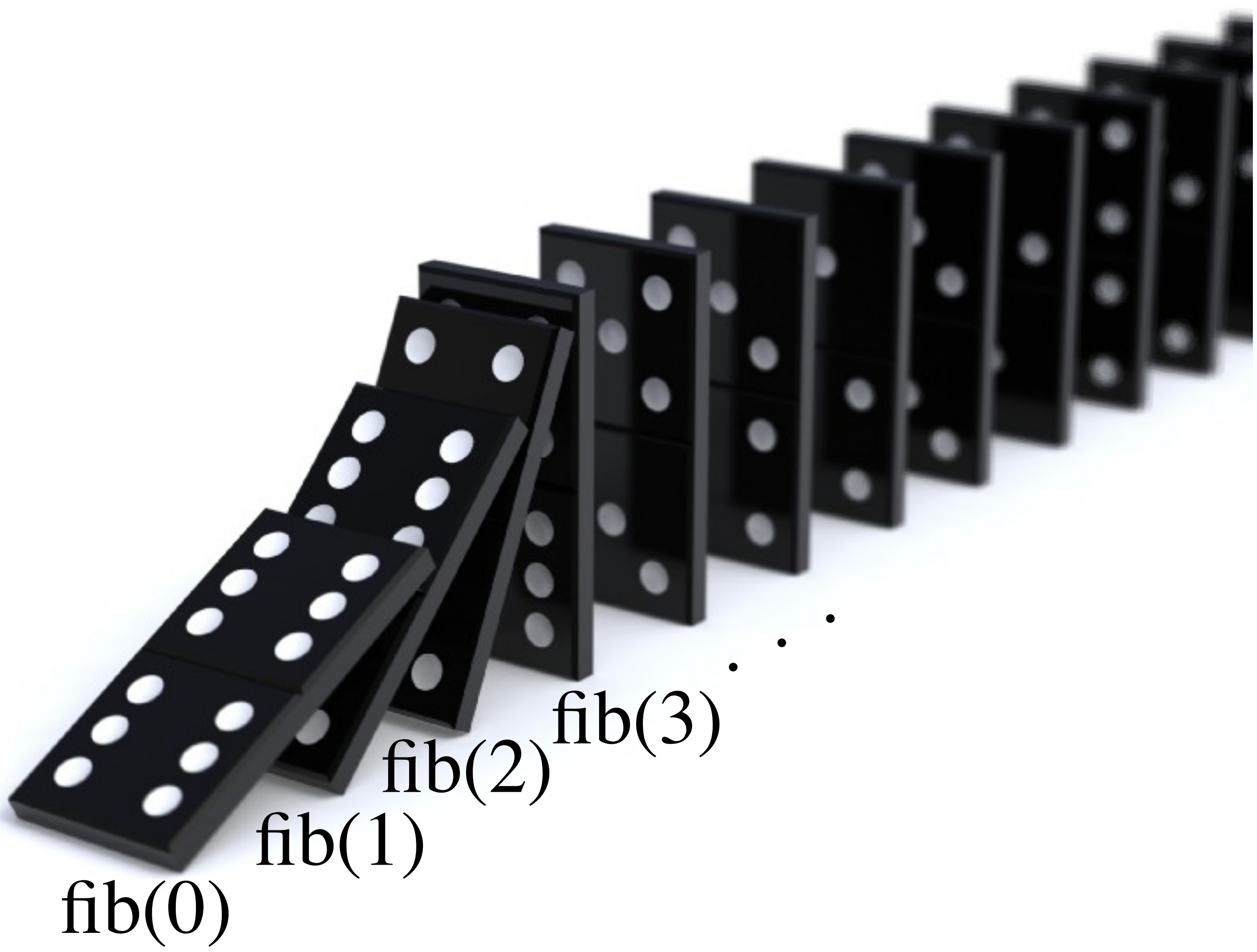
3 + 2

Unwinding the code

5

Recursion

fib(0), fib(1) —> fib(2) —> fib(3) —> fib(4) —> ...



The sweet thing about recursion

Do these 2 steps:

1. Base case:
Solve the "smallest" version of the problem (with no recursion).

2. Recursive call(s):
Correctly write the solution to the problem in terms of "smaller" version(s) of the same problem.

Your recursive function will always work!

Unwinding vs Trusting

Unwinding recursive functions:

- OK at first (for simple examples)
- Not OK once you understand the logic

Over time, you will start trusting recursion.

This trust is very important!

Recursion <u>**will**</u> earn your trust.

Unwinding vs Trusting

```
def fib(n):
    if (n == 0 or n == 1): return 1
    else: return fib(n-1) + fib(n-2)
```

You have to trust these will return the correct answer.

This is why recursion is so powerful.

You can assume every subproblem is solved for free!

Getting comfortable with recursion

1. See **<u>lot's</u>** of examples

2. Practice yourself

Getting comfortable with recursion

1. See <u>**lot's**</u> of examples

Recursive function design

Ask yourself:

If I had the solutions to the smaller instances for free, how could I solve the original problem?

Write the recursive relation:

e.g. fib(n) = fib(n-1) + fib(n-2)

Handle the base case:

A small version of the problem that does not require recursive calls.

Double check:

All your recursive calls make progress towards the base case(s) and they don't miss it.

Examples

Example: sum

Write a function that takes an integer n as input, and returns the sum of all numbers from 1 to n.

$$sum(n) = n + (n-1) + (n-2) + (n-3) + \ldots + 3 + 2 + 1$$

Example: sum

Write a function that takes an integer n as input, and returns the sum of all numbers from 1 to n.

$$\mathrm{sum}(n) = n + (n-1) + (n-2) + (n-3) + \ldots + 3 + 2 + 1$$

$$\mathrm{sum}(n) = n + \mathrm{sum}(n-1)$$

Example: sum

Write a function that takes an integer n as input, and returns the sum of all numbers from 1 to n.

```
def sum(n):
    if (n == 0): return 0
    else: return n + sum(n-1)
```

Example: sum in range

Write a function that takes integers n and m as input
(n <= m),
and returns the sum of all numbers from n to m.

$$sum(n, m) = n + (n+1) + (n+2) + \ldots + (m-1) + m$$

Example: sum in range

Write a function that takes integers n and m as input (n <= m),
and returns the sum of all numbers from n to m.

$$\text{sum}(n, m) = n + (n+1) + (n+2) + \ldots + (m-1) + m$$

$$\text{sum}(n, m) = \text{sum}(n, m-1) + m$$

Example: sum in range

Write a function that takes integers n and m as input (n <= m),
and returns the sum of all numbers from n to m.

$$\mathrm{sum}(n, m) = n + \underbrace{(n+1) + (n+2) + \ldots + (m-1) + m}_{\mathrm{sum}(n+1,\, m)}$$

Example: sum in range

Write a function that takes integers n and m as input (n <= m),
and returns the sum of all numbers from n to m.

$$\text{sum}(n, m) = n + (n+1) + (n+2) + \ldots + (m-1) + m$$

$$\text{sum}(n, m) = n + \text{sum}(n+1, m)$$

Example: sum in range

Write a function that takes integers n and m as input (n <= m),
and returns the sum of all numbers from n to m.

```
def sum(n, m):
    if (n == m): return n
    else: return n + sum(n+1, m)
```

Note: objects with recursive structure

Lists

0	1	2	4	5	5	6	8	9	9

Strings (a list of characters)

"Dammit I'm mad"

Problems related to these objects often have very natural recursive solutions.

Example: sumList(L)

Write a function that takes a list of integers as input and returns the sum of all the elements in the list.

3	5	2	6	9	1	5

sum(	3	5	2	6	9	1	5	) =

3 + sum(	5	2	6	9	1	5	)

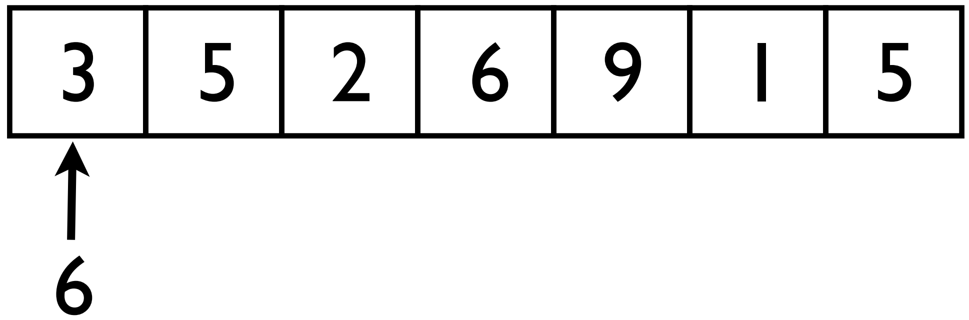
Example: sumList(L)

Write a function that takes a list of integers as input and returns the sum of all the elements in the list.

```
def sum(L):
    if (len(L) == 0): return 0
    else: return L[0] + sum(L[1:])
```

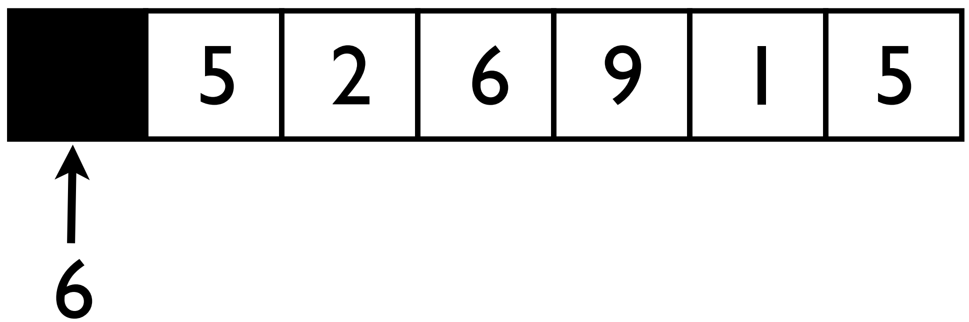
Example: isElement(L, e)

Write a function that checks if a given element is in a given list.



Example: isElement(L, e)

Write a function that checks if a given element is in a given list.



Example: isElement(L, e)

Write a function that checks if a given element is in a given list.

```
def isElement(L, e):
    if (len(L) == 0): return False
    else:
        if (L[0] == e): return True
        else: return isElement(L[1:], e)
```

This is linear search.

Example: isPalindrome(s)

Write a function that checks if a given string is a palindrome.

h a n n a h

Example: isPalindrome(s)

Write a function that checks if a given string is a palindrome.

should be palindrome

h a n n a h

Example: isPalindrome(s)

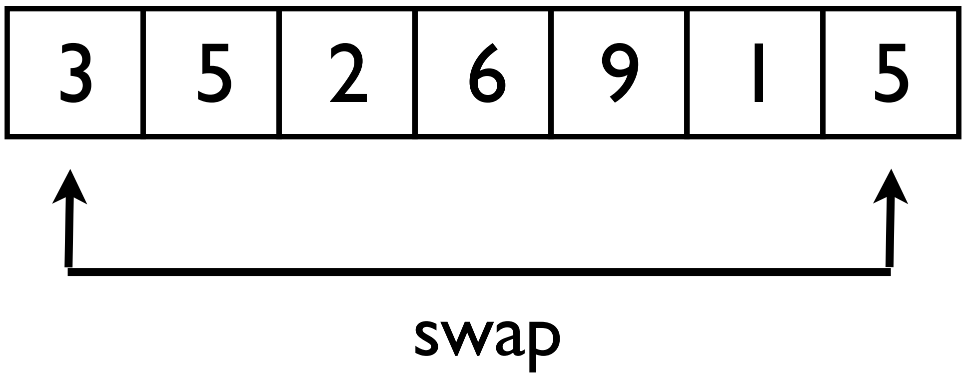
Write a function that checks if a given string is a palindrome.

```
def isPalindrome(s):
    if (len(s) <= 1): return True
    else:
        return (s[0] == s[len(s)-1]  and  isPalindrome(s[1:len(s)-1]))
```

Example: reverse array

Write a (non-destructive) function that reverses the elements of a list.

e.g. [1, 2, 3, 4] becomes [4, 3, 2, 1]



Example: reverse array

Write a (non-destructive) function that reverses the elements of a list.
e.g. [1, 2, 3, 4] becomes [4, 3, 2, 1]

5	5	2	6	9	1	3

reverse the middle

Example: reverse array

Write a (non-destructive) function that reverses the elements of a list.
e.g. [1, 2, 3, 4] becomes [4, 3, 2, 1]

```
def reverse(a):
    if (len(a) == 0 or len(a) == 1): return a
    else:
        return [a[-1]] + reverse(a[1:len(a)-1]) + [a[0]]
```

Example: findMax(L)

Write a function that finds the maximum value in a list.

3	5	2	6	9	1	5

Example: findMax(L)

Write a function that finds the maximum value in a list.

3	5	2	6	9	1	5

findMax

then compare it with 3

Example: findMax(L)

Write a function that finds the maximum value in a list.

```
def findMax(L):
    if (len(L) == 1): return L[0]
    else:
        m = findMax(L[1:])
        if (L[0] < m): return m
        else: return L[0]
```

if L = [ ], return None

Example: binary search

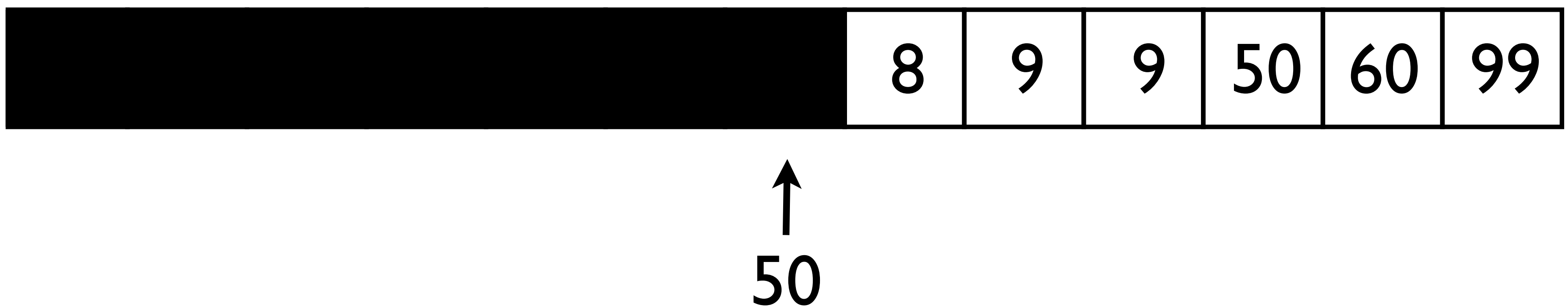
Write a function for binary search: find an element in a sorted list.

0	1	2	4	5	5	6	8	9	9	50	60	99

50

Example: binary search

Write a function for binary search: find an element in a sorted list.



Example: binary search

Write a function for binary search: find an element in a sorted list.

```
def binarySearch(a, element):
    if (len(a) == 0): return False
    mid = (start+end)//2
    if (a[mid] == element): return True
    elif (element < a[mid]):
        return binarySearch(a[:mid], element)
    else:
        return binarySearch(a[mid+1:], element)
```

Slicing too expensive here.

Example: binary search

```
def binarySearch(a, element, start, end):
    if (start >= end): return False
    mid = (start+end)//2
    if (a[mid] == element): return True
    elif (element < a[mid]):
        return binarySearch(a, element, start, mid)
    else:
        return binarySearch(a, element, mid+1, end)
```

Example: findMax(L)

Write a function that finds the maximum value in a list.

```
def findMax(L, start=0):
    if (start >= len(L)): return None
    elif (start == len(L)-1): return L[-1]
    else:
        m = findMax(L, start+1)
        if (L[start] < m): return m
        else: return L[start]
```

Common recursive strategies

With lists and strings, 2 common strategies:

Strategy 1:

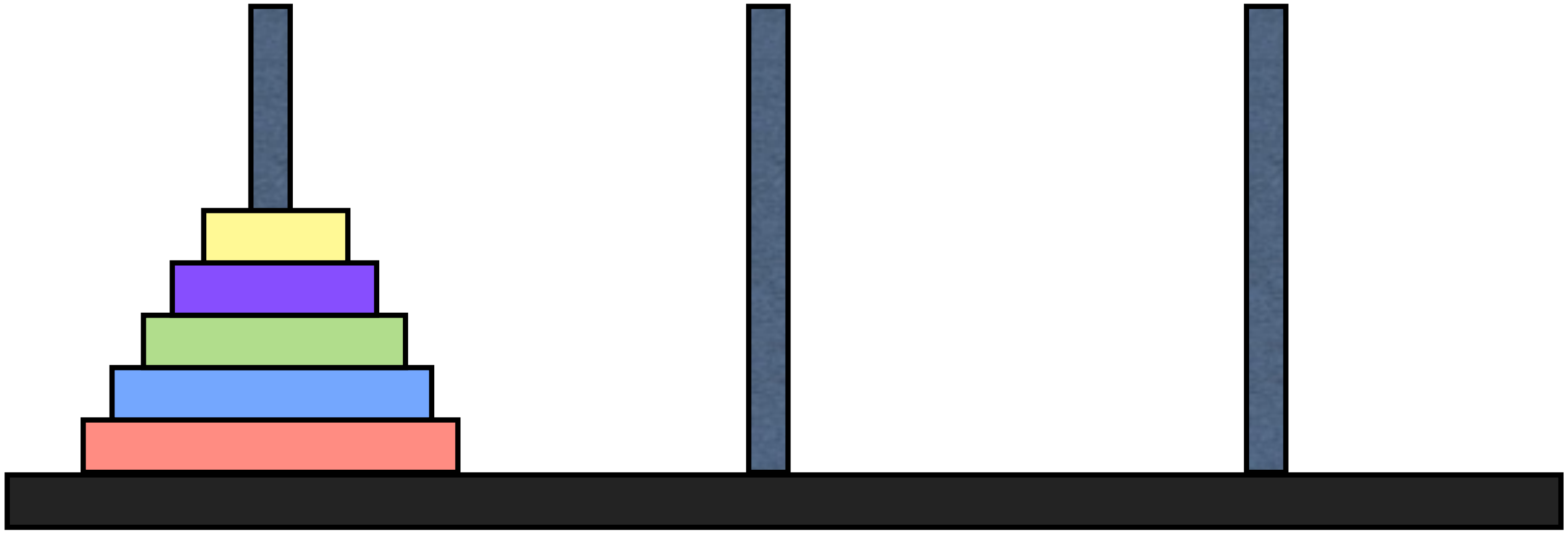
- Separate first or last index
- Use recursion on the remaining part

Strategy 2:

- Divide list or string in half
- Use recursion on each half, combine results.
  (or ignore one of the halves like in binary search)

One more example to really appreciate recursion

Example: Towers of Hanoi



Classic ancient problem:

N rings in increasing sizes.   3 poles.

Rings start stacked on Pole 1.

<u>Goal</u>: Move rings so they are stacked on Pole 3.

- Can only move one ring at a time.
- Can't put larger ring on top of a smaller ring.

Example: Towers of Hanoi

Example: Towers of Hanoi

Write a function

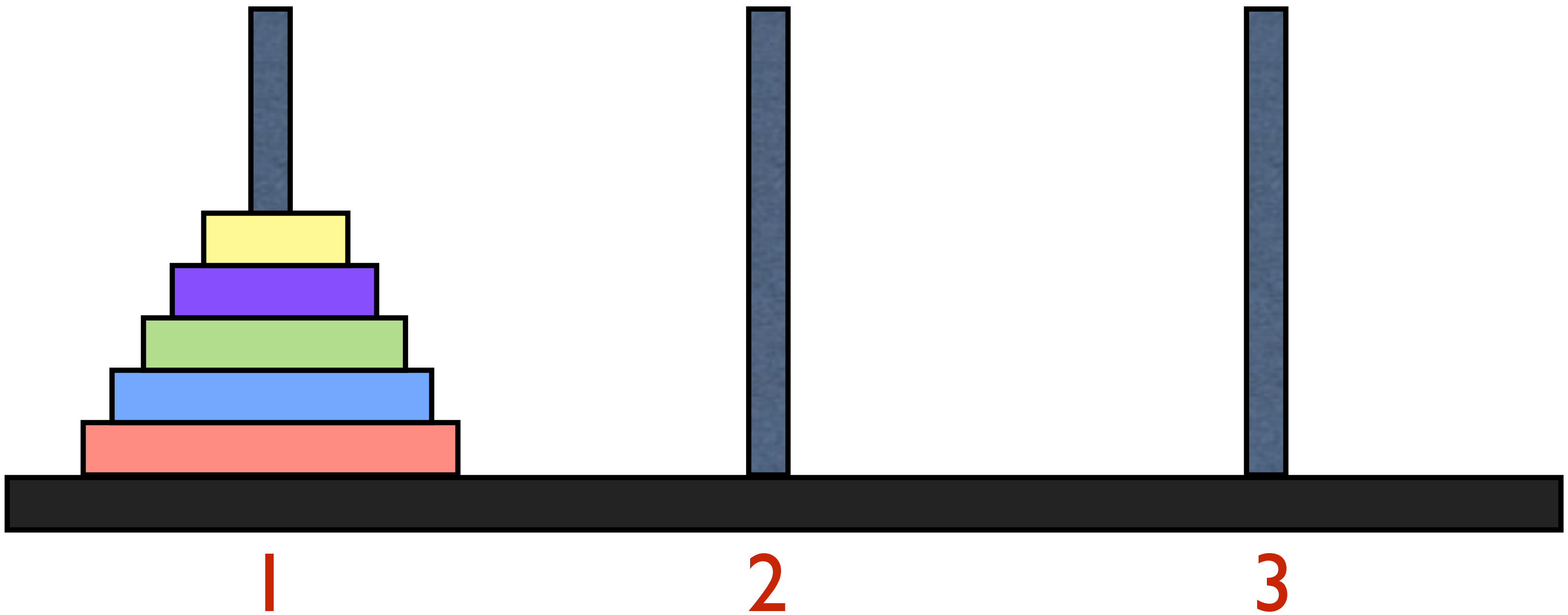
move (N, source, destination)   (integer inputs)

that solves the Towers of Hanoi problem
(i.e. moves the N rings from source to destination)
by printing all the moves.

move (3, 1, 3):

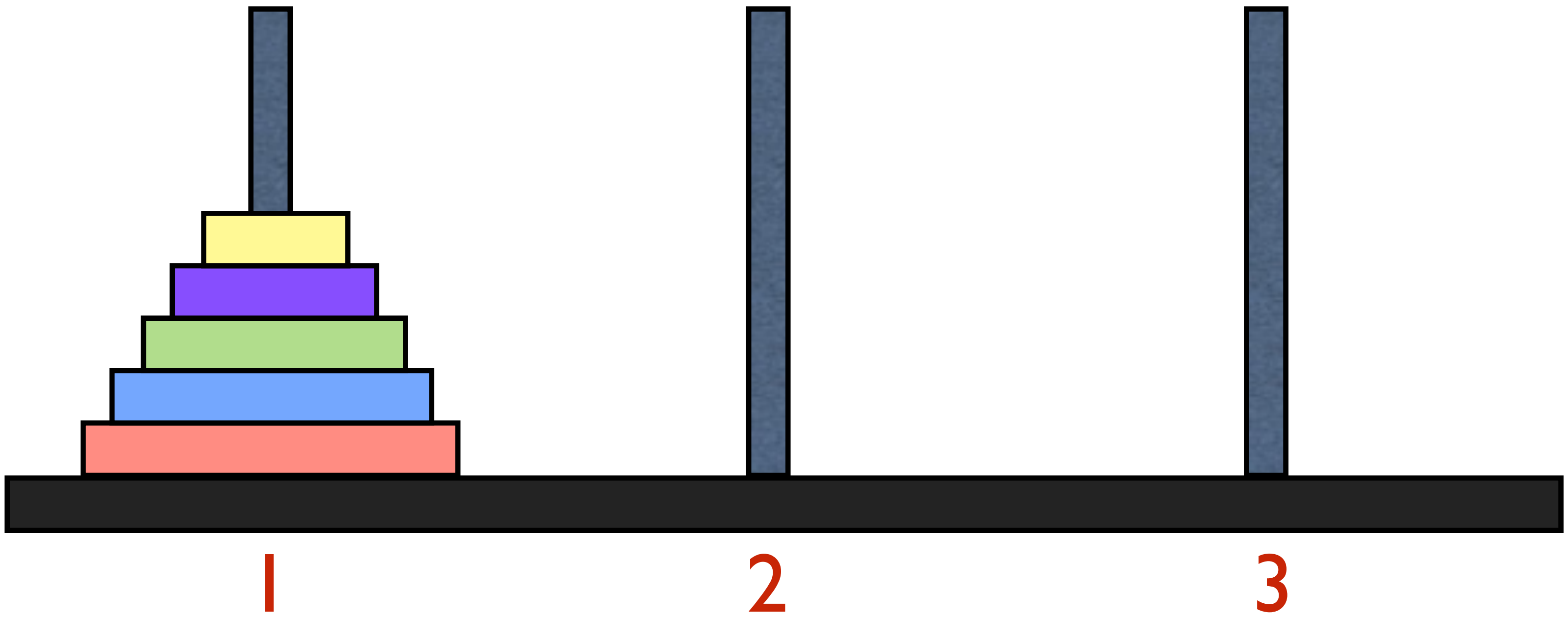
Move ring from Pole 1 to Pole 3
Move ring from Pole 1 to Pole 2
Move ring from Pole 3 to Pole 2
Move ring from Pole 1 to Pole 3
Move ring from Pole 2 to Pole 1
Move ring from Pole 2 to Pole 3
Move ring from Pole 1 to Pole 3

Example: Towers of Hanoi



<u>The power of recursion:</u> Can assume we can solve smaller instances of the problem for free.

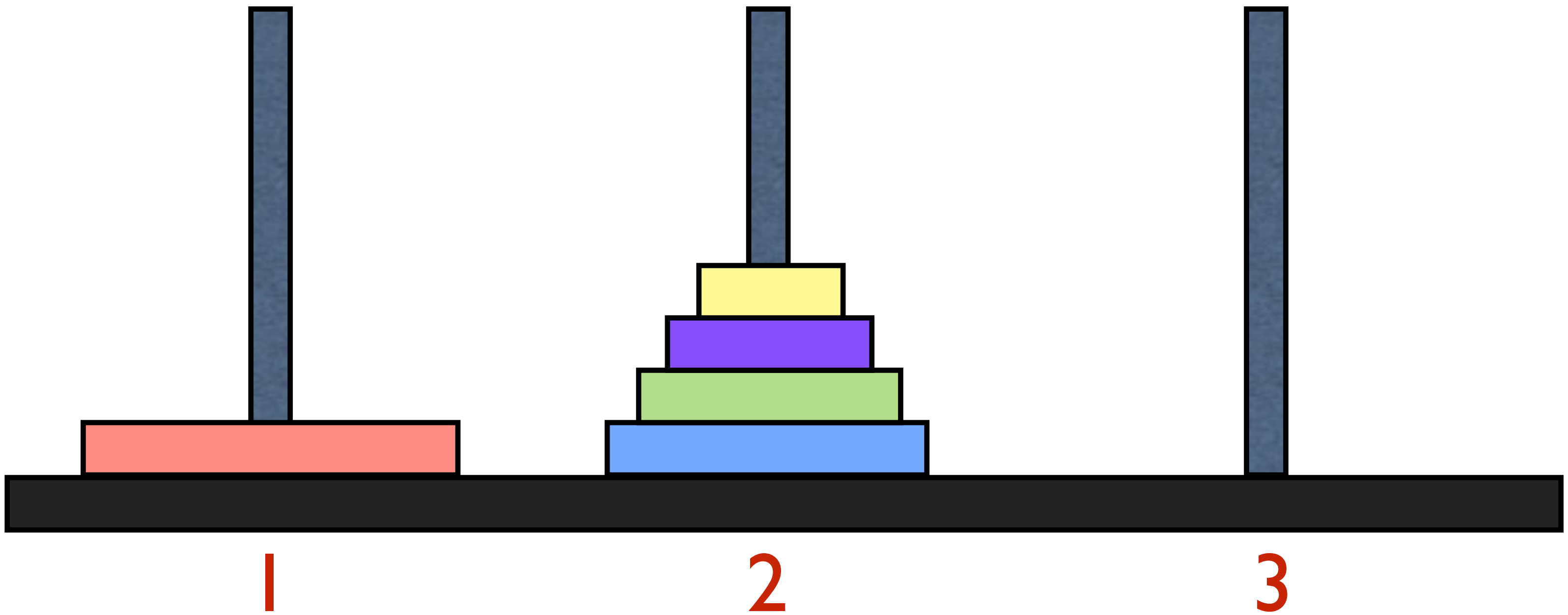
Example: Towers of Hanoi



<u>The power of recursion:</u> Can assume we can solve smaller instances of the problem for free.

- Move N-1 rings from Pole 1 to Pole 2.

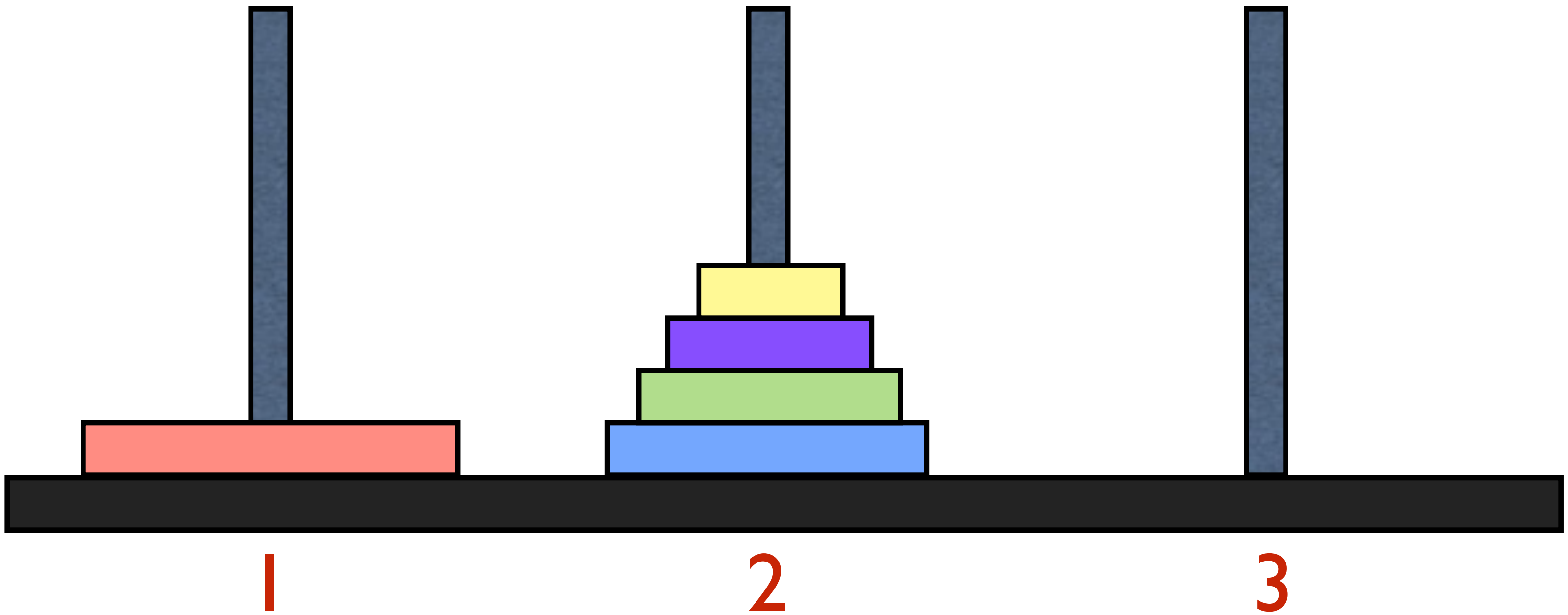
Example: Towers of Hanoi



<u>The power of recursion:</u> Can assume we can solve smaller instances of the problem for free.

- Move N-1 rings from Pole 1 to Pole 2.

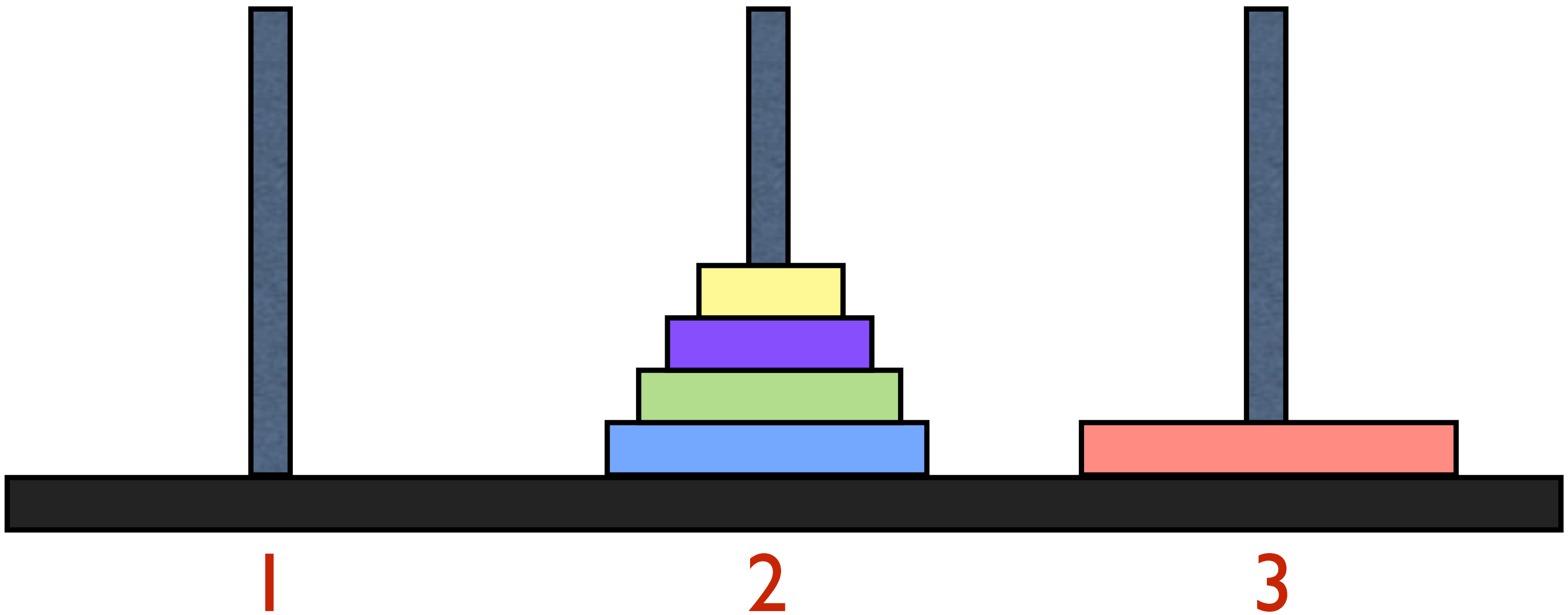
Example: Towers of Hanoi



The power of recursion: Can assume we can solve smaller instances of the problem for free.

- Move N-1 rings from Pole 1 to Pole 2.
- Move ring from Pole 1 to Pole 3.

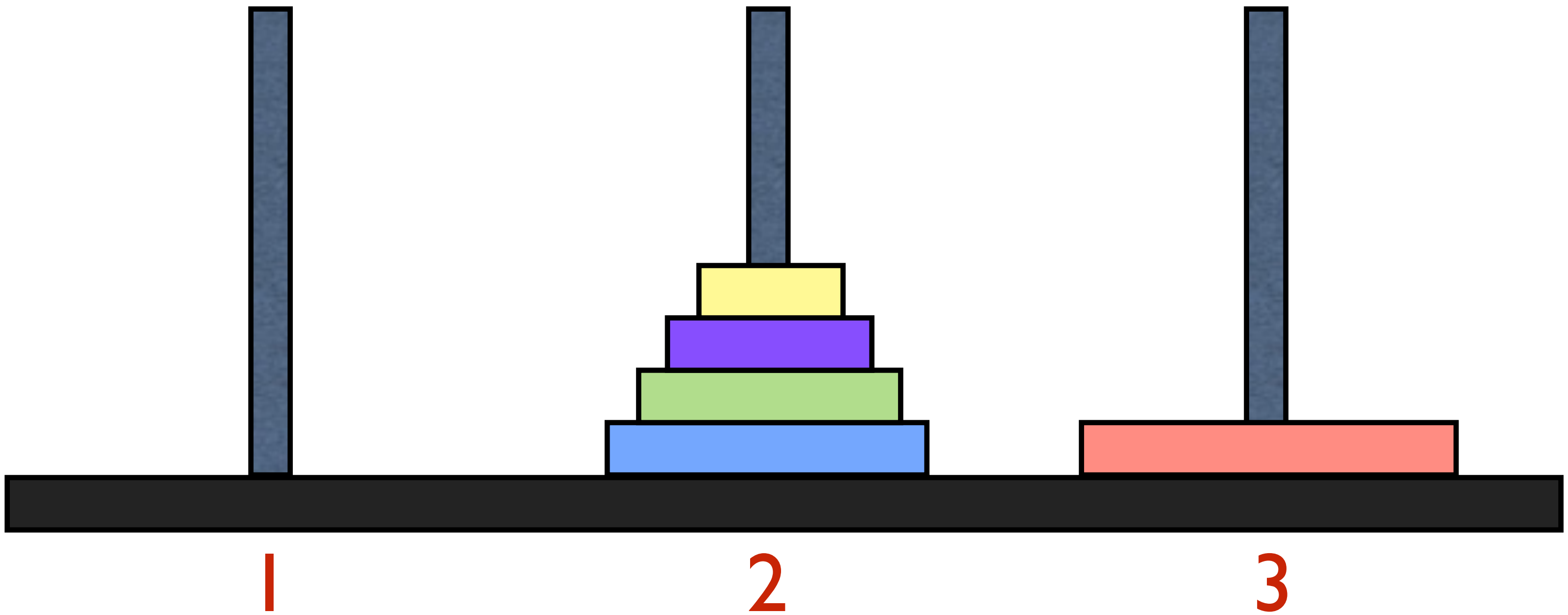
Example: Towers of Hanoi



The power of recursion: Can assume we can solve smaller instances of the problem for free.

- Move N-1 rings from Pole 1 to Pole 2.
- Move ring from Pole 1 to Pole 3.

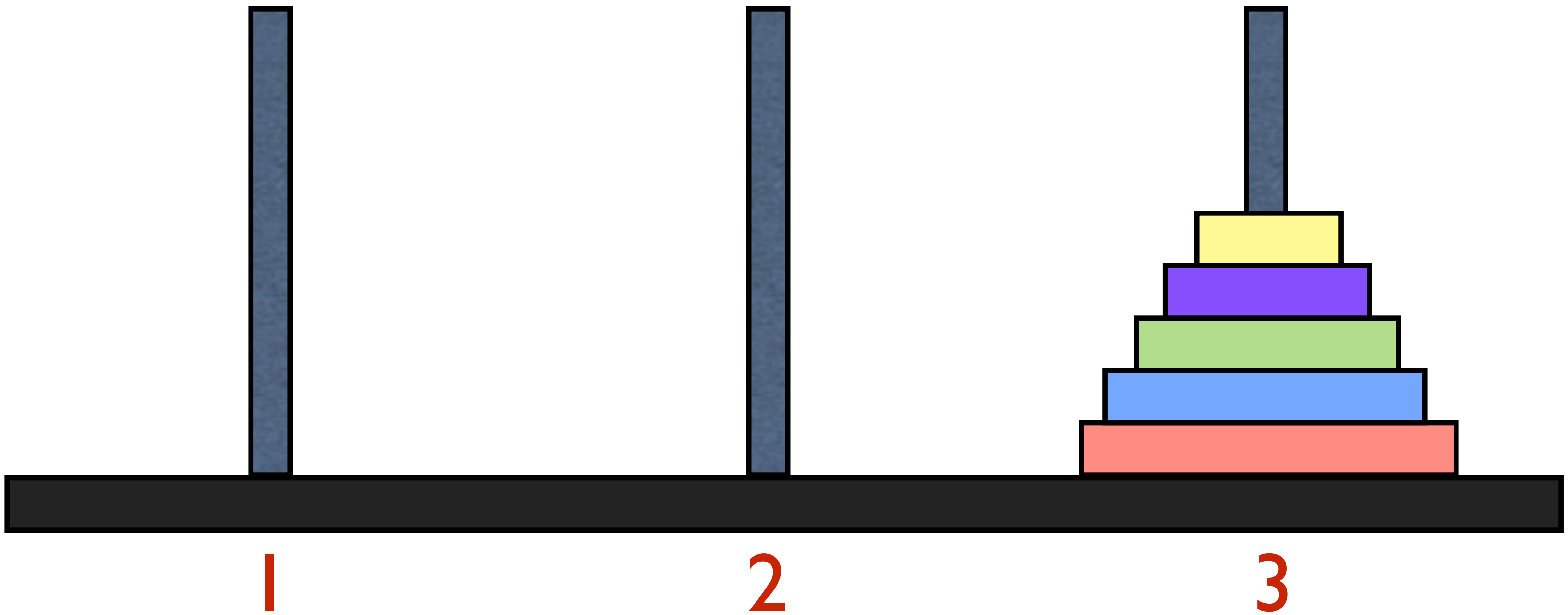
Example: Towers of Hanoi



The power of recursion: Can assume we can solve smaller instances of the problem for free.

- Move N-1 rings from Pole 1 to Pole 2.
- Move ring from Pole 1 to Pole 3.
- Move N-1 rings from Pole 2 to Pole 3.

Example: Towers of Hanoi



The power of recursion: Can assume we can solve smaller instances of the problem for free.

- Move N-1 rings from Pole 1 to Pole 2.
- Move ring from Pole 1 to Pole 3.
- Move N-1 rings from Pole 2 to Pole 3.

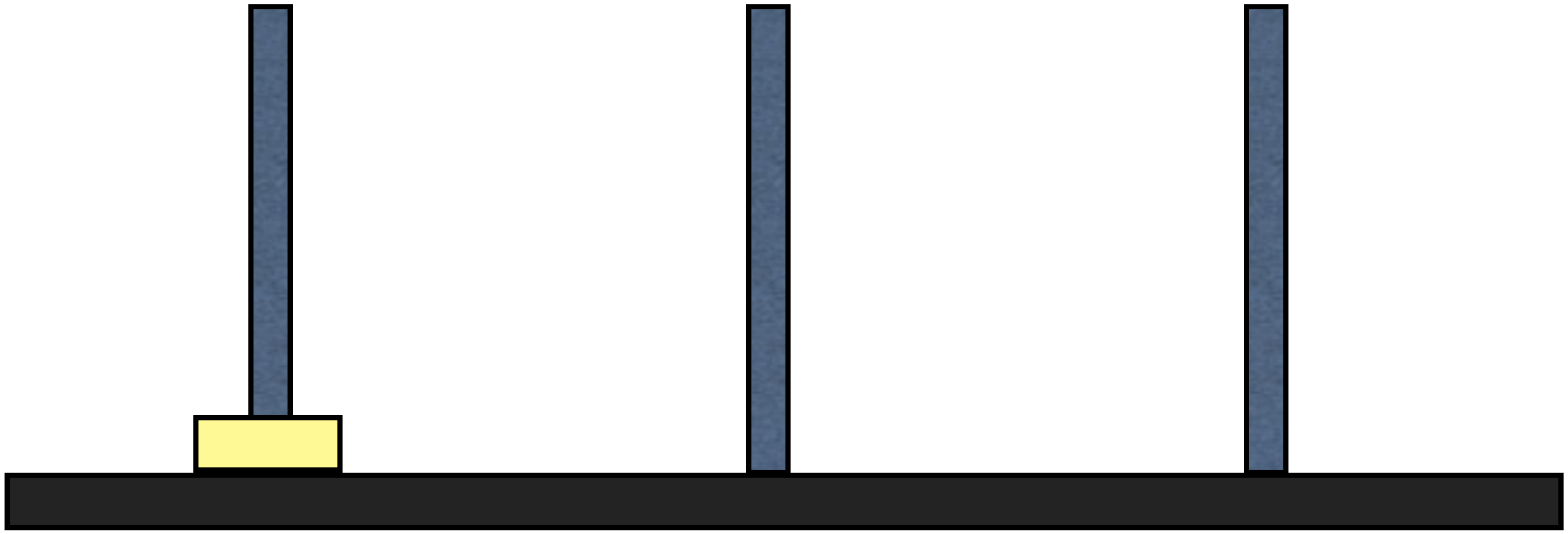
Example: Towers of Hanoi

```
move (N, source, destination):
    if(N > 0):
        Let temp be the index of other pole.
        move(N-1, source, temp)
        print ("Move ring from Pole " + source +
                " to Pole " + destination)
        move(N-1, temp, destination)
```

Challenge: Write the same program using loops

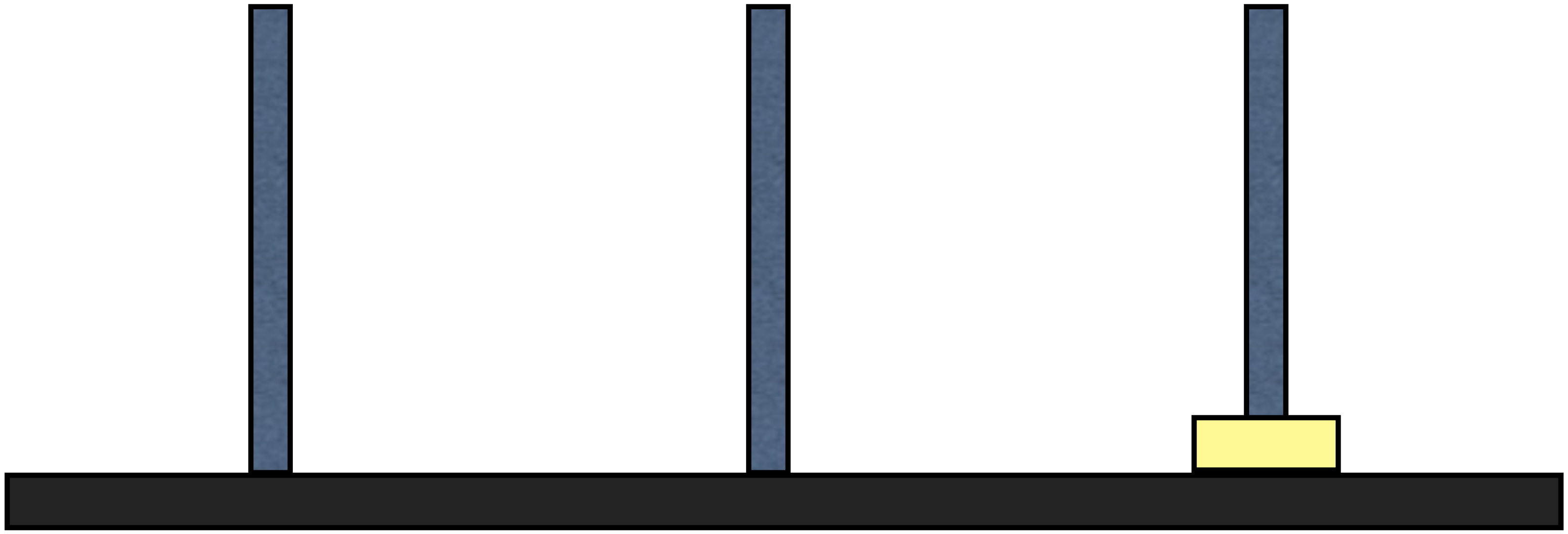
How/Why it works

```
move (N, source, dest):
    if(N > 0):
        Let temp be the index of other pole.
        move(N-1, source, temp)
        print ("Move ring from pole " + source + " to pole " + dest)
        move(N-1, temp, destination)
```



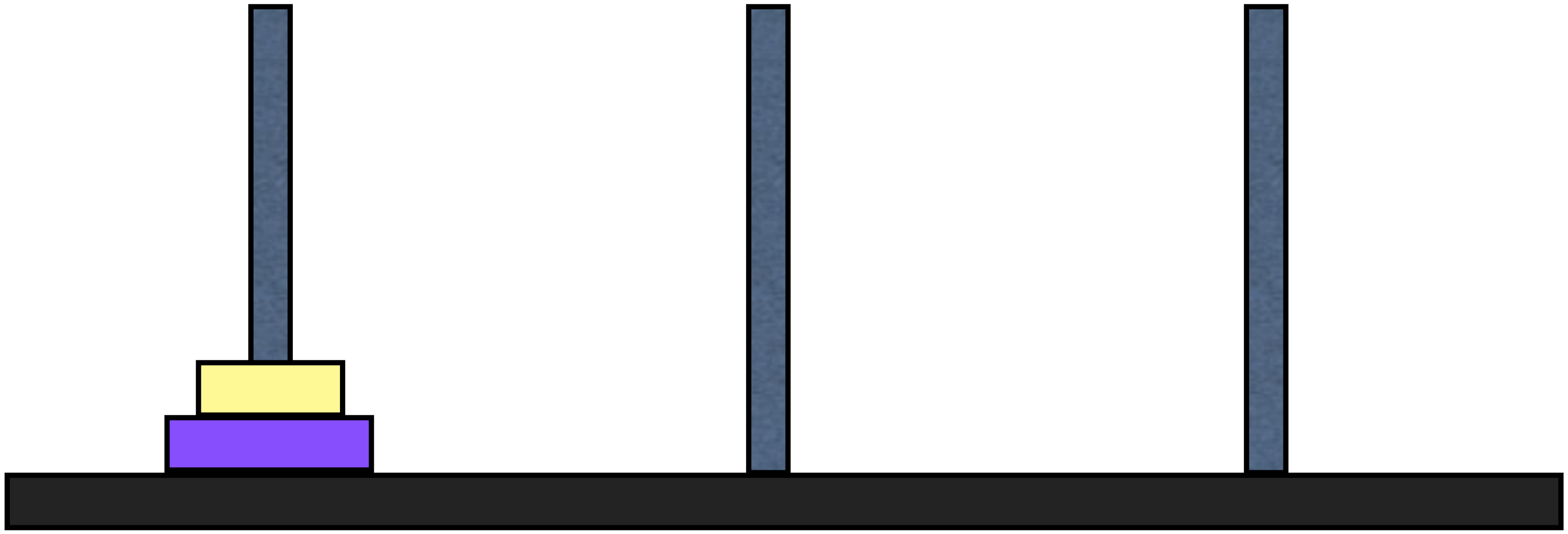
How/Why it works

```
move (N, source, dest):
    if(N > 0):
        Let temp be the index of other pole.
        move(N-1, source, temp)
        print ("Move ring from pole " + source + " to pole " + dest)
        move(N-1, temp, destination)
```



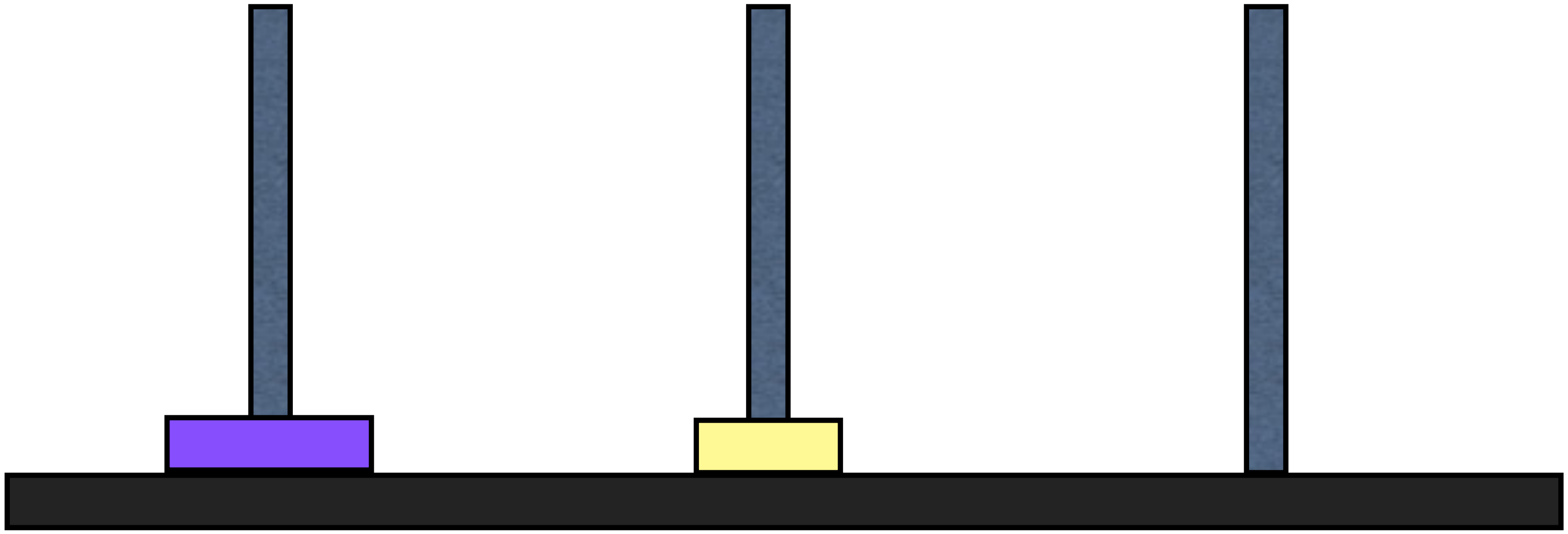
How/Why it works

```
move (N, source, dest):
  if(N > 0):
    Let temp be the index of other pole.
    move(N-1, source, temp)
    print ("Move ring from pole " + source + " to pole " + dest)
    move(N-1, temp, destination)
```



How/Why it works

```
move (N, source, dest):
  if(N > 0):
    Let temp be the index of other pole.
    move(N-1, source, temp)
    print ("Move ring from pole " + source + " to pole " + dest)
    move(N-1, temp, destination)
```



How/Why it works

move (N, source, dest):

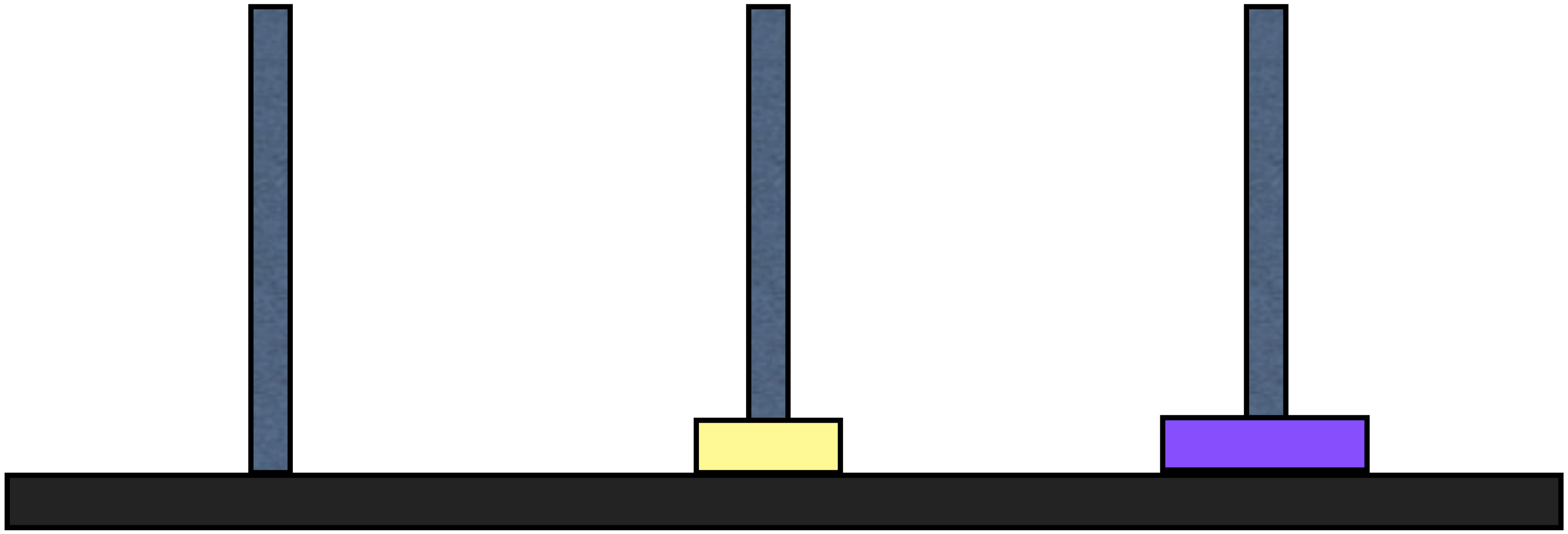
    if(N > 0):

        Let temp be the index of other pole.

        move(N-1, source, temp)

        print ("Move ring from pole " + source + " to pole " + dest)

        move(N-1, temp, destination)



How/Why it works

move (N, source, dest):

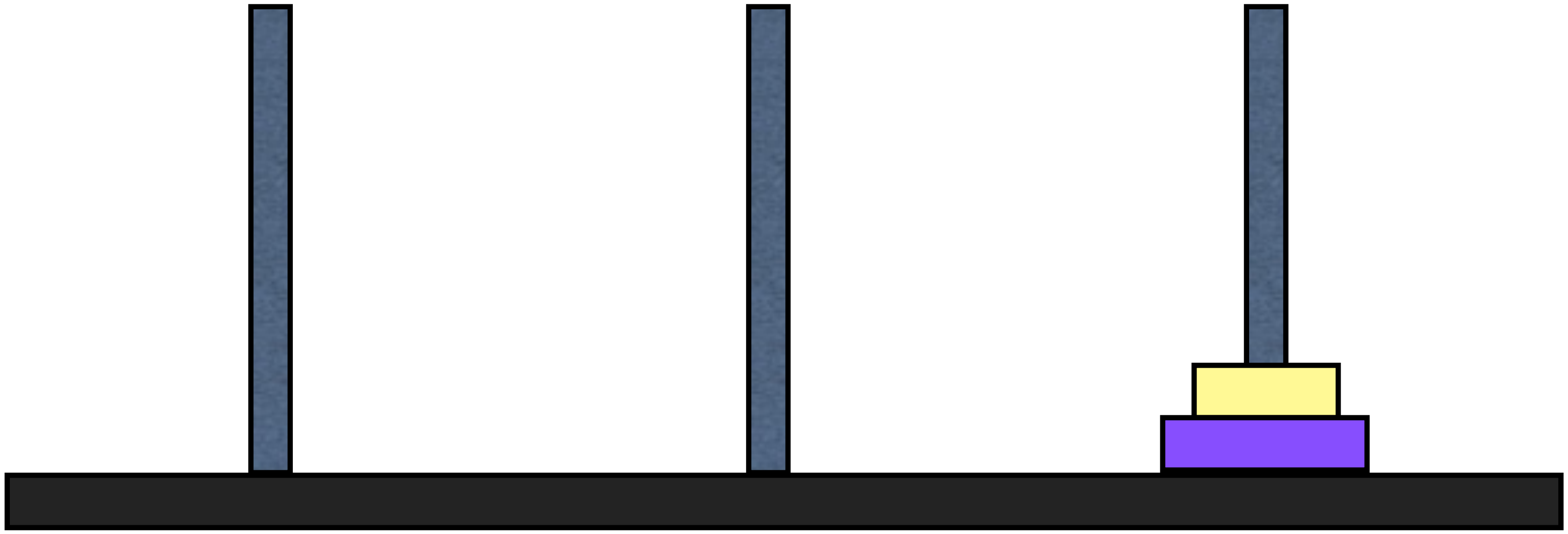
if(N > 0):

Let temp be the index of other pole.

move(N-1, source, temp)

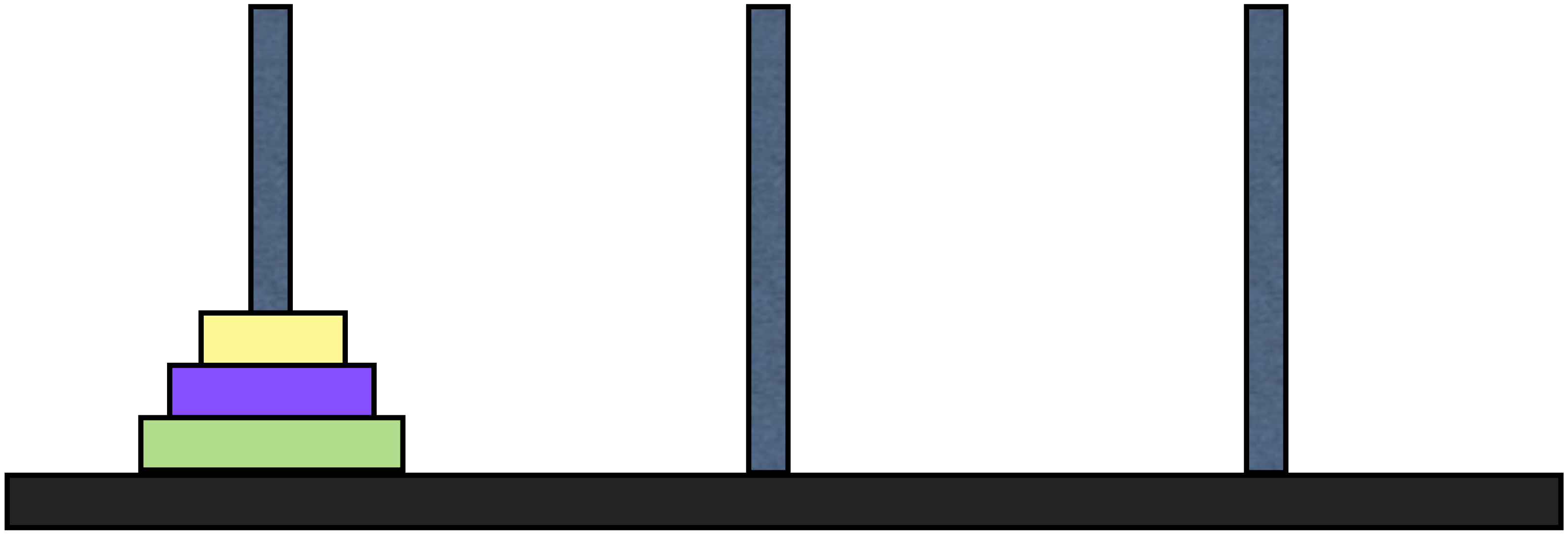
print ("Move ring from pole " + source + " to pole " + dest)

move(N-1, temp, destination)



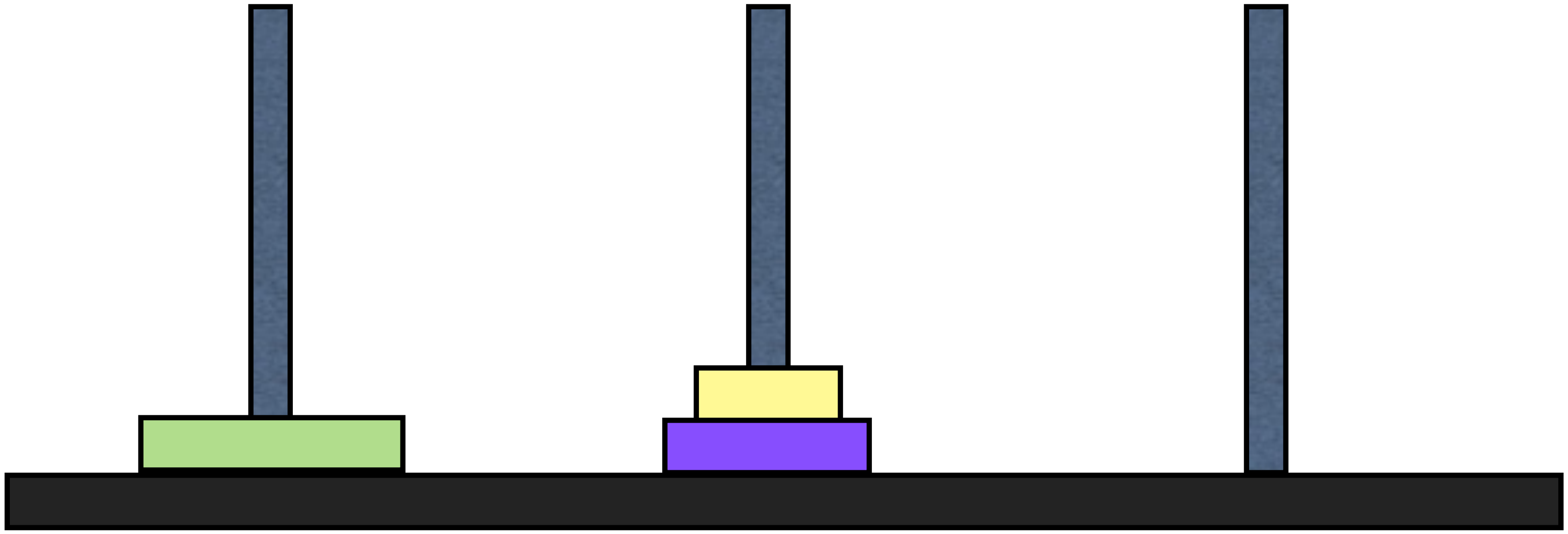
How/Why it works

```
move (N, source, dest):
    if(N > 0):
        Let temp be the index of other pole.
        move(N-1, source, temp)
        print ("Move ring from pole " + source + " to pole " + dest)
        move(N-1, temp, destination)
```



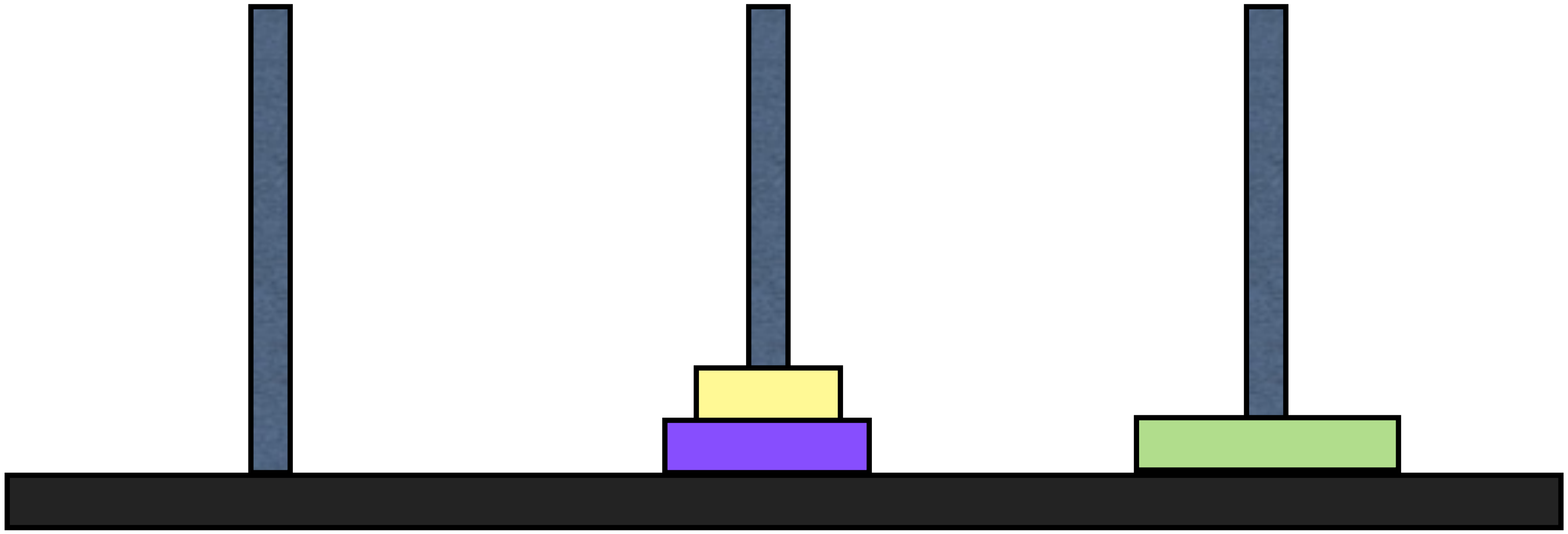
How/Why it works

```
move (N, source, dest):
    if(N > 0):
        Let temp be the index of other pole.
        move(N-1, source, temp)
        print ("Move ring from pole " + source + " to pole " + dest)
        move(N-1, temp, destination)
```



How/Why it works

```
move (N, source, dest):
    if(N > 0):
        Let temp be the index of other pole.
        move(N-1, source, temp)
        print ("Move ring from pole " + source + " to pole " + dest)
        move(N-1, temp, destination)
```



How/Why it works

move (N, source, dest):

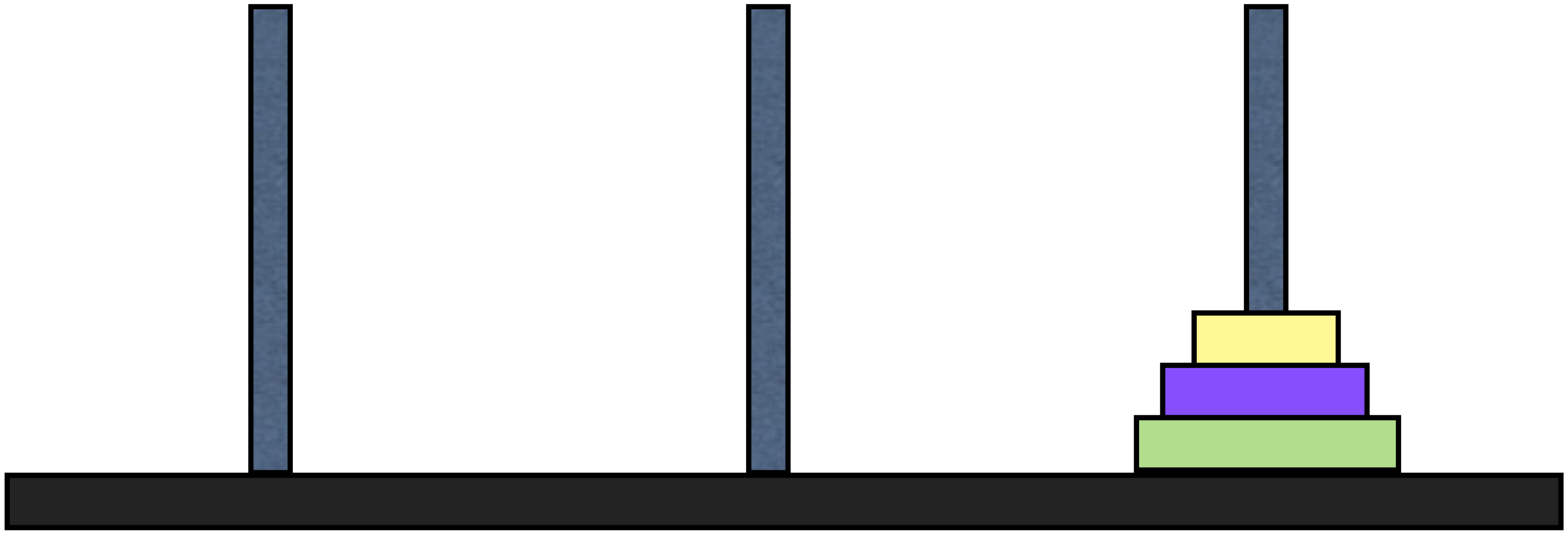
if(N > 0):

Let temp be the index of other pole.

move(N-1, source, temp)

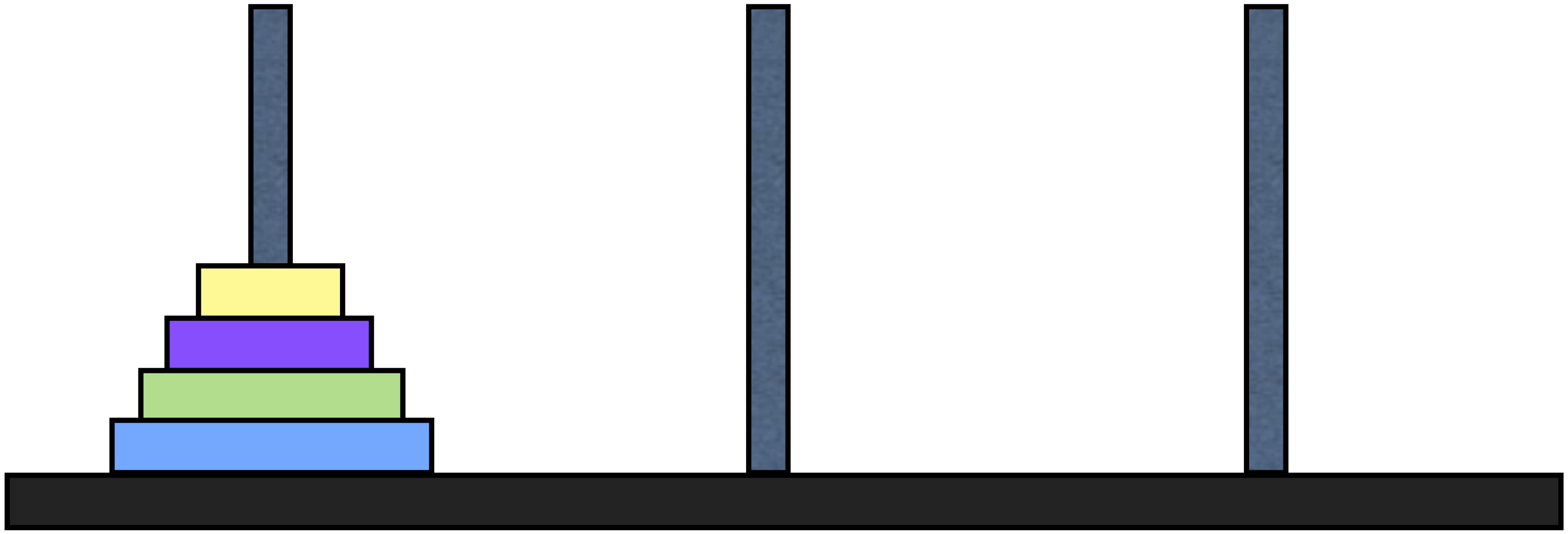
print ("Move ring from pole " + source + " to pole " + dest)

move(N-1, temp, destination)



How/Why it works

```
move (N, source, dest):
  if(N > 0):
    Let temp be the index of other pole.
    move(N-1, source, temp)
    print ("Move ring from pole " + source +  " to pole " + dest)
    move(N-1, temp, destination)
```



How/Why it works

move (N, source, dest):

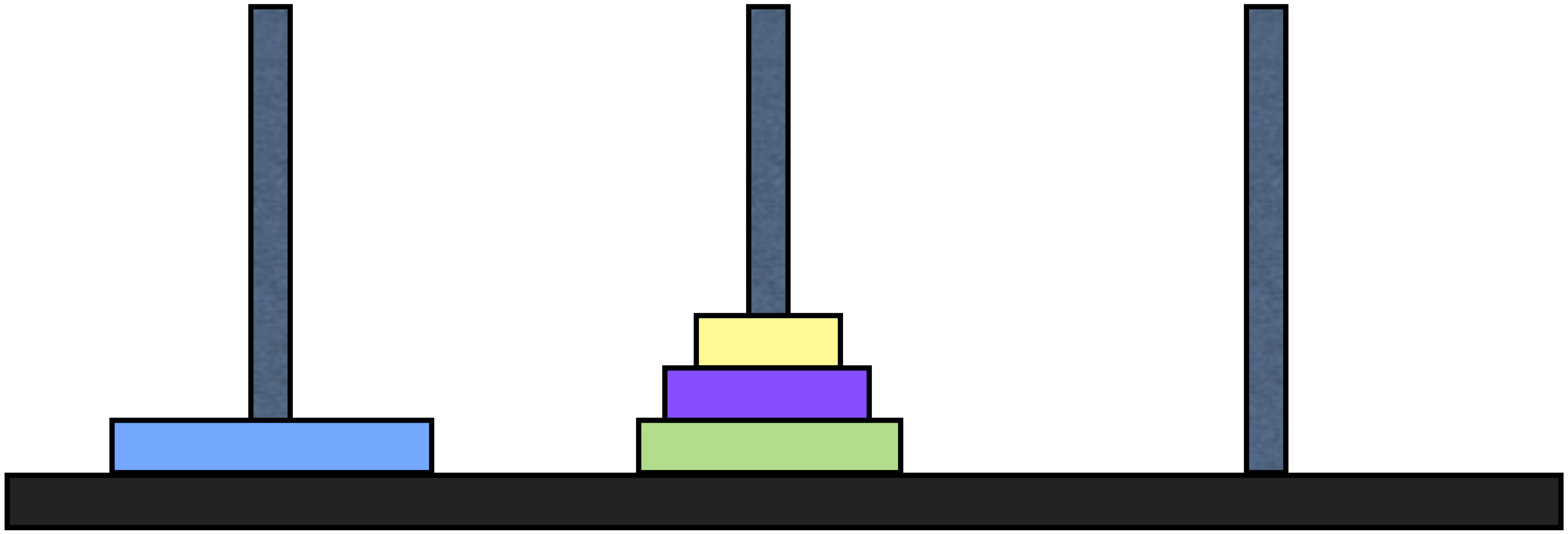
if(N > 0):

Let temp be the index of other pole.

move(N-1, source, temp)

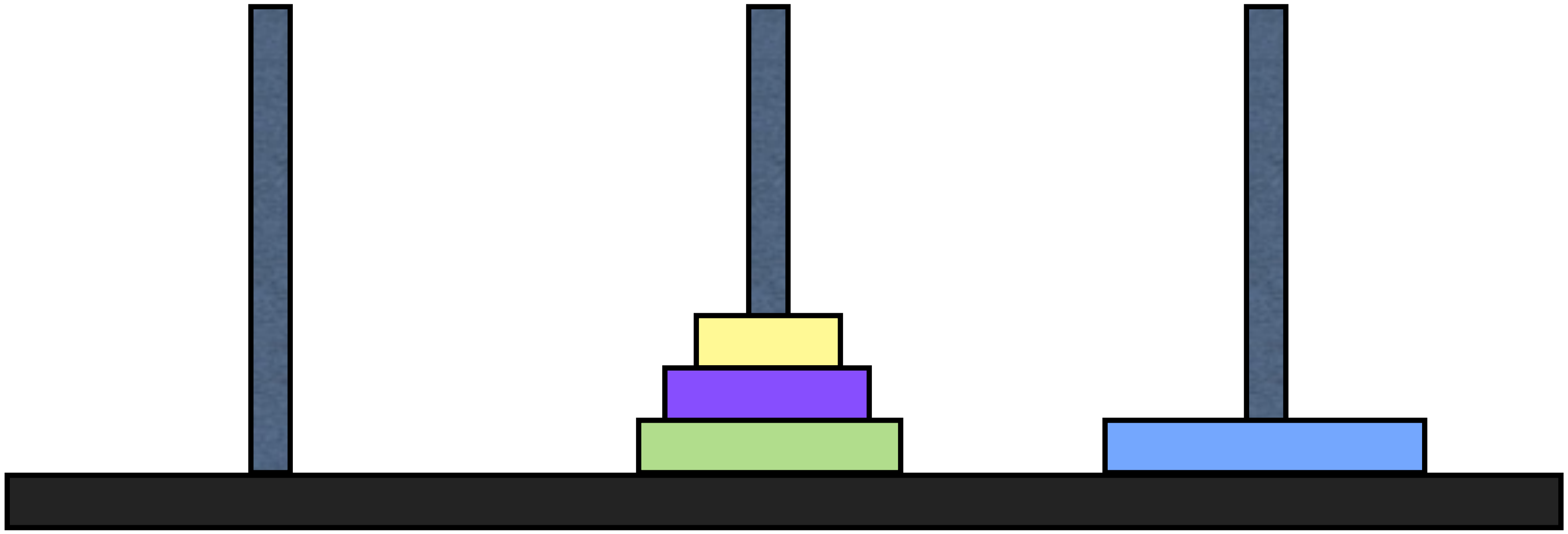
print ("Move ring from pole " + source + " to pole " + dest)

move(N-1, temp, destination)



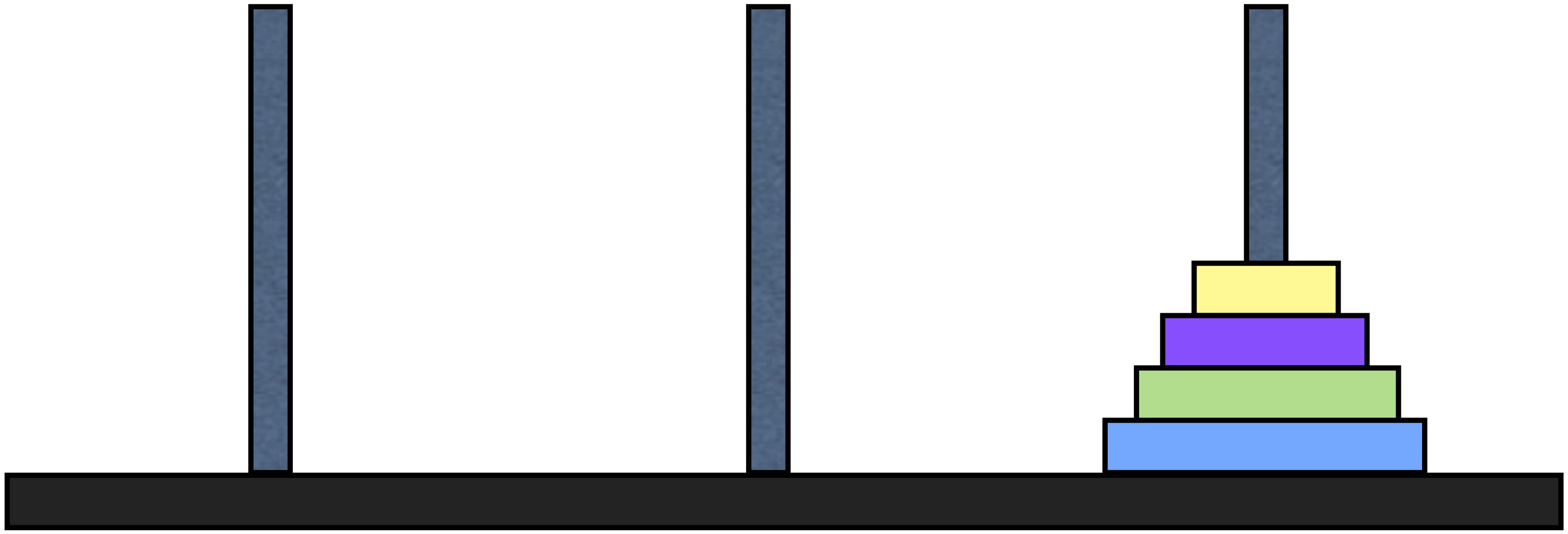
How/Why it works

```
move (N, source, dest):
  if(N > 0):
    Let temp be the index of other pole.
    move(N-1, source, temp)
    print ("Move ring from pole " + source + " to pole " + dest)
    move(N-1, temp, destination)
```

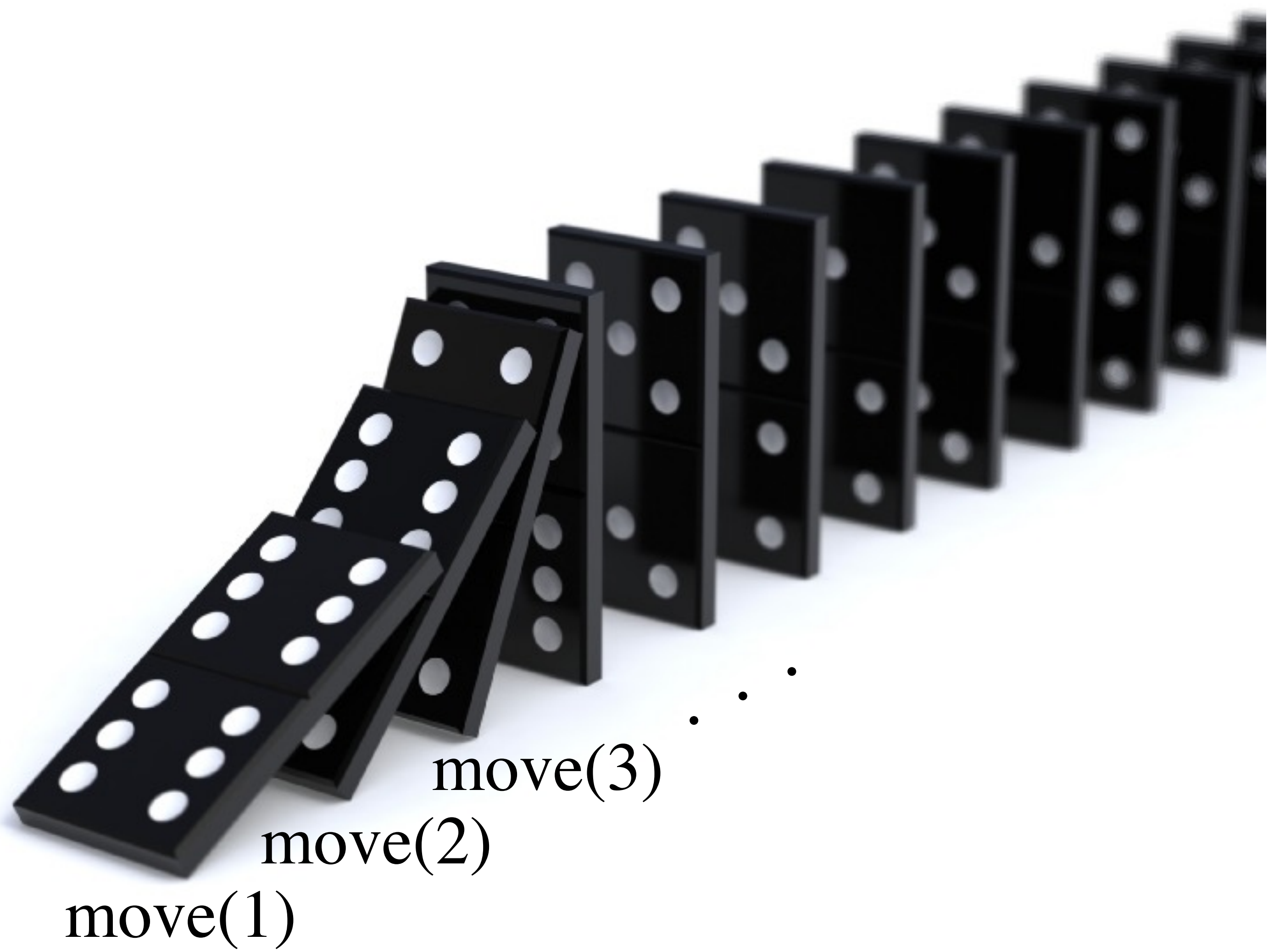


How/Why it works

```
move (N, source, dest):
    if(N > 0):
        Let temp be the index of other pole.
        move(N-1, source, temp)
        print ("Move ring from pole " + source + " to pole " + dest)
        move(N-1, temp, destination)
```



How/Why it works



Getting comfortable with recursion

1. See **<u>lot's</u>** of examples

2. Practice yourself

Getting comfortable with recursion

2. Practice yourself