15-112
Fundamentals of Programming

Week 9 - Lecture 2:
More Recursion and OOP Examples

March 17, 2016

More OOP

1. Creating our own data type

Step 1: Defining the properties/fields

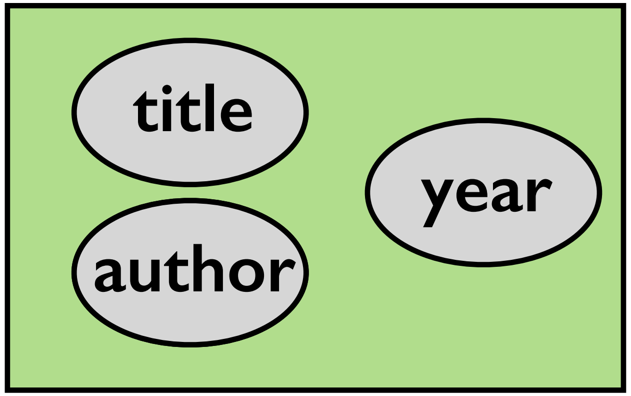
Step 2: Adding methods to our data type

2. OOP paradigm

Defining a data type (class) called Book

```
class Book(object):
    def __init__(self):
        self.title = None
        self.author = None
        self.year = None
```

Book class



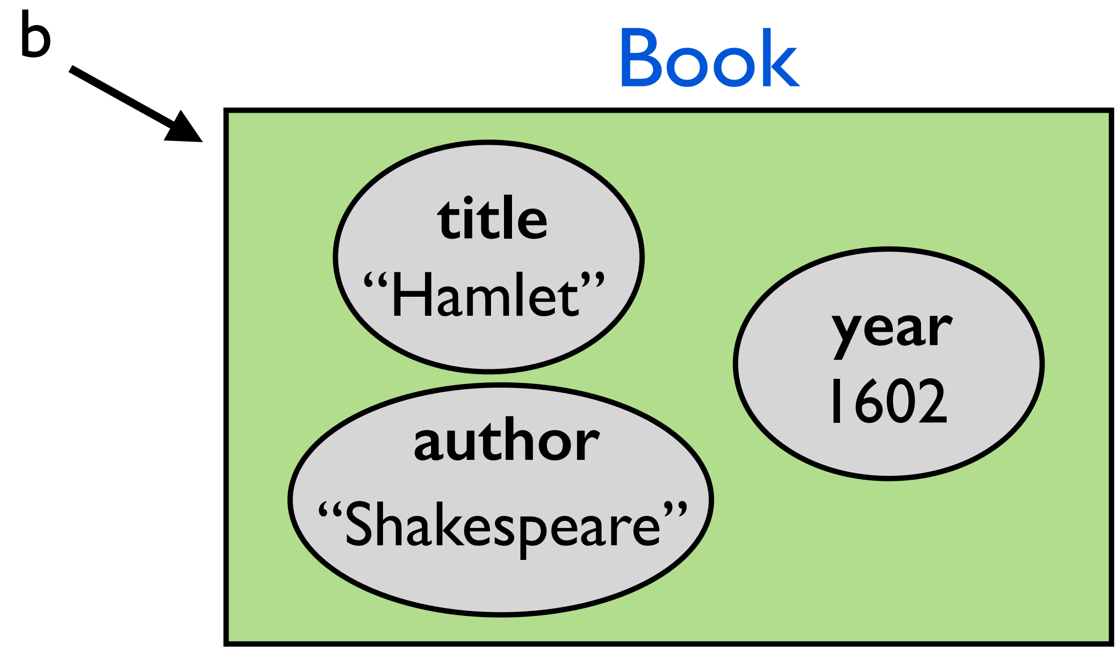
```
b = Book()
b.title = “Hamlet”
b.author = “Shakespeare”
b.year = 1602
```

Compare to:

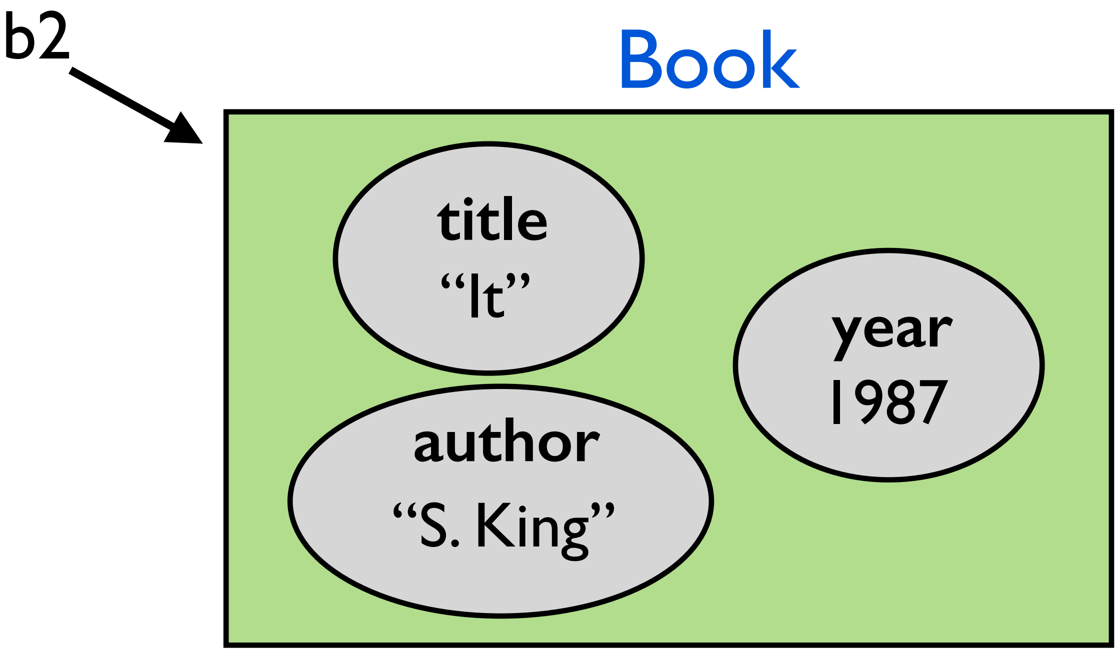
```
b = dict()
b[“title”] = “Hamlet”
b[“author”] = “Shakespeare”
b[“year”] = 1602
```

Creating 2 books

```
b = Book()
b.title = “Hamlet”
b.author = “Shakespeare”
b.year = 1602
```



```
b2 = Book()
b2.title = “It”
b2.author = “S. King”
b2.year = 1987
```



Other Examples

Imagine you have a website that allows users to sign-up.

You want to keep track of the users.

```
class User(object):
    def __init__(self, username, email, password):
        self.username = username
        self.email = email
        self.password = password
```

Other Examples

```
class Account(object):
    def __init__(self):
        self.balance = None
        self.numWithdrawals = None
        self.isRich = False
```

Account is the *type*.

```
a1 = Account()
a1.balance = 1000000
a1.isRich = True

a2 = Account()
a2.balance = 10
a2.numWithdrawals = 1
```

Creating different *objects*
of the same *type* (Account).

Other Examples

```
class Cat(object):
    def __init__(self, name, age, isFriendly):
        self.name = None
        self.age = None
        self.isFriendly = None
```

Cat is the *type*.

```
c1 = Cat(“Tobias”, 6, False)

c2 = Cat(“Frisky”, 1, True)
```

Creating different *objects* of the same *type* (Cat).

Other Examples

```
class Rectangle(object):
    def __init__(self, x, y, width, height):
        self.x = x
        self.y = y
        self.width = width
        self.height = height
```

Rectangle is the *type*.

```
r1 = Rectangle(0, 0, 4, 5)

r2 = Rectangle(1, -1, 2, 1)
```

Creating different *objects* of the same *type* (Rectangle).

Other Examples

```
class Aircraft(object):
    def __init__(self):
        self.numPassengers = None
        self.cruiseSpeed = None
        self.fuelCapacity = None
        self.fuelBurnRate = None
```

Aircraft is the *type*.

```
a1 = Aircraft()
a1.numPassengers = 305
…

a2 = Aircraft()
…
```

Creating different *objects* of the same *type* (Aircraft).

Other Examples

```
class Time(object):
    def __init__(self, hour, minute, second):
        self.hour = hour
        self.minute = minute
        self.second = second
```

Time is the *type*.

```
t1 = Time(15, 50, 21)
…

t2 = Aircraft(11, 15, 0)
…
```

Creating different *objects* of the same *type* (Time).

By the way, what is a Struct?

Basically a typeless container of variables (no methods).

1. **Creating our own data type**

    Step 1: Defining the properties/fields

    Step 2: Adding methods to our data type

2. **OOP paradigm**

Giving more power to our data type

Method: A function built into the data type.

It acts on the properties/fields/data members.

Usually 2 types:

1. Methods that return a value related to the fields. (reads and returns information)

2. Methods that modify the fields.

Example 1: Rectangle

```
class Rectangle(object):
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def getArea(self):
        return self.width*self.height
```

read/return data

```
    def getPerimeter(self):
        return 2*(self.width + self.height)
```

read/return data

```
    def doubleDimensions(self):
        self.width *= 2
        self.height *= 2
```

modify data

```
    def rotate90Degrees(self):
        (self.width, self.height) = (self.height, self.width)
```

modify data

Example 1: Rectangle

```
class Rectangle(object):
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def getArea(self):
        return self.width*self.height

    def getPerimeter(self):
        return 2*(self.width + self.height)

    def doubleDimensions(self):
        self.width *= 2
        self.height *= 2

    def rotate90Degrees(self):
        (self.width, self.height) = (self.height, self.width)
```

```
r = Rectangle(3, 5)
print (“The width is ”, r.width)
print (“The area is ”, r.getArea())
print (“The perimeter is ”, r.getPerimeter())
r.doubleDimensions()
print (“The perimeter is ”, r.getPerimeter())
```

Example 2: Employee

```
class Employee(object):
    def __init__(self, name, salary):
        self.name = name
        self.salary = salary

    def printEmployee(self):
        print (“Name: ”, self.name)
        print (“Salary: ”, self.salary)

    def getNetSalary(self):
        return 0.75*self.salary

    def isRich(self):
        return (self.salary > 100000)

    def salaryInFuture(self, years):
        return self.salary * 1.03**years

    def fire(self):
        self.salary = 0
```

Example 2: Employee

```
class Employee(object):
    def __init__(self, name, salary):
        self.name = name
        self.salary = salary

    def printEmployee(self):
        print (“Name: ”, self.name)
        print (“Salary: ”, self.salary)

    def getNetSalary(self):
        return 0.75*self.salary

    def isRich(self):
        return (self.salary > 100000)

    def salaryInFuture(self, years):
        return self.salary * 1.03**years

    def fire(self):
        self.salary = 0
```

```
e1 = Employee(“Frank Underwood”, 200000)
e1.printEmployee()
print (e1.isRich())
print (e1.salaryInFuture(10))
print (e1.fire())
print (e1.salary)
```

Example 3: Cat

```
class Cat(object):
    def __init__(self, weight, age, isFriendly):
        self.weight = weight
        self.age = age
        self.isFriendly = isFriendly

    def printInfo(self):
        print (“I weigh ”, self.weight, “kg.”)
        print (“I am ”, self.age, “ years old.”)
        if (self.isFriendly):
            print (“I am the nicest cat in the world.”)
        else:
            print (“One more step and I will attack!!!”)

    …
```

Example 3: Cat

```
…

def feed(self, food):
        self.weight += food
        print (“It was not Fancy Feast’s seafood”)
        self.wail()

def wail(self):
        print (“Miiiiaaaaawwwww”)
        self.moodSwing()

def moodSwing(self):
        self.isFriendly = (random.randint(0,1) == 0)

…
```

Example 3: Cat

```
frisky = Cat(4.2, 2, True)
tiger = Cat(102, 5, False)

frisky.printInfo()
tiger.printInfo()

frisky.feed(0.2)
tiger.feed(3)

frisky.printInfo()
tiger.printInfo()
```

1. **Creating our own data type**

   Step 1: Defining the properties/fields

   Step 2: Adding methods to our data type

2. **OOP paradigm**

The general idea behind OOP

1. Group together data together with the methods into one unit.

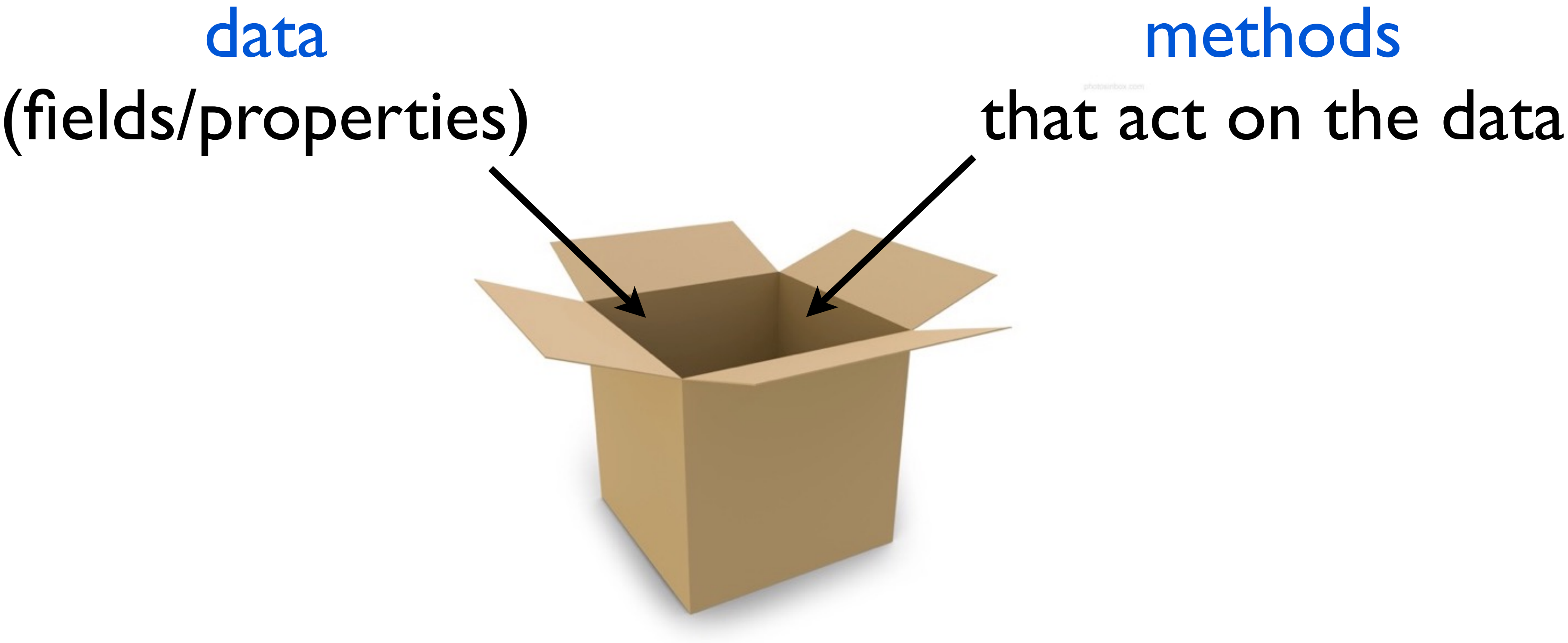
2. Methods represent the interface:

    - control how the object should be used.
    - hide internal complexities.

3. Design programs around objects.

Idea 1: group together data and methods

Encapsulate the data together with the methods that act on them.



data
(fields/properties)

methods
that act on the data

All in one unit

Idea 1 advantages

Adds another layer of organizational structure.

Our data types better correspond to objects in reality.

How we think about our program starts to correspond to how we think about objects in real life.

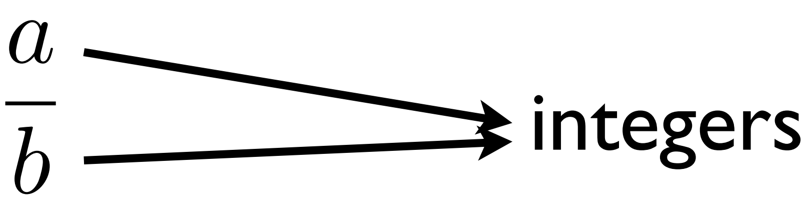
Your new data type is easily shareable.

- everything is in one unit.
- all you need to provide is a documentation.

Example: Representing rational numbers

Rational numbers: a number that can be expressed as a ratio of two integers.

Also called fractions.



$$\frac{a}{b}$$

integers

$a =$ numerator

$b =$ denominator (cannot be 0)

Example: Representing rational numbers

```
class Rational(object):
    def __init__(self, n, d):
        self.numerator = n
        self.denominator = d

    def toString(self):
        return str(self.numerator) + “ / ” + str(self.denominator)

    def toFloat(self):
        return self.numerator / self.denominator

    def simplify(self):
        # code for simplifying

    def add(self, other):
        # code for adding

    def multiply(self, other):
        # code for multiplying
    …
```

Example: Representing rational numbers

Everything you might want to do with rational numbers is packaged up nicely into one unit:

the new data type Rational.

The general idea behind OOP

1. Group together data together with the methods into one unit.

2. Methods represent the interface:

   - control how the object should be used.
   - hide internal complexities.

3. Design programs around objects.

Idea 2: Methods are the interface

Methods should be the only way to read and process the data/fields.

(shouldn’t access data members directly.)

If done right, the hope is that the code is:

- easier to handle/maintain
- easy to fix bugs

Can modify classes independently as long as the interface stays the same.

Expanding the Cat class (1/3)

```
class Cat(object):

    def __init__(self, n, w, a, f):
        self.name = n
        self.weight = w
        self.age = a
        self.isFriendly = f


   ...
```

Could do:

```
c = Cat("tiger", 98, 2, False)
c.weight = -1
```

But this is not processing data through the methods.

Expanding the Cat class (2/3)

```
    …

    def setWeight(self, newWeight):
        if (newWeight > 0):
            self.weight = newWeight

    def getWeight(self):
        return self.weight

    def getAge(self):
        return self.age

    def setAge(self, newAge):
        if(newAge >= 0):
            self.age = newAge

    …
```

```
c = Cat("tiger", 98, 2, False)
c.weight = -1
```

```
c = Cat("tiger", 98, 2, False)
c.setWeight(-1)
```

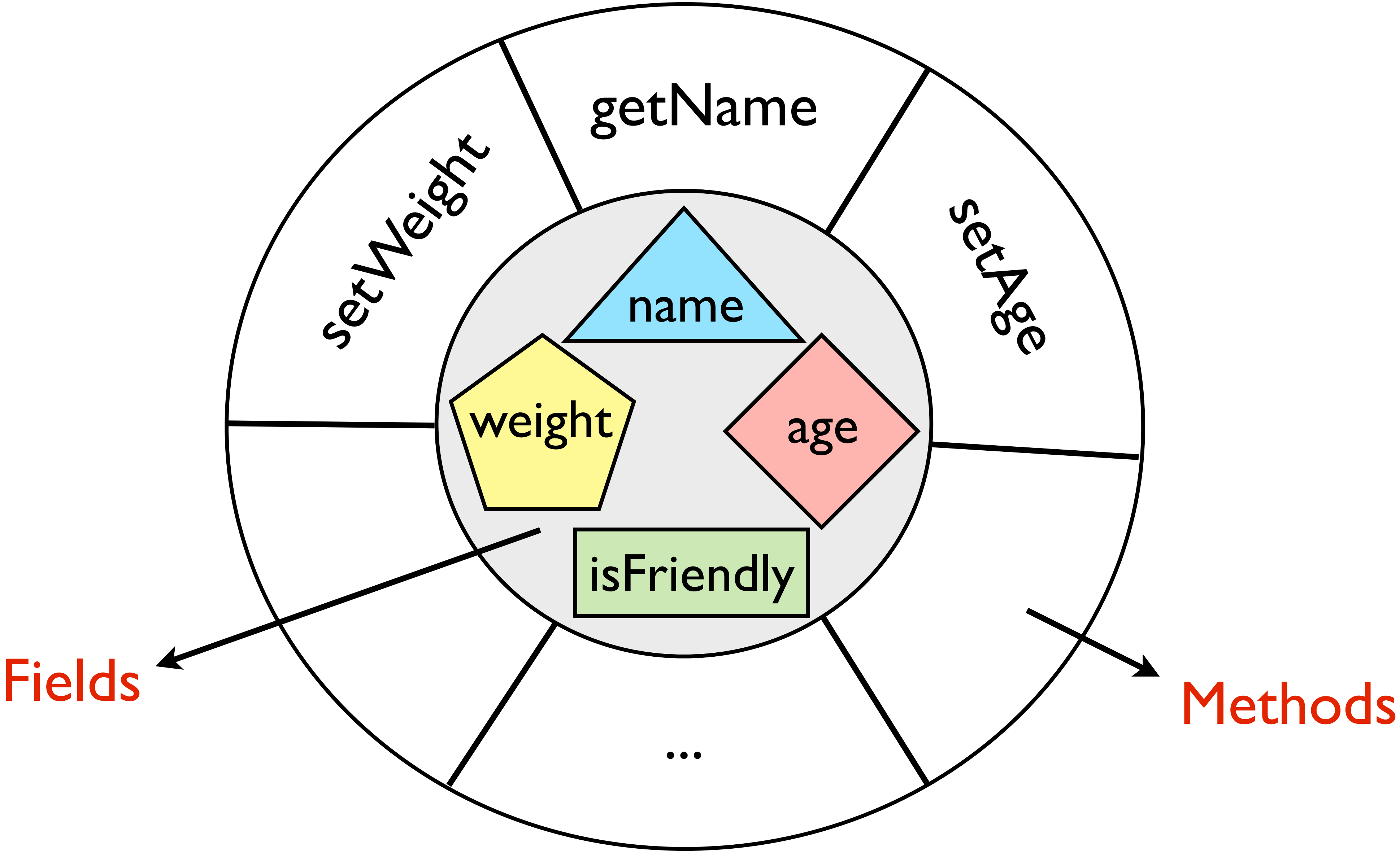
Expanding the Cat class (3/3)

```
...
def getName(self):
    return self.name

def getIsFriendly(self):
    return self.isFriendly

def feed(self, food):
    self.weight += food
    self.isFriendly = (random.randint(0,1) == 0)
```

There are no methods to directly change the name or isFriendly fields.

Idea 2: Methods are the interface

The Cat data type



The general idea behind OOP

1. Group together data together with the methods into one unit.

2. Methods represent the interface:

   - control how the object should be used.
   - hide internal complexities.

3. Design programs around objects.

Idea 3: Objects are at the center

Privilege data over action

Procedural Programming Paradigm

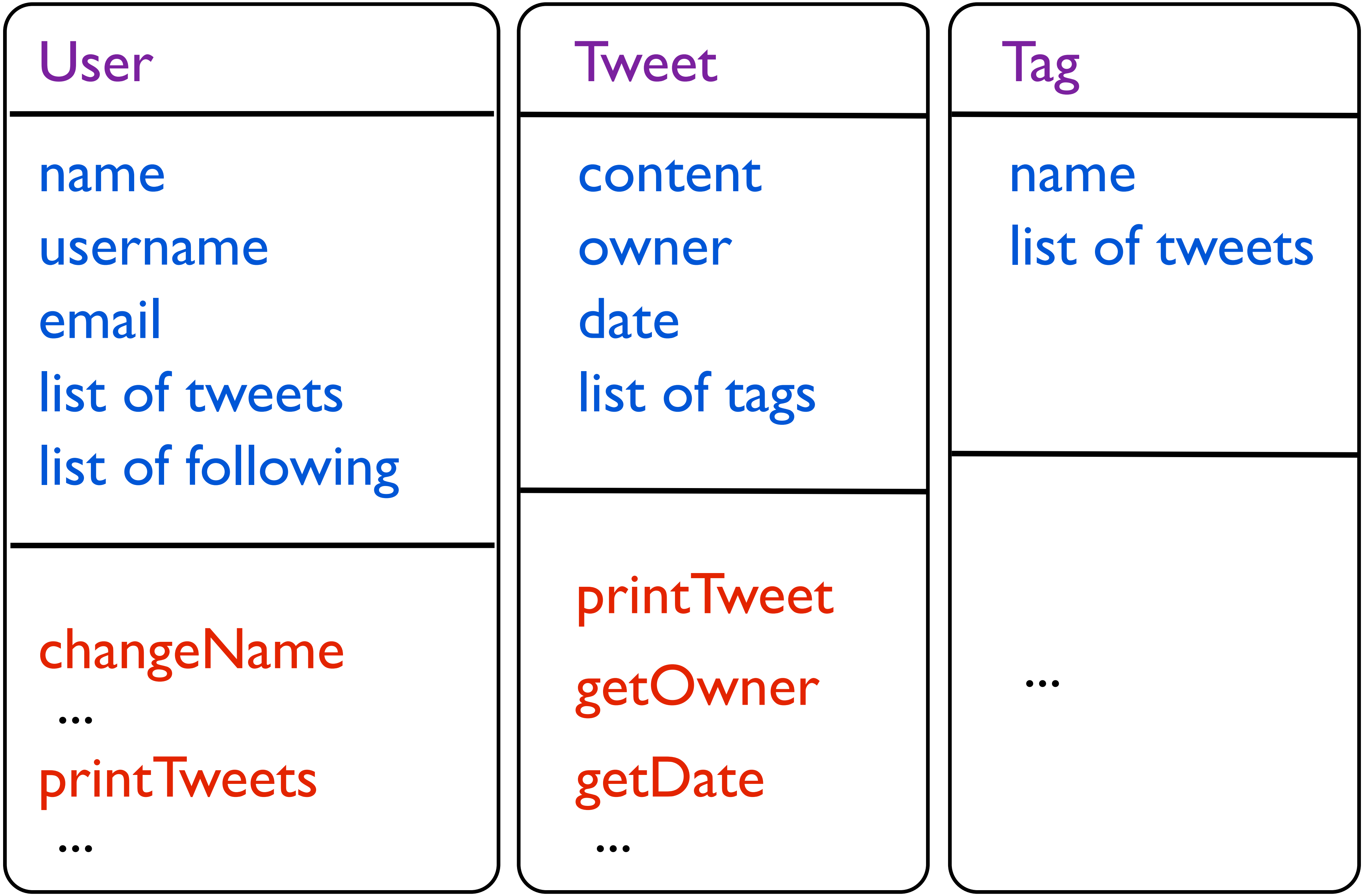
Decompose problem into a series of actions/functions.

Object Oriented Programming Paradigm

Decompose problem first into bunch of data types.

In both, we have actions and data types.
Difference is which one you end up thinking about first.

Simplified Twitter using OOP



User

name
username
email
list of tweets
list of following

changeName
…
printTweets
…

Tweet

content
owner
date
list of tags

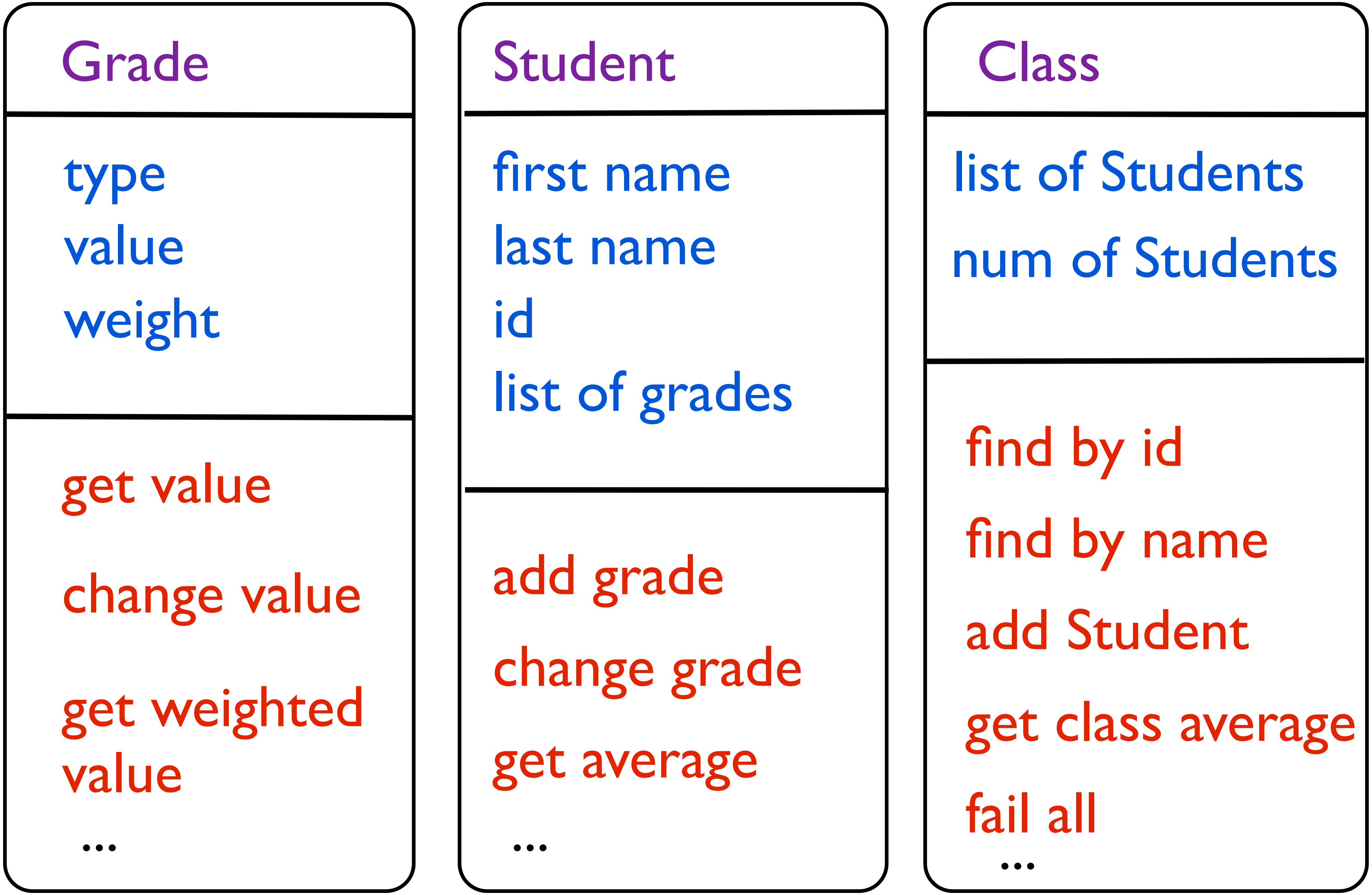
printTweet
getOwner
getDate
…

Tag

name
list of tweets

…

Managing my classes using OOP



Grade

type
value
weight

get value

change value

get weighted value

...

Student

first name
last name
id
list of grades

add grade

change grade

get average

...

Class

list of Students

num of Students

find by id

find by name

add Student

get class average

fail all

...

More Recursion

OOPy recursion example

```
class Person(object):

    def __init__(self, name):
        self.name = name
        self.children = [ ]
        self.isFriendly = True

    def numOfChildren(self):
        …

    def addChild(self, p):
        …

    def printChildren(self):
        …

    def numOfOffspring(self):
        …
```

Example: nthPrime(n)

```
def nthPrime(n):
    if (n == 0): return 2
    m = nthPrime(n-1) + 1
    while(True):
        if (isPrime(m)): return m
        m += 1
```

Can we do it **without** using a loop?

Example: nthPrime(n)

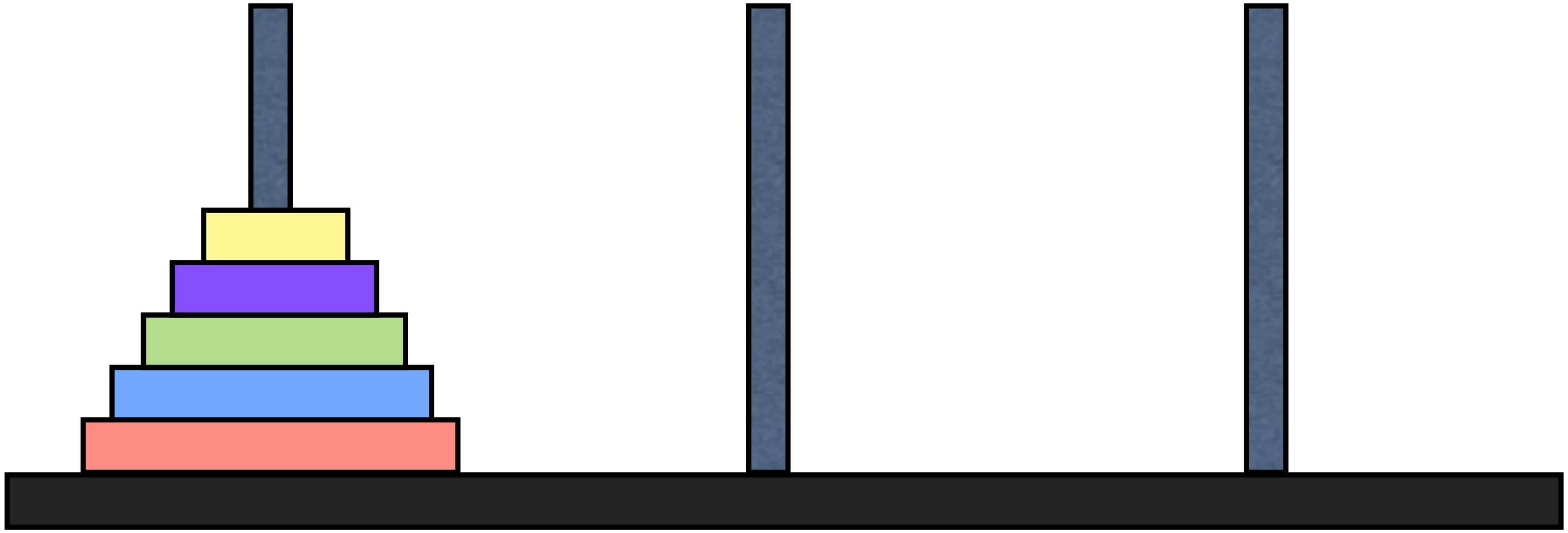
```
def nthPrime(n, start):
    # return the nth prime starting from the integer start
    if (n == 0 and isPrime(start)): return start
    elif (isPrime(start)):
        return nthPrime(n-1, start+1)
    else:
        return nthPrime(n, start+1)

# printing the 10th prime number
print(nthPrime(10, 2))
```

Example: nthPrime(n)

```
def nthPrime(n, start=2):
    # return the nth prime starting from the integer start
    if (n == 0 and isPrime(start)): return start
    elif (isPrime(start)):
        return nthPrime(n-1, start+1)
    else:
        return nthPrime(n, start+1)

# printing the 10th prime number
print(nthPrime(10))
```

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.

Goal: 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

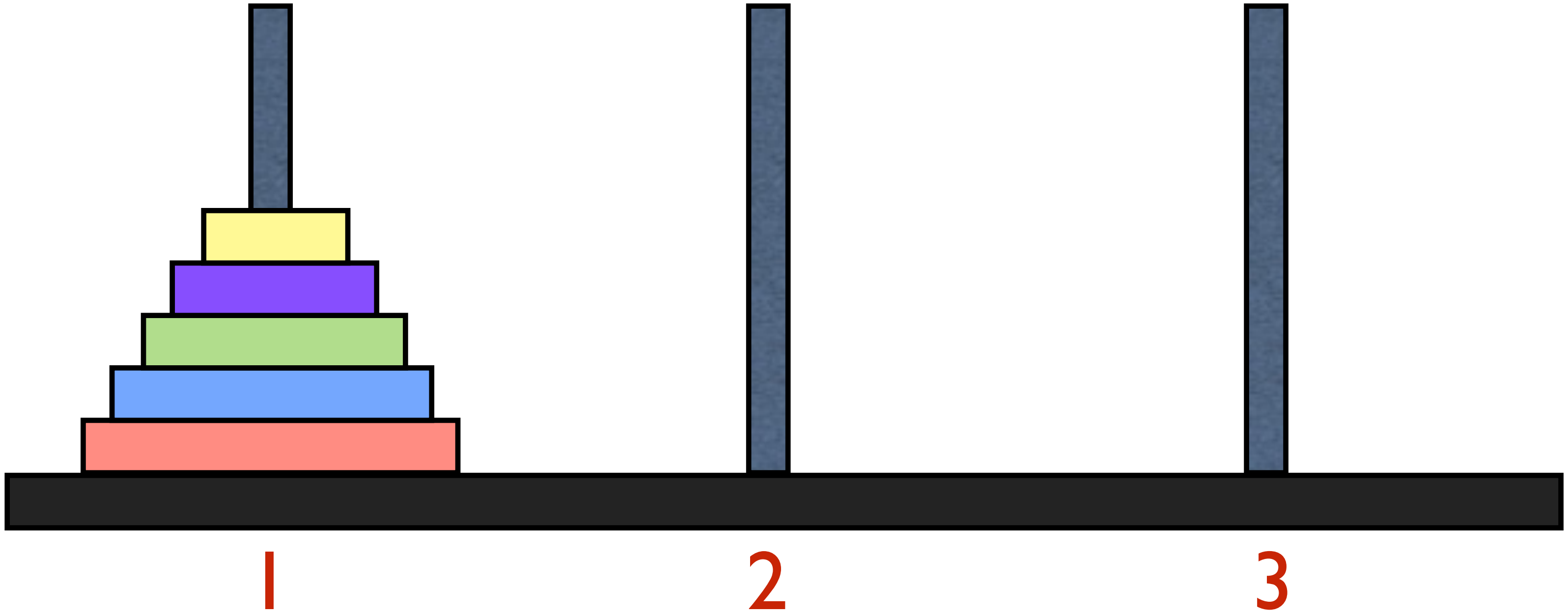
move (N, source, destination)   (integer inputs)

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):

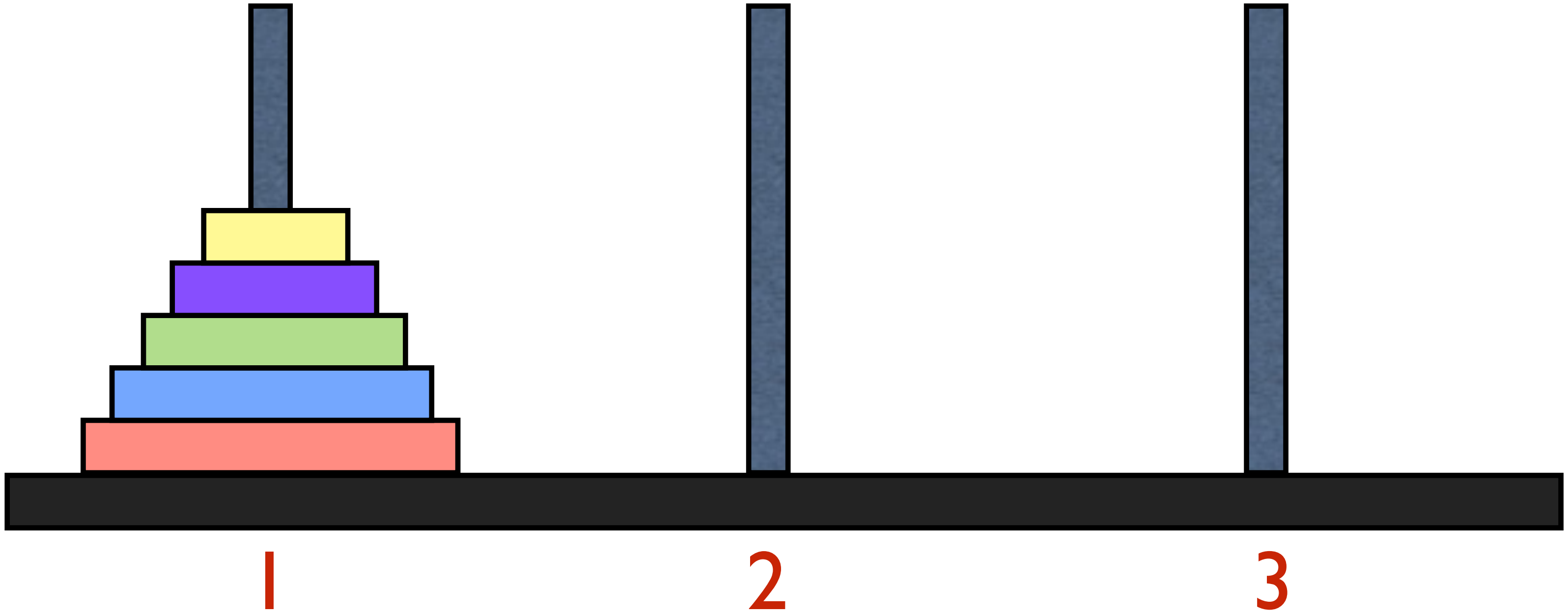
```
Move ring from Pole 1 to Pole 3
Move ring from Pole 1 to Pole 2
Move ring from Pole 3 to Pole 2
Move ring from Pole 1 to Pole 3
Move ring from Pole 2 to Pole 1
Move ring from Pole 2 to Pole 3
Move ring from Pole 1 to Pole 3
```

Example: Towers of Hanoi



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

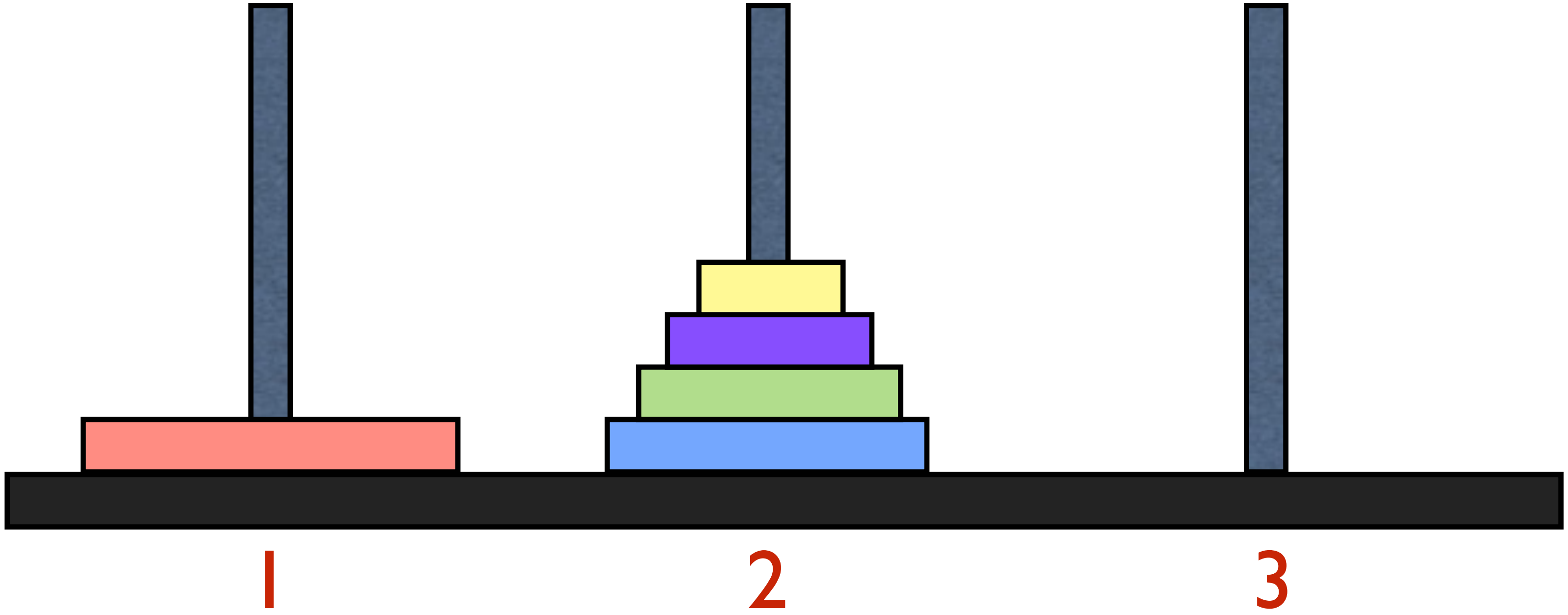
Example: Towers of Hanoi



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.

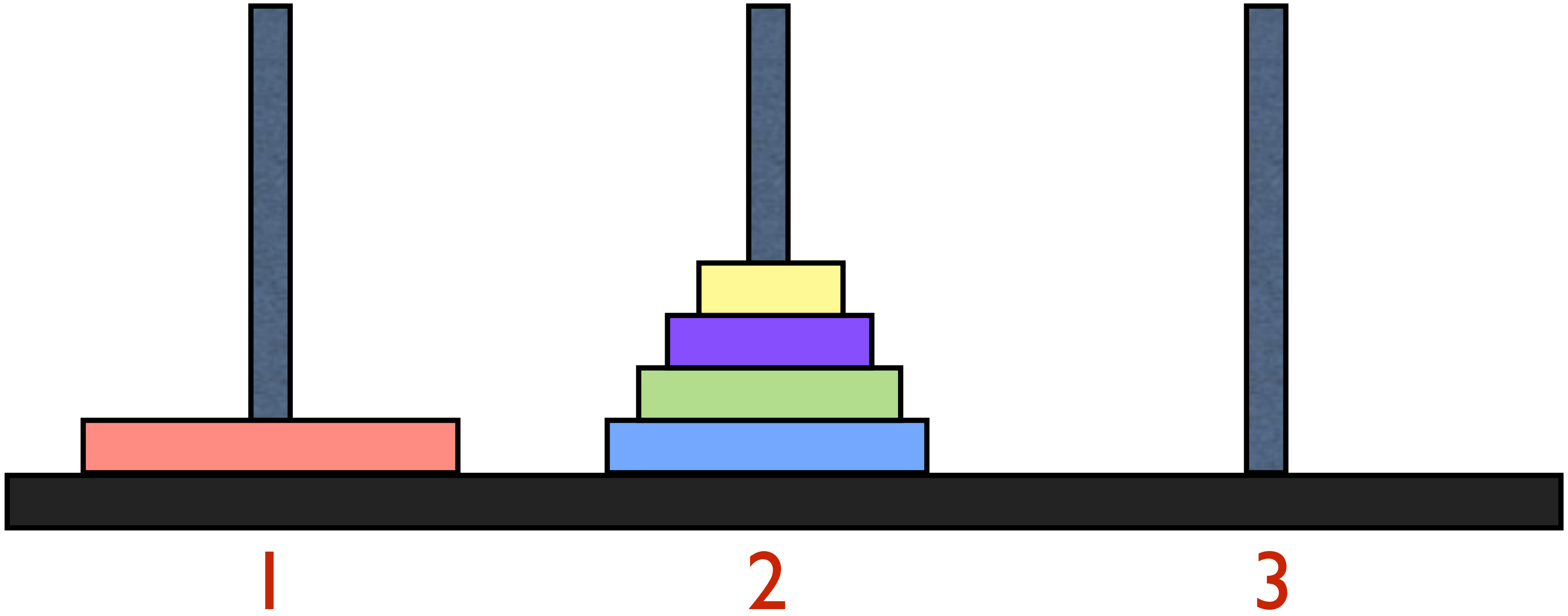
Example: Towers of Hanoi



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.

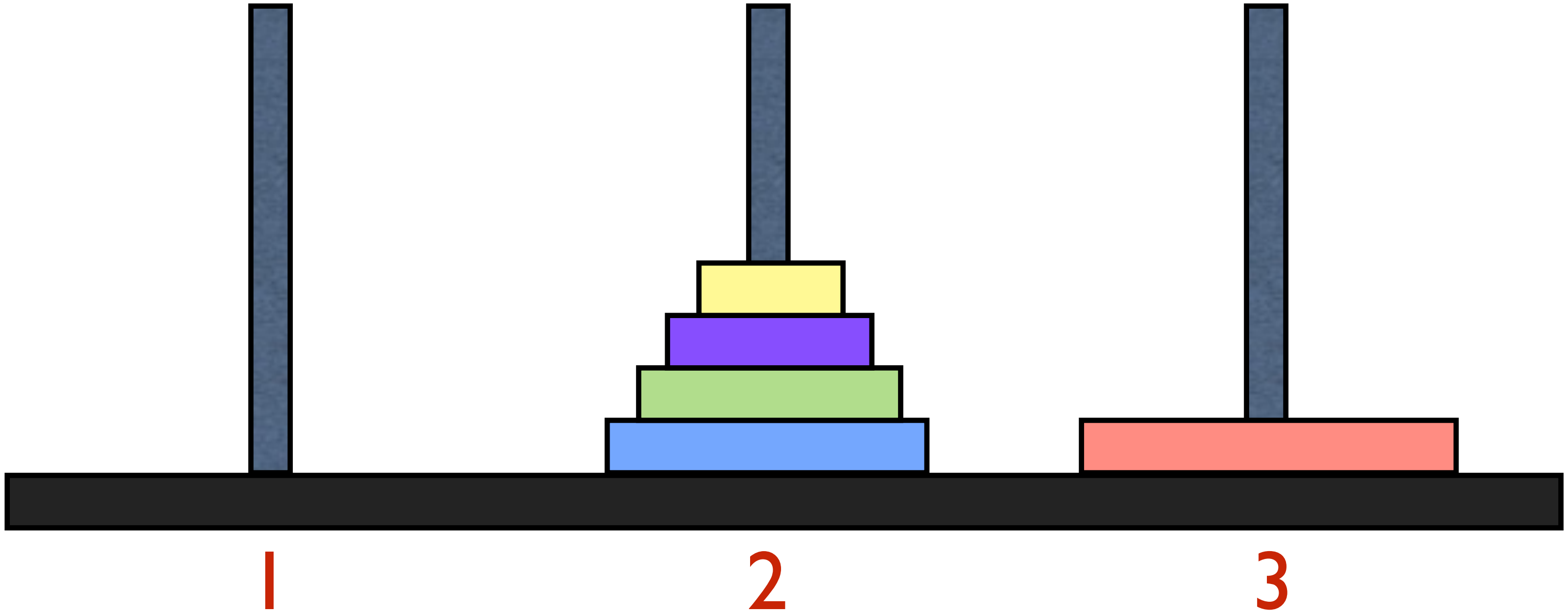
Example: Towers of Hanoi



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.

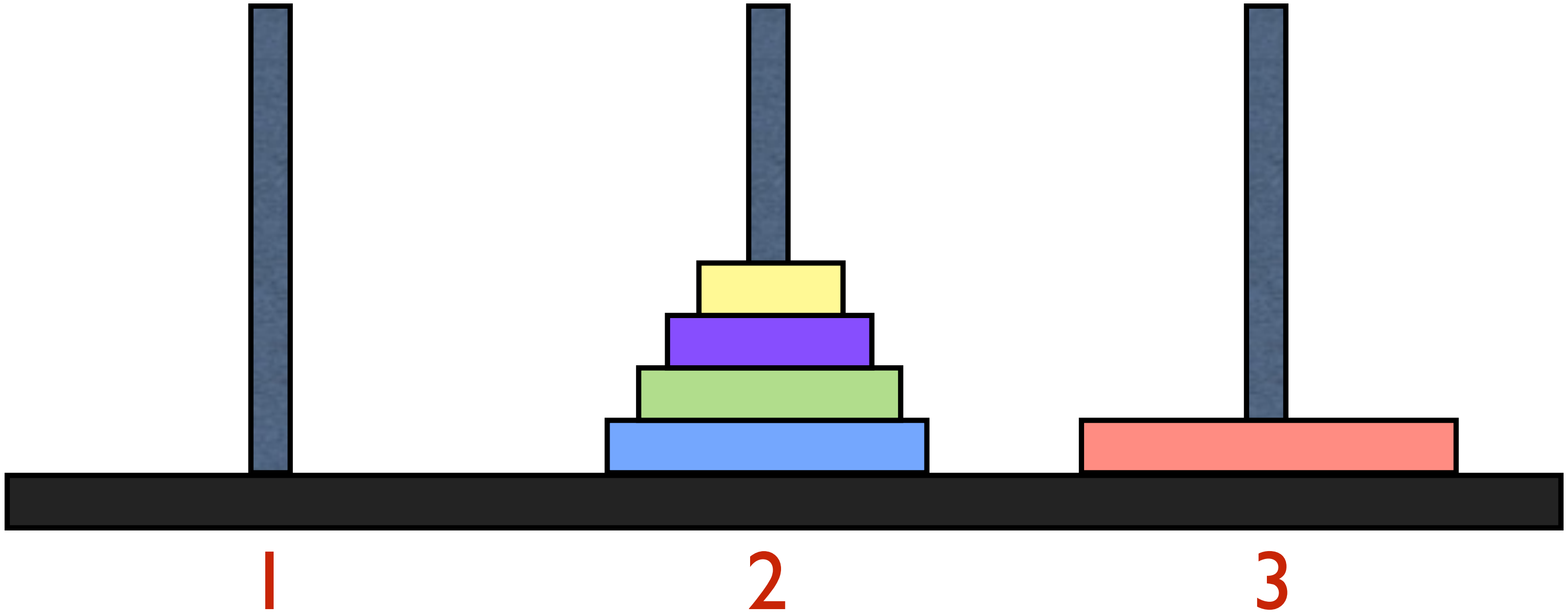
Example: Towers of Hanoi



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.

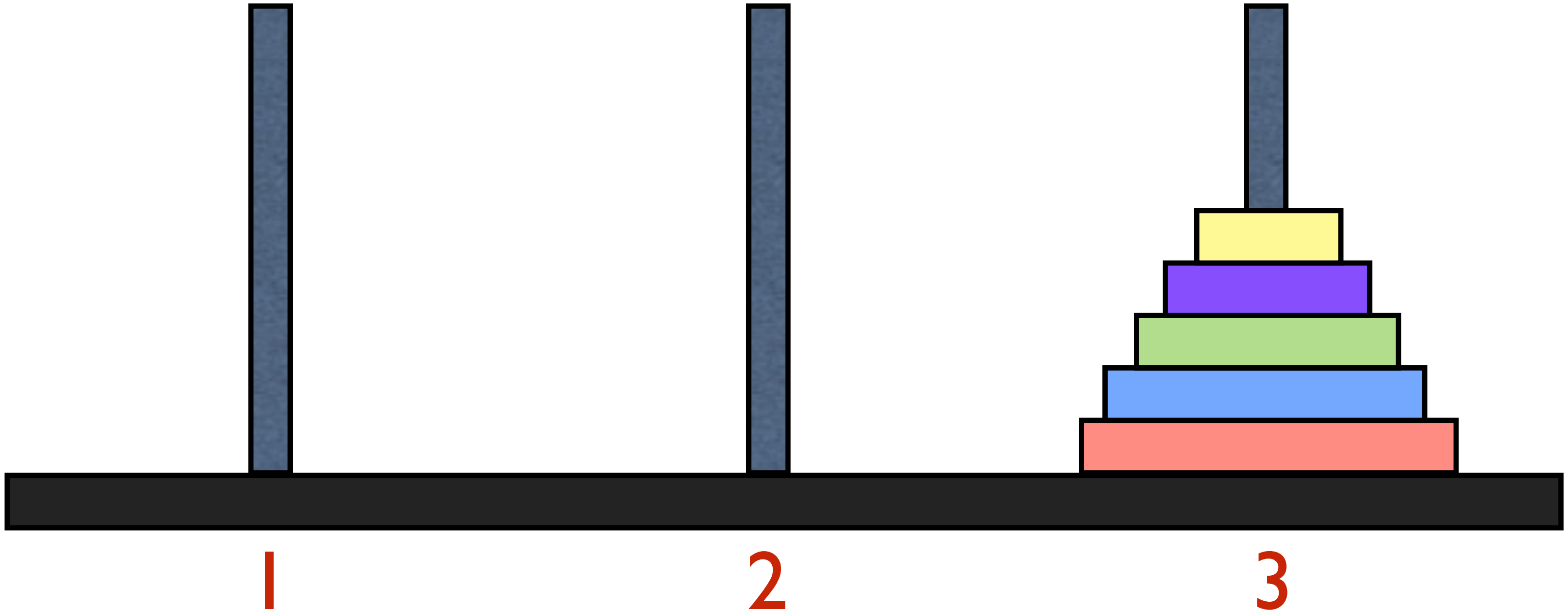
Example: Towers of Hanoi



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.

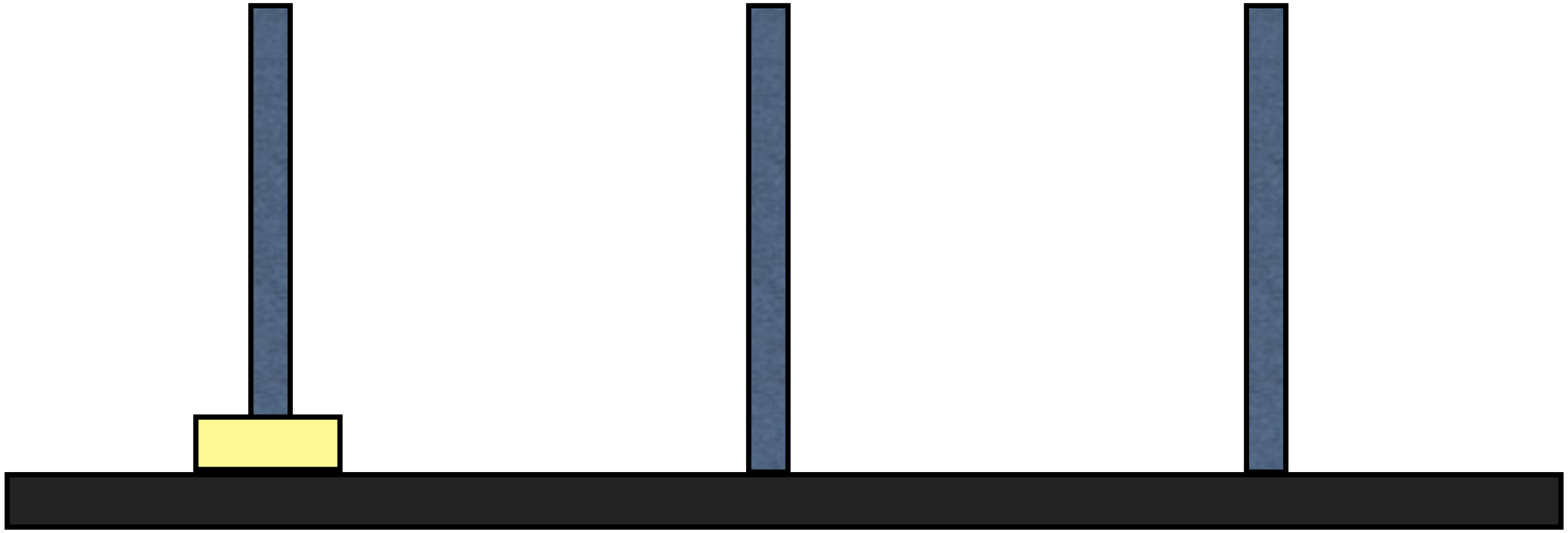
Example: Towers of Hanoi

```
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

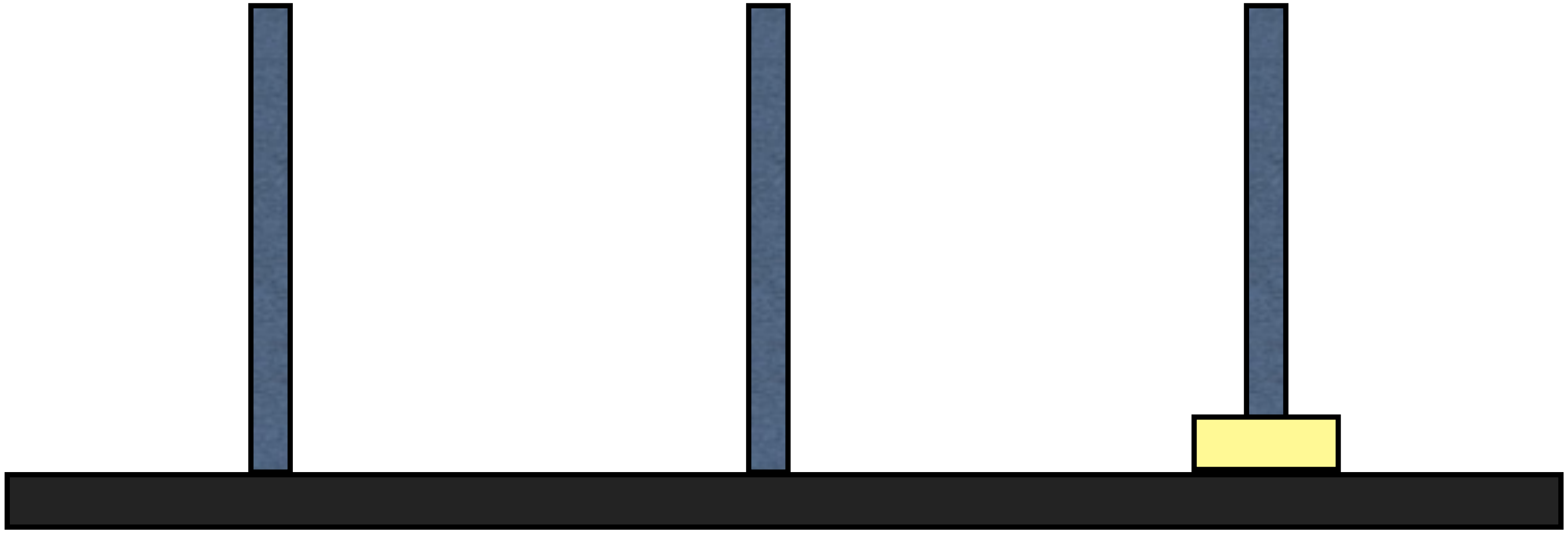
How/Why it works

```
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)
```



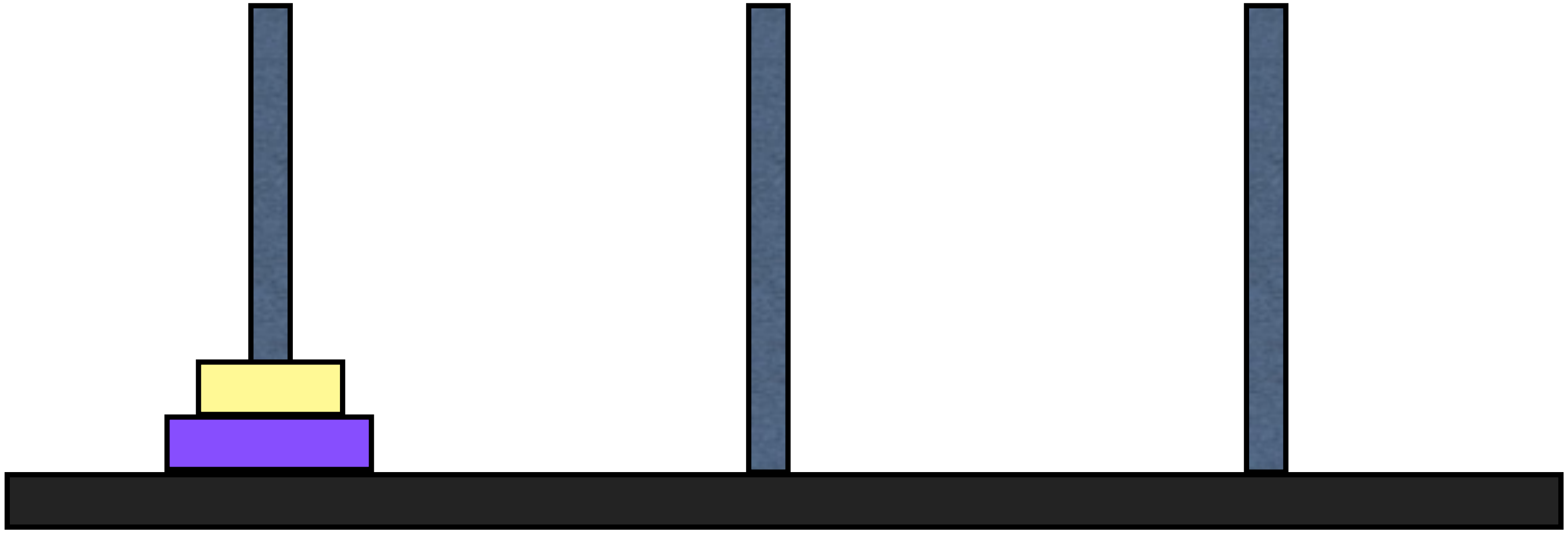
How/Why it works

```
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)
```



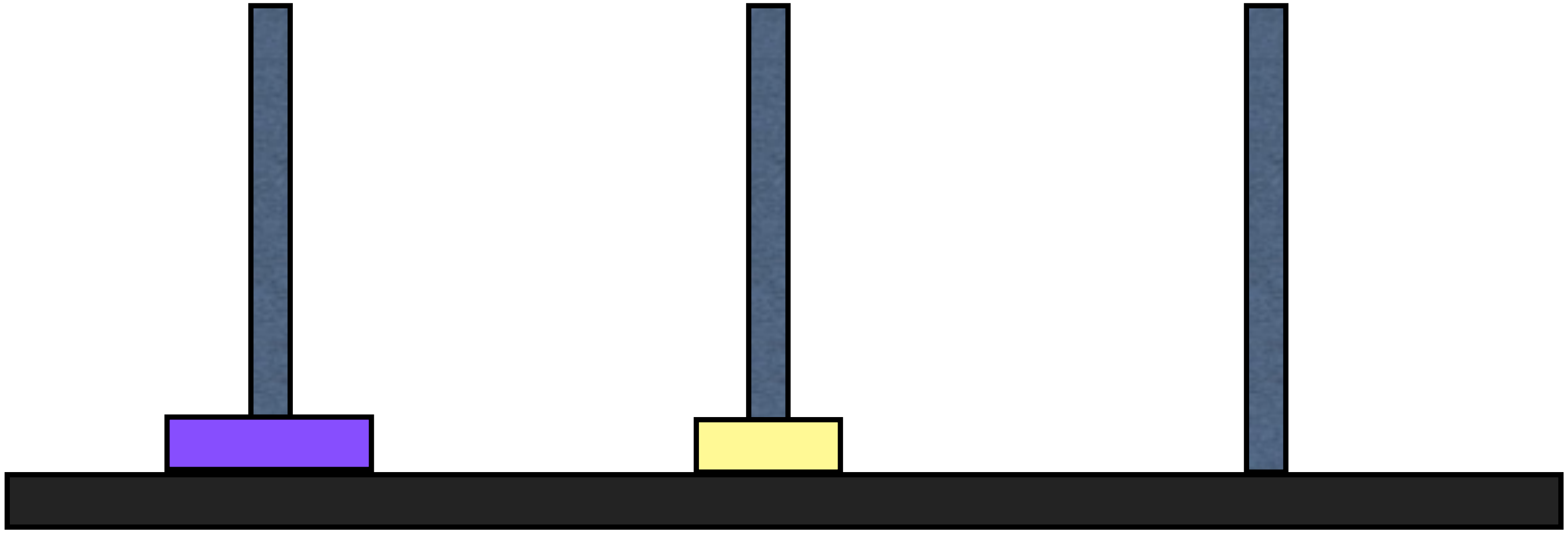
How/Why it works

```
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)
```



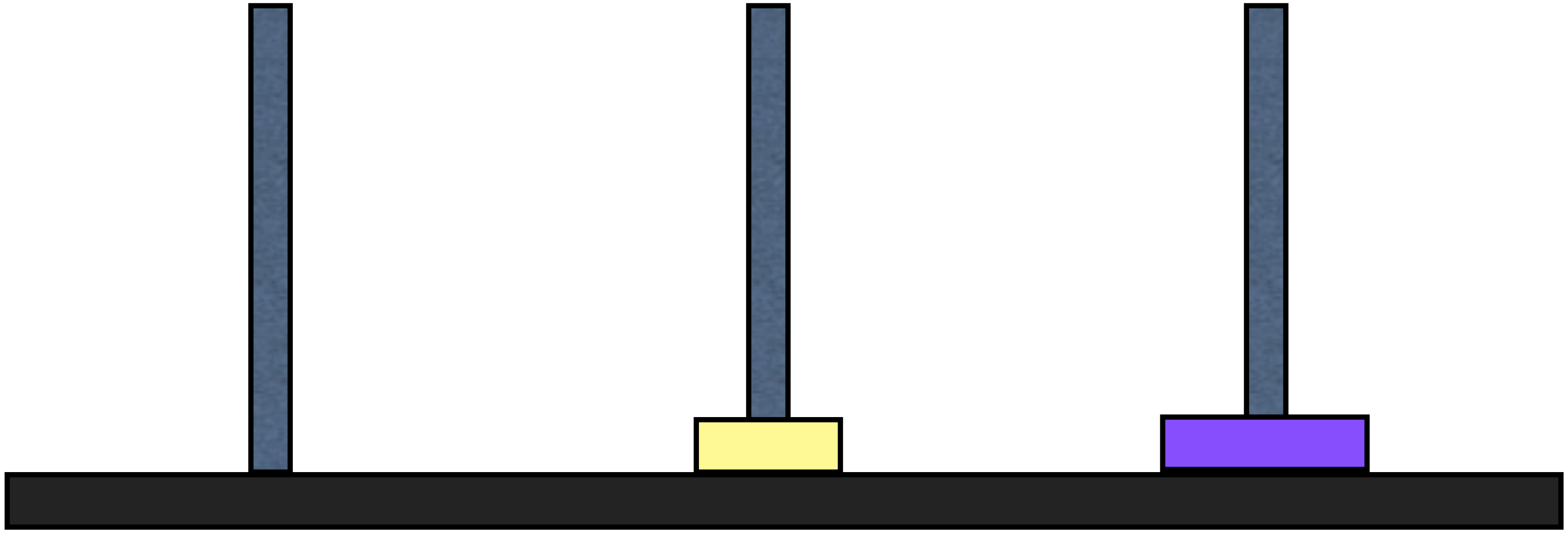
How/Why it works

```
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)
```



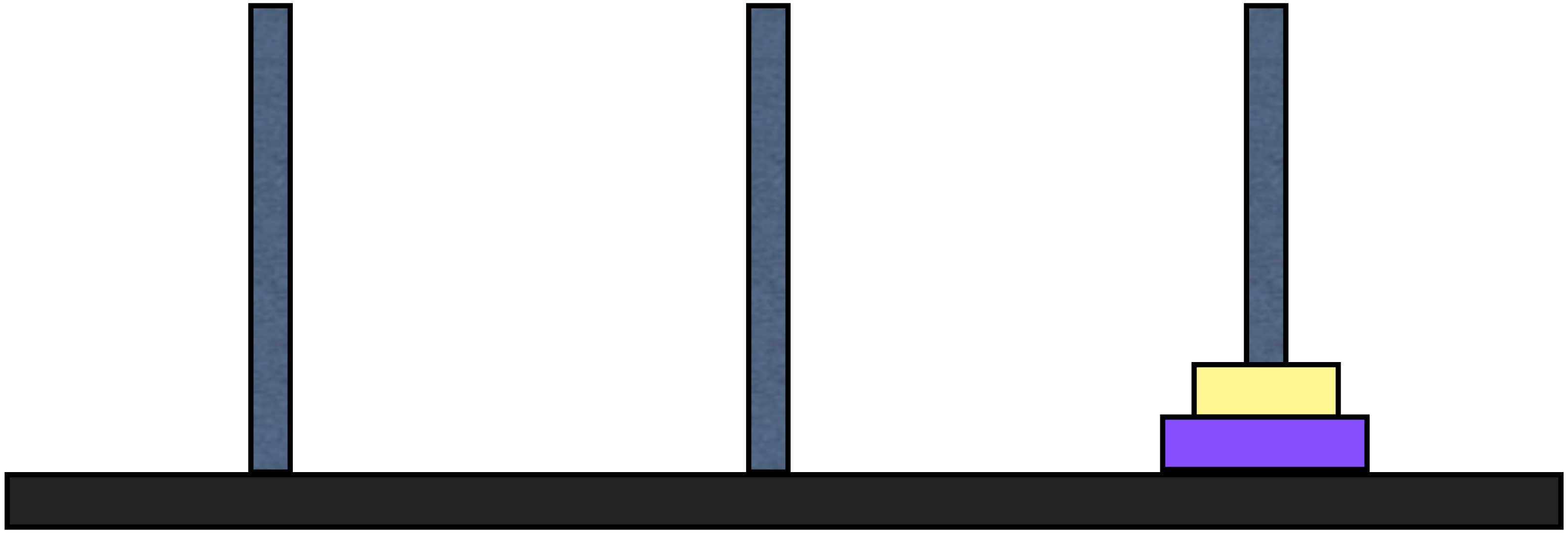
How/Why it works

```
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)
```



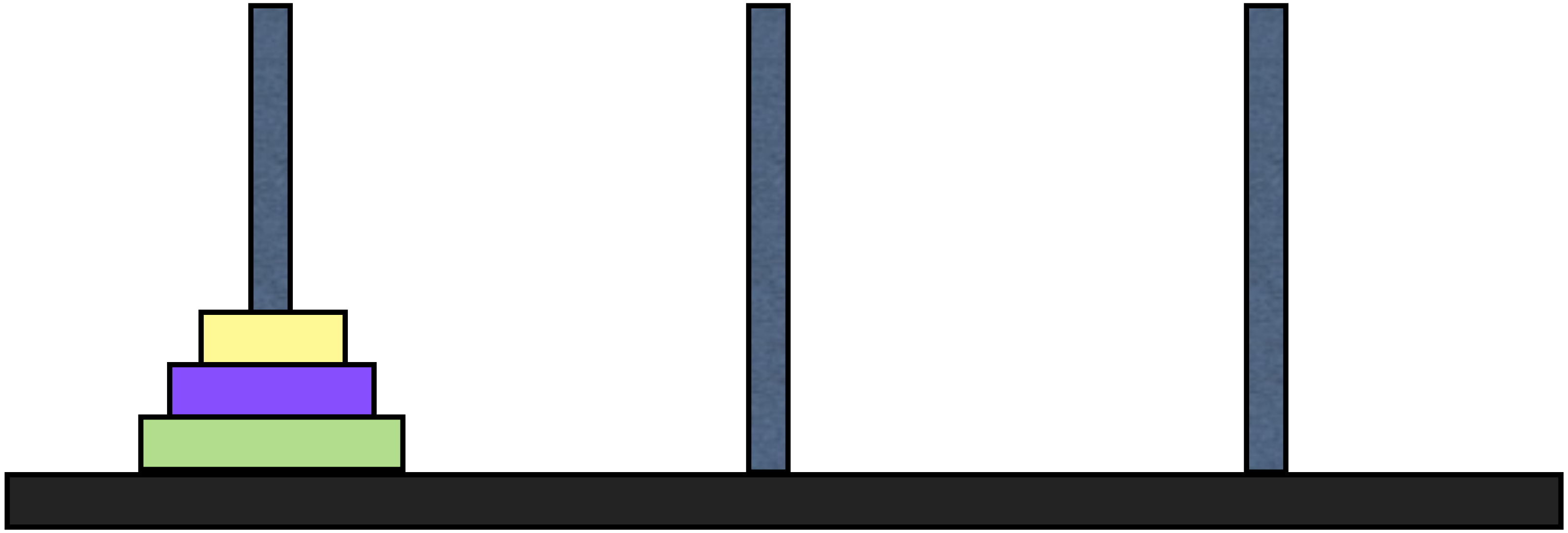
How/Why it works

```
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)
```



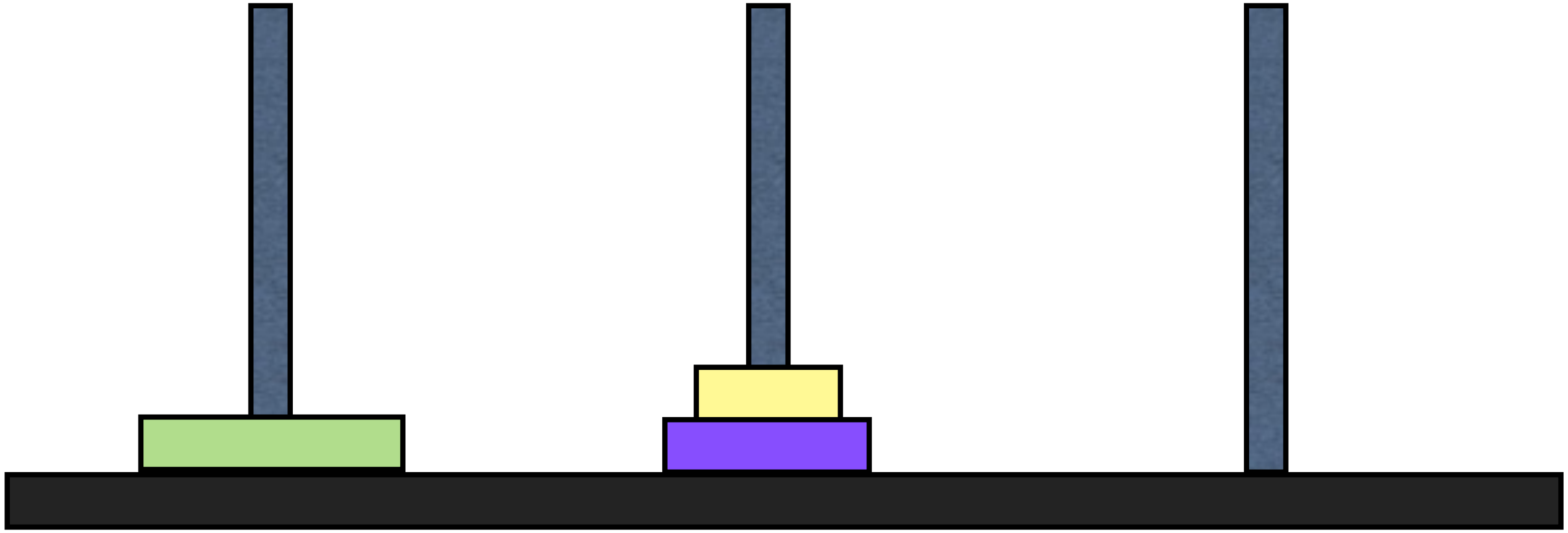
How/Why it works

```
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)
```



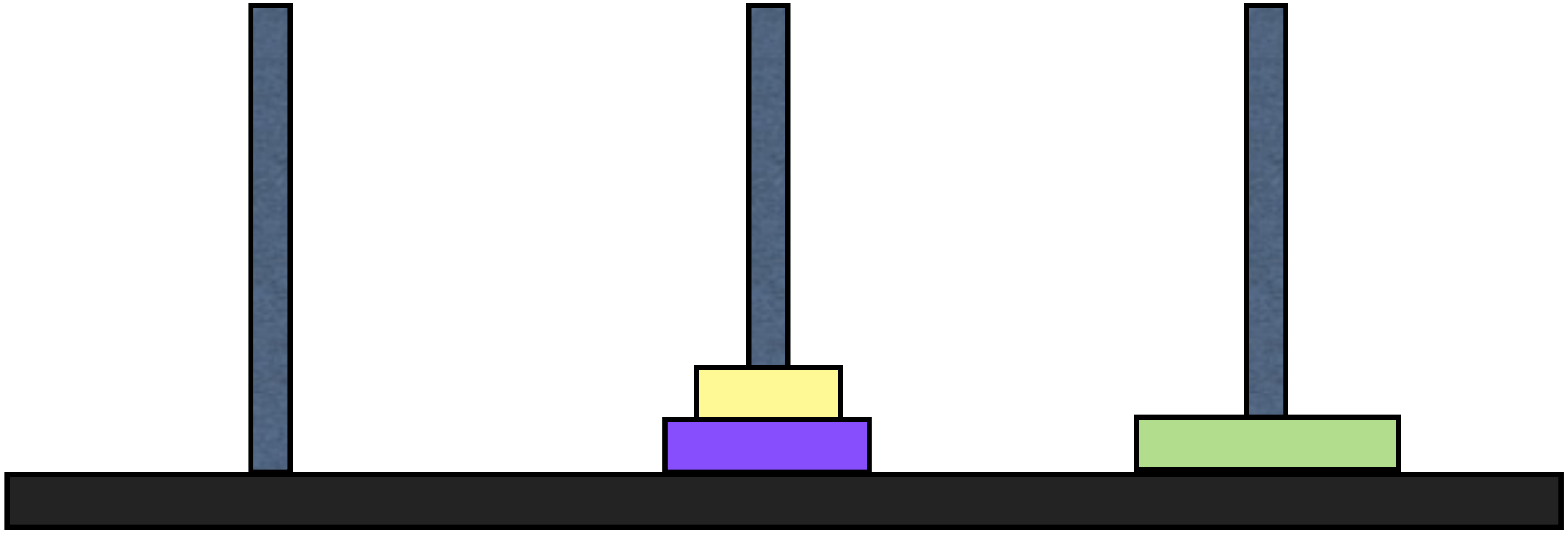
How/Why it works

```
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)
```



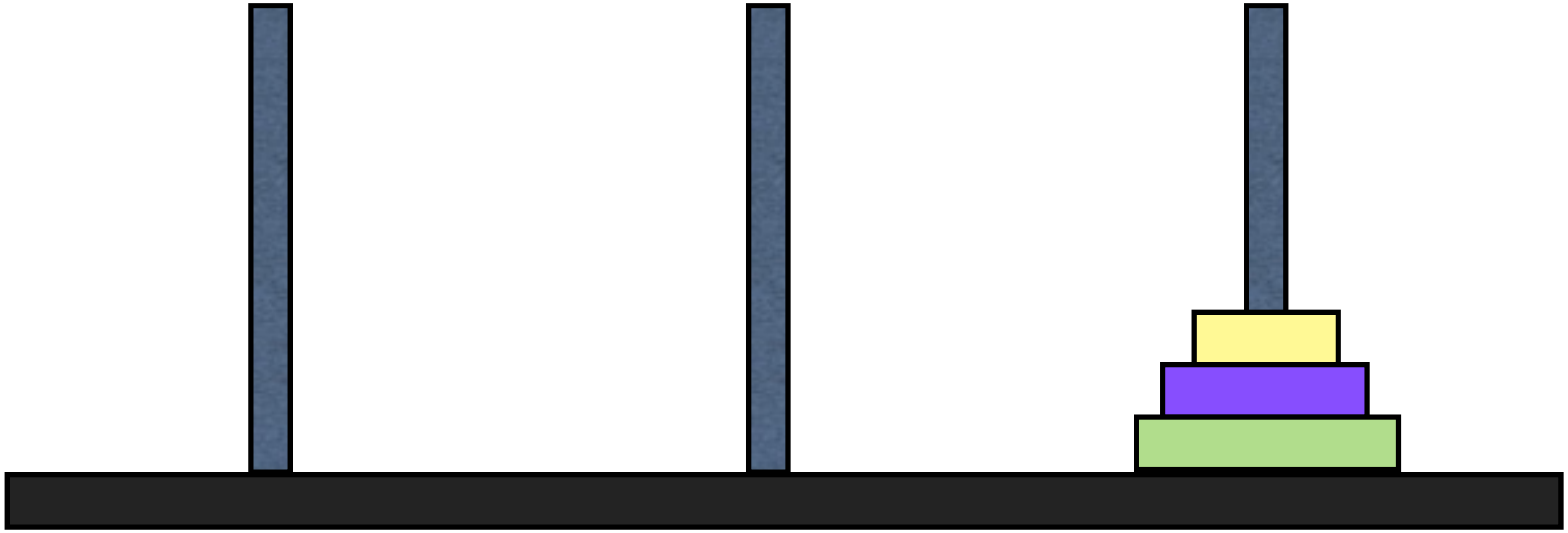
How/Why it works

```
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)
```



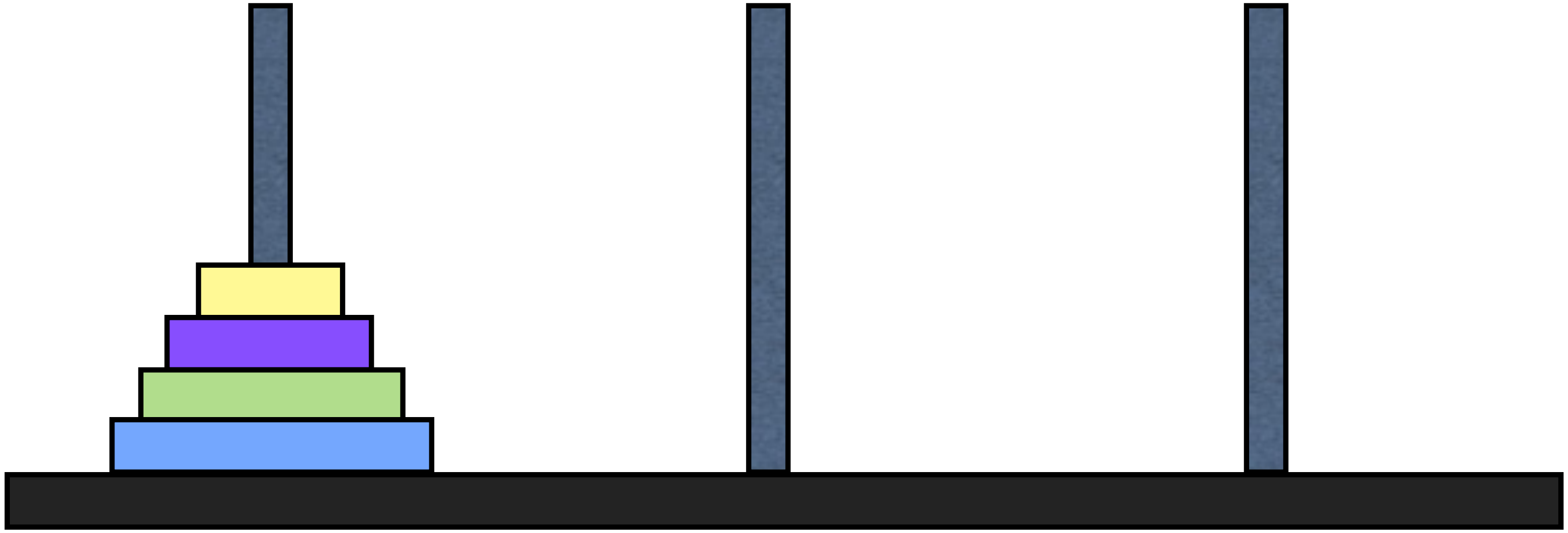
How/Why it works

```
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)
```



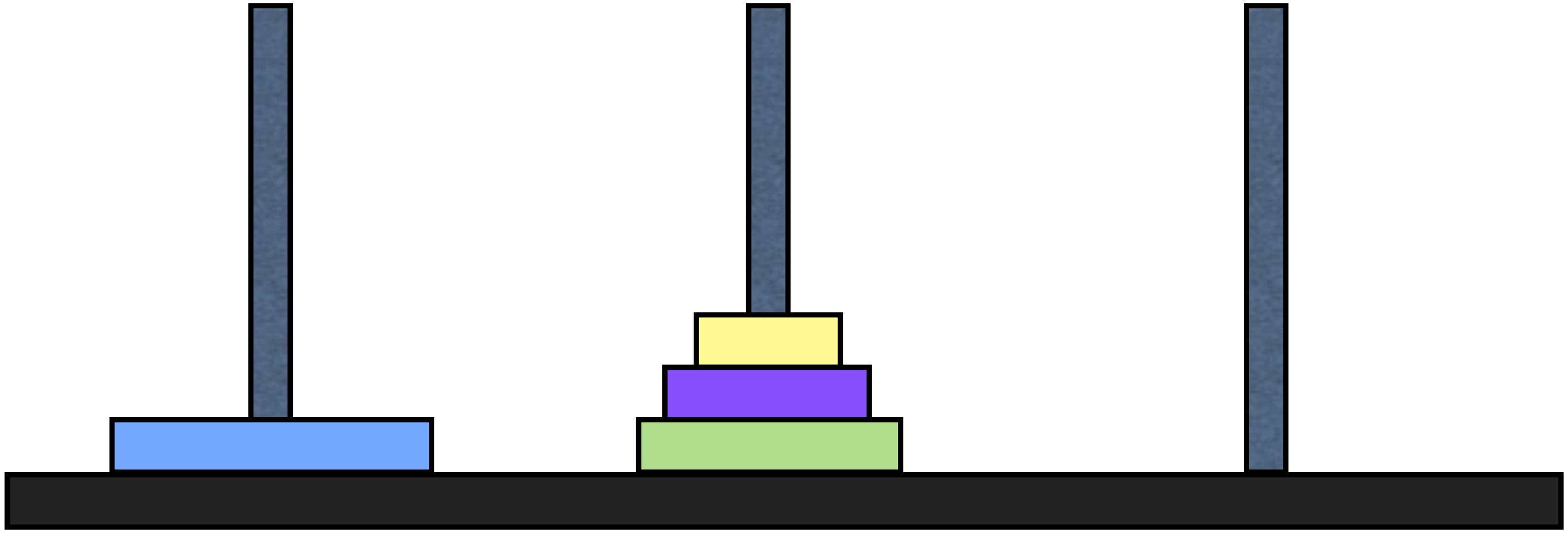
How/Why it works

```
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)
```



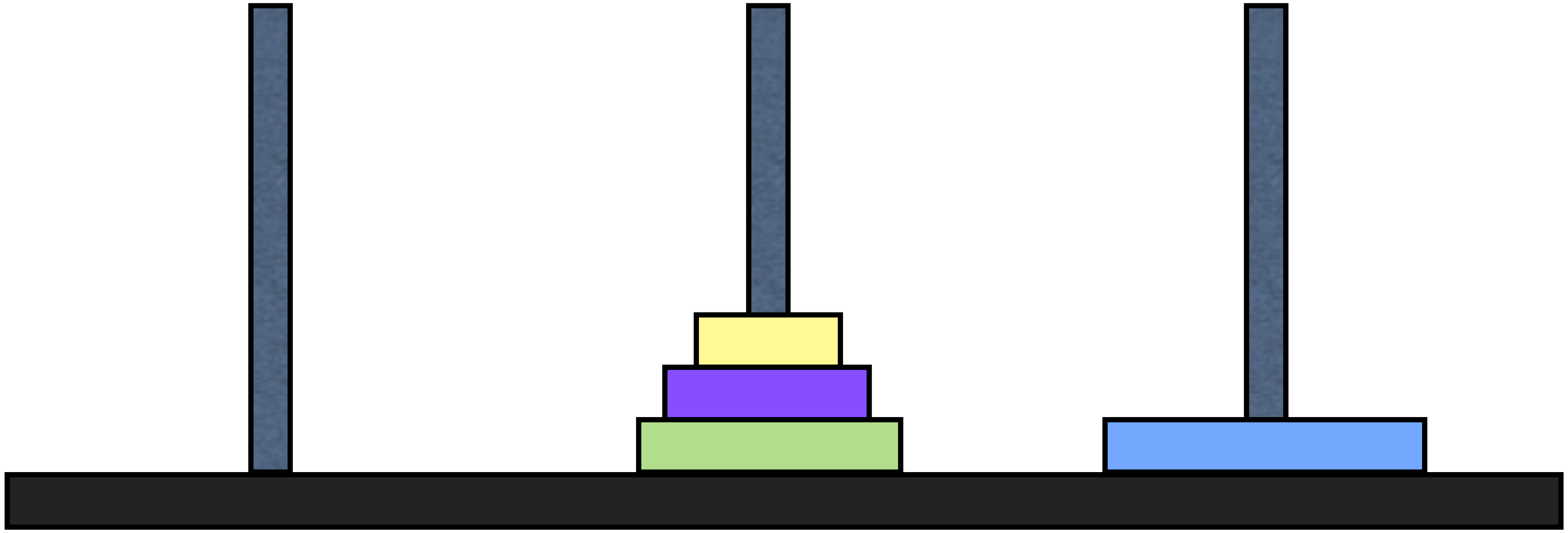
How/Why it works

```
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)
```



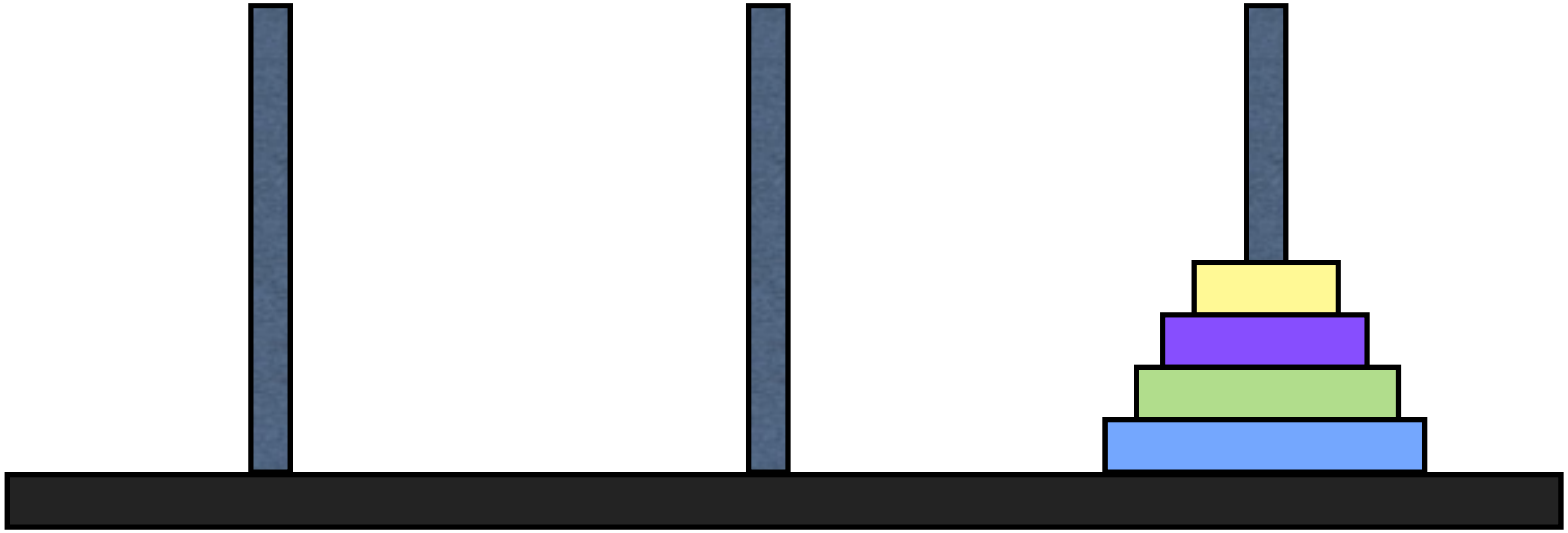
How/Why it works

```
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)
```

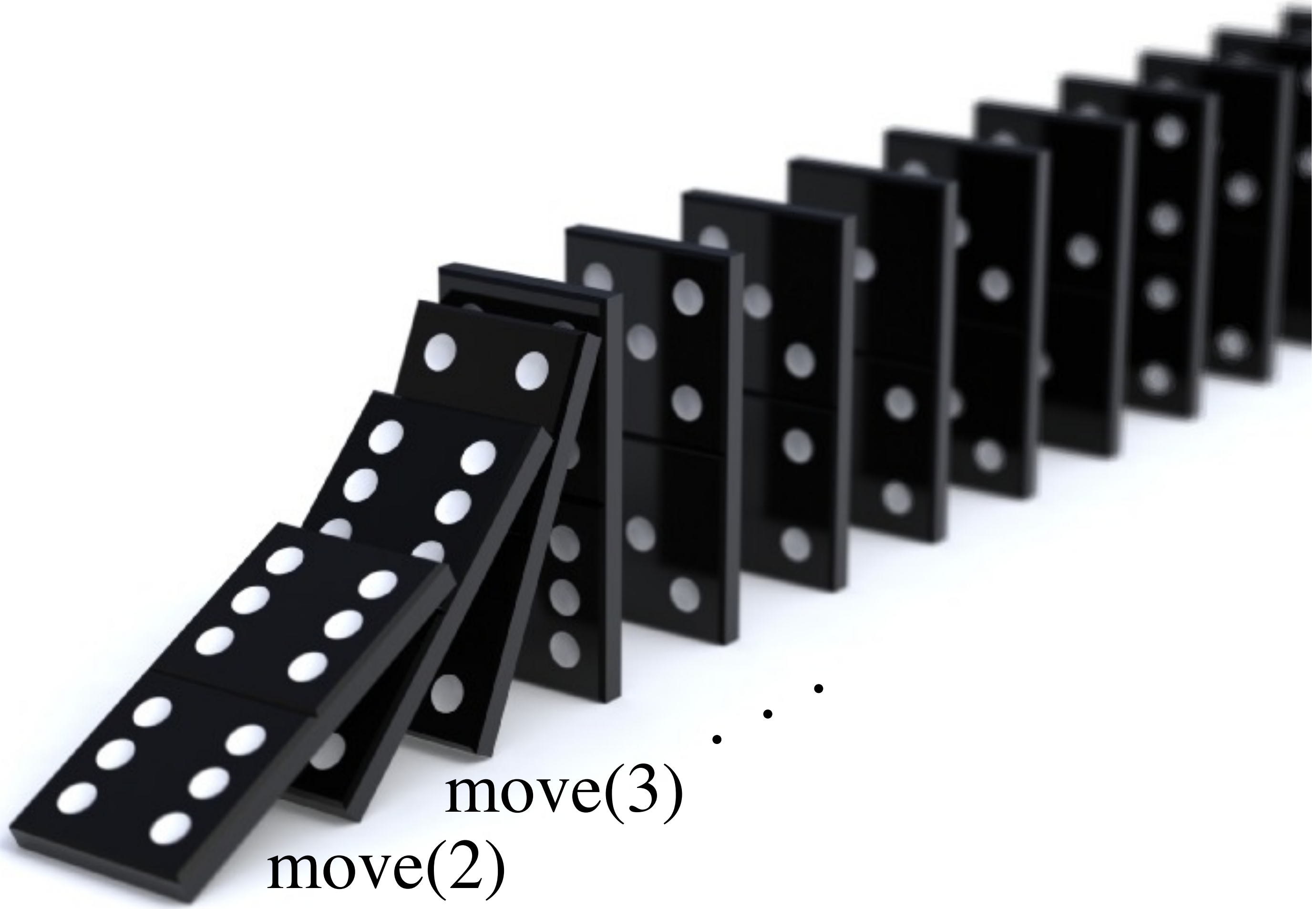


How/Why it works

```
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



. . .

move(3)

move(2)

move(1)