CSE11 Lecture 20

Fall 2013

Recursion

Recursion

- **recursion**: The definition of an operation in terms of itself.
  - Solving a problem using recursion depends on solving smaller or simpler occurrences of the same problem.

- **recursive programming**: Writing methods that call themselves to solve problems recursively.
  - An equally powerful substitute for *iteration* (loops)
  - Particularly well-suited to solving certain types of problems

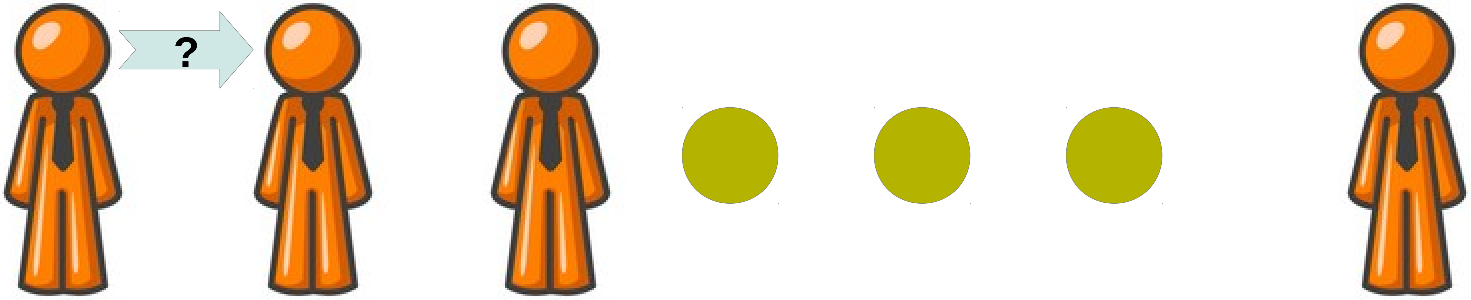
Why learn recursion?

- "Cultural experience" – think differently about problems
- Solves some problems more naturally than iteration
- Leads to elegant, simplistic, short code (when used well)
- Many programming languages ("functional" languages such as Scheme, ML, and Haskell) use recursion exclusively (no loops)

Simple Exercise

How many students total are directly to the left of you in your "row" of the classroom?

- You can only see the person right next to you
- But, You can ask that person a question and he/she can respond to you
- How can we solve this problem (recursively)?



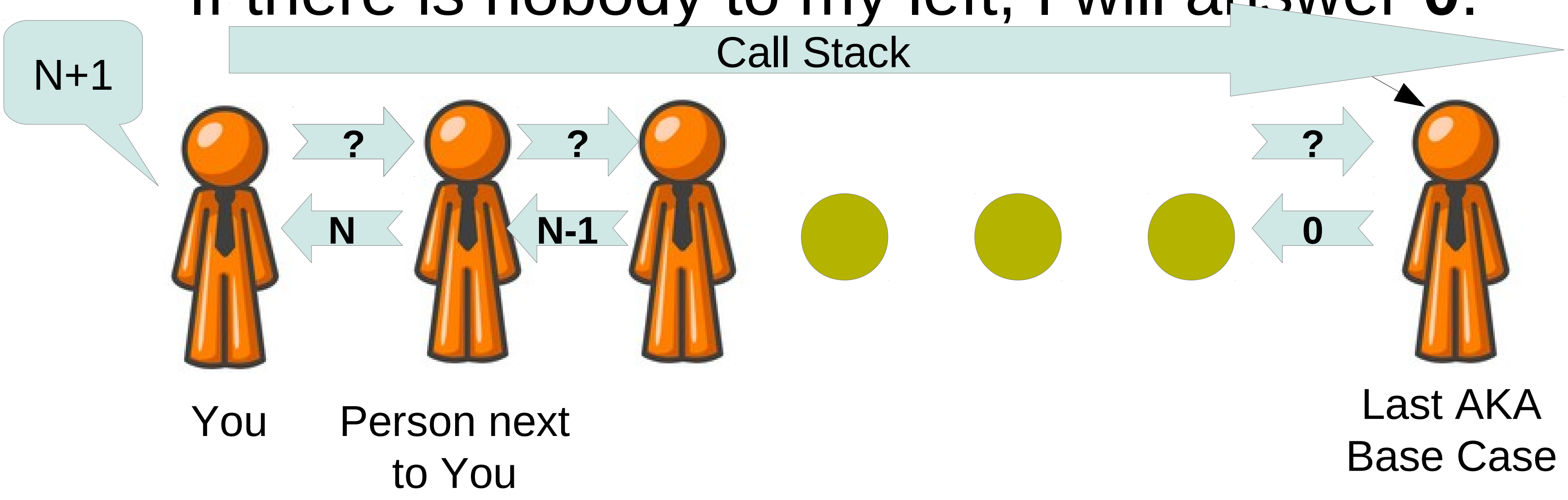
You

Person next to You

Last Person

Recursive algorithm

- Number of people to my left
  - If there is someone to my left,
    ask him/her how many people are to their left
    - When they respond with a value **N**, then I will answer **N + 1**.
  - If there is nobody to my left, I will answer **0**.



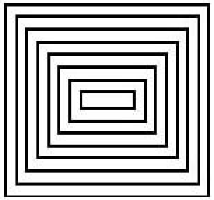
Recursion and cases

- Every recursive algorithm involves at least 2 cases:
  - **base case**: A simple occurrence that can be answered directly.
  - **recursive case**: A more complex occurrence of the problem that cannot be directly answered, but can instead be described in terms of smaller occurrences of the same problem.
  - Some recursive algorithms have *more than one* base or recursive case, but all *have at least one* of each

    .
  - A crucial part of recursive programming is identifying these cases.
  - Can also create/define Recursive Structures.

Complex objects

- How might you design a class called NestedRects of graphical objects that look like this?



- Requirements for the constructor:
  - – Like many graphical objects, takes 5 parameters:
    - x and y describing coordinates of upper left
    - width and height of outermost rectangle
    - canvas
  - – Spacing between rectangles is 4 pixels

Constructor for NestedRects

```
public NestedRects (double x double y,
                   double width, double height,
                   DrawingCanvas canvas) {
  new FramedRect(x, y, width, height, canvas);
  while (width >= 8 && height >= 8) {
      width = width - 8;
      height = height - 8;
      x = x + 4;
      y = y + 4;
      new FramedRect(x, y, width, height, canvas);
  }
}
```

Making NestedRects Useful

- Say that we want NestedRects objects to behave much like other graphical objects?
- NestedRects class should define methods like
  - moveTo()
  - removeFromCanvas()

But our constructor just draws the object

Need a way to keep track of entire collection of nested rectangles

Could use arrays for an iterative solution, but lets pretend we don't know about arrays.

Challenges

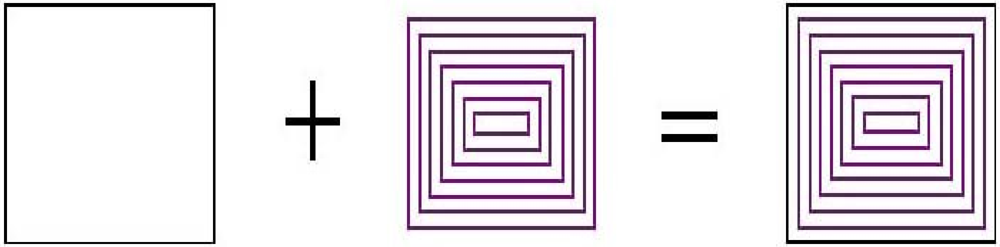
- Need to keep track of the rectangles in the collection
- Instance variables for each of the rectangles won't work:

  FramedRect rectangle1, rectangle2;

  We don't know how many there will be until a user specifies parameters when constructing one

A Recursive Solution

- A recursive structure consists of
  - A base structure (the simplest form of the structure)
  - A way to describe complex structures in terms of simpler structures of the same kind
- Let’s change the way we think about NestedRects
  - Rather than a series of FramedRects…
  - = outer FramedRect + a smaller NestedRects inside



NestedRects: a recursive def’n

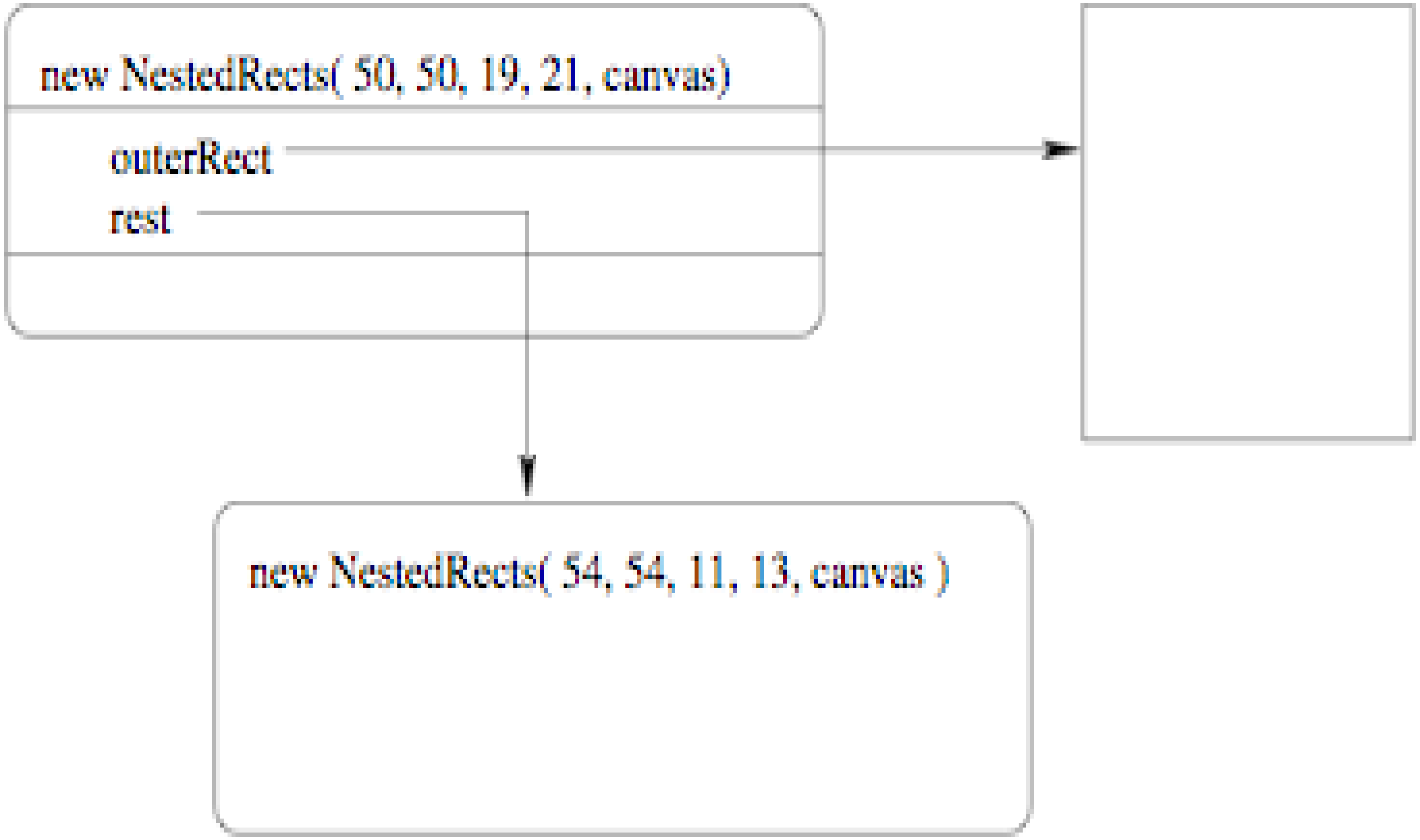
```
public class NestedRects {
    private FramedRect outerRect; // outermost rectangle
    private NestedRects rest;          // inner nested rects

    public NestedRects(double x, double y,
                          double width, double height,
                          DrawingCanvas canvas) {
        outerRect = new FramedRect(x, y, width, height, canvas);
        if (width >= 8 && height >= 8) {
                rest = new NestedRects(x+4, y+4, width-8,
                                             height-8, canvas);
        } else {
                rest = null; // nothing more to construct
        }
    }
```

```
        // Move nested rects to (x, y)
        public void moveTo(double x, double y) {
                outerRect.moveTo(x, y);
                if (rest != null) {
                        rest.moveTo(x+4, y+4);
                }
        }

        // Remove the nested rects from the canvas
        public void removeFromCanvas() {
                outerRect.removeFromCanvas();
                if (rest != null) {
                        rest.removeFromCanvas();
                }
        }
}
```

Tracing the execution of new NestedRects(50, 50, 19, 21, canvas);

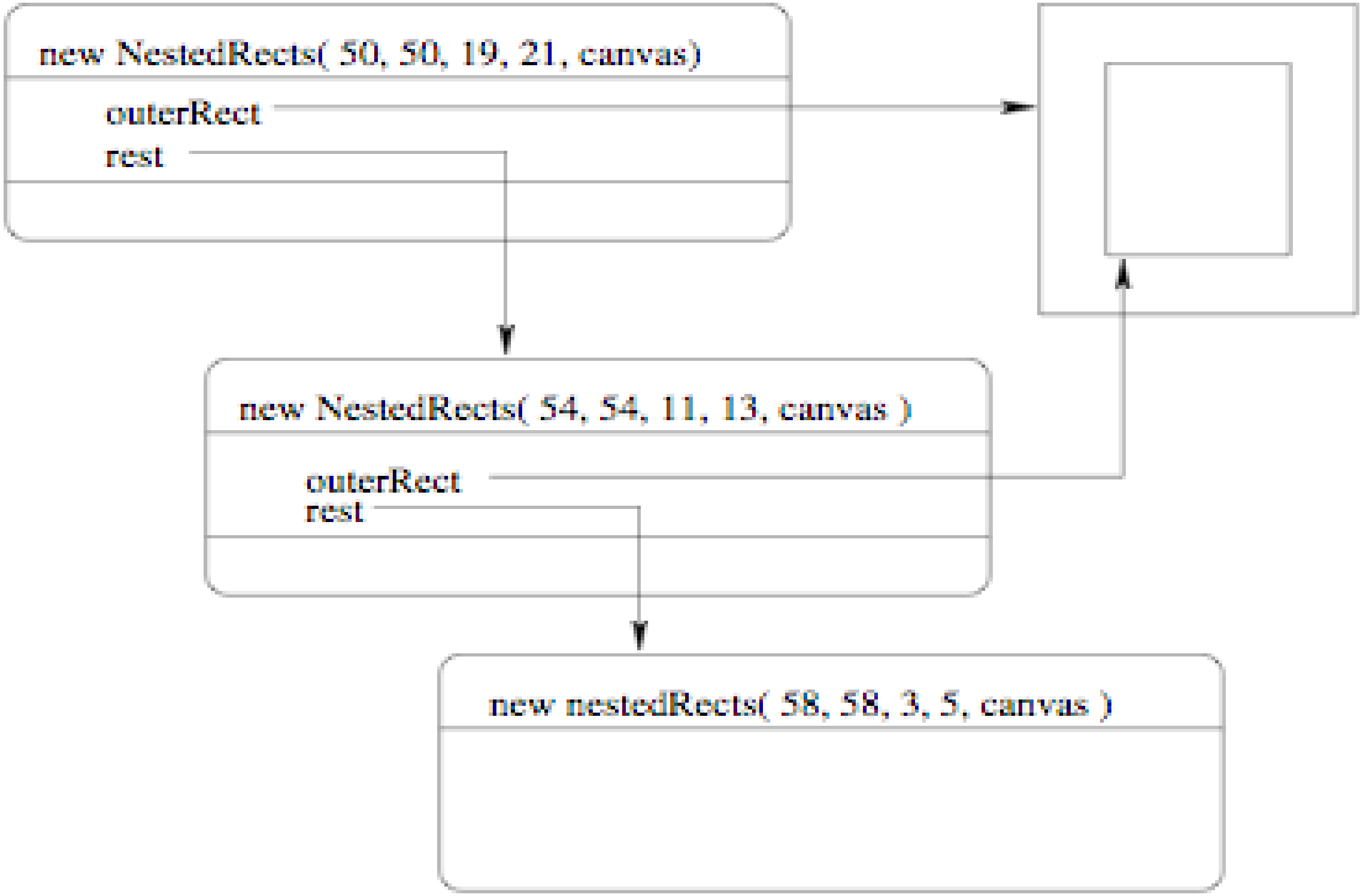


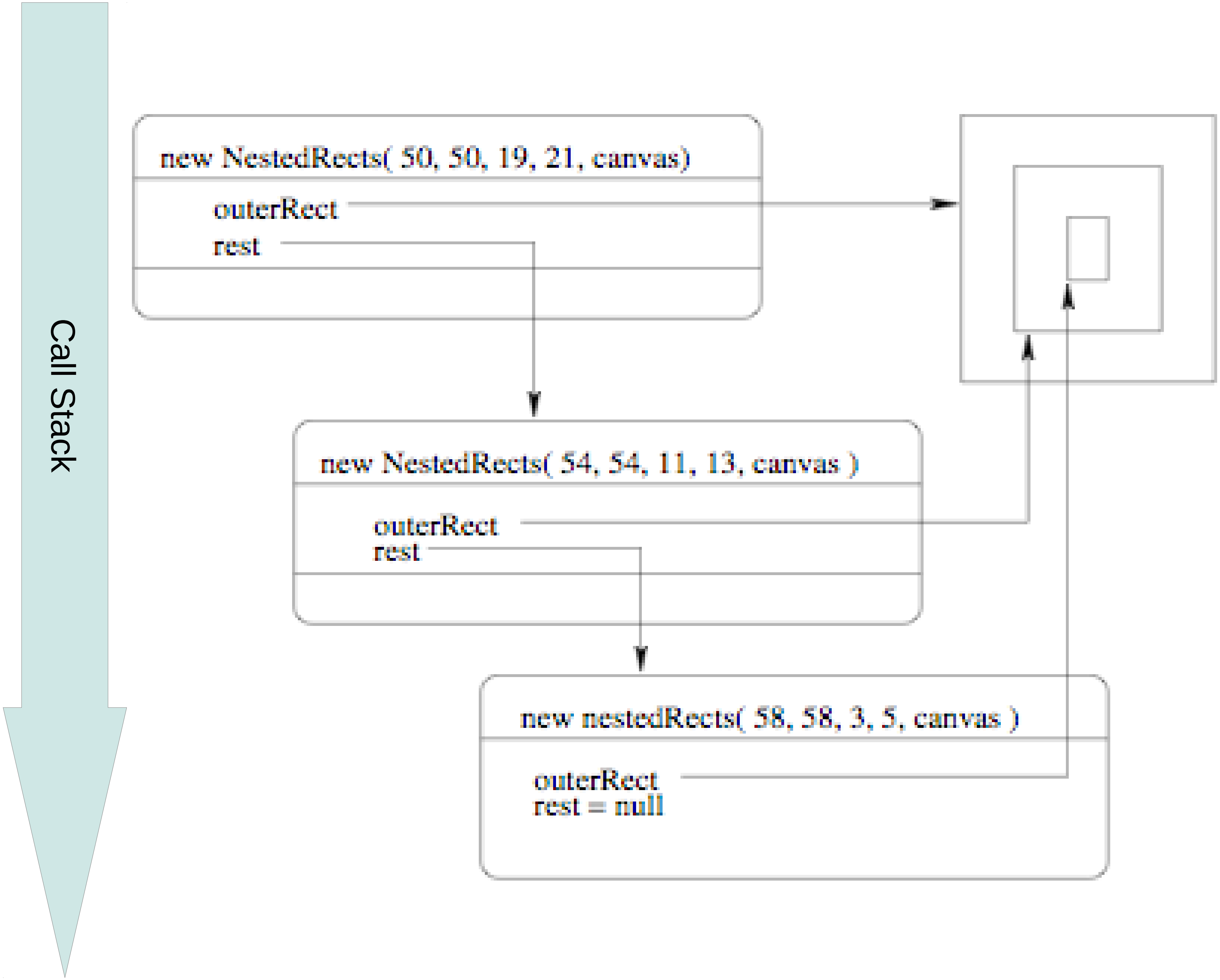
new NestedRects(50, 50, 19, 21, canvas)

outerRect

rest

new NestedRects(54, 54, 11, 13, canvas)


new NestedRects(50, 50, 19, 21, canvas)
outerRect
rest
new NestedRects(54, 54, 11, 13, canvas)
outerRect
rest
new nestedRects(58, 58, 3, 5, canvas)



Call Stack

new NestedRects(50, 50, 19, 21, canvas)

outerRect

rest

new NestedRects(54, 54, 11, 13, canvas)

outerRect

rest

new nestedRects(58, 58, 3, 5, canvas)

outerRect

rest = null

A Better Recursive Solution?

- moveTo and removeFromCanvas require checking whether rest is null
- Missing check will cause program to crash

- Can we write NestedRects to avoid the check for null?

Two Kinds of NestedRects

- “Normal” recursive case
  - outerRect
  - rest
- A special “simplest” NestedRects: **empty!**

Define a new class, BaseRects, representing an empty collection of FramedRects

A Simple Base Class
(an Empty Nested Rect)

```
public class BaseRects extends NestedRects2 {
    // Constructor has nothing to initialize
    public BaseRects() { }

    // Move nested rectangles to (x, y)
    public void moveTo(double x, double y) { }

    // Remove nested rectangles from canvas
    public void removeFromCanvas() {  }

}
```

A Base Class
(an Empty Nested Rect)

```
public class BaseRects extends NestedRects2 {
    // Constructor has nothing to initialize
    public BaseRects() { }

    // Move nested rectangles to (x, y)
    public void moveTo(double x, double y) { }

    // Remove nested rectangles from canvas
    public void removeFromCanvas() {  }

}
```

Revised Recursive Class

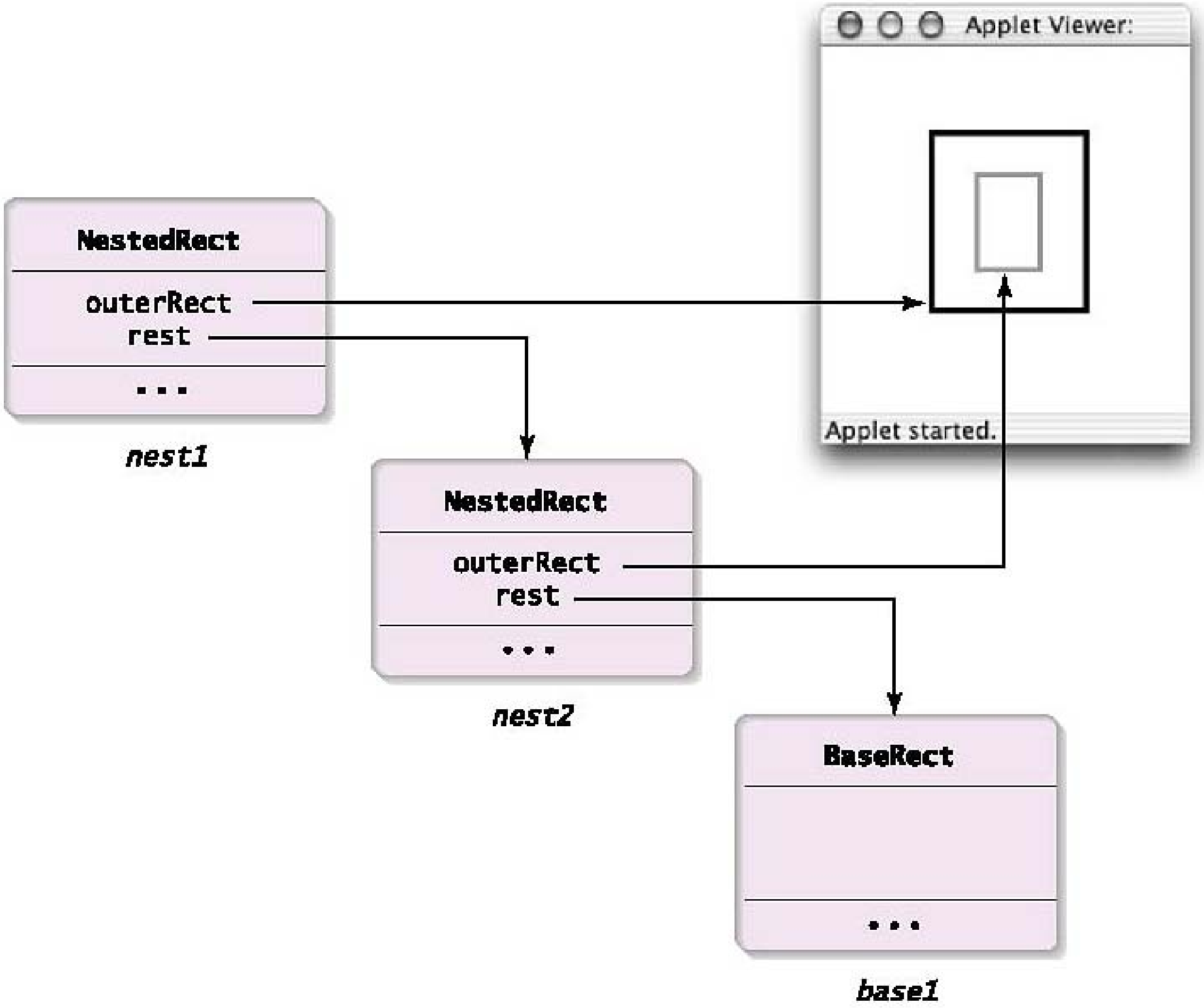
```
public class NestedRects2  {
    private FramedRect outerRect;                    // outermost rectangle
    private NestedRects2 rest;          // inner nested rects

    public NestedRects2(double x, double y,
                        double width, double height,
                        DrawingCanvas canvas) {
        outerRect = new FramedRect(x, y, width, height, canvas);
        if (width >= 8 && height >= 8) {
                rest = new NestedRects(x+4, y+4, width-8,
                                              height-8, canvas);
        } else { // construct a base object
                rest = new BaseRects();
        }
    }
```

```
    // Move nested rects to (x, y)
    public void moveTo(double x, double y) {
        outerRect.moveTo(x, y);
        rest.moveTo(x+4, y+4)
    }

    // Remove the nested rects from the canvas
    public void removeFromCanvas() {
        outerRect.removeFromCanvas();
        rest.removeFromCanvas();
    }
}
```

Evaluating
new NestedRects2(54, 54, 11, 13, canvas)



- Since objects of type BaseRects and NestedRects2 know how to “moveTo” and “removeFromCanvas” …

Checks for null are eliminated

'0' or Empty as the “base” case is often a good starting place for recursion

Designing recursive structures

Recursive structures built by defining classes for base and recursive cases

- Both implement same interface
- Base class
  - No instance variable has same type as interface or class
  - Generally easy to write
- Recursive class
  - At least one instance variable has same type as interface of class
  - Care needed to be sure methods terminate

Recursive Methods (or Algorithms)

- Can write recursive methods that are not part of recursive structures
- SolveMe (N) --> X + SolveMe(N-1)
- A very common use of recursion are so-called “divide and conquer” algorithms
  - Solve two problems, each of 1/2 the size of the original, then assemble the full answer from both parts
  - Sorting in Searching (Chapter 20)

Base case replaces Base class

Recursive methods

- Must include at least one base case
- Typically contain a conditional statement
  - At least one case is a recursive invocation
  - At least one case is a base case -- i.e., no recursive invocation of the method

- Without a BASE case you will recurse infinitely! (That's bad)

An example: Exponentiation

- Inspiration: Fast algorithms for exponentiation important to RSA algorithm for public key cryptography – calculate: B^k
- A simple (not fast!) recursive method:

```
// returns base raised to exponent as long as exponent >=0
public double simplePower(double base, int exponent) {
    if (exponent == 0) {
          return 1;
    } else {
          return base * simplePower(base, exponent-1);
    }
}
```

An example: Exponentiation

- Inspiration: Fast algorithms for exponentiation important to RSA algorithm for public key cryptography – calculate: B^k
- A simple (not fast!) recursive method:

```
// returns base raised to exponent as long as exponent >=0
public double simplePower(double base, int exponent) {
    if (exponent == 0) {
        return 1;
    } else {
        return base * simplePower(base, exponent-1);
    }
}
```

Rules for writing recursive methods

- Write the base case
  - No recursive call
- Write the recursive case
  - All recursive calls should go to simpler cases
  - Simpler cases must eventually reach base case

Applying rules to simplePower

- Base case: exponent == 0
  - Returns 1
  - Correct answer for raising base to the 0th power
  - No recursive invocation
- Recursive case: uses else clause
  - Recursive call involves smaller value for exponent
  - Recursive calls eventually reach base case of 0: exponent greater than 0 to start and always goes down by 1

Rules of Exponents

- Simple algorithm took advantage of these rules:
  - $\text{base}^{0} = 1$
  - $\text{base}^{exp+1} = \text{base} * \text{base}^{exp}$
- New algorithm will make use of this rule:
  - $\text{Base}^{m*n} = (\text{base}^{m})^{n}$

Let m = 2, n = exp/2

$$\text{base}^{exp} = (\text{base}^{2})^{exp/2}$$

Faster exponentiation

```
public double fastPower(double base, int exponent) {
        if (exponent == 0) {
                return 1;
        } else if ( exponent%2 == 1 ) {  // odd exponent
                return base * fastPower(base, exponent-1);
        } else {
                return fastPower(base* base, exponent/2);
        }
}
```

$base^{exp} = (base^2)^{exp/2}$

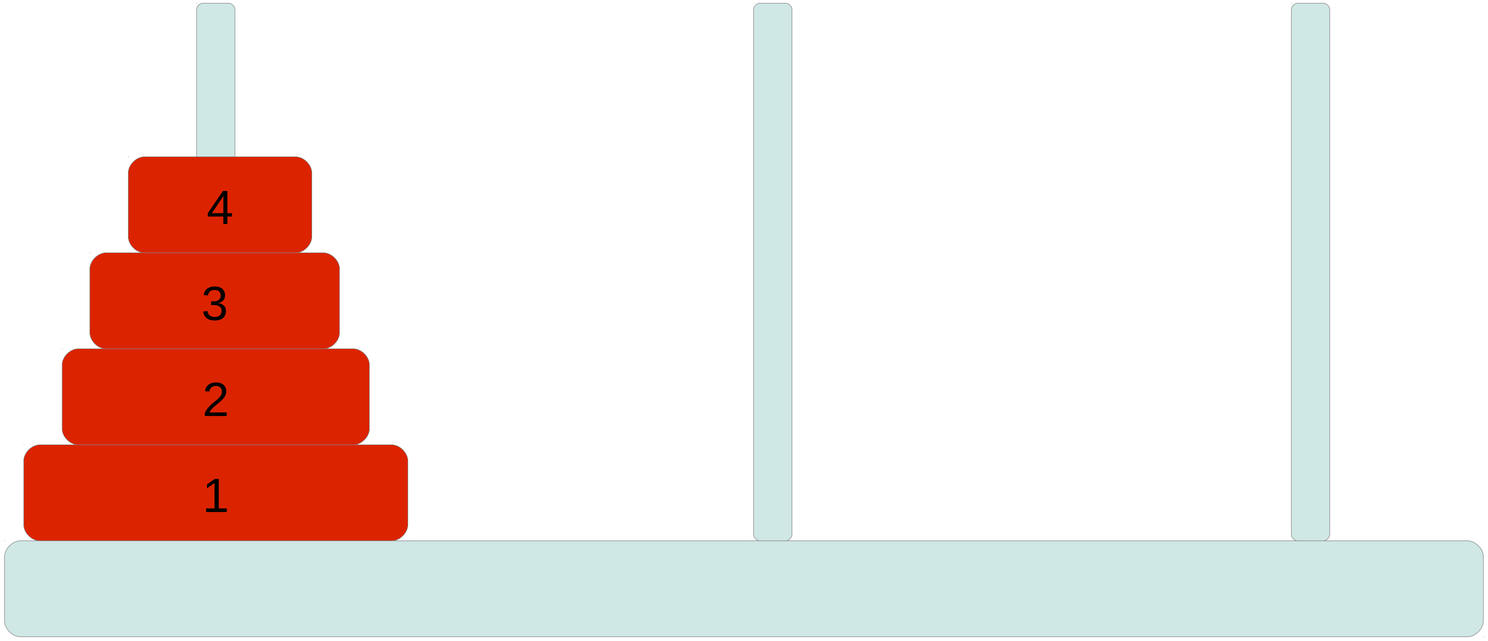
Tracing fastPower

```
fastPower(3, 16)
        = fastPower(9, 8)
        = fastPower(81, 4)
        = fastPower(6561, 2)
        = fastPower(43046721, 1)
        = 43046721 * fastPower(43046721, 0)
        = 43046721 * 1
        = 43046721
```

Only 5 multiplications!

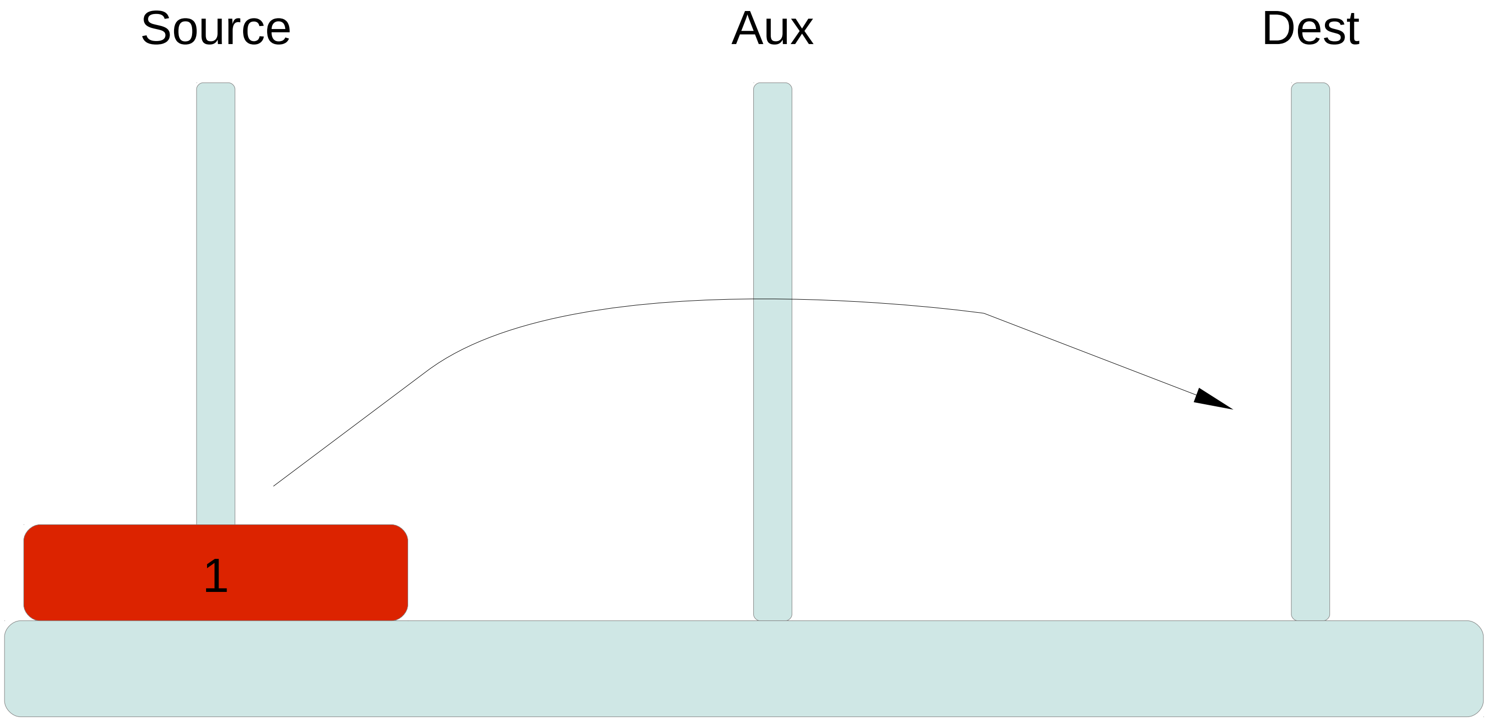
Division by 2 is fast and easy for computers

Towers of Hanoi



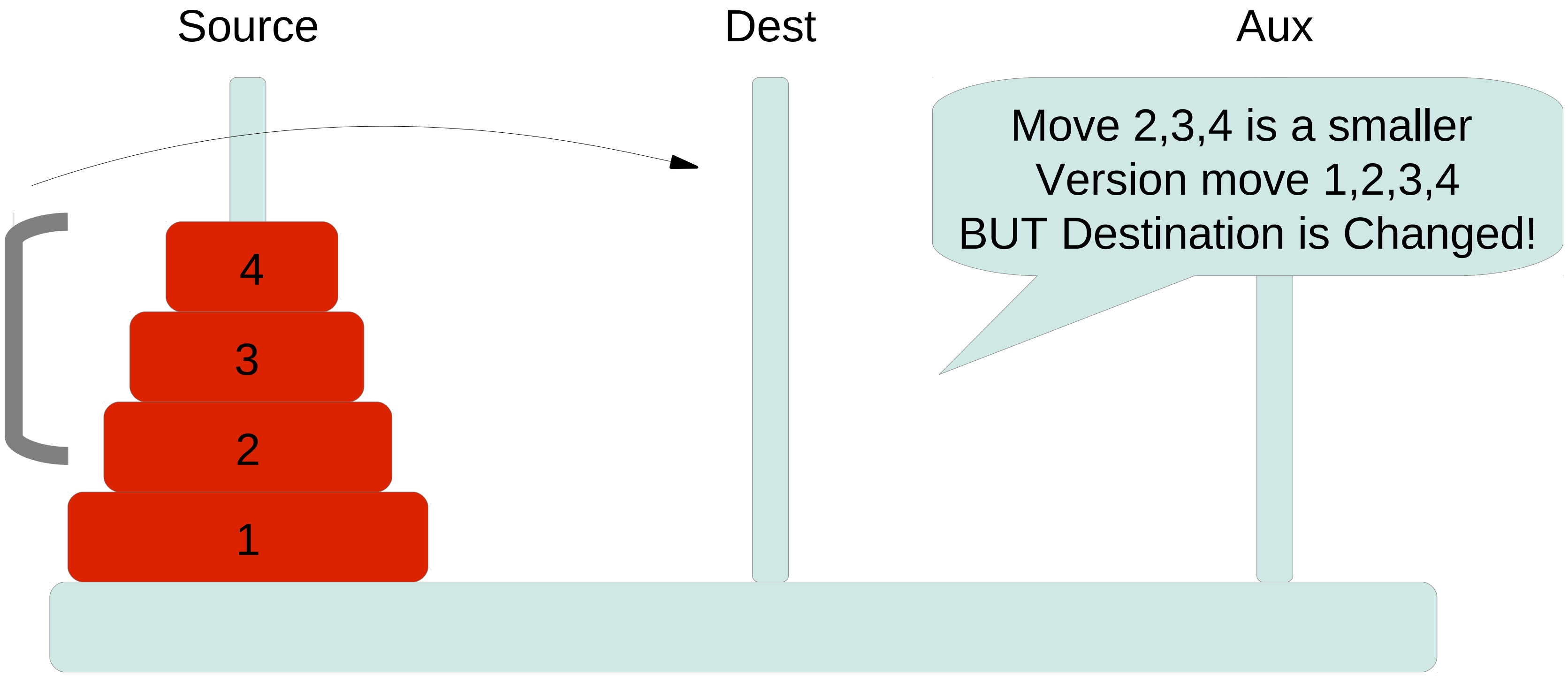
Move all disks from left to right peg
Move one at a time
Can only put smaller disks on empty or larger disks

Base Case



Move all disks from left to right peg
Move one at a time
Can only put smaller disks on empty or larger disks

1st Recursive Move

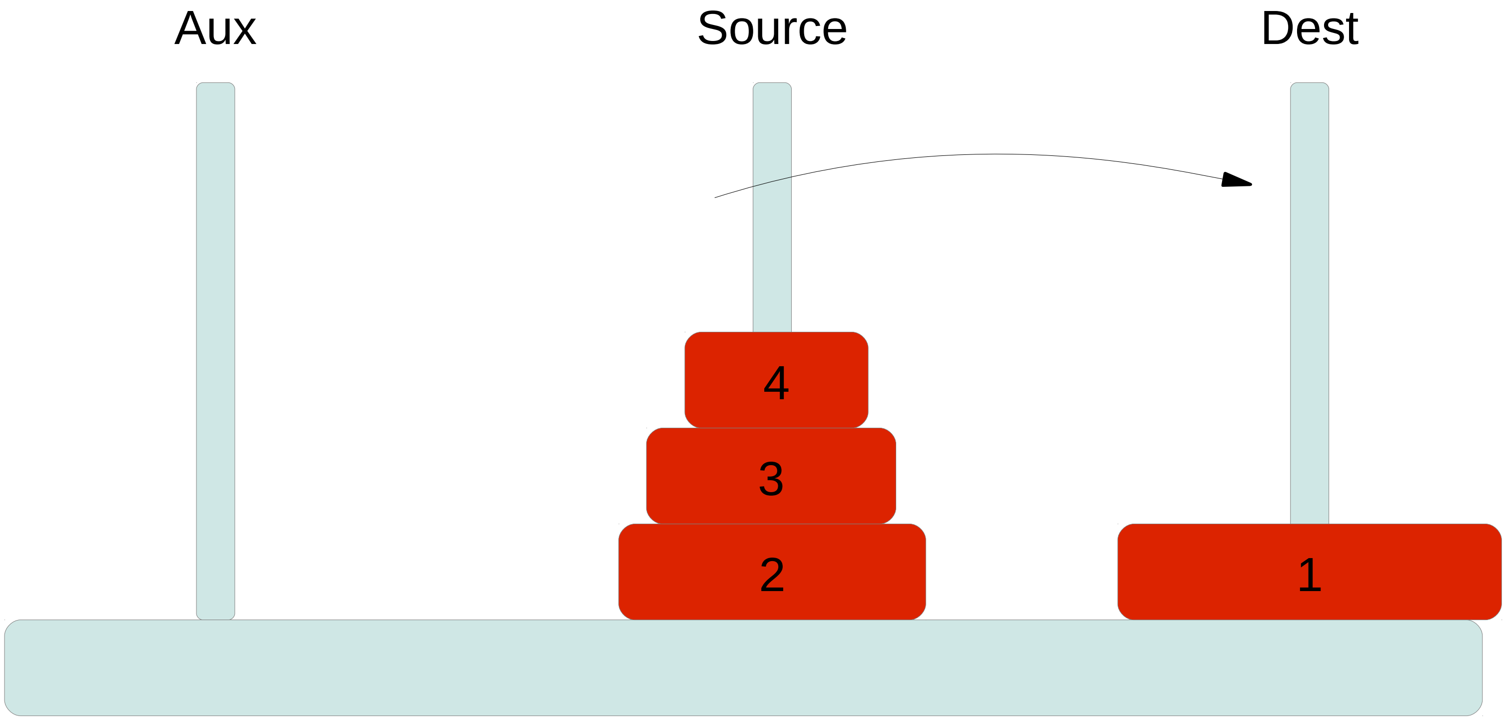


Move 2,3,4 is a smaller
Version move 1,2,3,4
BUT Destination is Changed!

Move All Disks from Left Tower to Right Tower =

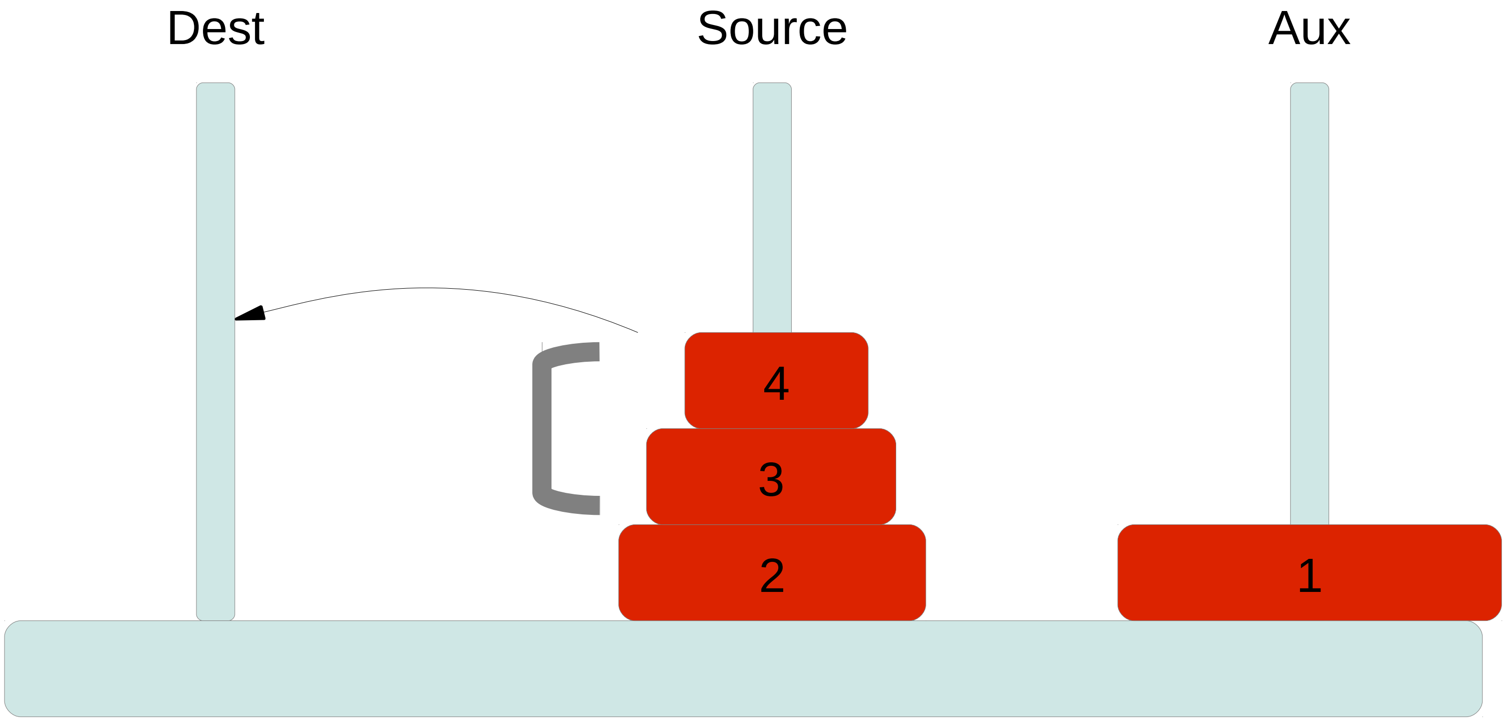
1. Move 2,3,4 to Middle (Recursive) + Move 1 to Right (base)
2. Then move Disks 2,3,4 (Recursive) to the Right Tower

Suppose we've accomplished Step ①



After ① has completed, then do ② . Howework explores this. Note both of these are recursive.

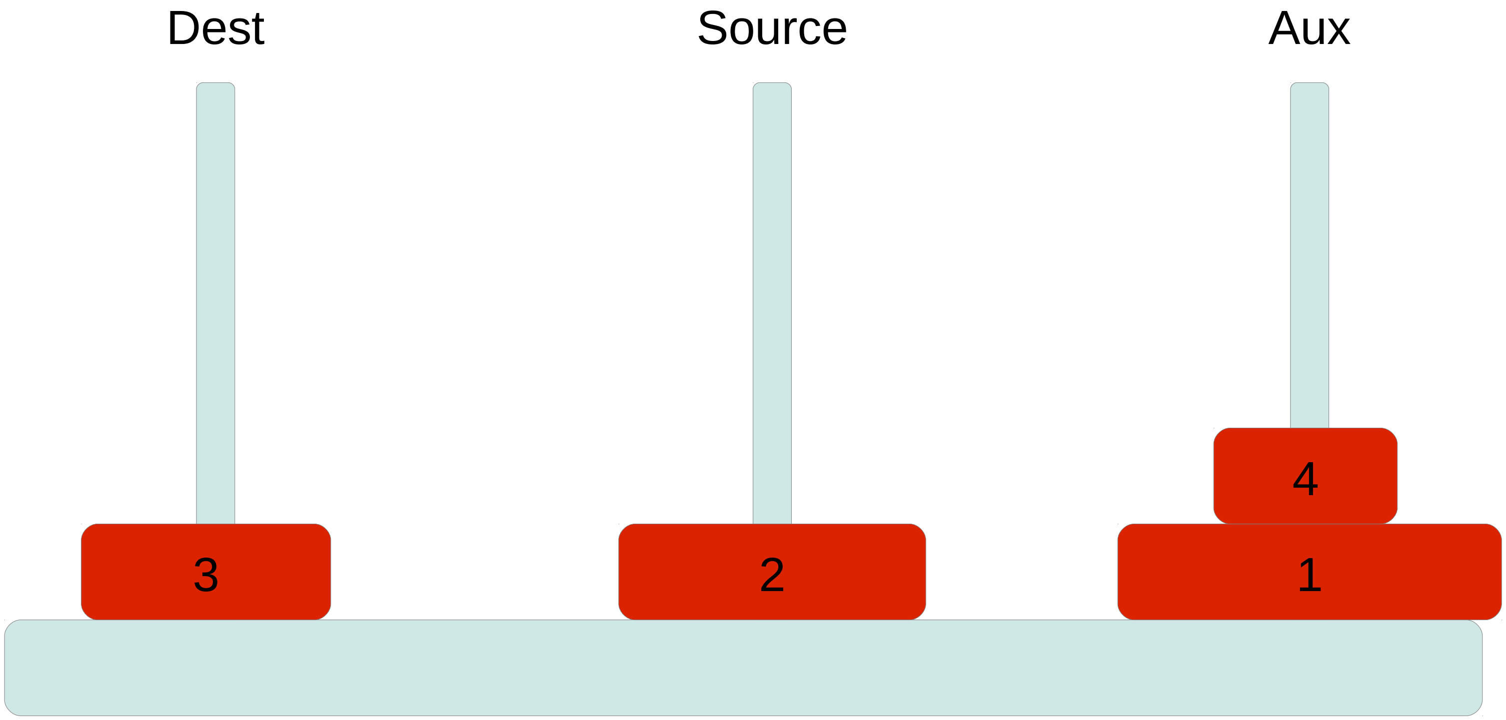
Recursive Move



To Move 2,3,4 to the Right Tower

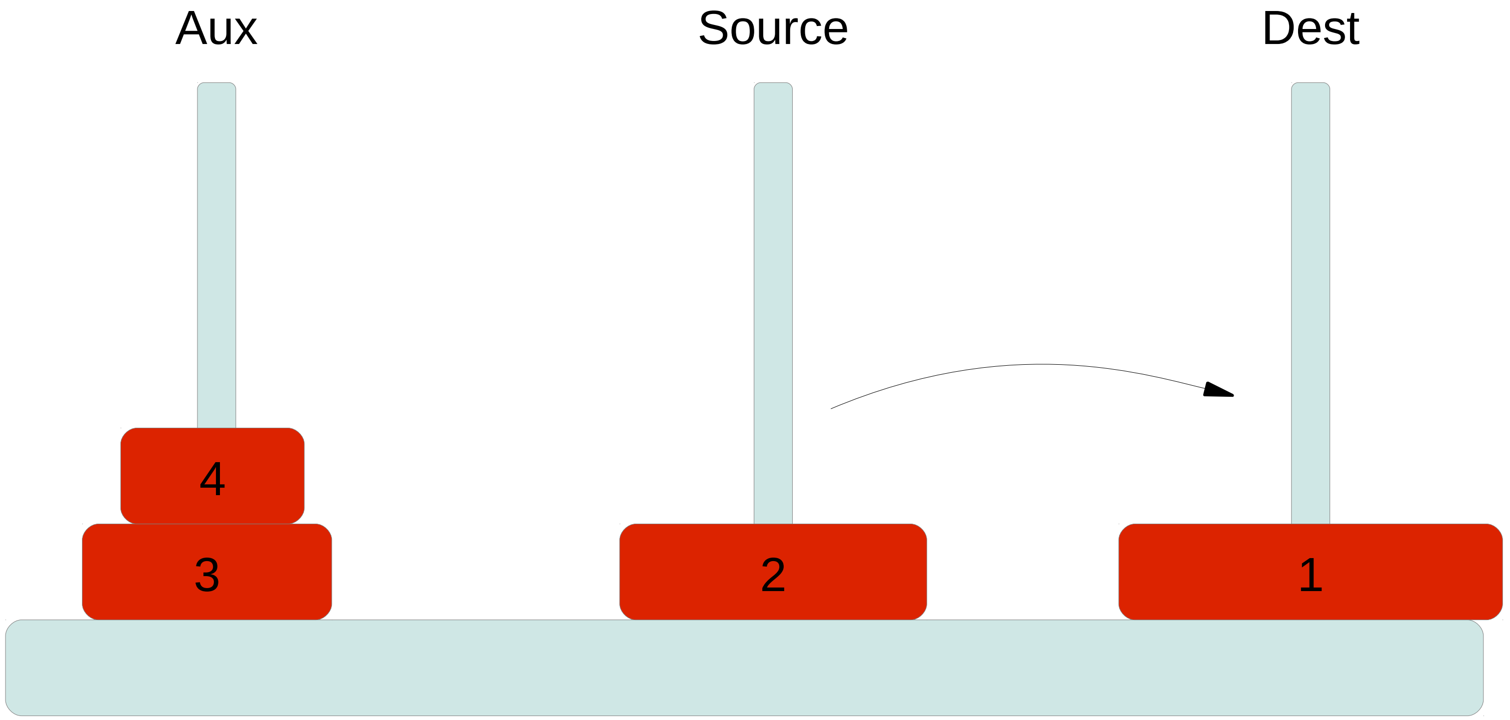
= Move 3,4 to the left tower + move 2 the Right Tower

Recursive Move



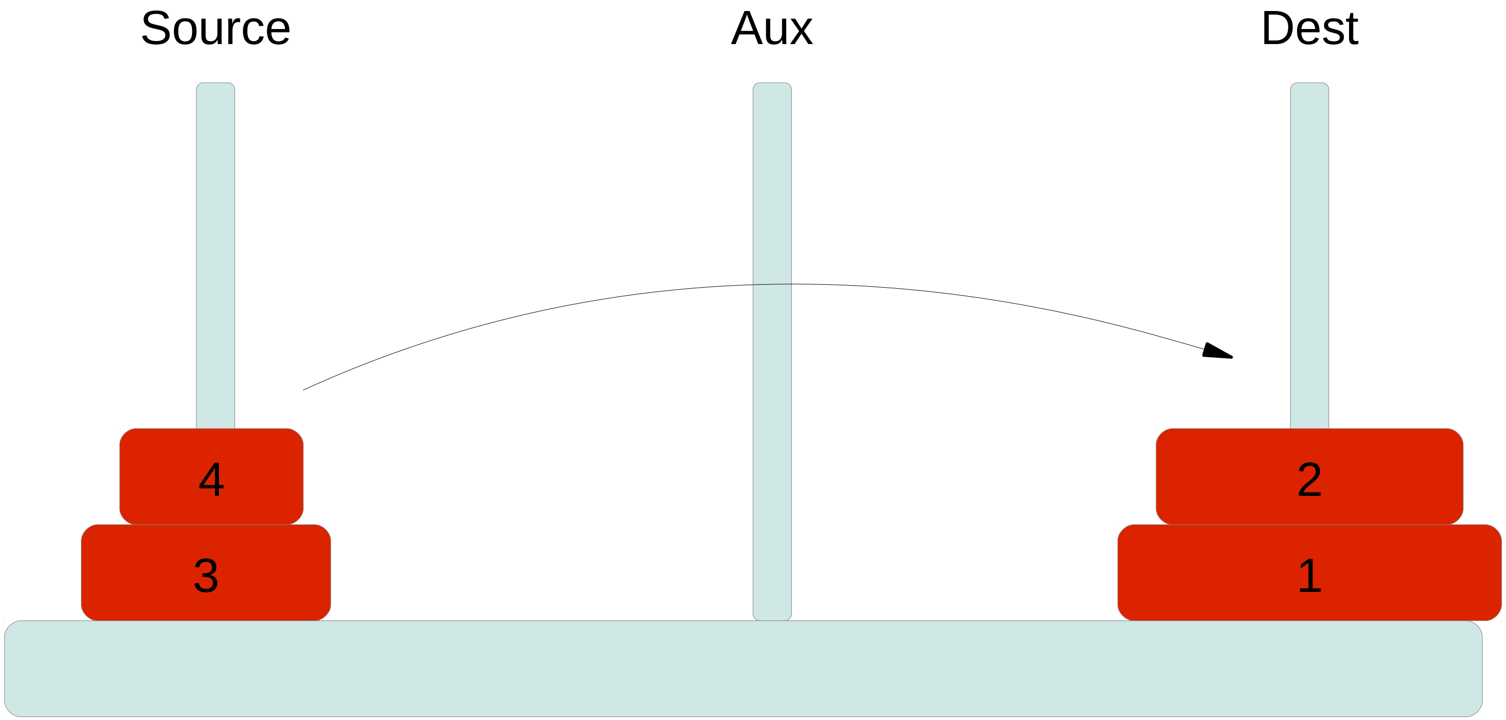
Moving 3,4 from middle tower to left tower

Recursive Move



Can Now move 2 to the Right Tower

Recursive Move

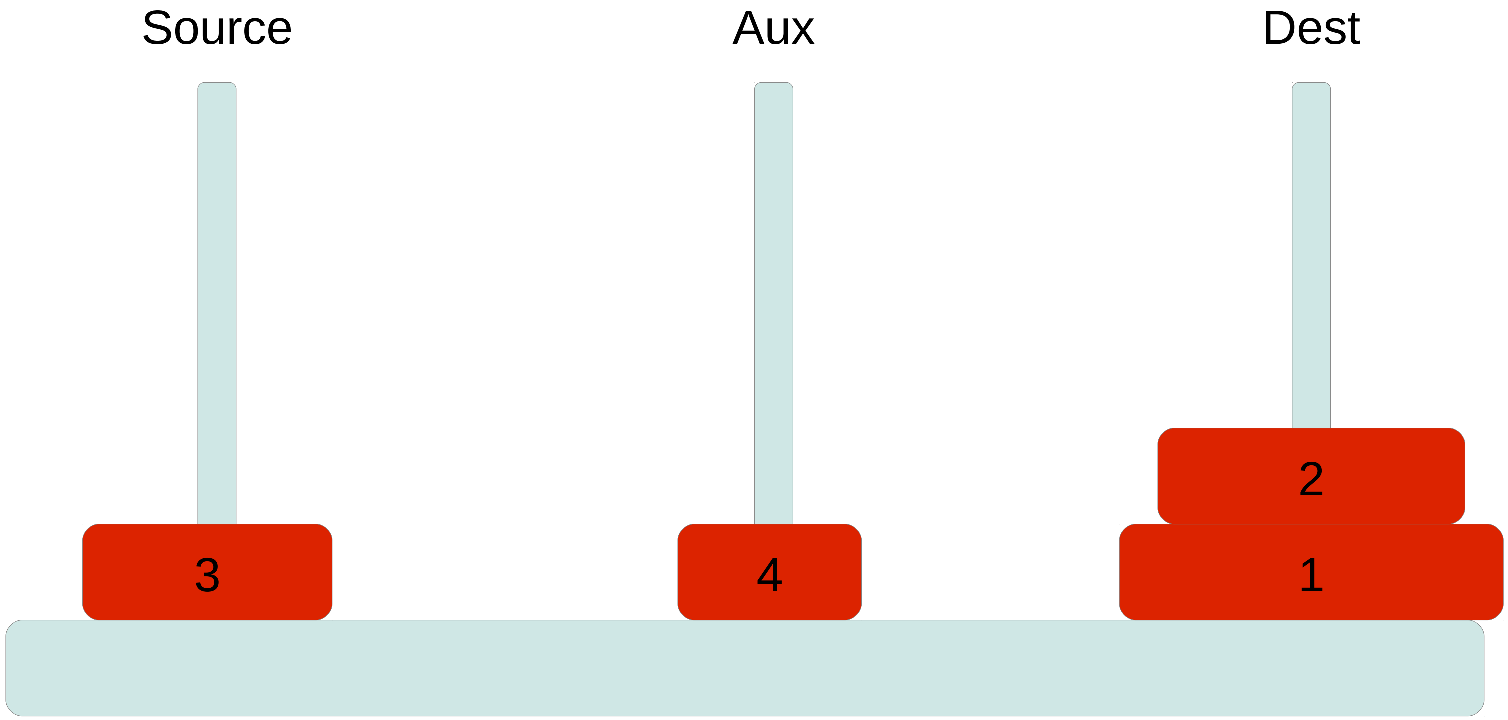


Move 3,4 to the Right Tower

= Move 4 to middle + move 3 to the Right

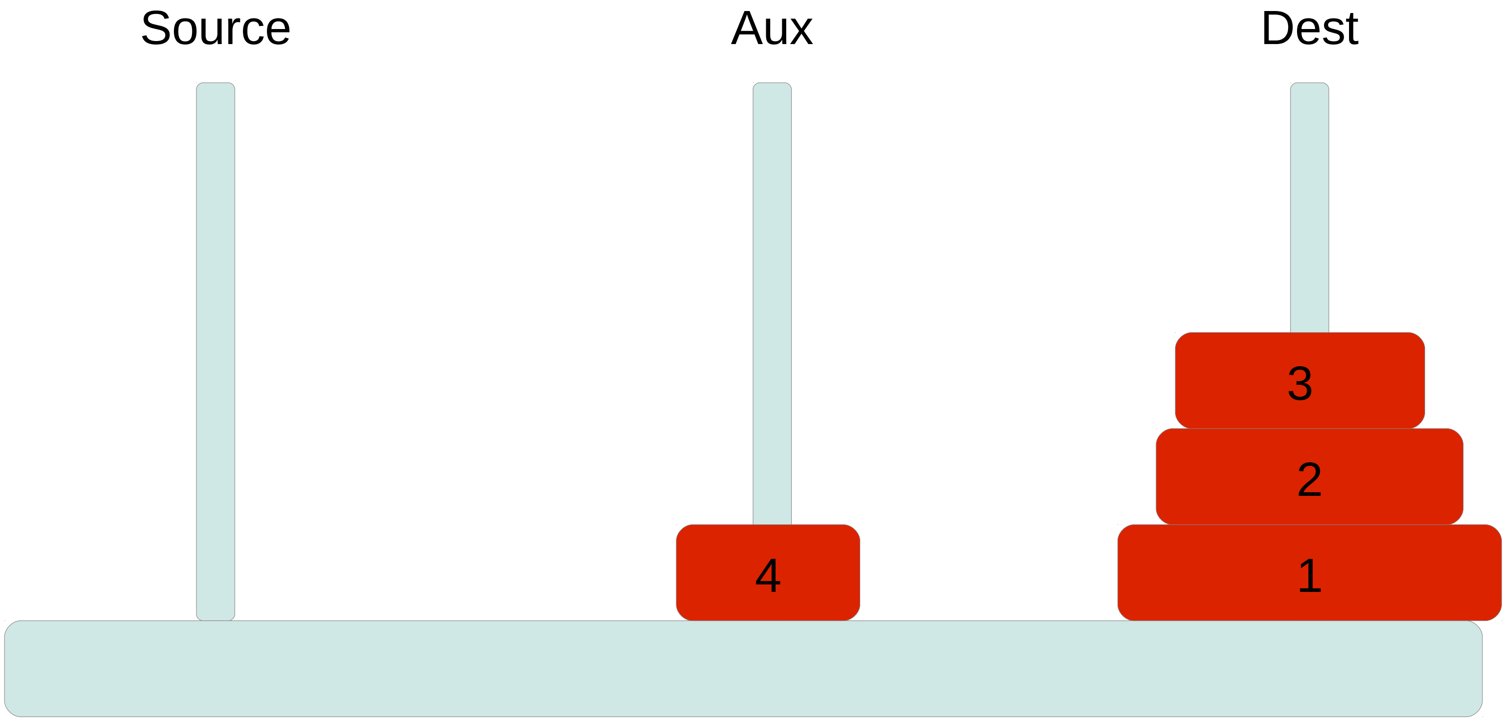
Then move 4 to the right

Recursive Move



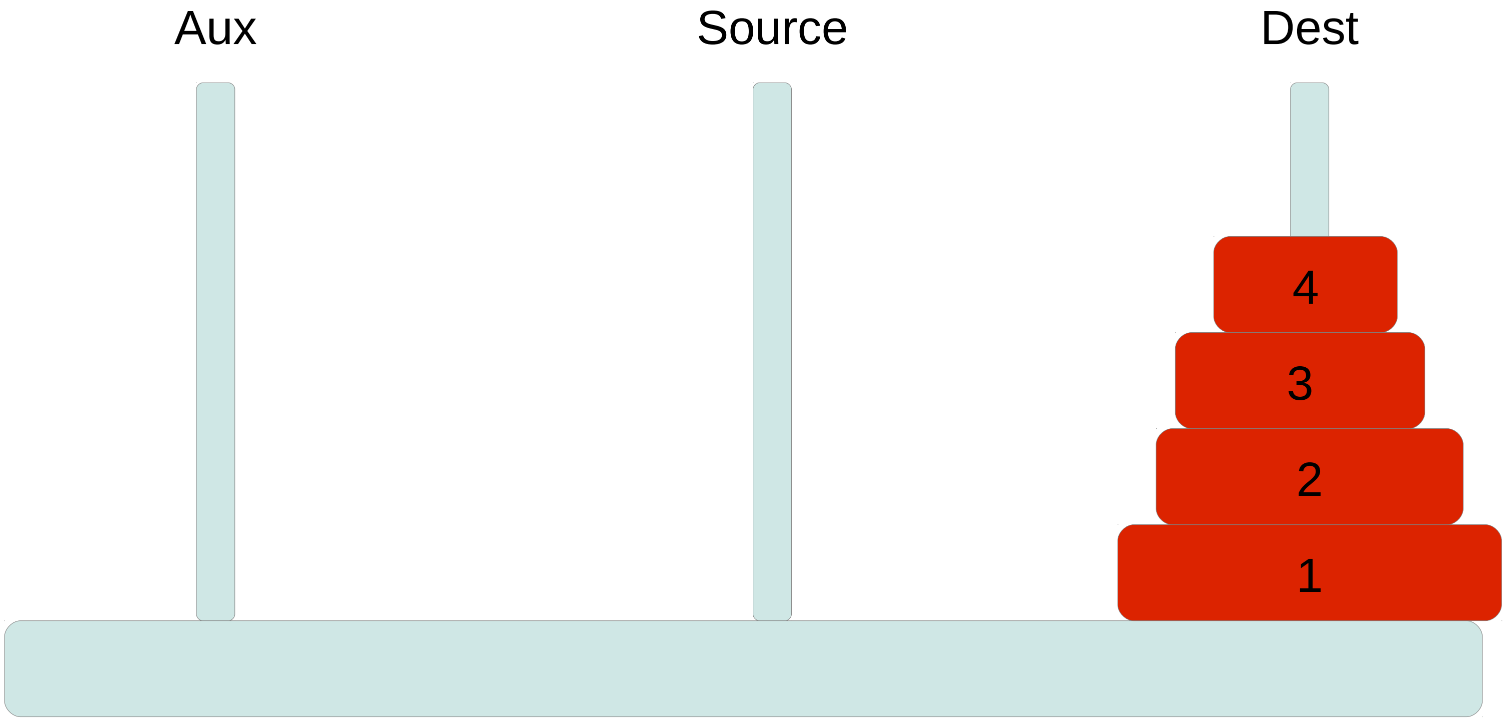
Finishing

Recursive Move



Move 4 to the Right (No recursion needed)

Recursive Move



Move all disks from left to right peg
Move one at a time
Can only put smaller disks on empty or larger disks