Lab

- http://www.youtube.com/watch?v=8TMBjfS8wY0&list=PL5405A3FD0F2046CD&index=14

- Implement Bubble Sort with Stacks

- Load StackBubbleSortLab.java into your ch03 materials in eclipse.

- Lets talk sorts, Wikipedia and comments.

Chapter 4: Recursion

4.1 Recursive Definitions, Algorithms and Programs

http://www.youtube.com/watch?v=fpuWkZs51aM

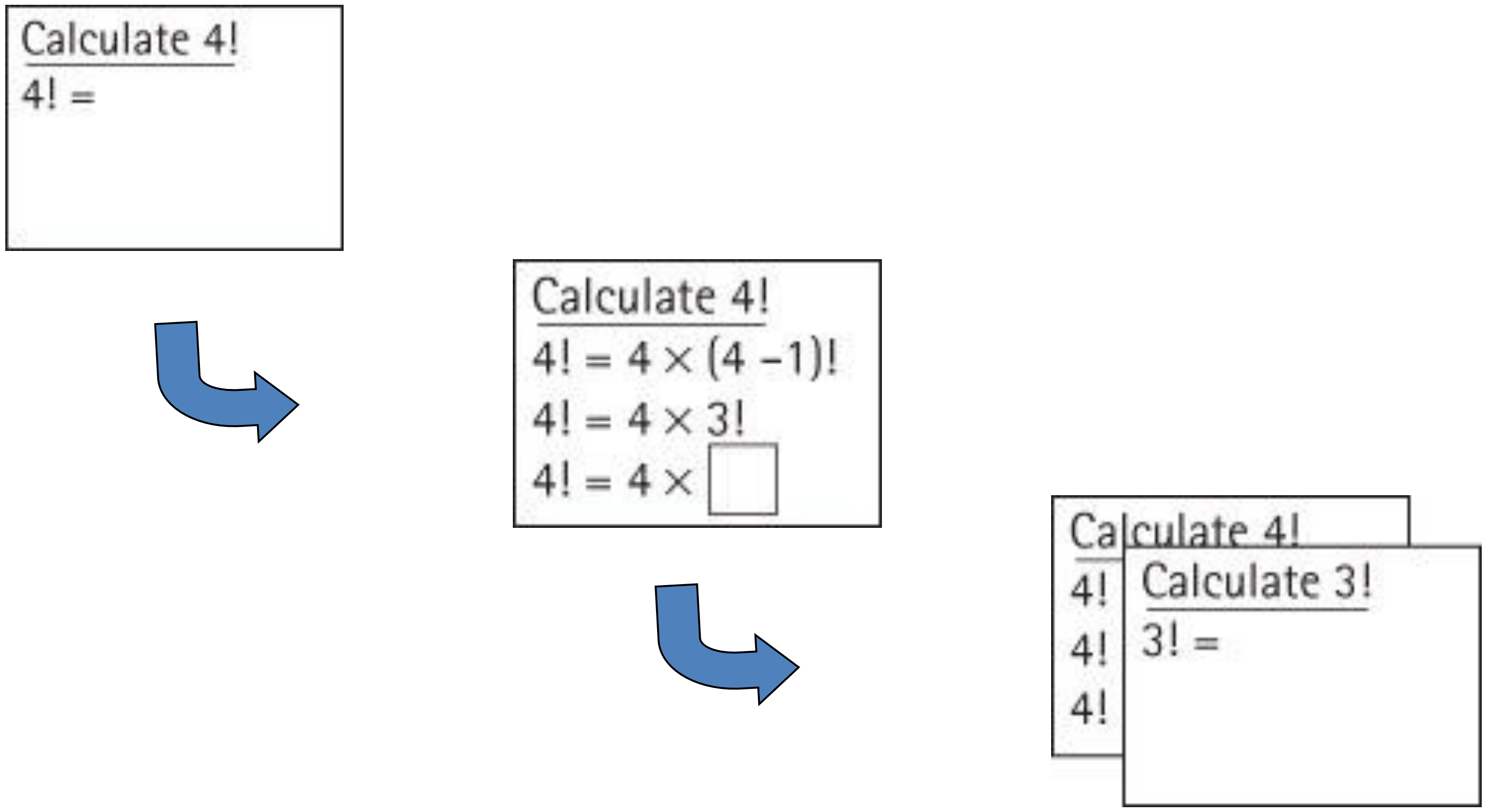
Recursive Definitions

- **Recursive definition** A definition in which something is defined in terms of smaller versions of itself. For example:
  - A *directory* is an entity in a file system which contains a group of files and other *directories*.
  - A *compound sentence* is a *sentence* that consists of two *sentences* joined together by a coordinating conjunction.
  - Example: Factorial

    n! = 1 if n = 0

    n X (n – 1)! if n > 0

Example: Calculate 4!



Calculate 4!

4! =

Calculate 4!

$4! = 4 \times (4-1)!$

$4! = 4 \times 3!$

$4! = 4 \times \boxed{}$

Calculate 4!

4!

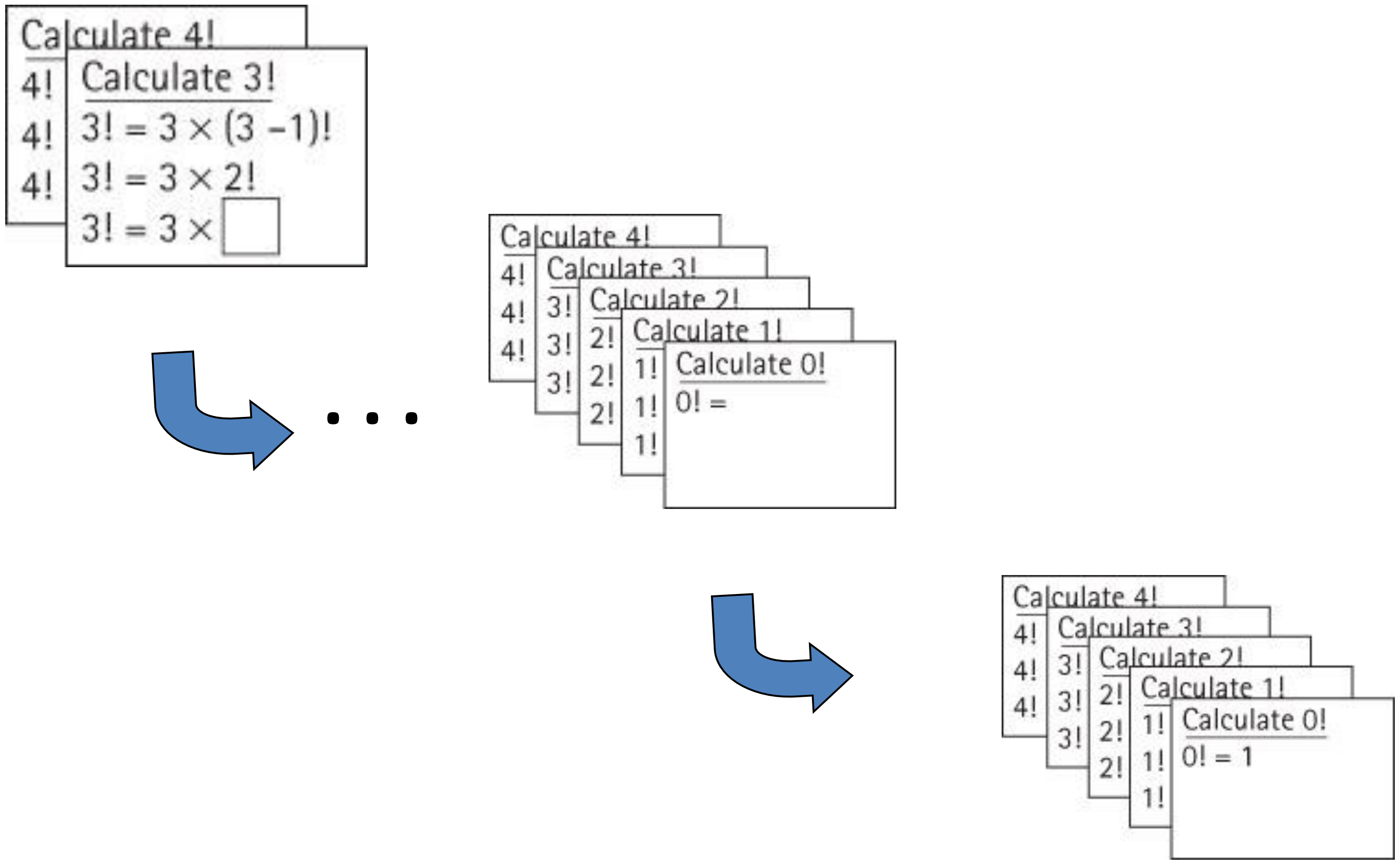
4!

4!

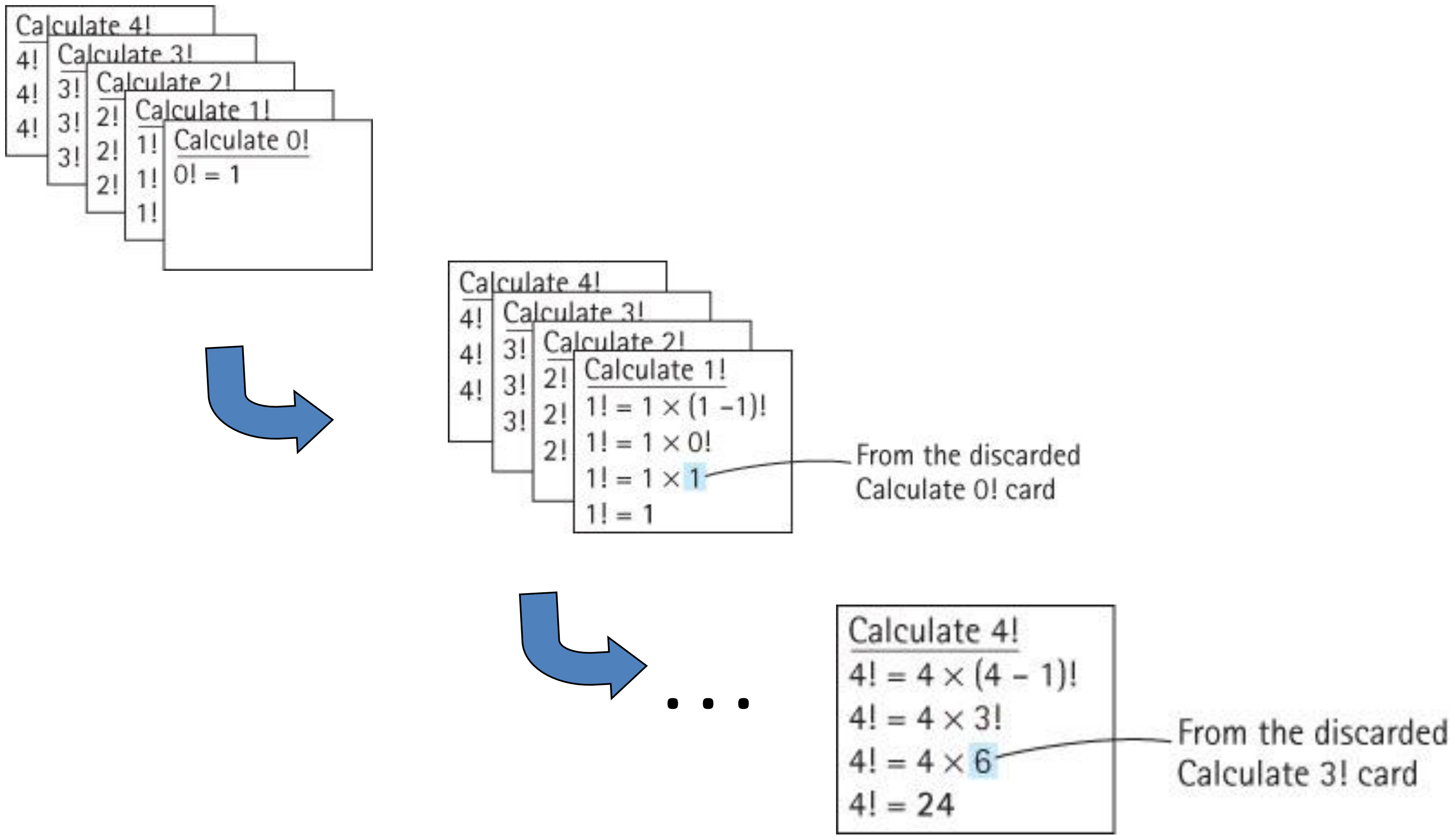
Calculate 3!

3! =

Example: Calculate 4!



Example: Calculate 4!



Recursive Algorithms

- **Recursive algorithm**
  - A solution that is expressed in terms of
    - smaller instances of itself and
    - a base case
- **General (recursive) case**
  - The solution is expressed in terms of a smaller version of itself
- **Base case**
  - The solution can be stated non-recursively

Examples of a Recursive Algorithm and a Recursive Program

```
Factorial (int n)
// Assume n >= 0
if (n == 0)
  return (1)
else
  return ( n * Factorial ( n – 1 ) )
```

```java
public static int factorial(int n)
// Precondition: n is non-negative
//
// Returns the value of "n!".
{
  if (n == 0)
    return 1;          // Base case
  else
    return (n * factorial(n - 1));     // General case
}
```

See tryFact.java

Augmented recursive factorial

```
private static int factorial(int n)
// Precondition: n is non-negative
//
// Returns the value of "n!".
{
  int retValue;   // return value
  System.out.println(indent + "Enter factorial " + n);
  indent = indent + "  ";
  if (n == 0)
    retValue = 1;
  else
    retValue = (n * factorial (n - 1));

  indent = indent.substring(2);
  System.out.println(indent + "Return " + retValue);

  return(retValue);
}
```

Output if argument is 9

```
Enter factorial 9
  Enter factorial 8
    Enter factorial 7
      Enter factorial 6
        Enter factorial 5
          Enter factorial 4
            Enter factorial 3
              Enter factorial 2
                Enter factorial 1
                  Enter factorial 0
                  Return 1
                Return 1
              Return 2
            Return 6
          Return 24
        Return 120
      Return 720
    Return 5040
  Return 40320
Return 362880
```

Recursion Terms

- **Recursive call** A method call in which the method being called is the same as the one making the call
- **Direct recursion** Recursion in which a method directly calls itself, like the factorial method.
- **Indirect recursion** Recursion in which a chain of two or more method calls returns to the method that originated the chain, for example method A calls method B which in turn calls method A

Iterative Solution for Factorial

- We have used the factorial algorithm to demonstrate recursion because it is familiar and easy to visualize. In practice, one would never want to solve this problem using recursion, since a straightforward, more efficient iterative solution exists:

```
public static int factorial(int n)
{
  int value = n;
  int retValue = 1;   // return value
  while (value != 0)
  {
    retValue = retValue * value;
    value = value - 1;
  }
  return(retValue);
}
```

Which is better for factorial?

	Iterative vs.	Recursive
Tends to use	Loops	Selection Statements
Control Structure	The loop	Branching structure
Local Variables	Uses more	Uses fewer
Efficiency	More efficient (Loop iteration is faster)	Less efficient (Method call is slower)

Test it! **Implement the following in tryFact.java**

Put the call to the factorial method**s** in a loop and test it.
Try 10, 100, 1000, and 10000 iterations.
Remember to…
Put in a startClock and stopClock and test the difference.
long startClock = new Date().getTime();

4.2 The Three Questions (Really four questions)

- Here we discuss three (four) questions to ask about any recursive algorithm or program.
- These questions can help us verify, design, and debug recursive solutions to problems.

Verifying Recursive Algorithms

To verify that a recursive solution works, we must be able to answer “Yes” to all three of these questions:

(1) *The Base-Case Question:*Is there a nonrecursive way out of the algorithm, and (2) does the algorithm work correctly for this base case?

(3) *The Smaller-Caller Question:* Does each recursive call to the algorithm involve a smaller case of the original problem, leading inescapably to the base case?

(4) *The General-Case Question:* Assuming the recursive call(s) to the smaller case(s) works correctly, does the algorithm work correctly for the general case?

We next apply these three questions to the factorial algorithm.

The Base-Case Question(s)

```
Factorial (int n)
// Assume n >= 0
if (n == 0)
  return (1)
else
  return ( n * Factorial ( n – 1 ) )
```

(1) Is there a nonrecursive way out of the algorithm, and (2) does the algorithm work correctly for this base case?

The base case occurs when *n* is 0. (So yes to the first question)

The Factorial algorithm then returns the value of 1, which is the correct value of 0!, and no further (recursive) calls to Factorial are made.

And yes to the second quesiton 0! = 1. 1 is returned when n = 0.

The Smaller-Caller Question

```
Factorial (int n)
// Assume n >= 0
if (n == 0)
  return (1)
else
  return ( n * Factorial ( n – 1 ) )
```

Does each recursive call to the algorithm involve a smaller case of the original problem, leading inescapably to the base case?

The parameter is *n* and the recursive call passes the argument *n - 1*. Therefore each subsequent recursive call sends a smaller value, until the value sent is finally 0.

At this point, as we verified with the base-case question, we have reached the smallest case, and no further recursive calls are made.

The answer is yes.

The General-Case Question

```
Factorial (int n)
// Assume n >= 0
if (n == 0)
  return (1)
else
  return ( n * Factorial ( n – 1 ) )
```

Assuming the recursive call(s) to the smaller case(s) works correctly, does the algorithm work correctly for the general case?

Assuming that the recursive call Factorial(n – 1) gives us the correct value of *(n - 1)!*, the *return* statement computes *n * (n - 1)!*.

This is the definition of a factorial,
so we know that the algorithm works in the general case.

The answer is yes.

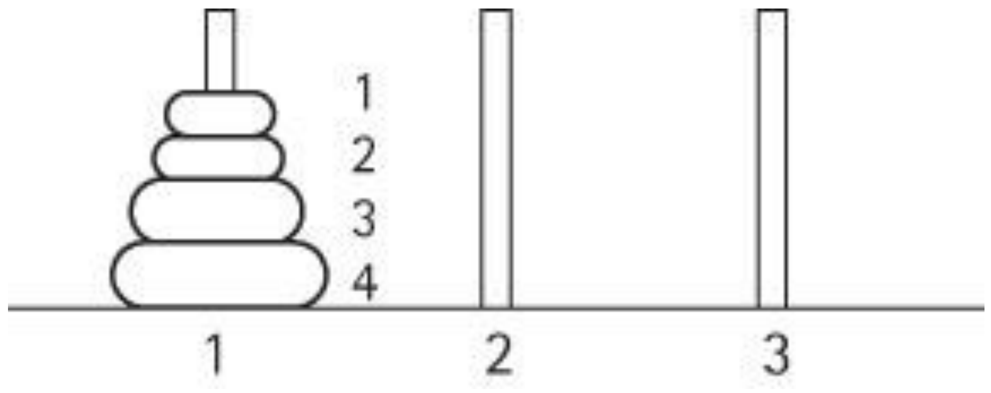
Constraints on input arguments

- Constraints often exist on the valid input arguments for a recursive algorithm.
  - For example, for Factorial, n must be >= 0.
- Use the following logic to determine constraints
  - Do any starting argument values exist such that the smaller call does not converge in on the base case.
    - These starting values are invalid!
    - If so, constrain your legal input to disallow these values

Steps for Designing Recursive Solutions

1. Get an exact definition of the problem to be solved.
2. Determine the size of the problem to be solved on each individual call to the method.
3. Identify and solve the base case(s) in which the problem can be expressed non-recursively. This ensures a yes answer to the base-case question(s).
4. Identify and solve the general case(s) correctly in terms of a smaller case of the same problem—a recursive call. This ensures yes answers to the smaller-caller and general-case questions.

4.3 Towers of Hanoi



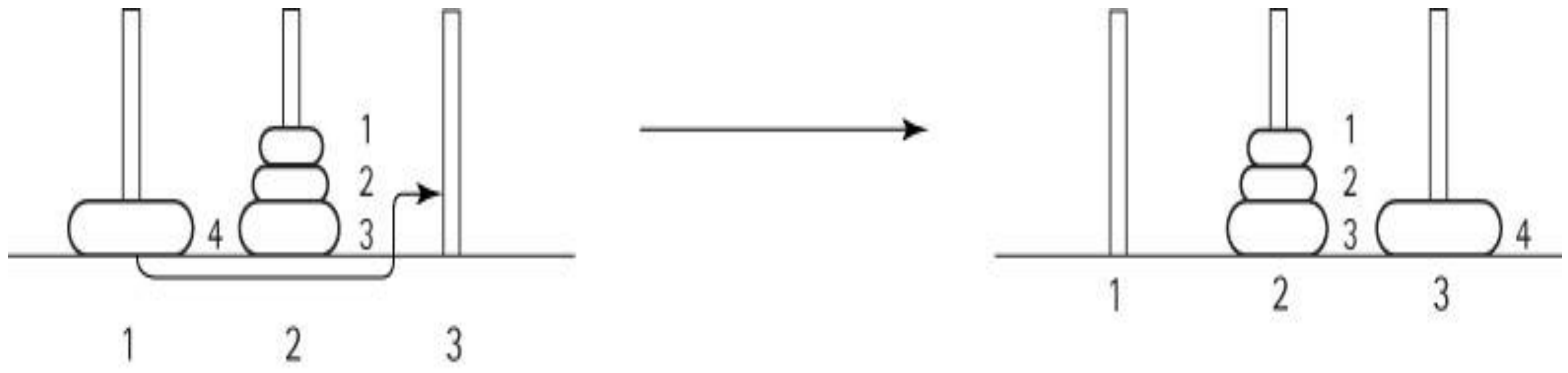
- Move the rings, one at a time, to the third peg.
- A ring cannot be placed on top of one that is smaller in diameter.
- The middle peg can be used as an auxiliary peg, but it must be empty at the beginning and at the end of the game.
- The rings can only be moved one at a time.

General Approach

To move the largest ring (ring 4) to peg 3, we must move the three smaller rings to peg 2 (this cannot be done with 1 move).

Assuming we can do this.

Then ring 4 can be moved into its final place:



Recursion

- Can you see that our assumption (that we move the three smaller rings to peg 2) involved solving a smaller version of the problem? We have solved the problem using recursion.

- The general recursive algorithm for moving *n* rings from the starting peg to the destination peg 3 is:

```
Move n rings from Starting Peg to Destination Peg
Move n - 1 rings from starting peg to auxiliary peg
Move the nth ring from starting peg to destination peg
Move n - 1 rings from auxiliary peg to destination peg
```

Recursive Method

```
public static void doTowers(
   int n,              // Number of rings to move
   int startPeg,       // Peg containing rings to move
   int auxPeg,         // Peg holding rings temporarily
   int endPeg      )   // Peg receiving rings being moved
{
  if (n > 0)
  {
    // Move n - 1 rings from starting peg to auxiliary peg
    doTowers(n - 1, startPeg, endPeg, auxPeg);

    System.out.println("Move ring from peg " + startPeg
             + " to peg " + endPeg);

    // Move n - 1 rings from auxiliary peg to ending peg
    doTowers(n - 1, auxPeg, startPeg, endPeg);
  }
}
```

Code and Demo

- Then exam review…