CSEN 202 – Introduction to Computer Programming

Lecture 5:
Methods and recursion

Prof. Dr. Slim Abdennadher,
Dr. Wael Abouelsaadat and
Dr Mohammed Abdel Megeed Salem
slim.abdennadher@guc.edu.eg

German University Cairo, Faculty of Media Engineering and Technology

March 11 - March 16, 2017

Why methods

Methods can be used to sub-divide an algorithm into smaller tasks

- To make the code better manageable
  - correctness, ease of debugging, extensibility
- To add clarity
  - separation of concerns, coherence, conceptual clarity, re-use of standard algorithms
- To make the code more compact
  - reuse of functionality in multiple places

How do methods work?

- Methods have two sides
  - Their definition, and their invocation.
- When invoked, data can be passed as arguments into the method. After execution, the method may return a value

Distinguishing methods

A method is uniquely identified by its signature.

- The signature consists of
  - The name of the method, and
  - The number, type, and position of its arguments

  Example:

```
public static int gcd (int a, long b) {...}
```

- The return type does not distinguish methods! The names of the arguments also do not distinguish methods!

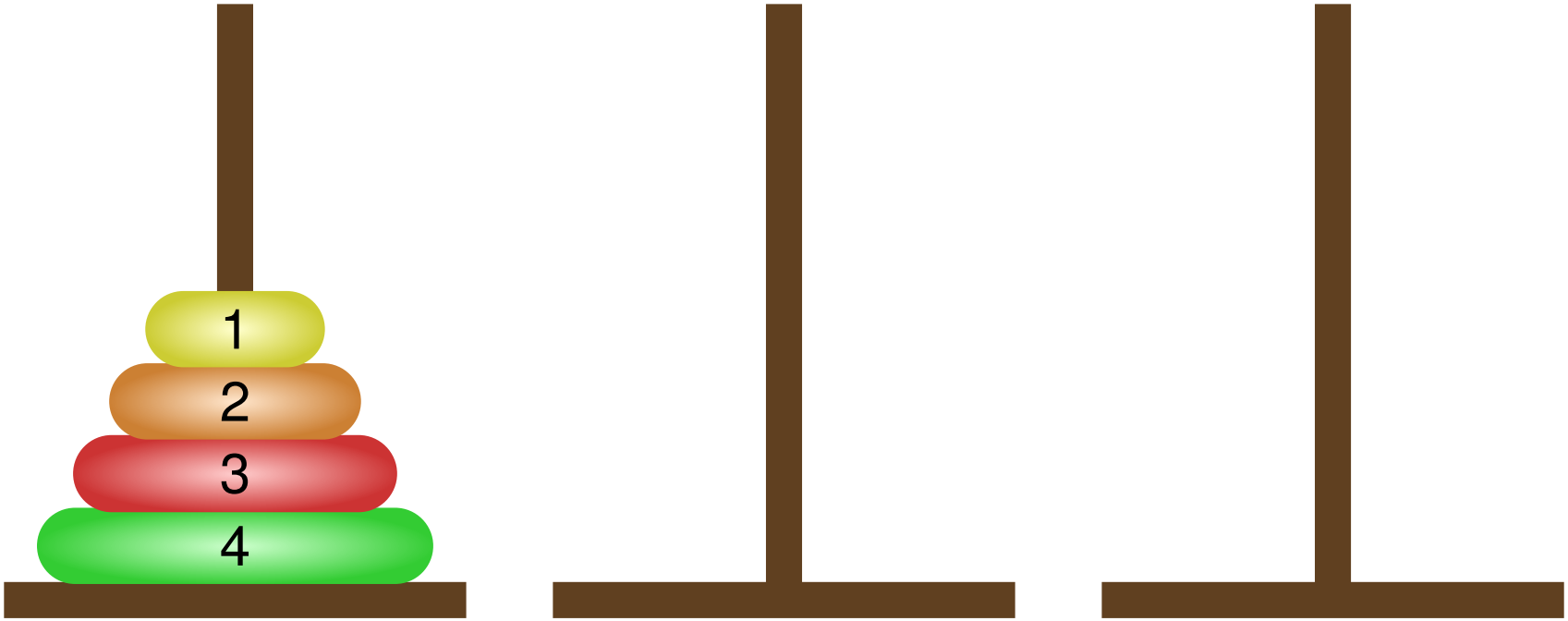
  Example (⚠):

```
public static double foo (int a, int b) {...}
public static int foo (int x, int y) {...}
```

- Creating multiple methods with the same name but different signatures is called overloading

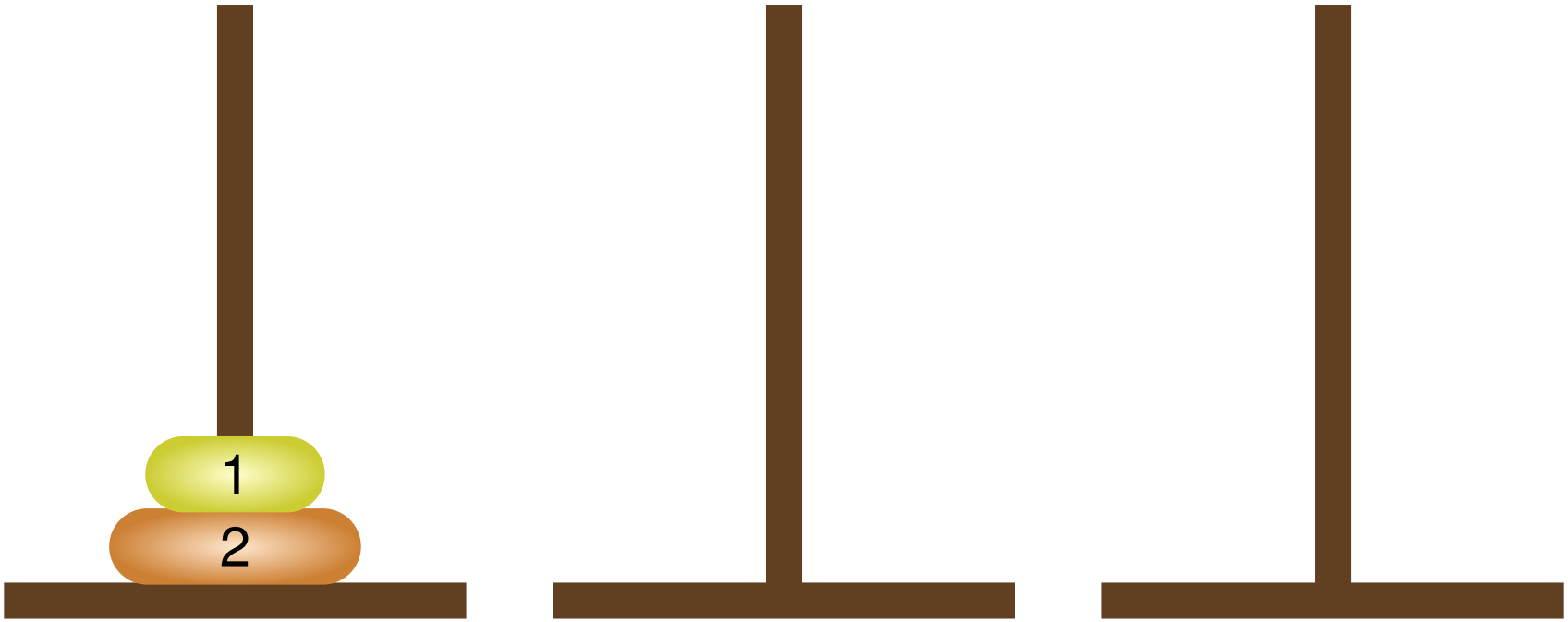
recursion

The towers of Hanoi



- Migrate all disks from the first to the third pole
- Move one disk at a time
- Never place a larger disk on top of a smaller disk

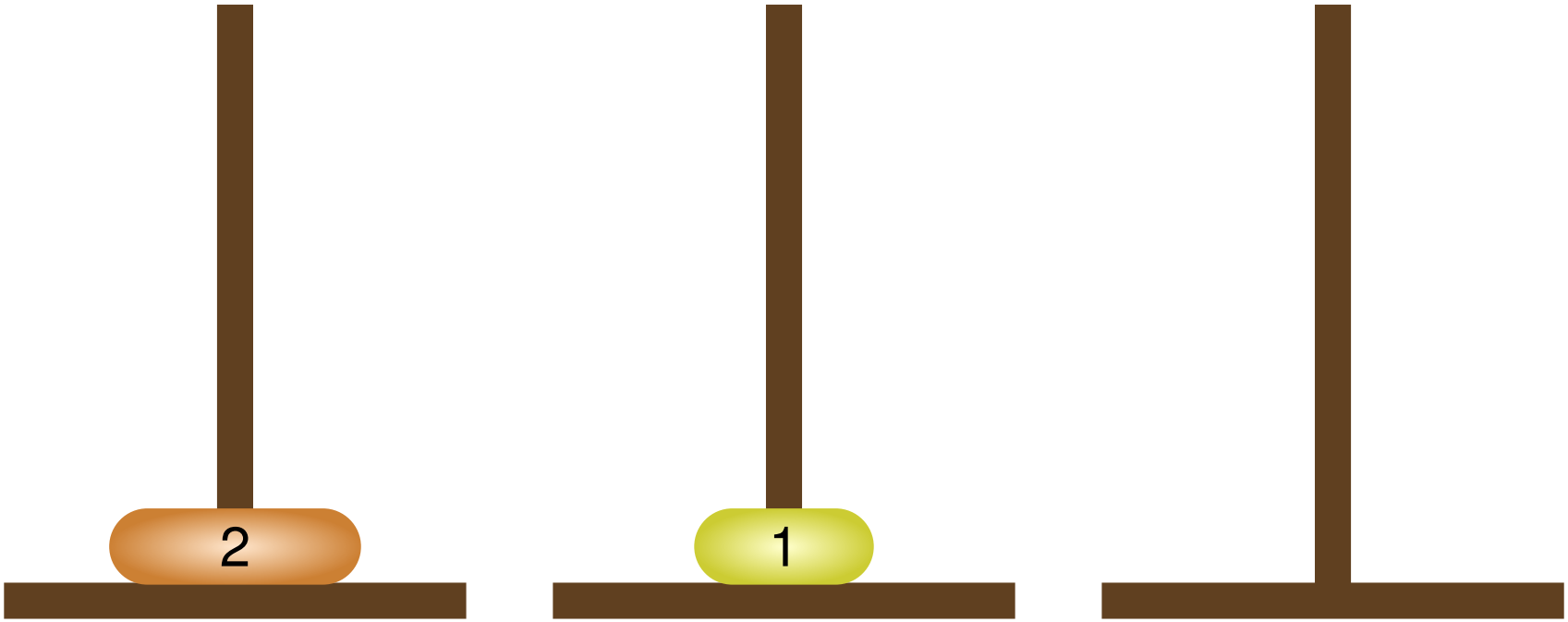
Example: 2 Discs



Graphics and implementation by Martin Hofmann and Berteun Damman

(see `http://www.texample.net/tikz/examples/towers-of-hanoi/`)

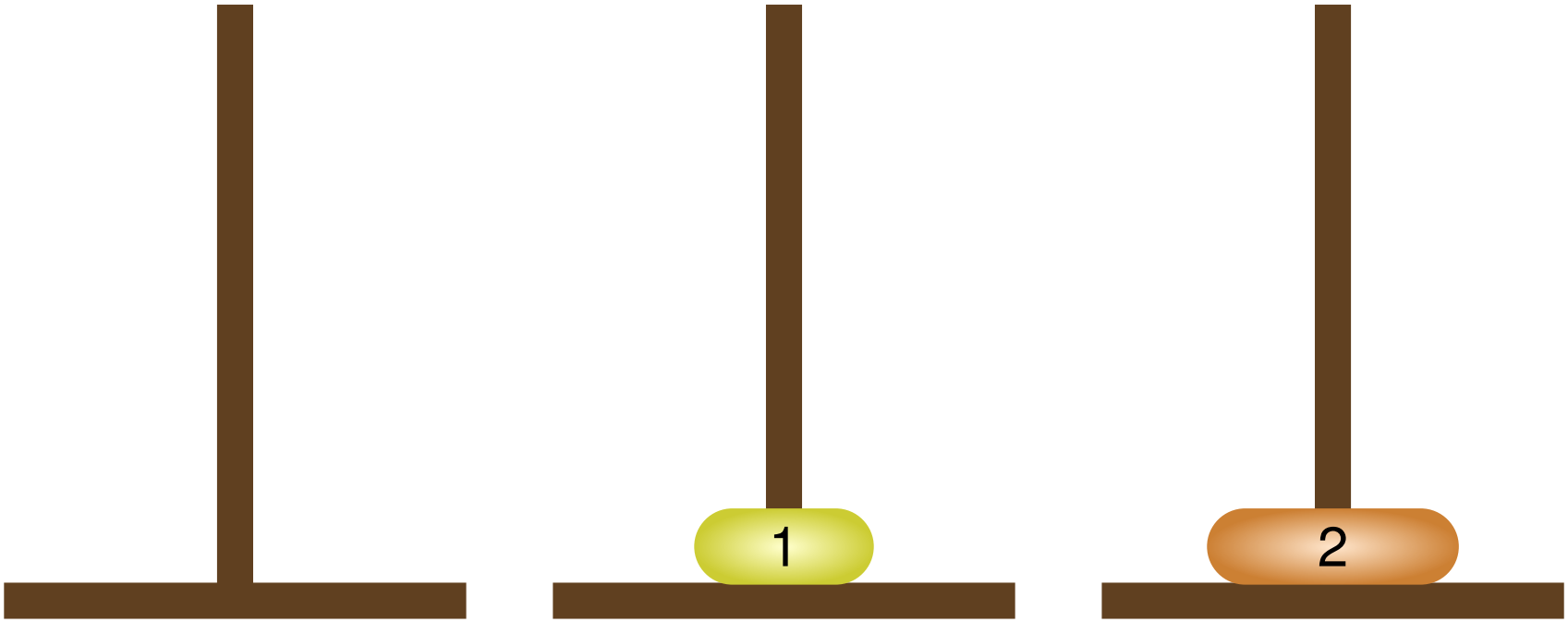
Example: 2 Discs



Graphics and implementation by Martin Hofmann and Berteun Damman

(see `http://www.texample.net/tikz/examples/towers-of-hanoi/`)

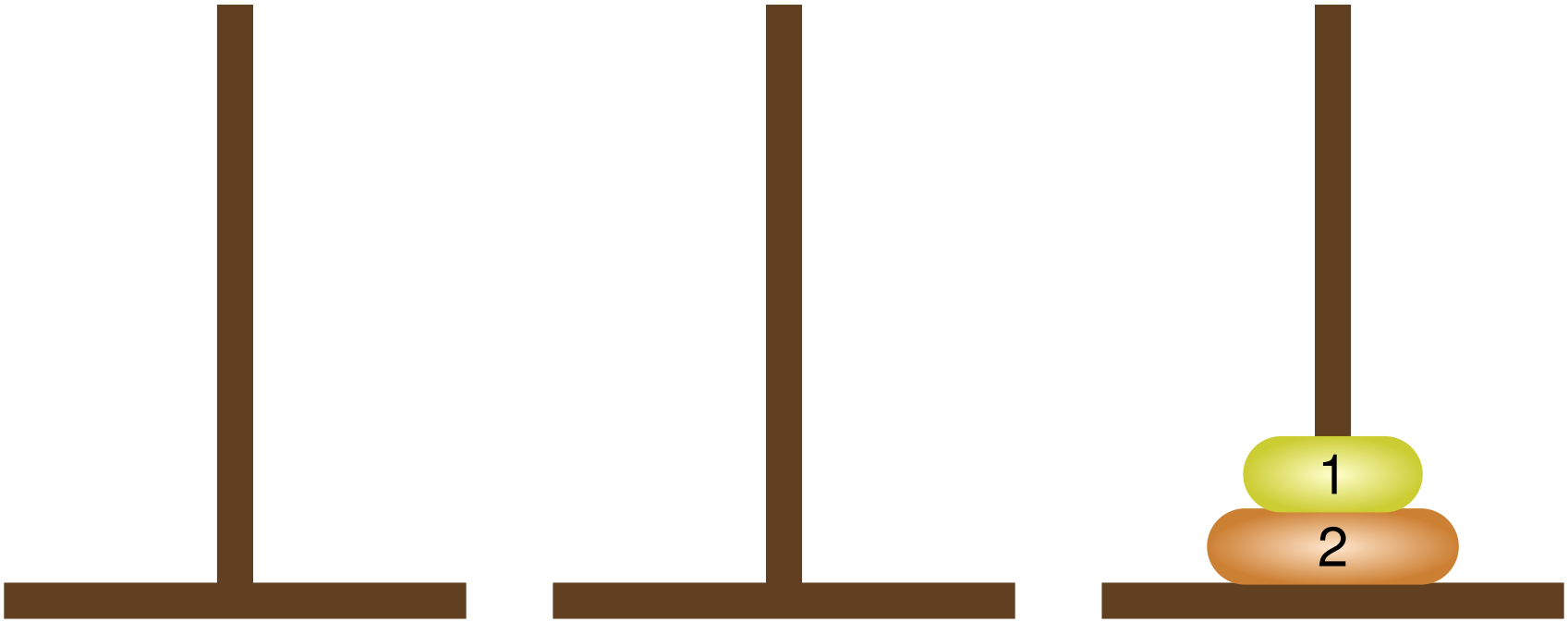
Example: 2 Discs



Graphics and implementation by Martin Hofmann and Berteun Damman

(see `http://www.texample.net/tikz/examples/towers-of-hanoi/`)

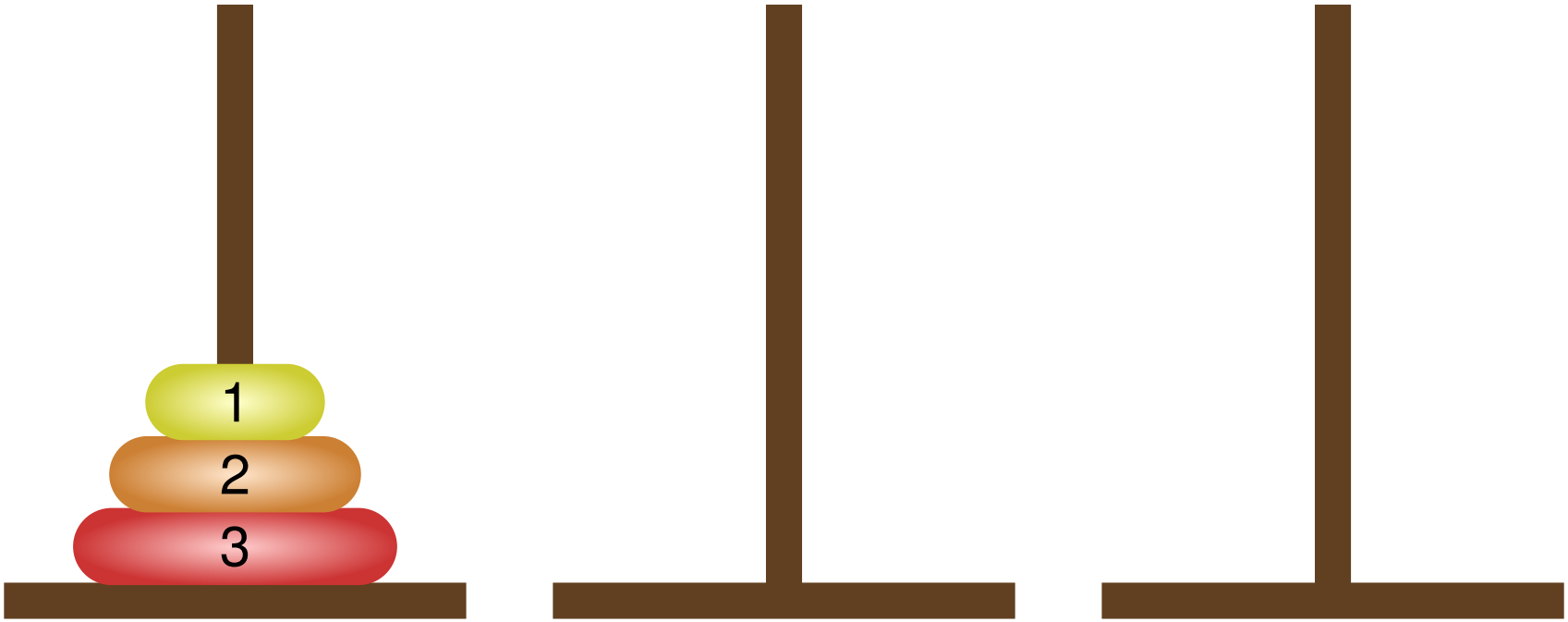
Example: 2 Discs



Graphics and implementation by Martin Hofmann and Berteun Damman

(see `http://www.texample.net/tikz/examples/towers-of-hanoi/`)

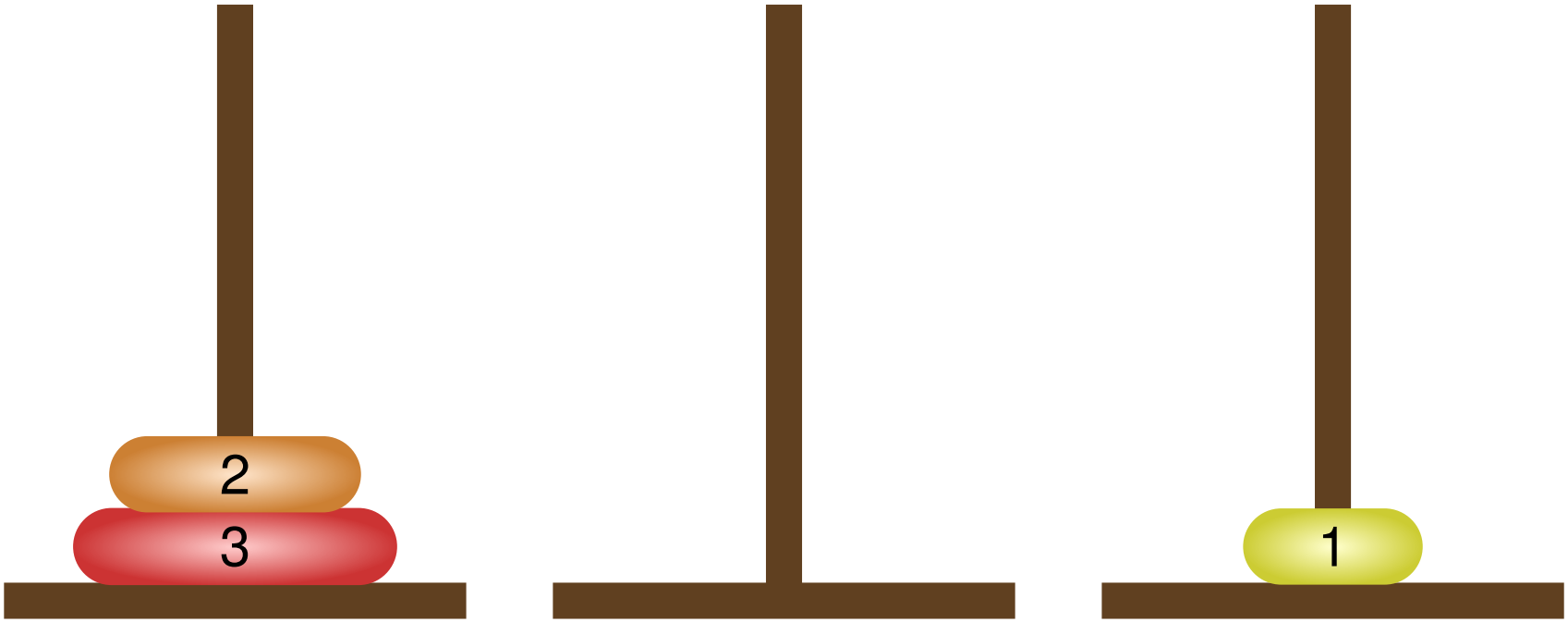
Example: 3 Discs



Graphics and implementation by Martin Hofmann and Berteun Damman

(see `http://www.texample.net/tikz/examples/towers-of-hanoi/`)

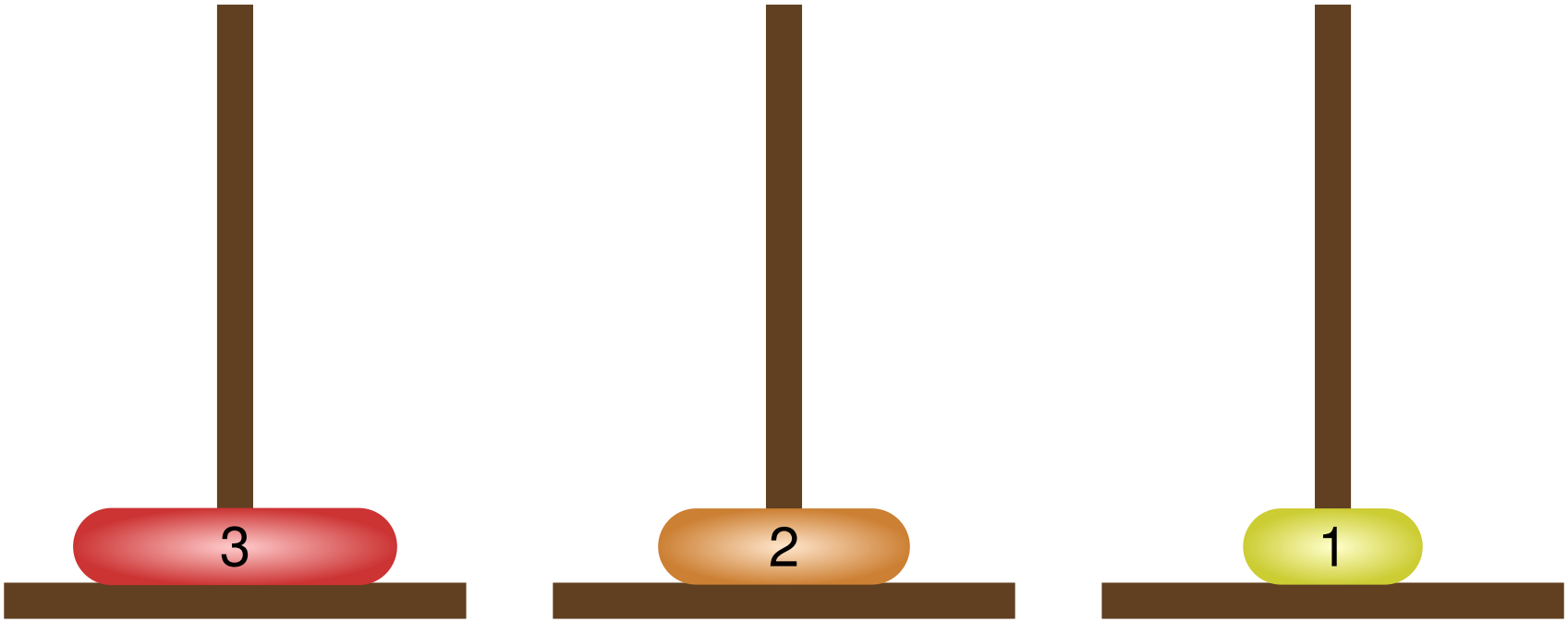
Example: 3 Discs



Graphics and implementation by Martin Hofmann and Berteun Damman

(see `http://www.texample.net/tikz/examples/towers-of-hanoi/`)

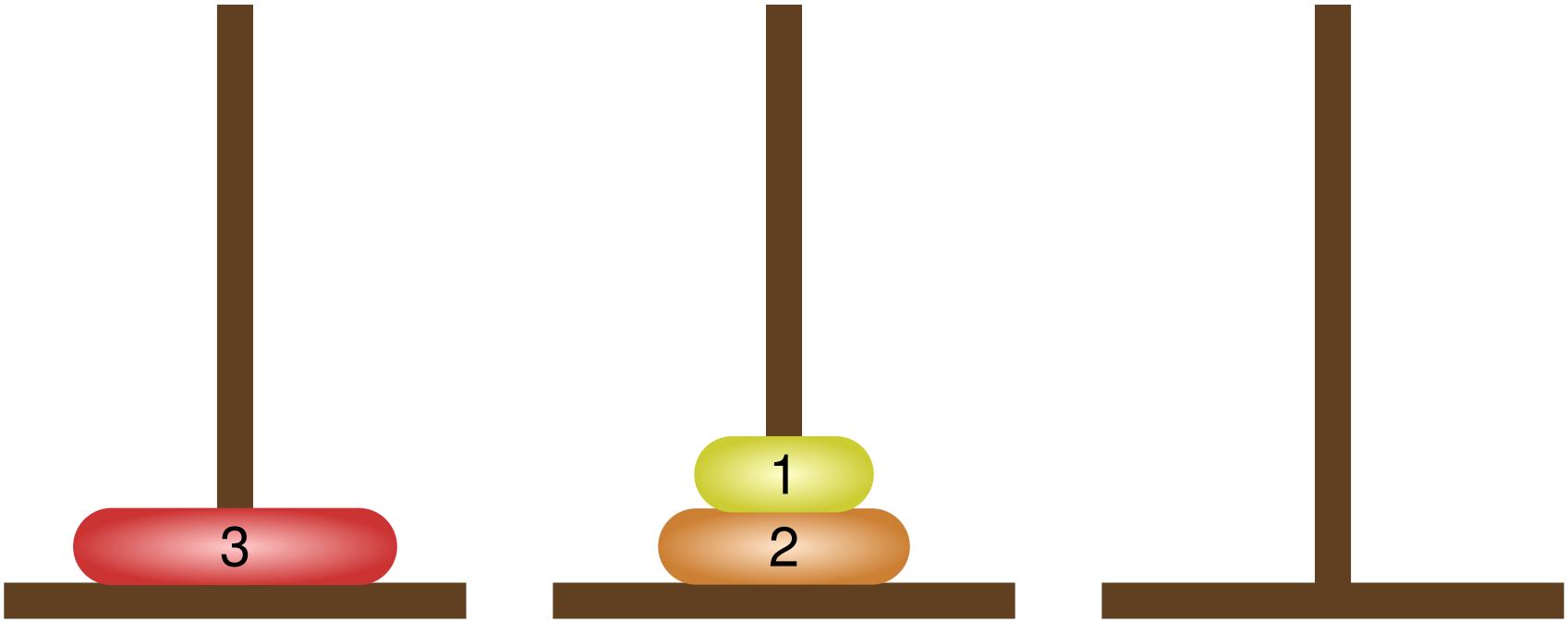
Example: 3 Discs



Graphics and implementation by Martin Hofmann and Berteun Damman

(see `http://www.texample.net/tikz/examples/towers-of-hanoi/`)

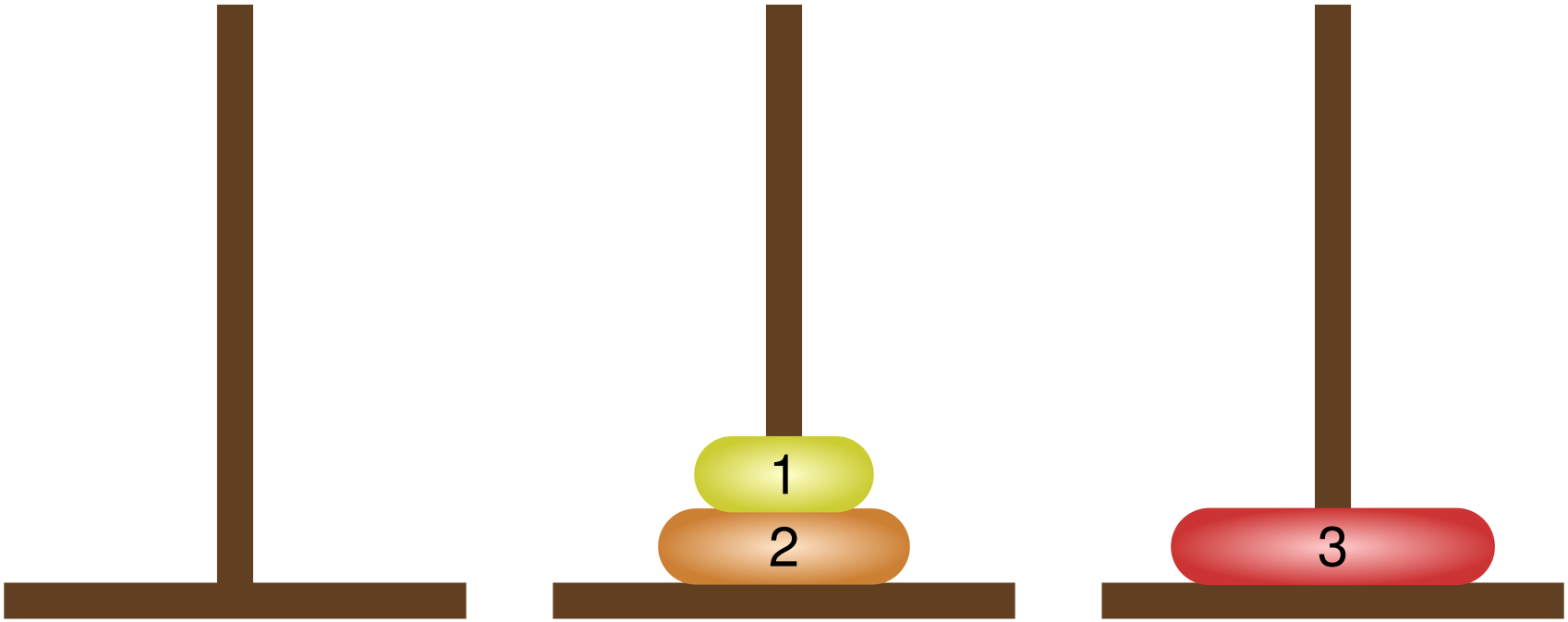
Example: 3 Discs



Graphics and implementation by Martin Hofmann and Berteun Damman

(see `http://www.texample.net/tikz/examples/towers-of-hanoi/`)

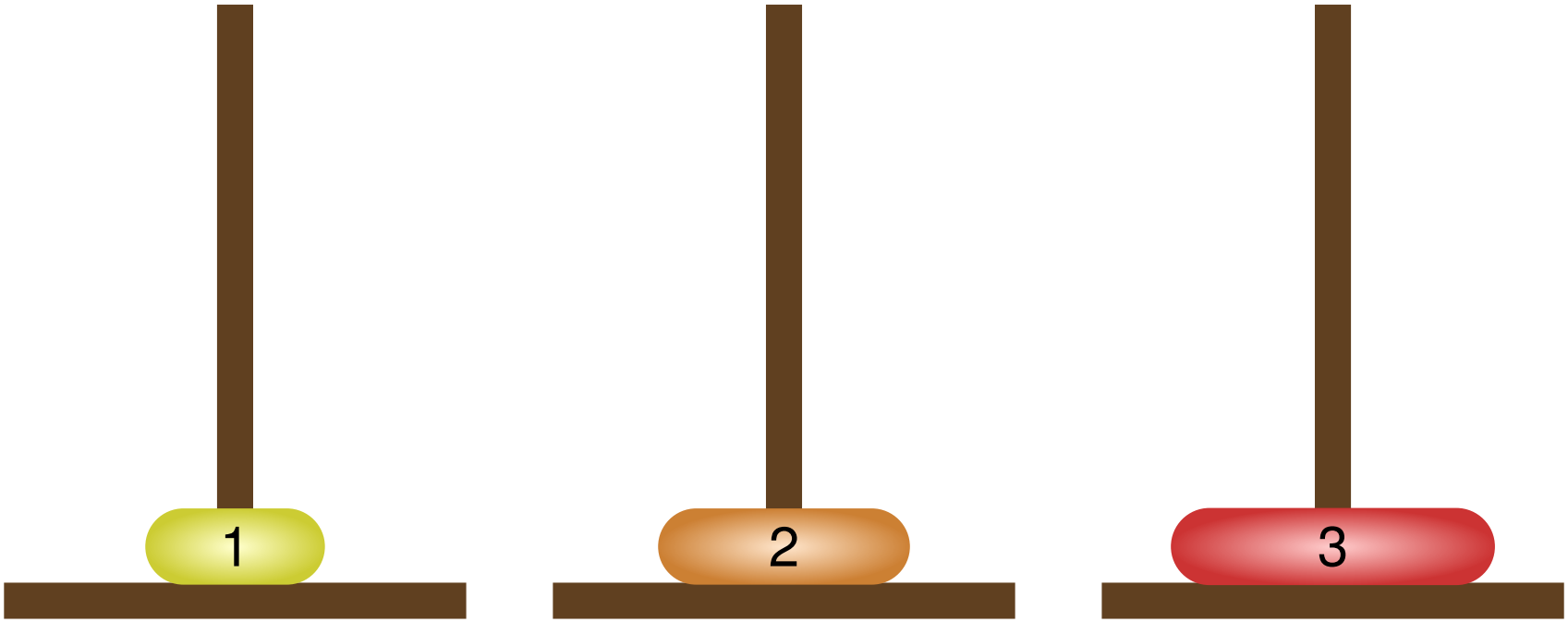
Example: 3 Discs



Graphics and implementation by Martin Hofmann and Berteun Damman

(see `http://www.texample.net/tikz/examples/towers-of-hanoi/`)

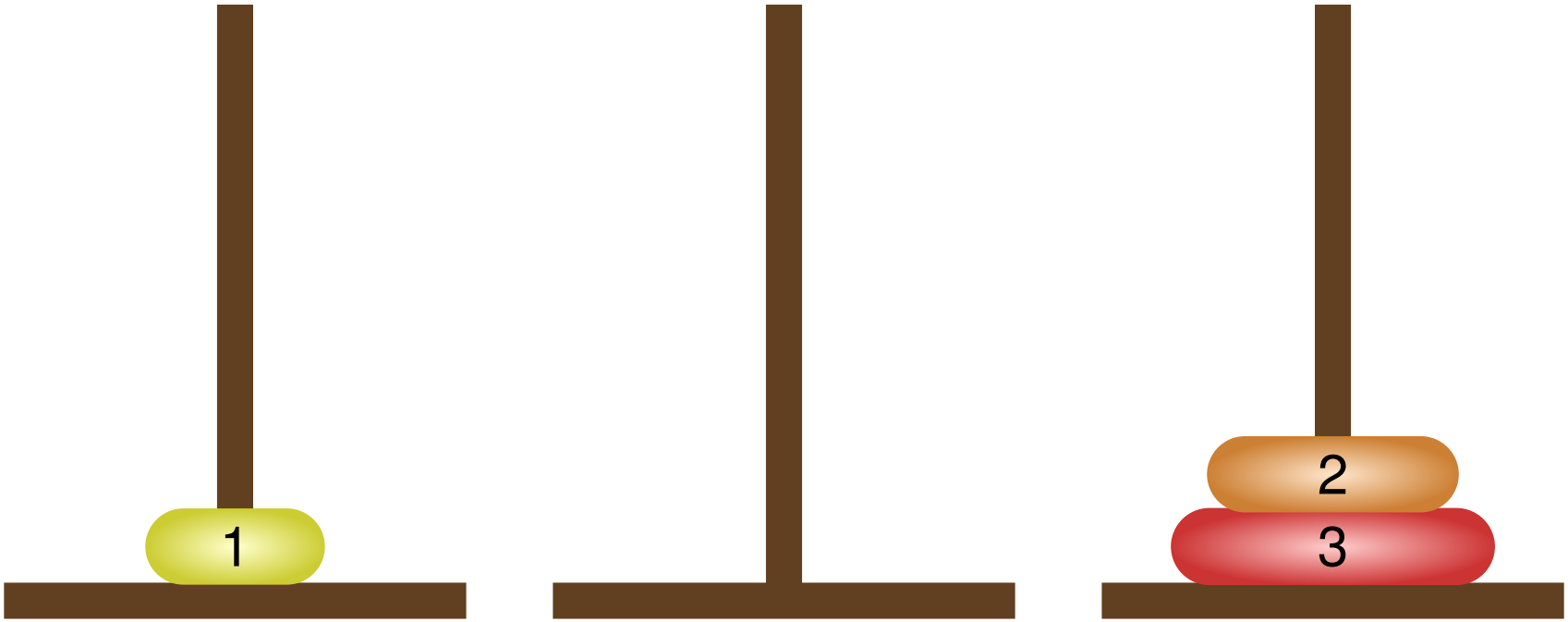
Example: 3 Discs



Graphics and implementation by Martin Hofmann and Berteun Damman

(see `http://www.texample.net/tikz/examples/towers-of-hanoi/`)

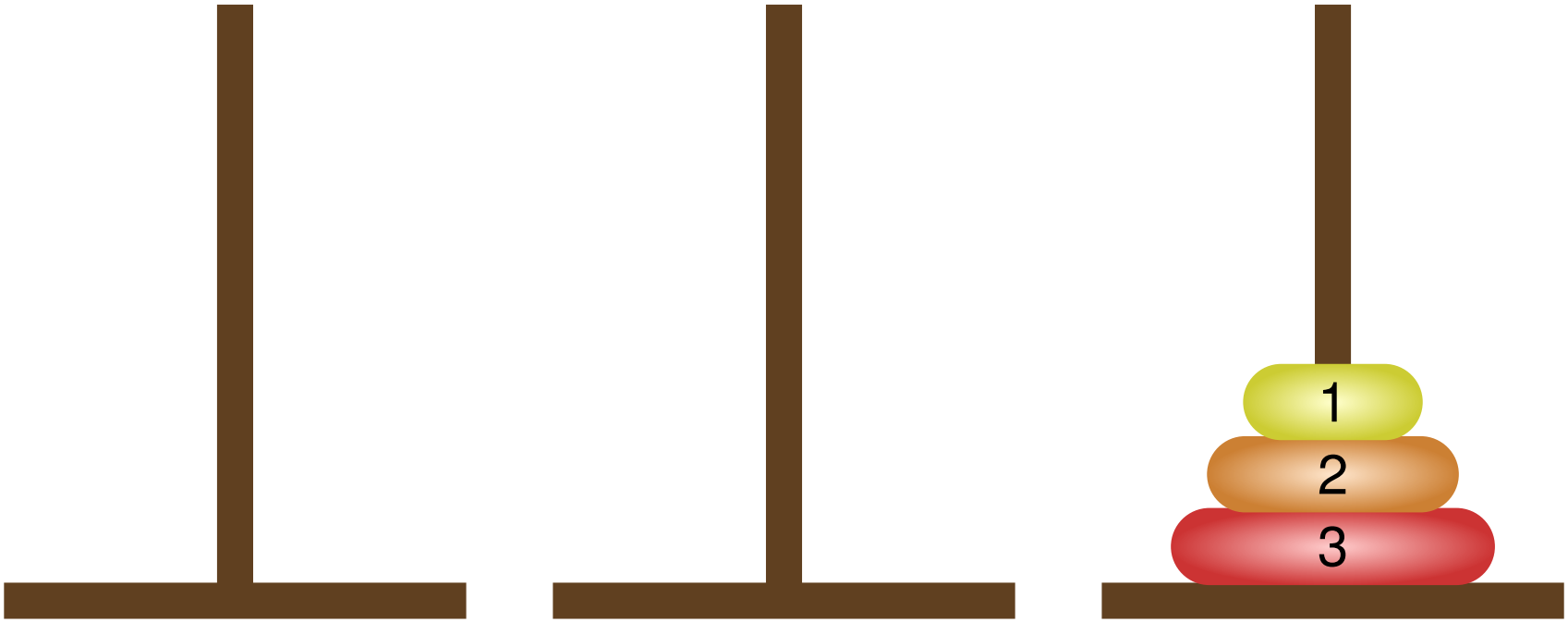
Example: 3 Discs



Graphics and implementation by Martin Hofmann and Berteun Damman

(see `http://www.texample.net/tikz/examples/towers-of-hanoi/`)

Example: 3 Discs

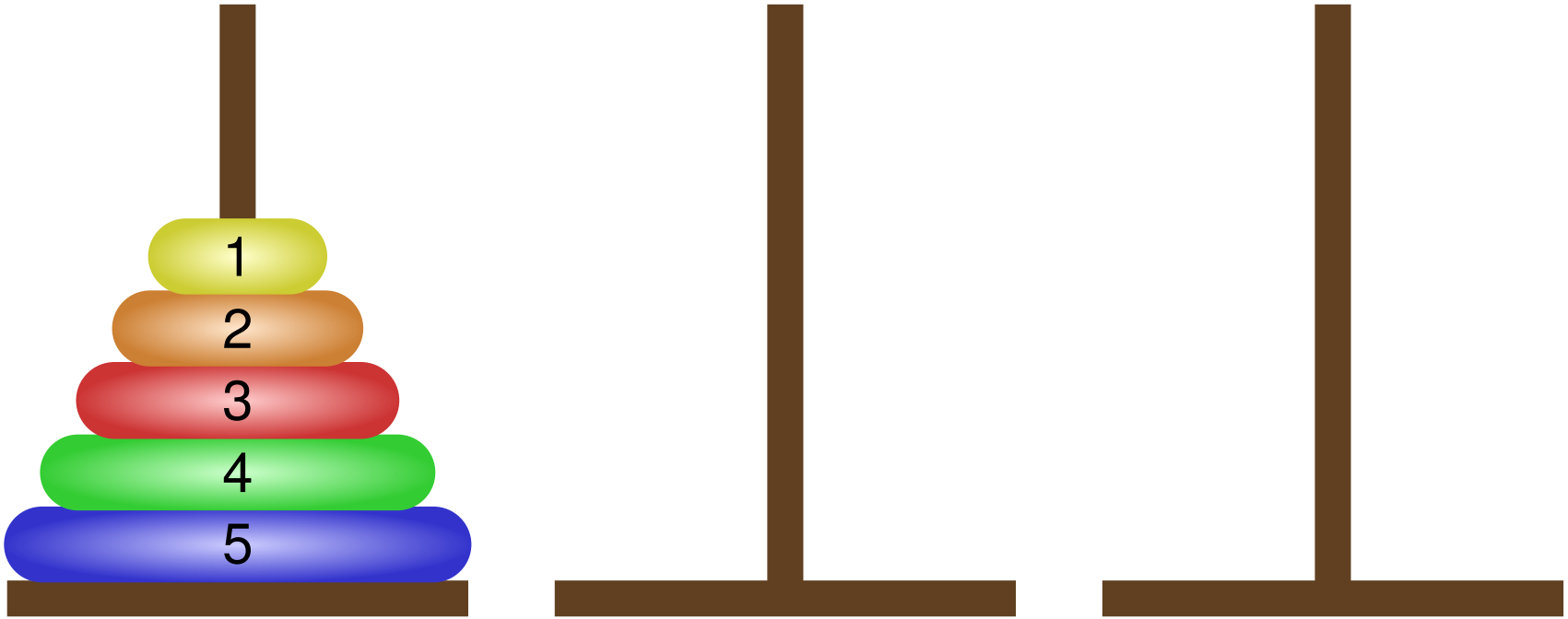


Graphics and implementation by Martin Hofmann and Berteun Damman

(see `http://www.texample.net/tikz/examples/towers-of-hanoi/`)

Problem:

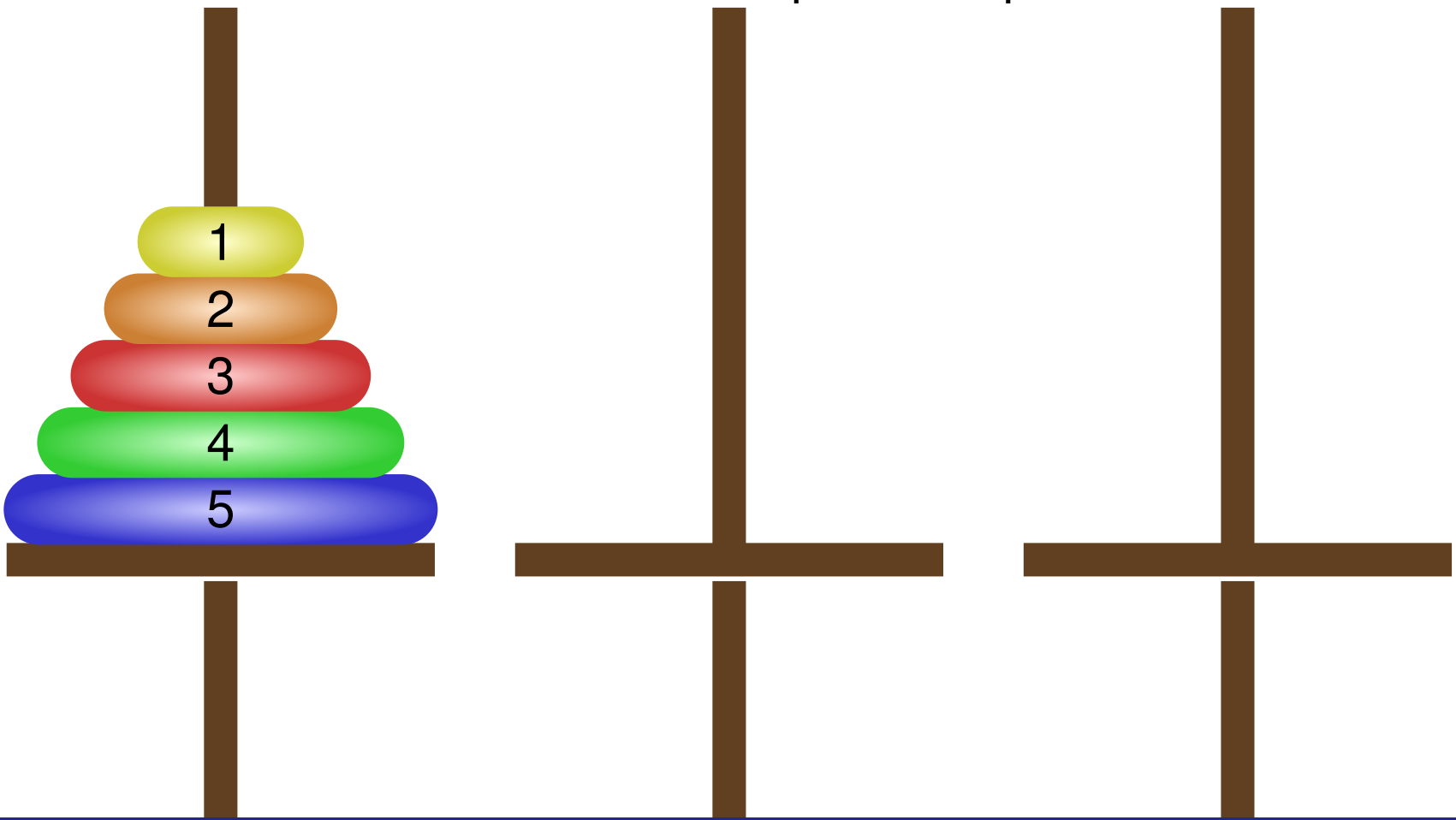
Now imagine a tower with five disks



Where to start??

Idea

To move a tower of five disks from pole 1 to pole 3



Being lazy

Do we have a solution?

- We know how to move five disks if we know how to move four disks
- Likewise, we know how to move four disks if we know how to move three disks
- Generally, we know how to move n disks if we know how to move $n - 1$ disks
- Moving zero disks is trivial

Summary

- To move n disks from pole x to pole z we need to
  1. move $n-1$ disks from pole x to pole y,
  2. move 1 disk from pole x to pole z, and
  3. move $n-1$ disks from pole y to pole z.
- There is no fundamental difference between moving n or $n-1$ disks. The same method can be used.
- A method that calls itself (self referential) is called recursive

Code

```
public class Hanoi {

  public static void main (String[] args) {
    Scanner sc = new Scanner (System.in);
    System.out.print ("Enter number of disks: ");
    int n = sc.nextInt ();
    hanoi (n, 1, 2, 3);
  }

  public static void hanoi (int n, int s, int o, int d) {
    if (n <= 0)
      return;
    hanoi (n - 1, s, d, o);
    System.out.println ("move disk " + n + " from tower " + s
        + " to tower " + d);
    hanoi (n - 1, o, s, d);
  }
}
```

Recursion

- Recursion $\equiv$ Self reference
  - Recursive structures resemble themselves
  - Recursive definition make reference to themselves
- Well-known from the definition of mathematical functions

$$f(x,n) = \begin{cases} 1 & \text{if } n = 0 \\ x \times f(x, n-1) & \text{if } n > 0 \end{cases}$$

$$g(x) = \begin{cases} 1 & \text{if } x = 0 \\ x \times g(x-1) & \text{if } x > 0 \end{cases}$$

Example: Factorial

The factorial is defined for all non-negative integers:

$$fac(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \times fac(n-1) & \text{if } n > 0 \end{cases}$$

Example:

$$\begin{aligned} fac(4) &= 4 \times fac(3) \\ &= 4 \times (3 \times fac(2)) \\ &= 4 \times (3 \times (2 \times fac(1))) \\ &= 4 \times (3 \times (2 \times (1 \times fac(0)))) \\ &= 4 \times (3 \times (2 \times (1 \times 1))) \\ &= 4 \times (3 \times (2 \times 1)) \\ &= 4 \times (3 \times 2) \\ &= 4 \times 6 \\ &= 24 \end{aligned}$$

Static view of recursion

$$fac(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \times fac(n-1) & \text{if } n > 0 \end{cases}$$

- A math-like definition is translated into a Java method.
- You think about text and syntax.
- You don't think about run time.

Static view of recursion

The code:

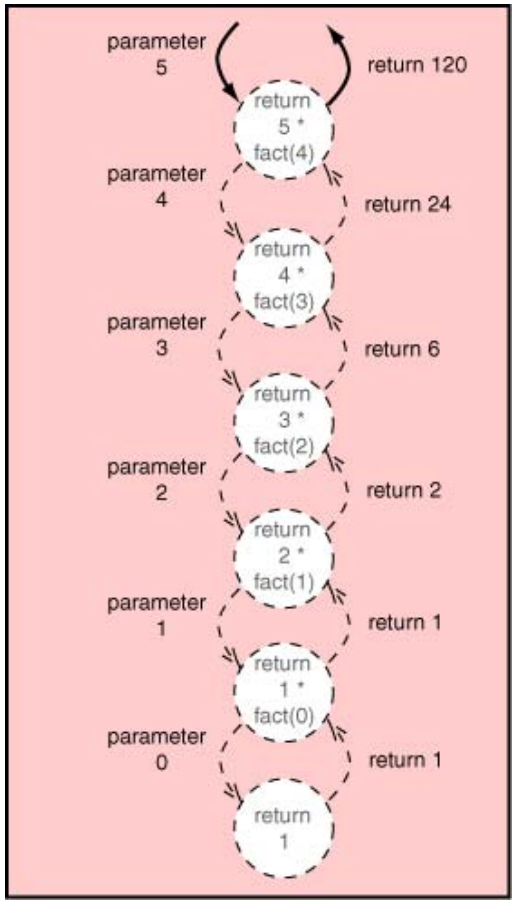
```
import java.util.Scanner;

public class Factorial {

  public static void main (String[] args) {
    Scanner sc = new Scanner (System.in);
    System.out.print ("Enter␣number:␣");
    int n = sc.nextInt();
    System.out.println("Factorial␣of␣" + n + "␣is␣" + fac (n));
  }

  public static long fac (long n) {
    if (n <= 0) return 1l;
    return n * fac (n - 1);
  }
}
```

Dynamic thinking



Two views of recursion

- Write the code using the static viewpoint.
- Test and debug the code using the dynamic viewpoint.

Here is another version of the Java method:

```
public static long fac (long n) {
  return n * fac (n - 1);
}
```

- From a static viewpoint, what is wrong?
  Answer: The base case $fac(1) = 1$ was left out of the Java code.
- From a dynamic viewpoint, what is wrong?
  Answer: The method always calls for another invocation regardless of the value of the parameter, so the chain of activation keeps growing until system resources run out.

How to construct a recursive algorithm

1. Find a way to divide the whole task, so that it becomes manageable.
2. Identify the recursion anchor: What is the trivial solution step and what is its associated condition?
3. Identify the recursion step: How can the problem be made (slightly) smaller?
4. Make sure that the problem reduction 3 eventually leads to the trivial case 2.

Potential problem

Problem: Recursion might not terminate

```
public static void recurForever () {
  recurForever ();
}
```

Solution: Choose appropriate condition for recursion and ensure progress is made towards making it **`false`** eventually (analogous to **`while`**).

Termination

Consider the following code:

```
public static long fac (long n) {
  if (n == 0) return 1l;
  return n * fac (n - 1);
}
```

- What would happen if `fac (-6)` was called?
- `fac (-6)` would grow the invocation chain without limit!
- Defensive programming means to anticipate such problems and design the code accordingly

```
public static long fac (long n) {
  if (n <= 0) return 1l;
  return n * fac (n - 1);
}
```

Recursion *vs.* iteration

Both recursion and iteration allow to executing statements multiple times

$$fac(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \times fac(n-1) & \text{if } n > 0 \end{cases}$$

```
int fac (int n)
{
  if (n <= 0)
    return 1;
  else
    return n * fac (n - 1);
}
```

```
int fac (int n)
{
  int product = 1;
  for (int j = 1; j <= n; j++)
    product *= j;
  return product;
}
```

Recursion *vs.* iteration

- Iteration
  - As long as the condition is true the loop is executed.
  - When the loop body has been executed for the last time, the loop completely terminates.
- Recursion
  - As long as the recursion condition is true the method is called again.
  - When the recursion anchor has been reached, no further recursion occurs
  - However, all recursive calls then unfold backwards, possibly leading to the execution of further code.

Recursion *vs.* iteration

- Although the use of recursion often leads to algorithms which
  - are shorter, more elegant, more readable
  - can be naturally developed
- It should not be used without consideration
  - comes with a cost for method calls in terms of
    - memory space (saving local variables in execution context)
    - execution speed (saving return address, register values, . . .)

Simple iteration is better dealt with a simple loop

Typically, however, the performance penalty is negligible and well worth the extra clarity (less bugs, . . .)

Rabbits

Problem:

- A female rabbit matures after 2 months.
- Each mature female rabbit produces 1 baby female rabbit the first day of every month.

Month	1	2	3	4	5	6	7	8	9	10
Rabbits	1	1	2	3	5	8	13	21	34	55

Solution:

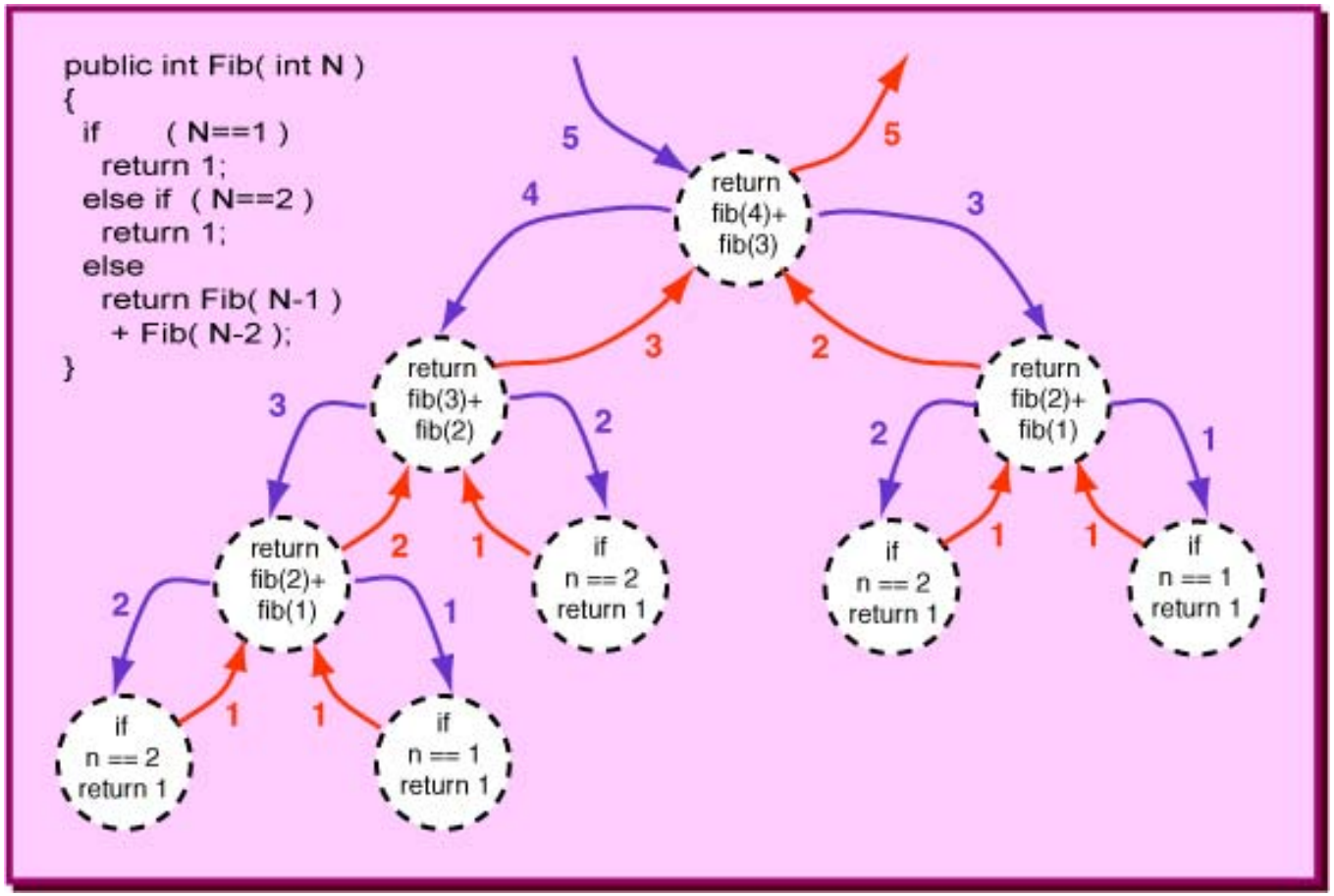
- Problem was studied and solved by a famous mathematician, Fibonacci.
- Function:

$$F(1) = 1$$
$$F(2) = 1$$
$$F(n) = F(n-1) + F(n-2)$$

Rabbits

```
public static int fib (int n) {
  if (n <= 2)
    return 1;
  return fib (n - 1) + fib (n - 2);
}
```

Rabbits


public int Fib(int N)
{
if (N==1)
return 1;
else if (N==2)
return 1;
else
return Fib(N-1)
+ Fib(N-2);
}
return fib(4)+ fib(3)
return fib(3)+ fib(2)
return fib(2)+ fib(1)
return fib(2)+ fib(1)
if n == 2 return 1
if n == 1 return 1
if n == 2 return 1
if n == 2 return 1
if n == 1 return 1
5
5
4
3
3
2
3
2
2
1
2
1
2
1
1
1
1
1

Coming up

- Next topic: Classes and objects