

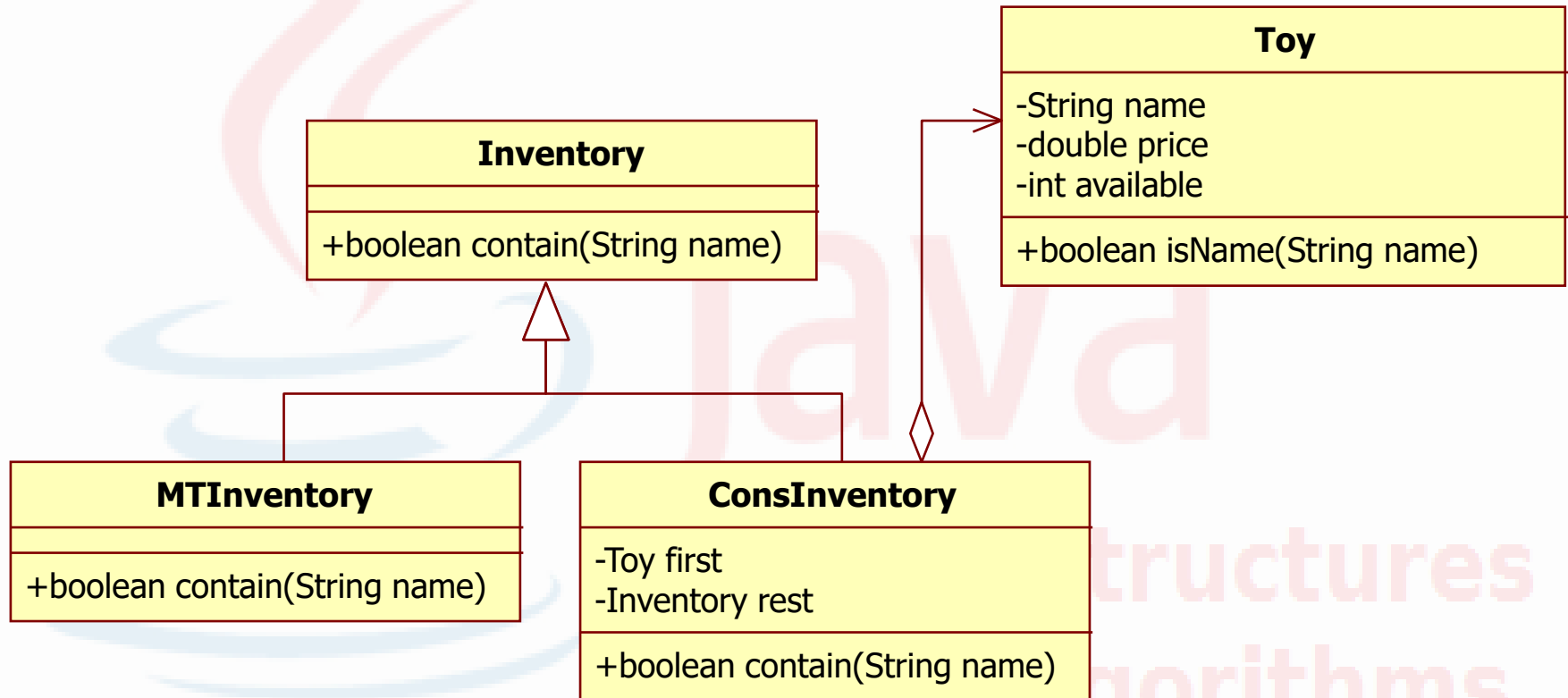


FACULTY OF INFORMATION TECHNOLOGY

# DATA STRUCTURES (CTDL)

Semester 1, 2022/2023

# Example 1. (BP Review)



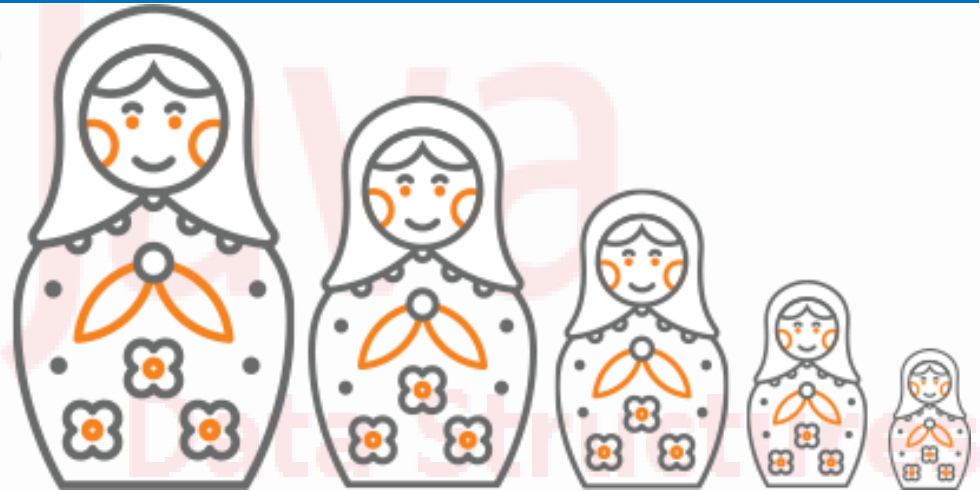
## contains() for MTInventory and ConsInventory

```
//in class MTInventory
public boolean contains(String toyName) {
    return false;
}
```

```
// in class ConsInventory
public boolean contains(String toyName) {
    return this.first.isName(toyName)
        || this.rest.contains(toyName);
}
```

```
//in class Toy
public boolean isName(String toyName) {
    return this.name.equals(toyName);
}
```

# Recursion



and Algorithms

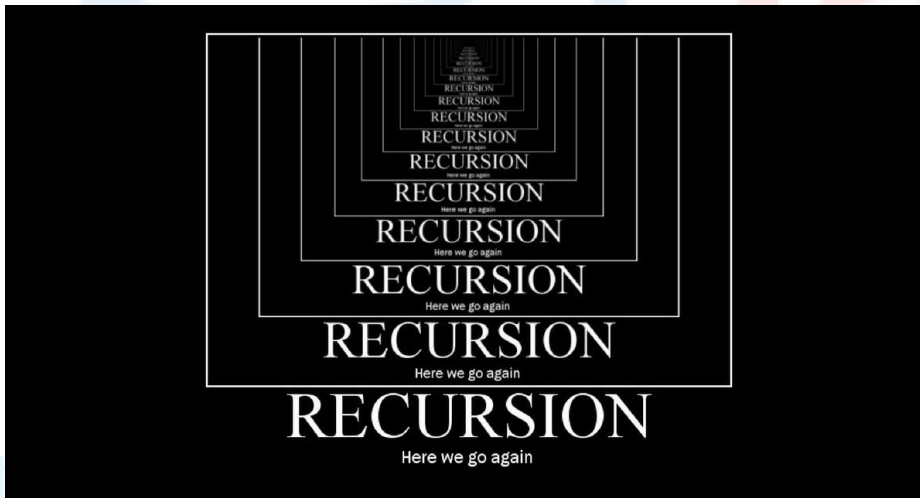
<http://www.javaguides.net>

# What is recursion?

- ▶ A method of solving a problem where the solution depends on solutions to smaller instances of the same problem
- ▶ Recursion is the process of defining something in terms of itself.
- ▶ An algorithm is recursive if it **calls itself** to do part of its work. It includes 2 parts:
  - The **base case** handling a simple input that can be solved **without resorting to a recursive call**
  - The **recursive part** containing **one or more recursive calls** to the algorithm

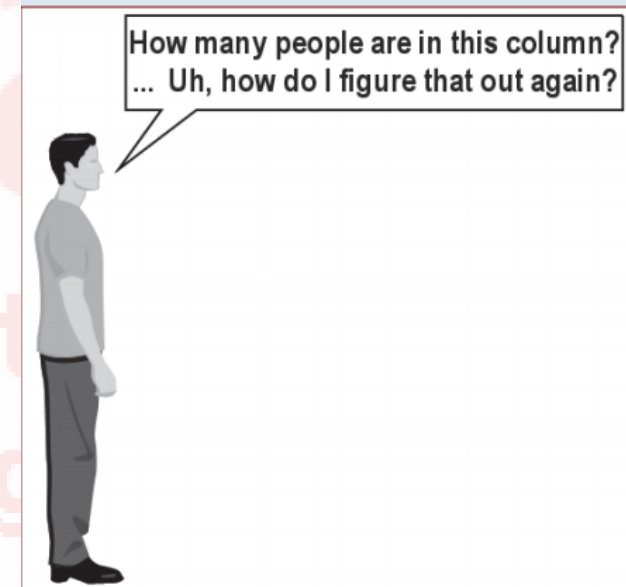
# What is recursion?

- ▶ A recursive algorithm must eventually **terminate**.
- ▶ A recursive algorithm must have at least one **base case**, or **stopping case**.
- ▶ A base case does not execute a recursive call.



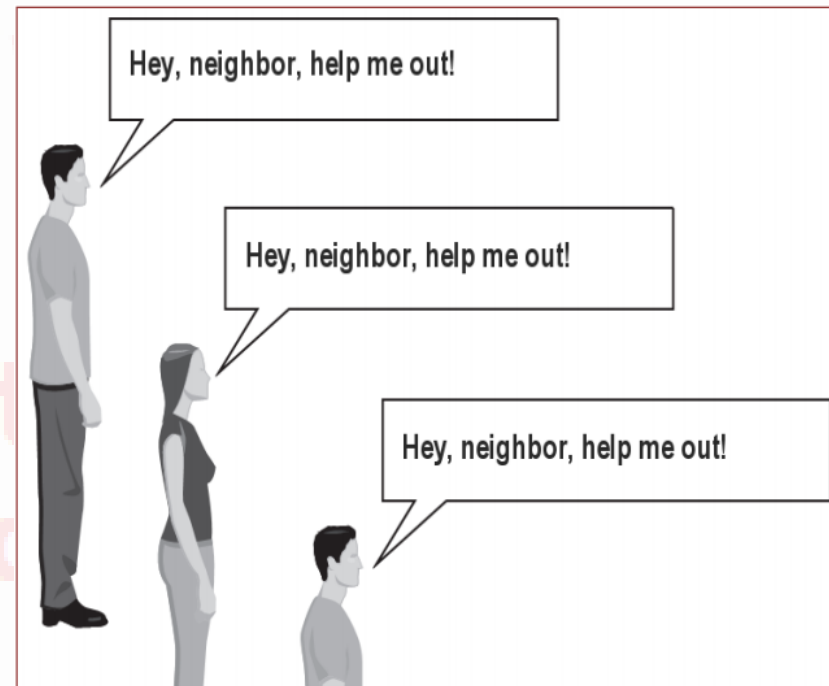
# Exercise

- ▶ How many students total are directly behind you in your "column" of the classroom?
- ▶ You have poor vision → you can see only the people right next to you. So, you **can't just look back and count**.
- ▶ But you are allowed to ask questions of the person next to you.
- ▶ How can we solve this problem?  
( **recursively!** )



# The recursion idea

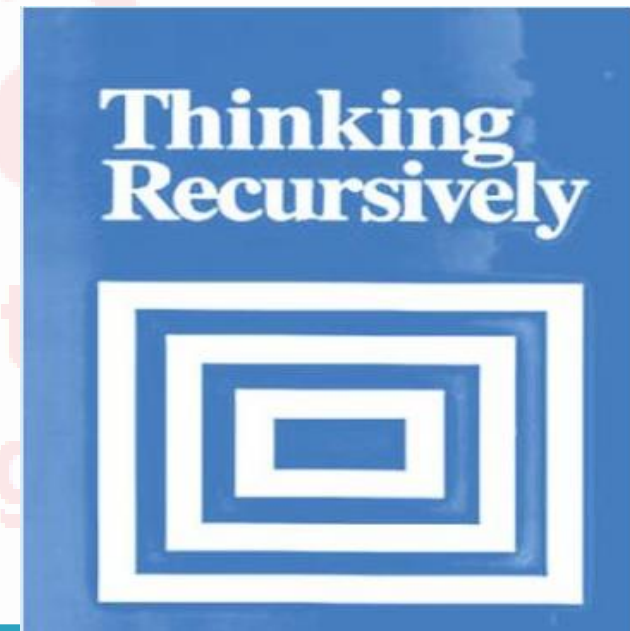
- ▶ Recursion is all about breaking a big problem into **smaller occurrences** of that same problem.
  - Each person can solve **a small part of the problem**.
  - What is a small version of the problem that would be easy to answer?
  - What **information from a neighbor** might help me?





# Recursive algorithm

- ▶ Number of people behind me:
  - If there is **someone behind me**, ask him/her how many people are behind him/her.
  - When they respond with a value **N**, then I will answer  **$N + 1$** .
  - If there is nobody behind me, then I will answer **1**.



<http://www.javaguides.net>

# Example

```
package lab2_recursion;

public class RecursionExample1 {
    static void p() {
        System.out.println("hello");
        p();
    }

    public static void main(String[] args) {
        p();
    }
}
```

hello  
hello  
...  
java.lang.StackOverflowError

# Types of recursion



Java

Data Structures  
and Algorithms

<http://www.javaguides.net>

# Types of recursion

- ▶ **Linear recursion**: makes at most **one** recursive call each time it is invoked.
- ▶ **Binary recursion**: algorithm makes **two** recursive calls.
- ▶ **Multiple recursion**: method may make (potentially more than two) recursive calls.

# Linear recursion

**Example:**

```
public int linearSum(int[] array, n) {  
    if (n == 1)  
        return array[0];  
    else  
        return linearSum(array, n-1) + array[n-1];  
}
```

Data Structures  
and Algorithms

<http://www.javaguides.net>

# Binary recursion

## Algorithm:

```
public int binarySum(int[] array, int i, int n) {  
    if (n == 1)  
        return array[i];  
    else  
        return binarySum(array, i, [n/2]) +  
               binarySum(array, i+[n/2], [n/2]);  
}
```

<http://www.javaguides.net>

# Factorial

- **Factorial**: the factorial of a positive integer **n**, denoted by **n!**, is the product of all positive integers less than or equal to **n**:

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

**Base case**

**Recursive part**

## Pseudocode (recursive):

function factorial is:

input: integer  $n$  such that  $n \geq 0$

output:  $[n \times (n-1) \times (n-2) \times \dots \times 1]$

1. if  $n$  is 0, return 1

2. otherwise, return  $[n \times \text{factorial}(n-1)]$

end factorial

# Factorial (cont.)

- ▶ The function can also be written as a recurrence relation:

$$b_n = n b_{n-1}$$

$$b_0 = 1$$

Computing the recurrence relation for  $n = 4$ :

$$\begin{aligned} b_4 &= 4 * b_3 \\ &= 4 * (3 * b_2) \\ &= 4 * (3 * (2 * b_1)) \\ &= 4 * (3 * (2 * (1 * b_0))) \\ &= 4 * (3 * (2 * (1 * 1))) \\ &= 4 * (3 * (2 * 1)) \\ &= 4 * (3 * 2) \\ &= 4 * 6 \\ &= 24 \end{aligned}$$



# Factorial (cont.)

## ► Implemented Java code:

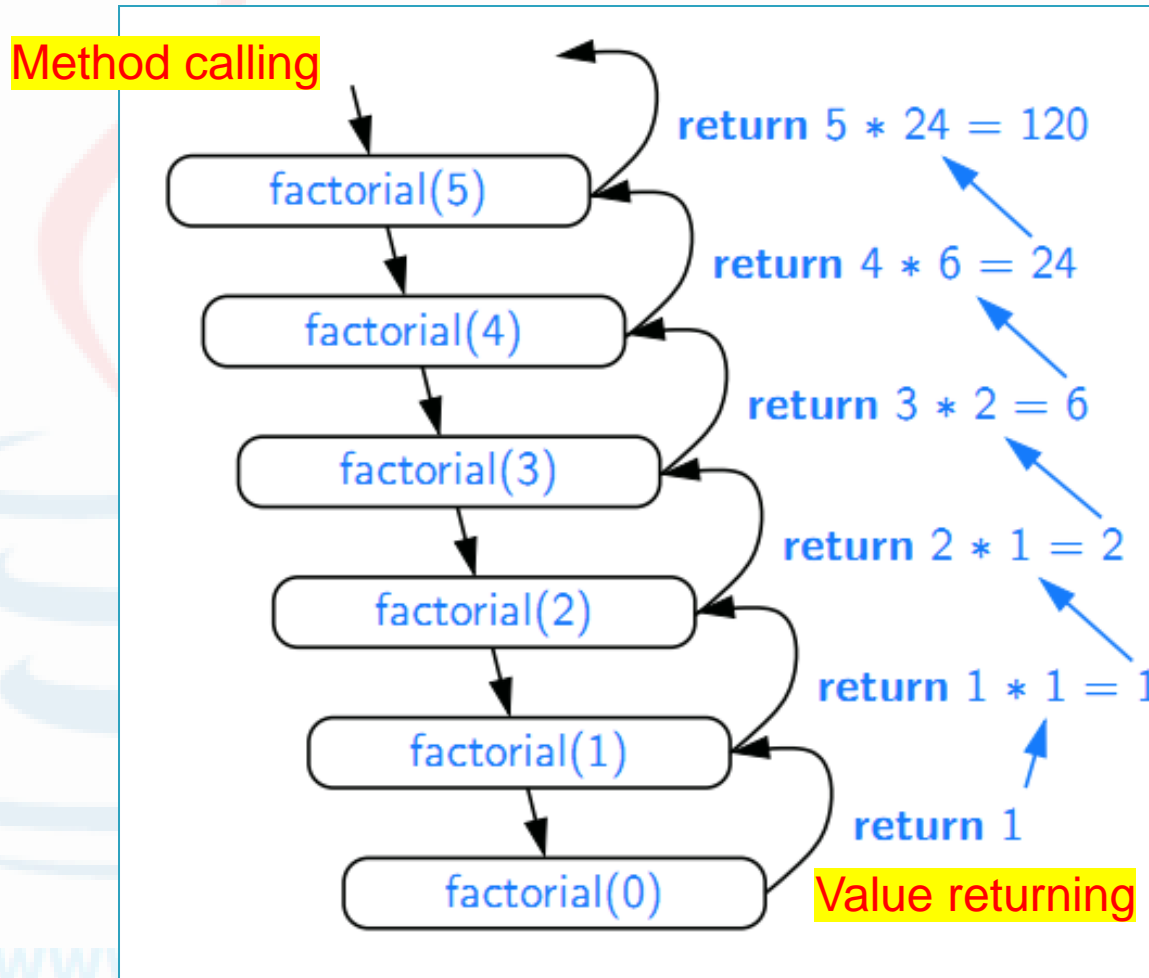
```
1 public static int factorial(int n) throws IllegalArgumentException {  
2     if (n < 0)  
3         throw new IllegalArgumentException();    // argument must be nonnegative  
4     else if (n == 0)  
5         return 1;                                // base case  
6     else  
7         return n * factorial(n-1);              // recursive case  
8 }
```

Data Structures  
and Algorithms

<http://www.javaguides.net>

# Factorial (cont.)

- ▶ A recursion trace for the call **factorial(5)**



# Factorial (cont.)

- ▶ This factorial function can also be described **without using recursion** :

## Pseudocode (iterative):

```
function factorial is:  
  input: integer  $n$  such that  $n \geq 0$   
  output:  $[n \times (n-1) \times (n-2) \times \dots \times 1]$   
  
  1. create new variable called running_total with a value of 1  
  
  2. begin loop  
    1. if  $n$  is 0, exit loop  
    2. set running_total to (running_total  $\times$   $n$ )  
    3. decrement  $n$   
    4. repeat loop  
  
  3. return running_total  
  
end factorial
```

**Recursive version  
Or  
Iterative version?**

# Why recursion?

- ▶ Avoidance of unnecessary calling of functions.
- ▶ A **substitute for iteration** where the iterative solution is very complex.
- ▶ Extremely useful when applying the same solution.
- ▶ Leads to elegant, simplistic, short Java code (when used well).

<http://www.javaguides.net>

# When to Use Recursion Rather Than Iteration?

Data Structures  
and Algorithms



<http://www.javaguides.net>

# When to Use Recursion Rather Than Iteration

- ▶ Common reasons for using recursion:
  - The **problem** is naturally recursive (e.g. Fibonacci)
  - The **data** is naturally recursive (e.g. filesystem)
  - Take more advantage of immutability:
    - **all variables are final**, **all data is immutable**, and the recursive methods are all pure functions → they do not mutate anything.
- ▶ Recursion is that it may take more space than an iterative solution (**downside**)

# Recursive Problems vs. Recursive Data

- ▶ The **problem structure** lends itself naturally to a **recursive definition**.
  - Ex.: factorial, Fibonacci, ...
- ▶ The data you are operating on is inherently **recursive in structure**.
  - Ex.: A filesystem consists of named *files*. Some files are *folders*, which can contain other files

# Common Mistakes in Recursive

- ▶ The **base case is missing entirely**, or
  - the problem needs more than one base case but not all the base cases are covered.

```
public static int factorial(int n) {  
    return n * factorial(n - 1);  
}
```

Data Structures  
and Algorithms



<http://www.javaguides.net>



# Common Mistakes in Recursive

- ▶ The recursive step doesn't reduce to a smaller subproblem, so the recursion doesn't converge.

```
public static int getFibonacci(int n) {  
    if (n == 0) {  
        return 0;  
    }  
    if (n == 1) {  
        return 1;  
    }  
    int f_n2 = getFibonacci(n),  
    int f_n1 = getFibonacci(n);  
    return f_n2 + f_n1;  
}
```

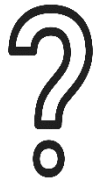


# Helper Methods

- ▶ For a given array of integers, implement a method, called **reversePrint** to display the array reversely.

```
public static void reversePrint(int[] arr) {  
    //TODO  
}
```

- ▶ Ex. arr= { 1, 2, 3, 4, 5 } → { 5, 4, 3, 2, 1 }
- ▶ How to implement **reversePrint** method?
  - It's easy if using **iterative approach**
  - How about **recursive approach**?

A yellow rectangular box containing the words "NO WAY" in a bold, black, stylized font. The letter "O" in "NO" is red with a white center.A large, black, hand-drawn style question mark.

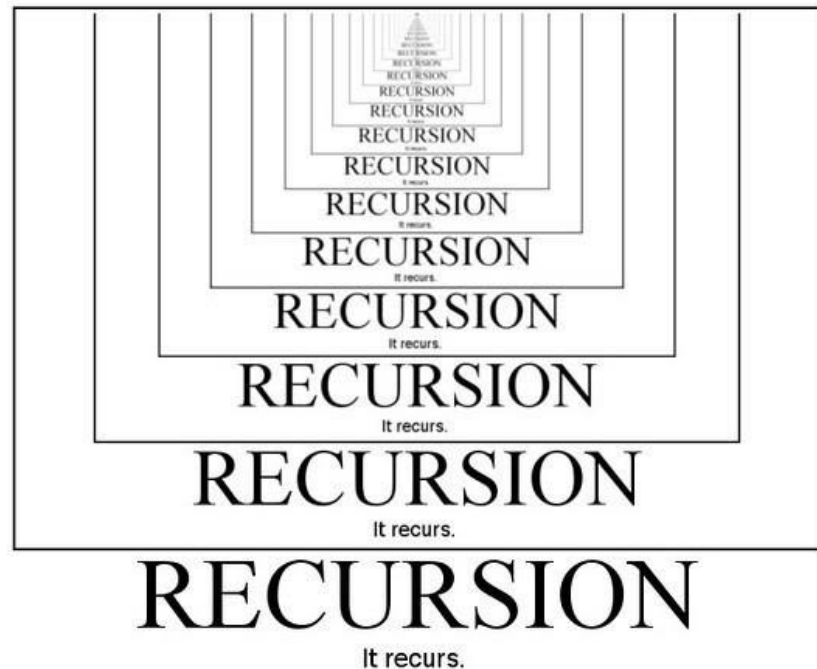
# Helper Methods

- ▶ For a given array of integers, implement a method, called **reversePrint** to display the array reversely.
- ▶ **Hint:** using a helper method

```
public static void reversePrint(int[] arr) {  
    reversePrintHelp(arr, arr.length);  
}
```

```
public static void reversePrintHelp(int[] arr, int n) {  
    //TODO  
}
```

# Other recursive problems



# Recursive sum

- ▶ Calculation arithmetic series ( sigma ) recursive Sum

$$\sum_{x=1}^n x$$

```
public int sigma(int n) {  
    // TODO  
    return 0;  
}
```

```
public static int sigma(int n) {  
    if (n <= 1)  
        return n;  
    else  
        return n + sigma(n - 1);  
} // sigma
```

Iterative approach???

# Calculation power

- ▶ How to calculate power:

$$x^y = \underbrace{x * x * \dots * x}_{y \text{ times}}$$

```
public static int power(int x, int y) {  
    //TODO  
    return 0;  
} // power
```

```
public static int power(int x, int y) {  
    if (y == 0)  
        return 1;  
    else  
        return x * power(x, y - 1);  
} // power
```

Iterative approach???

# Calculation product

- ▶ How to calculate product?

$$x * y$$

$$x * y = \underbrace{x + x + x + x + \dots + x}_{y \text{ times}}$$

```
public static int recMult(int x, int y) {  
    // TODO  
    return 0;  
} // recMult
```

# Printing stars

- ▶ If the method **stars1** is called with the value **3**, is it equivalent to the method **stars2**?

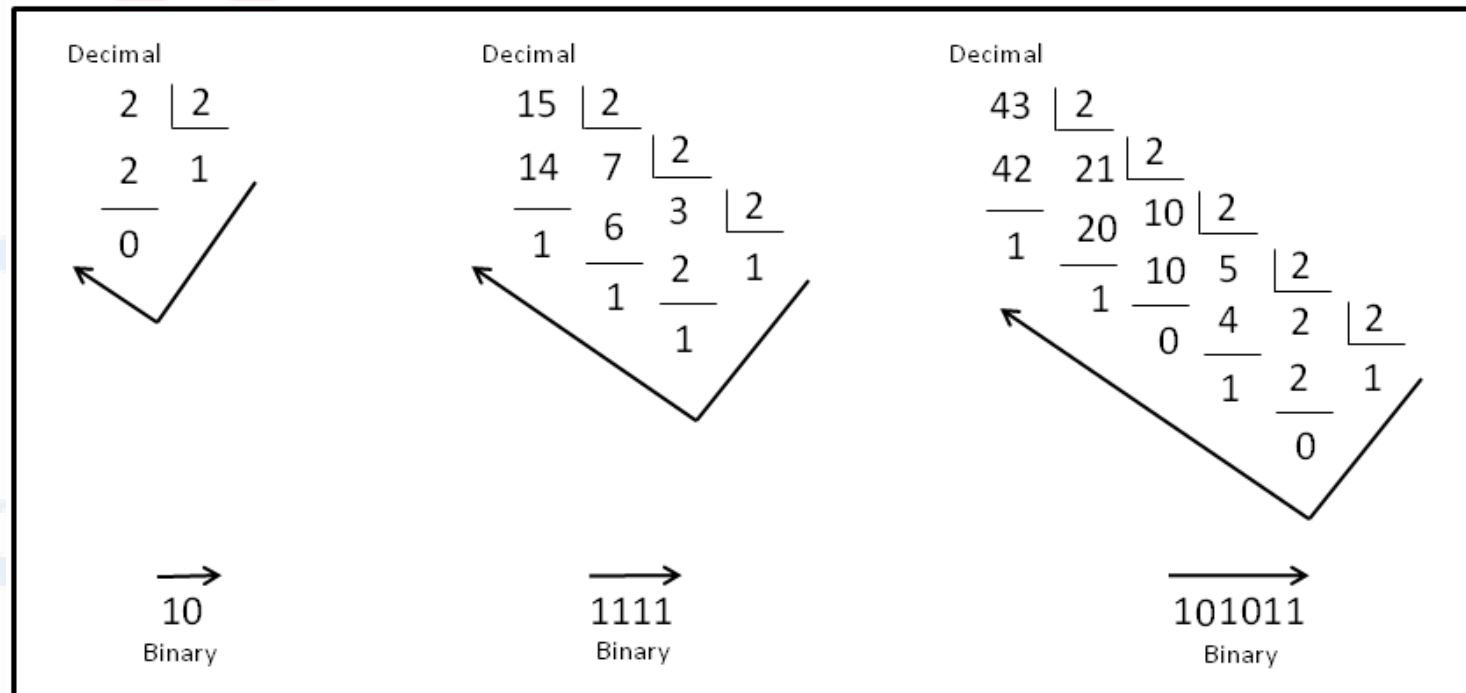
```
public static void stars1(int n) {  
    if (n < 1)  
        return;  
    System.out.print(" * ");  
    stars1(n - 1);  
} // stars1  
  
public static void stars2(int n) {  
    if (n > 1)  
        stars2(n - 1);  
    System.out.print(" * ");  
} // stars2
```

Explain!



# Decimal to binary number using recursion

- ▶ Given a decimal number as input, how to convert the given decimal number into equivalent binary number.



# Decimal to binary number using recursion

- ▶ Given a decimal number as input, how to convert the given decimal number into equivalent binary number.

```
findBinary(decimal)
    if (decimal == 0)
        binary = 0
    else
        binary = decimal % 2 + 10 * (findBinary(decimal / 2))
```

Data Structures  
and Algorithms

<http://www.javaguides.net>

# Decimal to binary number

## ► Other approaches:

- Using **Integer.toString(number)**: returns a string representation of the integer argument as an unsigned integer in binary
- Using **iterative method**?



<http://www.javaguides.net>

# Drawing an English Ruler

- ▶ How to draw the markings of a typical English ruler?
- ▶ The length of the tick designating a whole inch as **the major tick length**.
- ▶ Between the marks for whole inches, the ruler contains a series of **minor ticks**, placed at intervals of  $1/2$  inch,  $1/4$  inch, and so on.
- ▶ As the size of **the interval decreases by half**, the **tick length decreases by one**.

# Drawing an English Ruler (cont.)

---- 0

-

--

-

----

-

--

-

----- 1

-

--

-

----

-

--

-

----- 2

(a)

----- 0

-

--

-

----

-

--

-

-----

-

--

-

----

-

--

-

----- 1

(b)

--- 0

-

--

-

--- 1

-

--

-

--- 2

-

--

-

--- 3

(c)

Three sample outputs of an English ruler drawing: (a) a 2-inch ruler with major tick length 4; (b) a 1-inch ruler with major tick length 5; (c) a 3-inch ruler with major tick length 3.



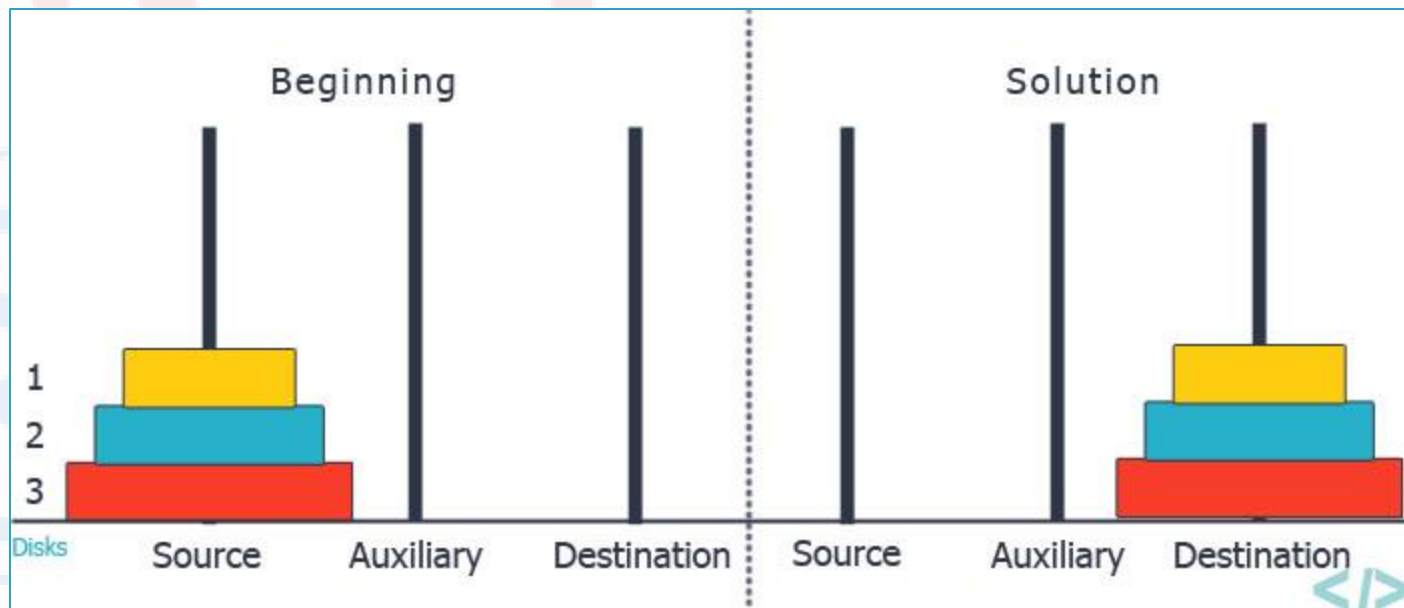
# Towers of Hanoi

- ▶ A mathematical puzzle where we have **three rods** and **n disks**. Rules:
  - Only **one disk can be moved at a time**.
  - Each move consists of **taking the upper disk from one of the stacks and placing it on top of another stack** i.e. a disk can only be moved if it is the uppermost disk on a stack.
  - No disk may be placed on top of a smaller disk.

<http://www.javaguides.net>

# Towers of Hanoi (cont.)

- ▶ The minimum number of moves required to solve is  $2^n - 1$ , where  $n$  is the number of discs.



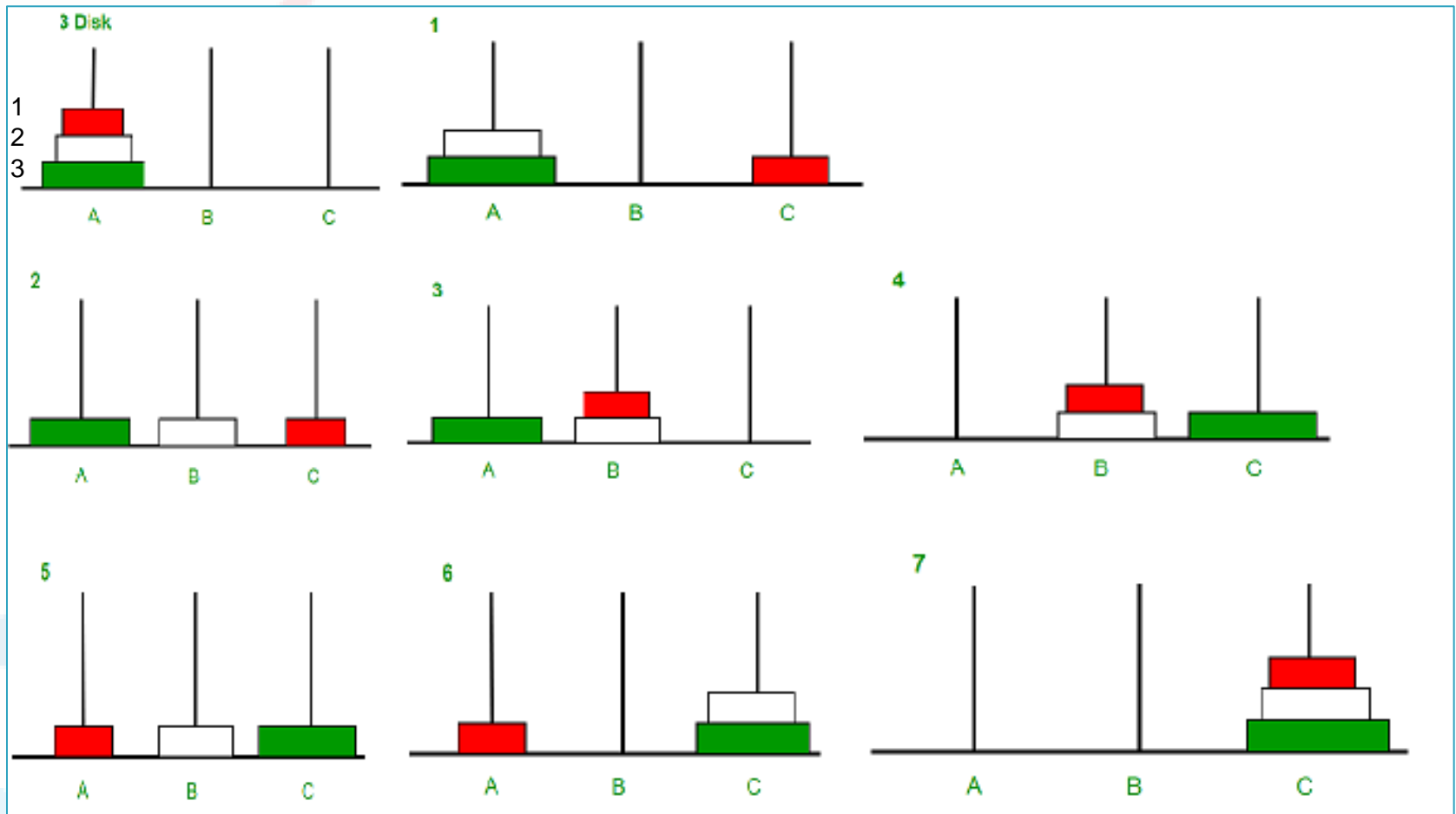
# Towers of Hanoi (cont.)

- ▶ Label the pegs A, B, C
- ▶ Let  $n$  be the total number of discs
- ▶ Number the discs from 1 (smallest, topmost) to  $n$  (largest, bottommost)
- ▶ To move  $n$  discs from rod A to rod C:
  - **Step 1.** move  $n-1$  discs from A to B. This leaves disc  $n$  alone on peg A
  - **Step 2.** move disc  $n$  from A to C
  - **Step 3.** move  $n-1$  discs from B to C



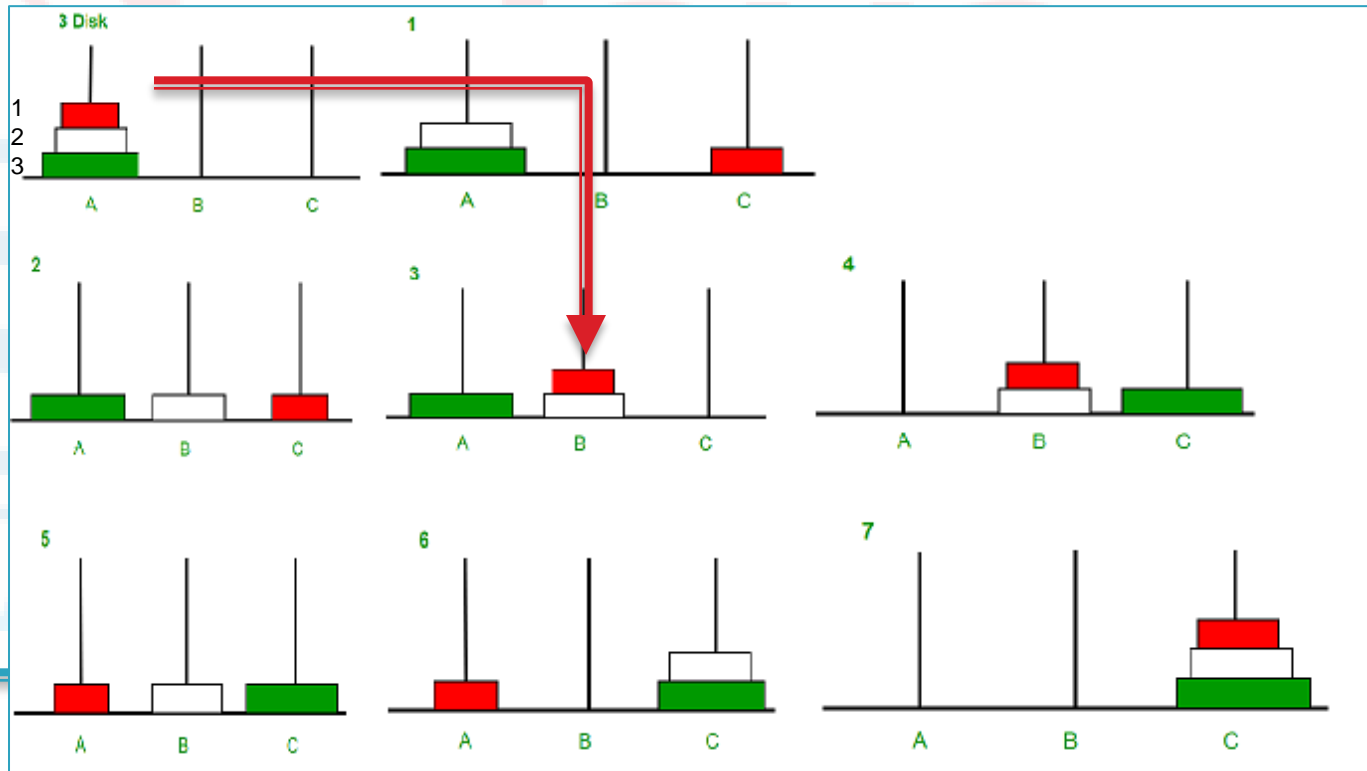
# Towers of Hanoi (cont.)

- ▶ Image illustration for 3 discs :



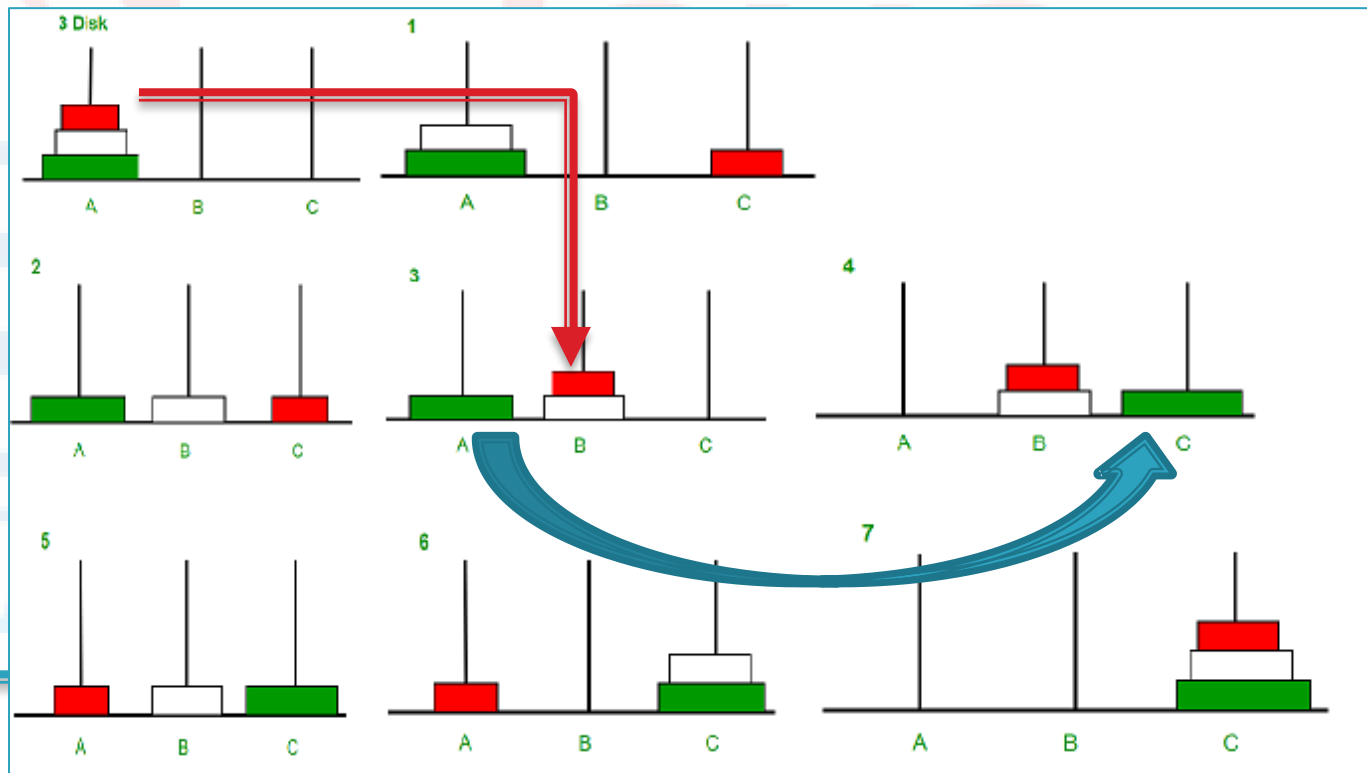
# Towers of Hanoi: Think recursively

- ▶ For given 3 discs with order: **disk 1** < **disk 2** < **disk 3**. How to move all disks to C from A?
- Step 1: Move discs 2 and smaller from peg A (*source*) to peg B (*spare*), using peg C (*dest*) as a spare



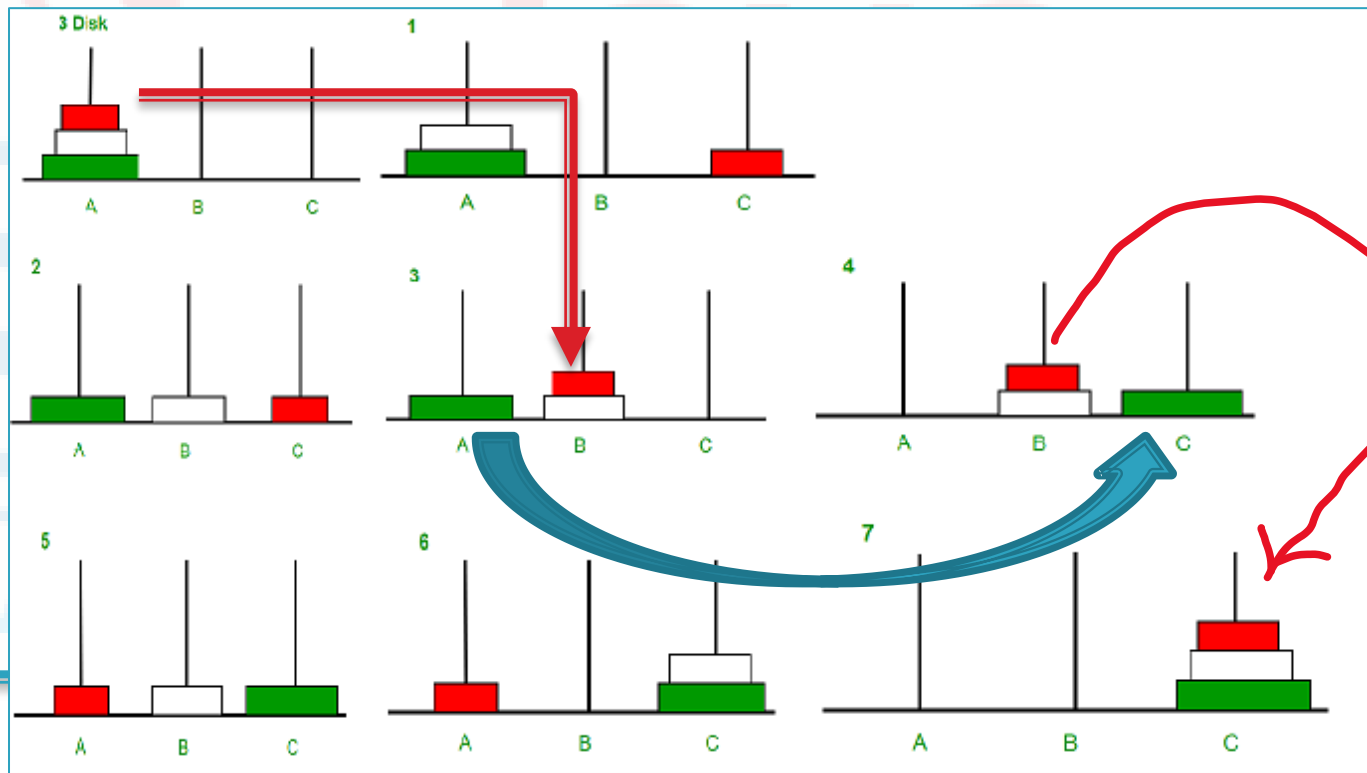
# Towers of Hanoi (cont.): Think recursively

- ▶ For given 3 discs with order: **disk 1 < disk 2 < disk 3**. How to move all discs to C from A?
  - Step 2: With all the smaller discs on the spare peg, we can move disk 3 from peg A (*source*) to peg C (*dest*).



# Towers of Hanoi (cont.): Think recursively

- ▶ For given 3 discs with order: **disk 1 < disk 2 < disk 3**. How to move all disks to C from A?
  - **Step 3**: We want discs 1 and smaller moved from peg C (*spare*) to peg B (*dest*)



# Towers of Hanoi (cont.)

- ▶ Suppose *source=A*, *dest=C*, and *spare=B*, *disk* represents the number of disks.

```
FUNCTION MoveTower(disk, source, dest, spare):  
IF disk == 1, THEN:  
    move disk from source to dest  
ELSE:  
    MoveTower(disk - 1, source, spare, dest)    // Step 1 above  
    move disk from source to dest                // Step 2 above  
    MoveTower(disk - 1, spare, dest, source)    // Step 3 above  
END IF
```

Data Structures  
and Algorithms

<http://www.javaguides.net>

# Fibonacci

- ▶ Fibonacci: next number is the sum of previous two numbers
- ▶ Ex. 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...

$$F_0 = 0$$

$$F_1 = 1$$

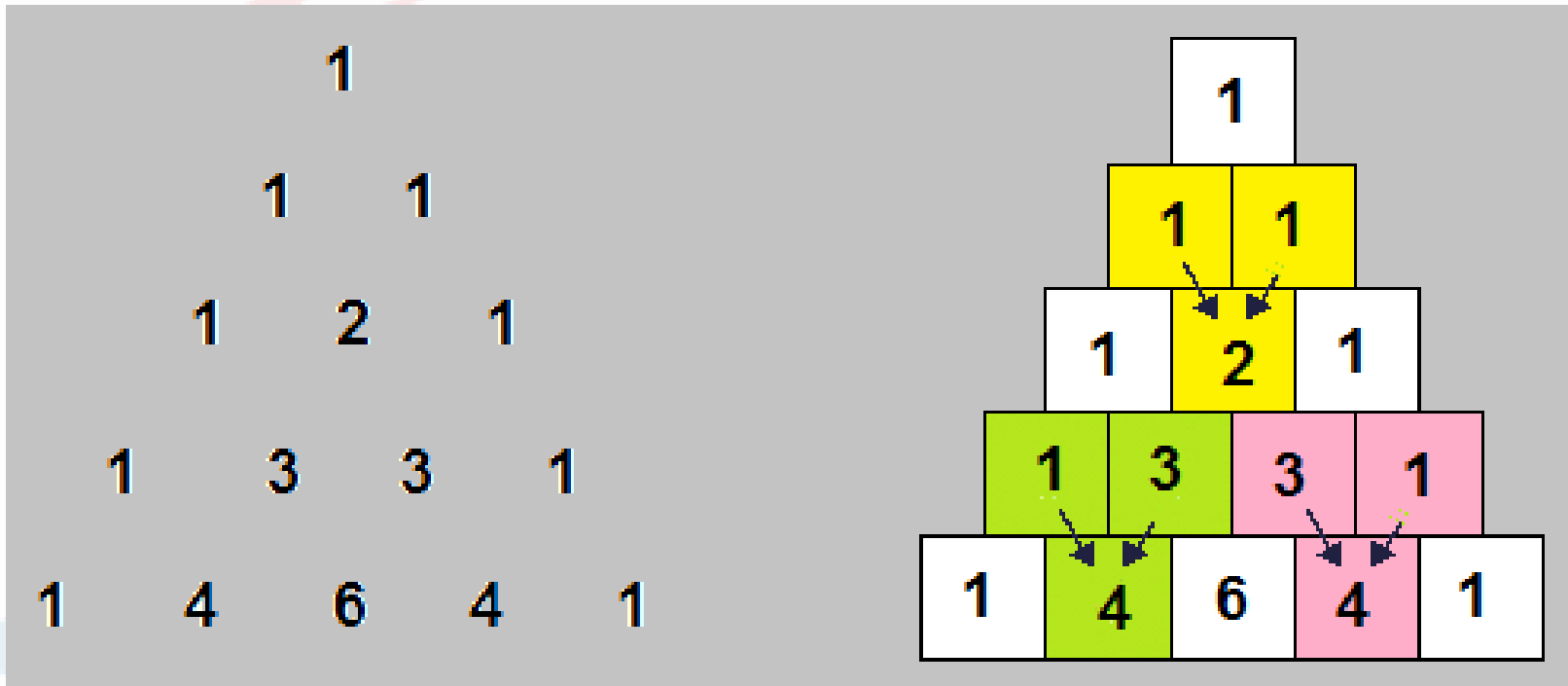
$$F_n = F_{n-2} + F_{n-1} \quad \text{for } n > 1.$$

- ▶ Two approaches:
  - Fibonacci Series **without using recursion**
  - Fibonacci Series **using recursion**



# Pascal's triangle

- ▶ Pascal's triangle: a triangular array of the binomial coefficients



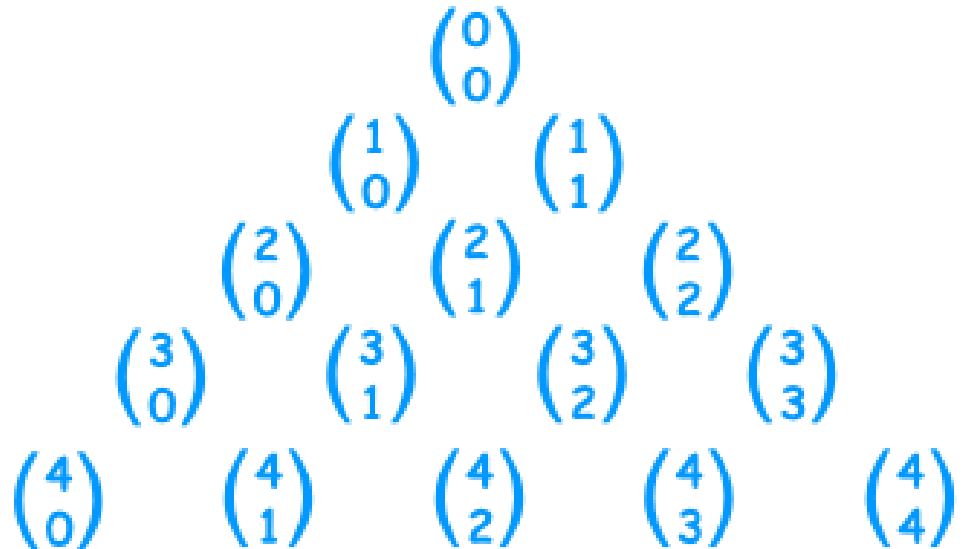
# Pascal's triangle (cont.)

It is commonly called "n choose k" and written like this:  $\binom{n}{k} = \frac{n!}{k!(n-k)!}$

Notation: "n choose k" can also be written  $C(n,k)$ ,  ${}^nC_k$  or even  ${}_nC_k$ .

$$\binom{0}{0} = 1$$

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$





# Pyramid

```
  1
 2 2
3 3 3
4 4 4 4
5 5 5 5 5
6 6 6 6 6 6
7 7 7 7 7 7 7
8 8 8 8 8 8 8 8
9 9 9 9 9 9 9 9 9
```

**Pyramid Pattern-1**

```
  1
 1 2
1 2 3
1 2 3 4
1 2 3 4 5
1 2 3 4 5 6
1 2 3 4 5 6 7
1 2 3 4 5 6 7 8
1 2 3 4 5 6 7 8 9
```

**Pyramid Pattern-2**

```
  *
 * *
* * *
* * * *
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
```

**Pyramid Pattern-3**

```
      1
    1 2 1
  1 2 3 2 1
1 2 3 4 3 2 1
 1 2 3 4 5 4 3 2 1
1 2 3 4 5 6 5 4 3 2 1
 1 2 3 4 5 6 7 6 5 4 3 2 1
1 2 3 4 5 6 7 8 7 6 5 4 3 2 1
```

**Pyramid Pattern-4**

```
      9
    8 9 8
  7 8 9 8 7
6 7 8 9 8 7 6
 5 6 7 8 9 8 7 6 5
4 5 6 7 8 9 8 7 6 5 4
3 4 5 6 7 8 9 8 7 6 5 4 3
2 3 4 5 6 7 8 9 8 7 6 5 4 3 2
1 2 3 4 5 6 7 8 9 8 7 6 5 4 3 2 1
```

**Pyramid Pattern-5**

```
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
```

**Inverted Pyramid Pattern-6**

```
9 9 9 9 9 9 9 9 9 9
8 8 8 8 8 8 8 8 8
7 7 7 7 7 7 7 7
6 6 6 6 6 6 6
5 5 5 5 5 5
4 4 4 4 4
3 3 3
2 2
1
```

**Inverted Pyramid Pattern-7**



## Pyramid Pattern Programs in Java

# Christmas tree

```

      *
    * * *
  * * * * *
* * * * * * *
    * * *
  * * * * *
* * * * * * *
* * * * * * * * *
    * * * * *
  * * * * * * *
* * * * * * * * *
    *
  *
* * * * * * *

```



# Algebra problems

1.  $S(n) = 1 - 2 + 3 - 4 + \dots + ((-1)^{(n+1)}).n$ ,  $n > 0$

2.  $S(n) = 1 + 1.2 + 1.2.3 + \dots + 1.2.3 \dots n$ ,  $n > 0$

3.  $S(n) = 1^2 + 2^2 + 3^2 + \dots + n^2$ ,  $n > 0$

4.  $S(n) = 1 + 1/2 + 1/(2.4) + 1/(2.4.6) + \dots + 1/(2.4.6.2n)$ ,  $n \geq 0$





# FACULTY OF INFORMATION TECHNOLOGY

