Recursion

Dr. Abdallah Karakra

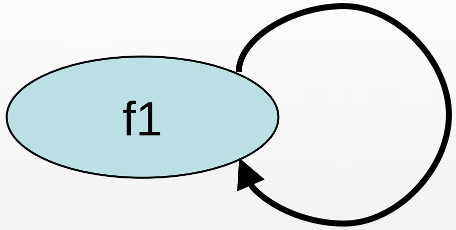
Computer Science Department

COMP242

Thursday, February 29, 2024

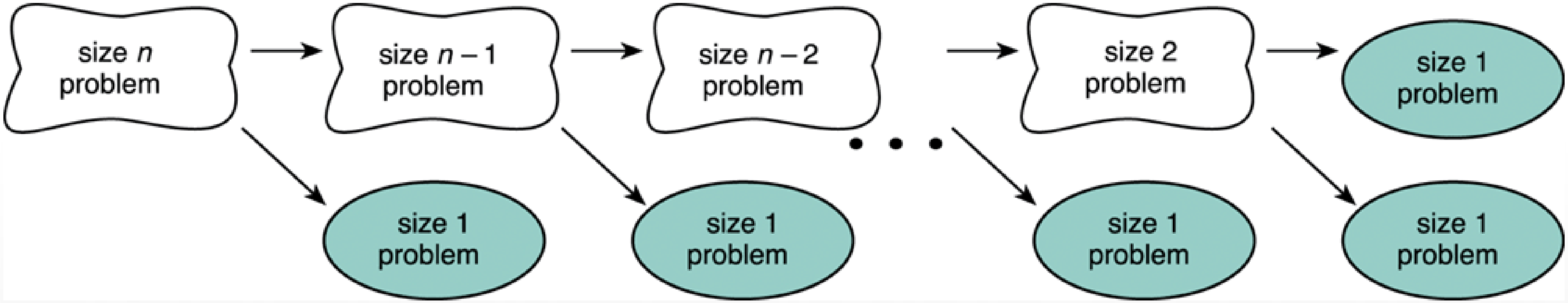
Introduction to Recursion

- A recursive function is one that calls itself.



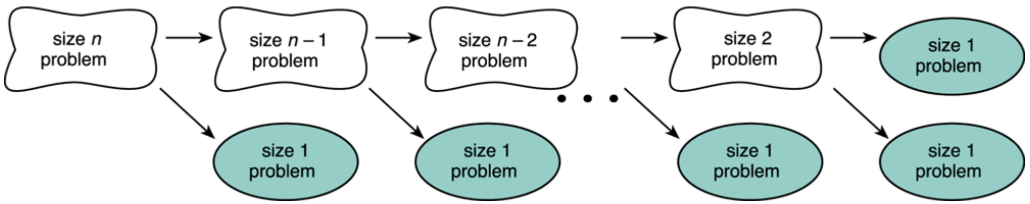

```
public void message()
{
        System.out.println("This is a recursive function");
        message();
}
```

Splitting a Problem into Smaller Problems



- Assume that the problem of size 1 can be solved easily (i.e., the simple case).
- We can recursively split the problem into a problem of size 1 and another problem of size n-1.

Splitting a Problem into Smaller



Let f(x)=f(x -1)+3 , f(0)=4 , find f(7)

f(7) = f(7-1)+3 → f(7)=f(6)+3

f(6) = f(6-1)+3 → f(6)=f(5)+3

f(5) = f(5-1)+3 → f(5)=f(4)+3

f(4) = f(4-1)+3 → f(4)=f(3)+3

f(3) = f(3-1)+3 → f(3)=f(2)+3

f(2) = f(2-1)+3 → f(2)=f(1)+3

f(1) = f(1-1)+3 → f(1)=f(0)+3

f(7)=22+2=25

f(6)=19+3=22

f(5)=16+3=19

f(4)=13+3=16

f(3)=10+3=13

f(2)=7+3=10

f(1)=4+3=7

f(0)=4

Base case

Recursive Problem

The function below displays the string "This is a recursive function.", and then calls itself.

```
public void message()
{
        System.out.println("This is a recursive function");
        message();
}
```

Recursive Problem

- The function is like an **infinite loop** because there is **no code to stop it from repeating.**
- Like a loop, a recursive function **must have some algorithm to control the number of times it repeats.**

Recursion

- Like a loop, a recursive function must have some algorithm to control the number of times it repeats. Shown below is a modification of the **`message`** function. It passes an integer argument, which holds the number of times the function is to call itself.

```
Public void message(int times)
{
        if (times > 0)
        {
                System.out.println("This is a recursive function");
                message(times - 1);
        }

}
```

Recursion

- The function contains an `if/else` statement that controls the repetition.
- As long as the `times` argument is greater than zero, it will display the message and call itself again. Each time it calls itself, it passes `times - 1` as the argument.

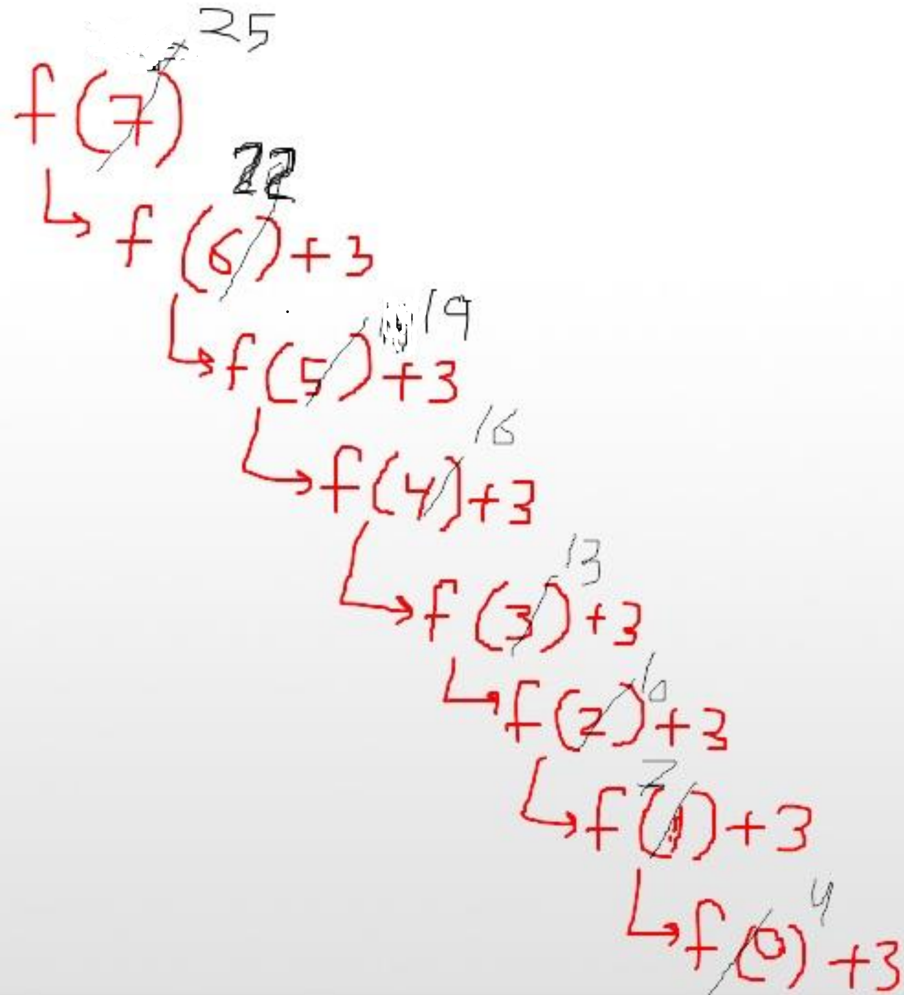
Recursive Function

Let f(x)=f(x -1)+3 , f(0)=4 , find f(7)

```
public int f(int x)
{

  if (x == 0)
      return 4; //base case

  else

      return f(x-1)+3;

}
```

Recursive function terminates when a base case is met.

Trace of f(x)=f(x -1)+3



Recursive Function Factorial

In mathematics, the notation n! represents the factorial of the number n. The factorial of a number is defined as:

$$
\text{fact }(n) = \begin{cases} 1 & , \quad n = 0 \\ n * \text{fact }(n-1) & , \quad n > 0 \end{cases}
$$

```
In other words,

n! =   1 * 2 * 3 * ... * n                  if n > 0

             1                              if n = 0
```

Recursive Function Factorial

The following Java function implements the recursive definition of factorial:

```
/** Return the factorial for the specified number */
public static long factorial(int n) {
  if (n == 0) // Base case                                    base case
    return 1;
  else
    return n * factorial(n - 1); // Recursive call            recursion
}
```

Trace of fact = factorial(3);

factorial(3) 6

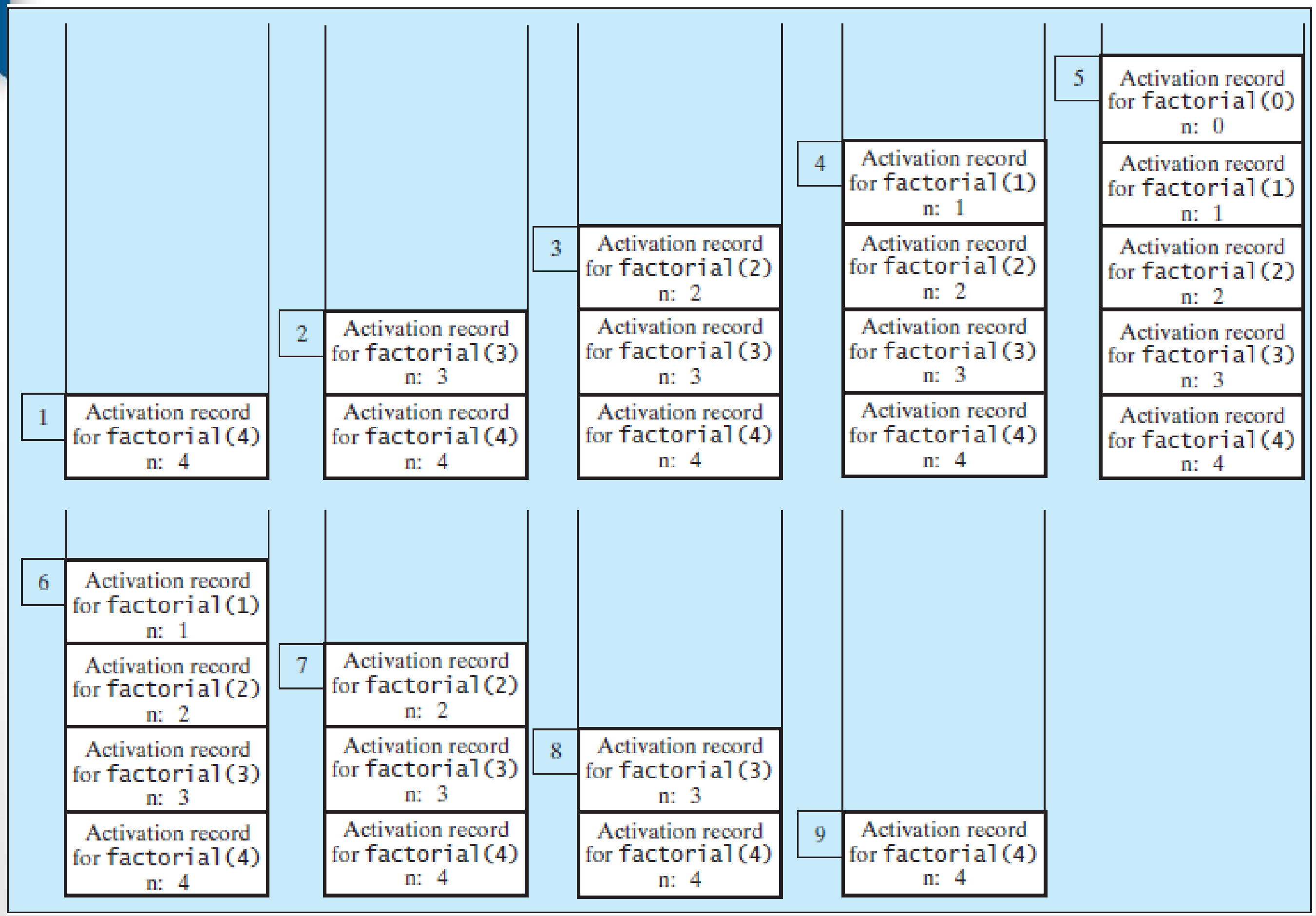
→ 3 * factorial(2) 2

→ 2 * factorial(1) 1

→ 1 * factorial(0) 1

Trace of fact = factorial(4);

The Stack "FILO"



When `factorial(4)` is being executed, the `factorial` method is called recursively, causing stack space to dynamically change.

Recursive Function Power

$$\text{Power}(x,y) = \begin{cases} 1 & , \quad y = 0 \\ X*\text{power}(x,y-1) & , \quad y>0 \end{cases}$$

Recursive Function Power

The following Java function implements the recursive definition of power:

```
public static int power(int x, int y)

{
  if (y == 0)

       return 1;

  else

       return x * power(x, y - 1);
}
```

Recursive Function Power

```
Power(2,3)          8
 └→ 2 * Power(2,2) = 8          4
      └→ 2 * Power(2,1) = 4     2
           └→ 2 * Power(2,0) = 2     1
```

Recursive Function fibonacci

the Fibonacci sequence 1, 1, 2,3, 5, 8, 13, 21, 34,…………

$a_{n:}$ 1, 1, 2, 3, 5, 8, 13, 21, 34,…………
n: 1, 2, 3, 4, 5, 6, 7, 8 , 9,…………

$a_1 = 1$, $a_2 = 1$, $a_3 = a_1 + a_2 = 2$, $a_n = a_{n-1} + a_{n-2}$

Recursive Function fibonacci

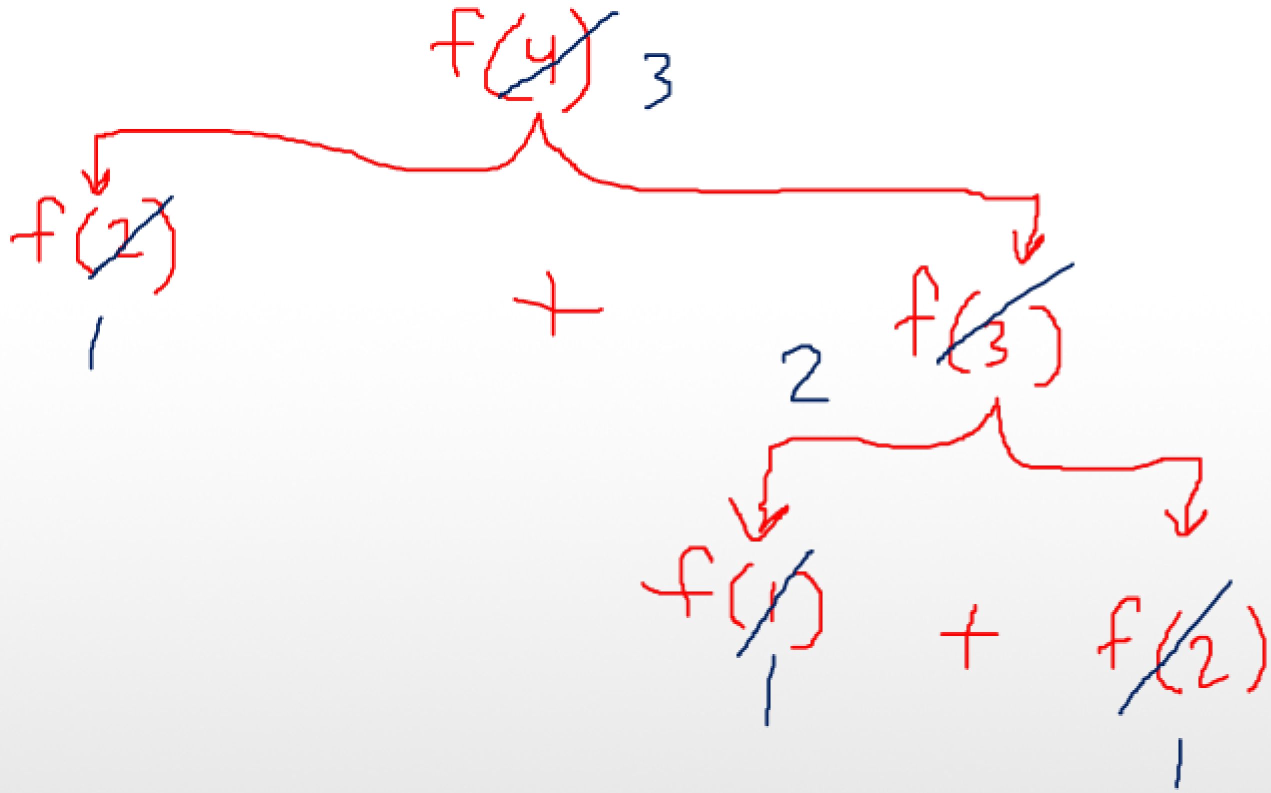
$$
\text{fibonacci}(n) = \begin{cases} 1 & , n = 1 \\ 1 & , n = 2 \\ \text{fibonacci}(n-2) + \text{fibonacci}(n-1) & , n > 2 \end{cases}
$$

Recursive Function fibonacci

the Fibonacci sequence 1, 1, 2,3, 5, 8, 13, 21, 34,…………

```
public static long  fibonacci(int n)
{
    if (n==1 || n==2)
        return 1;
    else
        return fibonacci(n-2)+fibonacci(n-1);


}
```

Trace of fibonacci = fibonacci(4)



You have to know !

The code of a recursive method must be structured to handle both **the base case** and **the recursive case**

Each call sets up **a new execution environment, with new parameters and new local variables**

You have to know !

You **must be able to determine when recursion is the correct technique** to use

Recursion has the overhead of multiple method invocations

However, for some problems recursive solutions are often more simple and elegant than iterative solutions

The main advantage is usually simplicity. The main disadvantage is often that the algorithm may require large amounts of memory if the depth of the recursion is very large.

You have to know !

Iterative Factorial

```
public long factorial(int n) {
        long product = 1;
        for(int i = 1; i <= n; i++) {
                product *= i;
                System.out.println("Product " + product);
        }
        return product;
}
```

Recursive Factorial

(Write the previous method without using a loop!)

```
public long factorial(int n) {
        if(n == 1) {
                return 1;
        }
        return n*(factorial(n-1));
}
```

Try running this for large values of n!

Comparison

1. Some problems are more easily solved recursively.

2. Recursion can be highly inefficient as resources are allocated for each method invocation.

If you can solve it iteratively, you usually should!

You have to know !

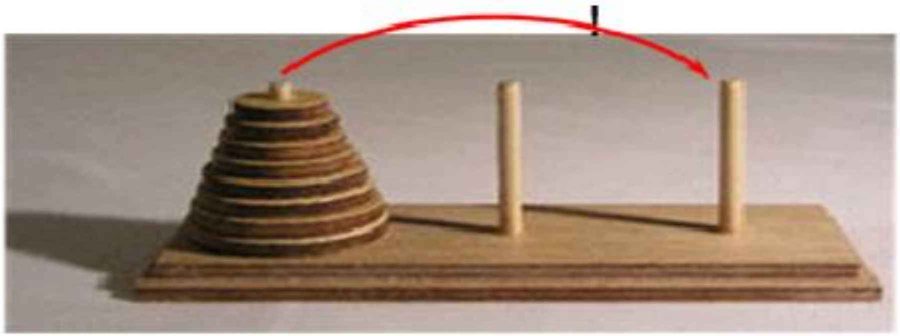
Recursion

```java
public static long  fib(int n)
{
    if (n==1 || n==2)
       return 1;
    else
       return fib(n-2)+fib(n+1);


}
```

Iteration

```java
public static long fib(int n) {
   int f0 = 0, f1 = 1, currentFib;

   if (n == 0) return 0;
   if (n == 1) return 1;

   for (int i = 2; i <= n; i++) {
     currentFib = f0 + f1;
     f0 = f1;
     f1 = currentFib;
   }

   return f1;
 }
```

Case Study: Towers of Hanoi



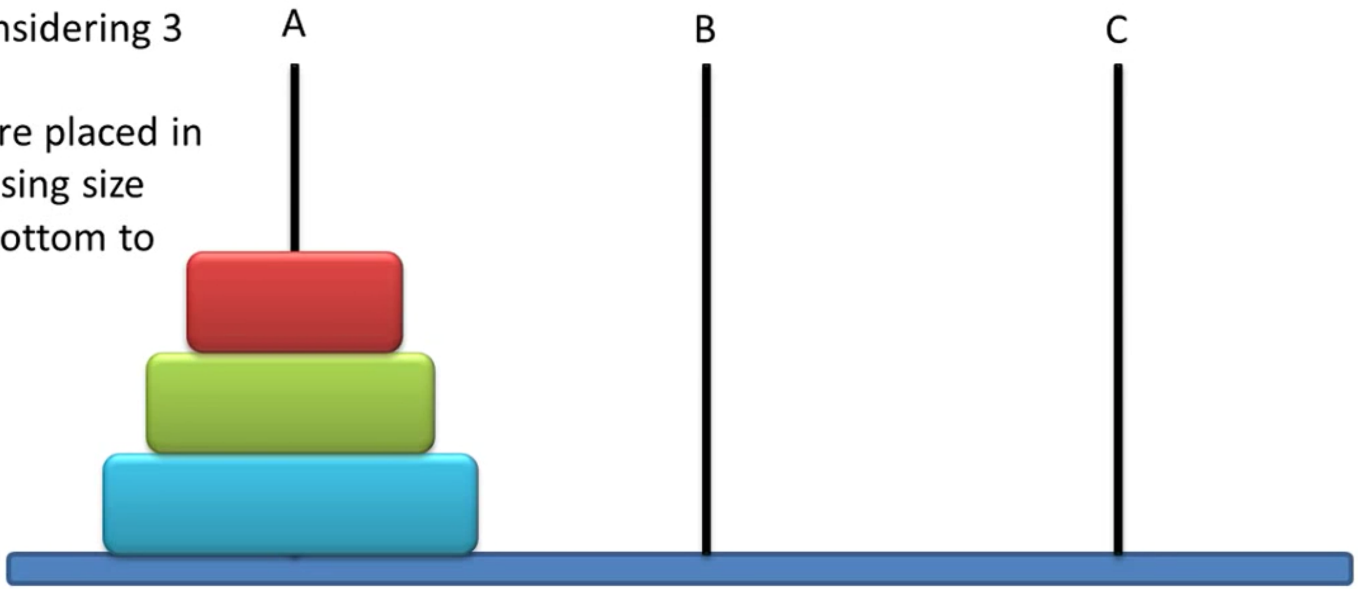
The "Towers of Hanoi" puzzle was invented by the french mathematician Édouard Lucas

- Tower of Hanoi is a very famous game. In this game there **are 3 pegs and N number of disks** placed one over the other in decreasing size.
- The objective of this game is to move the disks one by one from the first peg to the last peg (Only the top disk can be moved each time).
- There is only ONE condition, we can not place a bigger disk on top of a smaller disk.

Towers of Hanoi: Example

The 3 pegs are labeled A, B and C.

In this example we are considering 3 disks.
They are placed in decreasing size from bottom to top.



Our objective is to move the disks from peg A to peg C in such a way that they are in the same order:
RED then GREEN then BLUE
from top to bottom as they are in peg A.

Towers of Hanoi

Before solving the example, let's learn how to solve this problem with lesser amount of disks, say → 1 or 2

Towers of Hanoi for 1 disk

- ❑ If we have only one disk, then it can easily be moved from source to destination peg

Towers of Hanoi for 2 disks

- ❑First, we move the smaller (top) disk to aux peg.
- ❑Then, we move the larger (bottom) disk to destination peg.
- ❑And finally, we move the smaller disk from aux to destination peg

How to solve Tower Of Hanoi?

To solve this game we will follow 3 simple steps recursively.

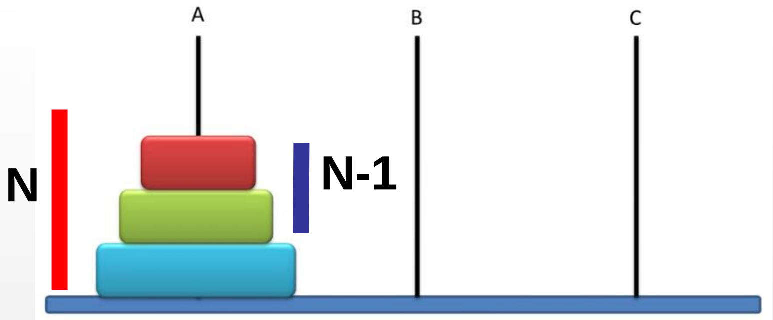
We will use a general notation:
T(N, Beg, Aux, End)

***T** denotes our procedure*
***N** denotes the number of disks*
***Beg** is the initial peg*
***Aux** is the auxiliary peg* (only to help moving the disks).
***End** is the final peg*

Towers of Hanoi

Steps

1. T (N-1, Beg, End, Aux)
2. Move (1, Beg, Aux, End)
3. T (N-1, Aux, Beg, End)



Step 1 − Move top (N-1) disks from **Beg** to **Aux** peg
Step 2 − Move 1 disk from **Beg** to **End** peg
Step 3 − Move top (N-1) disks from **Aux** to **End** peg

Towers of Hanoi: Algorithm

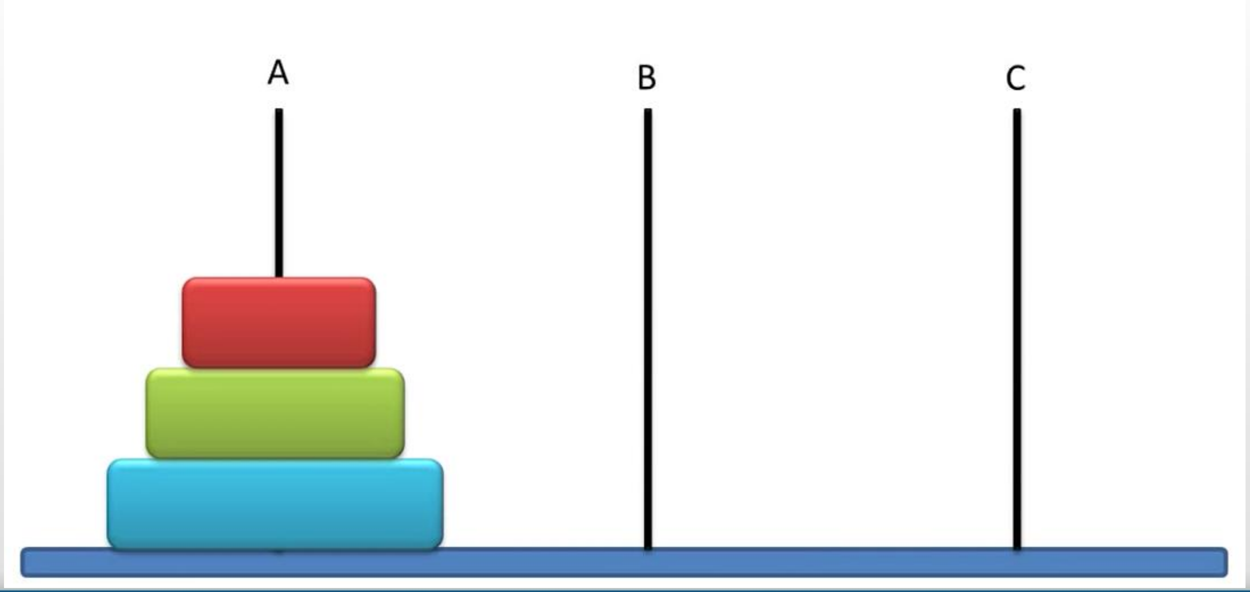
Let's solve this game together. ☺

Towers of Hanoi: Algorithm

We have 3 disks Red, Green and Blue all placed in peg A.

So, N = 3 (Number of disks)

Therefore, we will start with T(3, A, B, C)



Towers of Hanoi: Algorithm

N = 3
Beg = A
Aux = B
End = C

We will follow the 3 steps recursively to find the moves.

Step 1: T (N-1, Beg, End, Aux)
Step 2: Move (1, Beg, Aux, End)
Step 3: T (N-1, Aux, Beg, End)

T(3, A, B, C) ← We will apply the 3 steps on this

Towers of Hanoi: Algorithm

$$T(n,A,B,C) = \begin{cases} T(n-1,A,C,B) \\ A \rightarrow C \\ T(n-1,B,A,C) \end{cases}, \quad n > 0$$

$$T(n,Beg,Aux,end) = \begin{cases} T(n-1,Beg,end,Aux) \\ Beg \rightarrow end \\ T(n-1,Aux,beg,end) \end{cases}$$

Beg Aux end

$T(3,A,B,C)$

# of dishs

- $T(2,A,C,B)$ →
  - $T(1,A,B,C)$ →
    - $T(0,A,C,B)$
    - $A \rightarrow C$ (1)
    - T(0,B,A,C)
  - $A \rightarrow B$ (2)
  - $T(1,C,A,B)$ →
    - $T(0,C,B,A)$
    - $C \rightarrow B$ (3)
    - $T(0,A,C,B)$
- $A \rightarrow C$ (4)
- $T(2,B,A,C)$ →
  - $T(1,B,C,A)$ →
    - $T(0,B,A,C)$
    - $B \rightarrow A$ (5)
    - $T(0,C,B,A)$
  - $B \rightarrow C$ (6)
  - $T(1,A,B,C)$ →
    - $T(0,A,C,B)$
    - $A \rightarrow C$ (7)
    - $T(0,B,A,C)$

$A \rightarrow C$
$A \rightarrow B$
$C \rightarrow B$
$A \rightarrow C$
$B \rightarrow A$
$B \rightarrow C$
$A \rightarrow C$

# of moves $= 2^n - 1 = 2^3 - 1 = 8 - 1 = \underline{\underline{7}}$

# of calls $= 2^{n+1} - 1 = 2^4 - 1 = 16 - 1 = \underline{\underline{15}}$

Towers of Hanoi: Algorithm

Moves

A → C

A → B

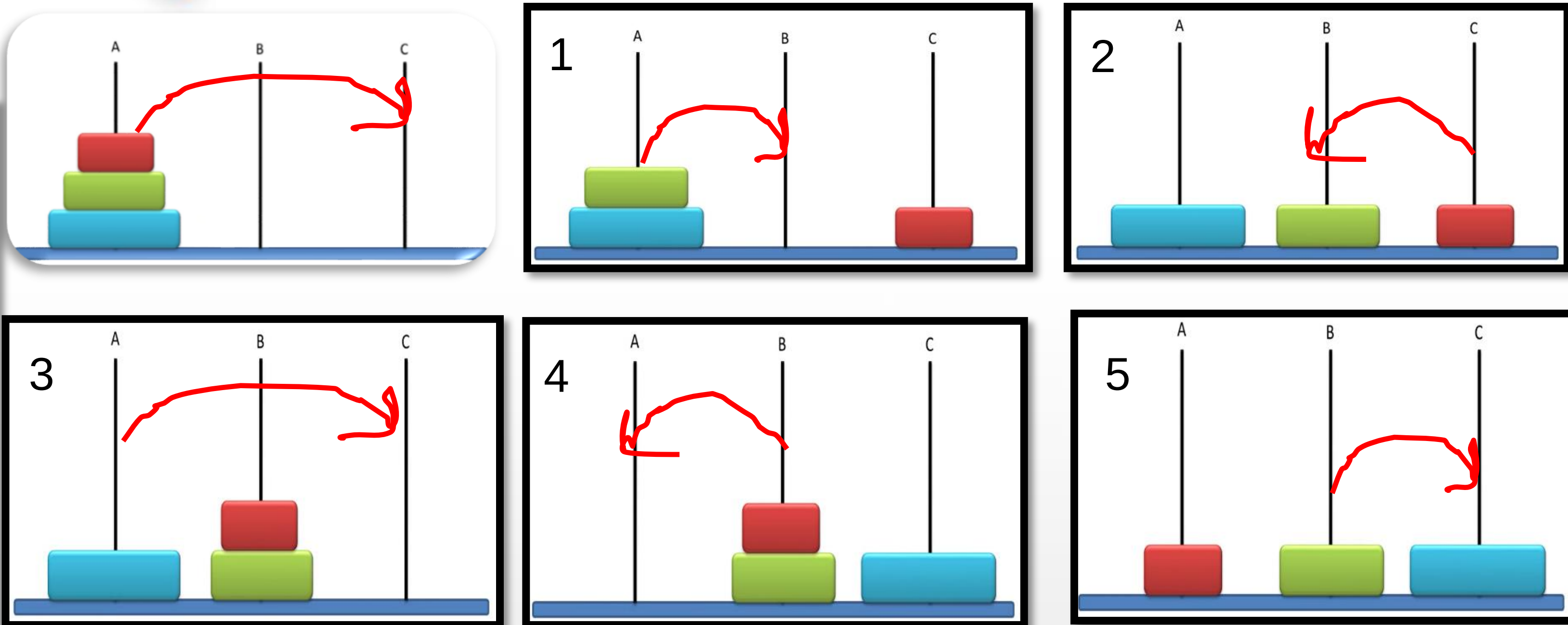
C → B

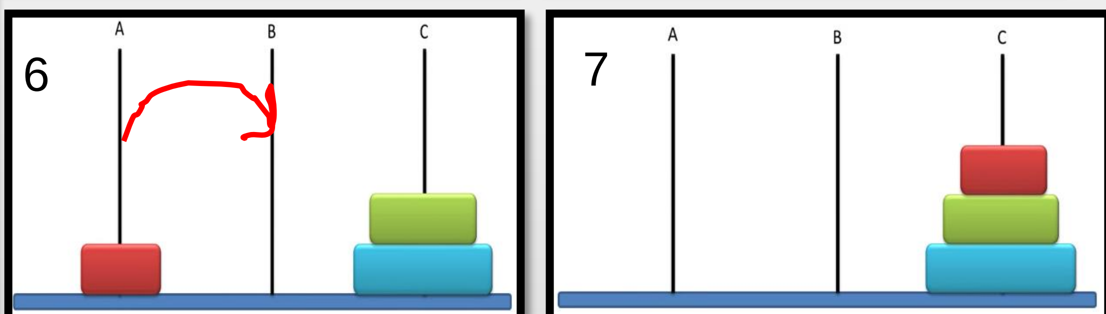
A → C

B → A

B → C

A → C





Tower of Hanoi puzzle with n disks can be solved in minimum 2^n-1 steps

puzzle with 3 disks has taken $2^3 - 1 = 7$ steps

Towers of Hanoi: Algorithm

```
public static void T(int n,char A, char B, char C){

  if (n>0){
       T(n-1,A,C,B);
       System.out.println(A+" ---> "+C);
       T(n-1,B,A,C);
  }

}
```

# of disks= n

Beg = A

Aux = B

End = C

Extra Exercises

Problem 1 **:**

Write a recursive method **`isPalindrome`** that takes a string and returns whether the string is the same forwards as backwards.

```
public static boolean isPalindrome(String s)
{
    // Remove spaces and convert the string to lowercase to handle cases insensitively.
    s = s.replaceAll("\\s", "").toLowerCase();

    // Base case: if the length is 0 or 1, it's a palindrome.
    if (s.length() <= 1) {
        return true;
    } else {
        // Check the first and last characters for equality.
        if (s.charAt(0) == s.charAt(s.length() - 1)) {
            // Recurse on the substring excluding the first and last characters.
            return isPalindrome(s.substring(1, s.length() - 1));
        } else {
            // If the first and last characters do not match, it's not a palindrome.
            return false;
        }
    }
}
```

Problem 2 :

Given integers a and b where a >= b, find their greatest common divisor ("GCD"), which is the largest number that is a factor of both a and b.

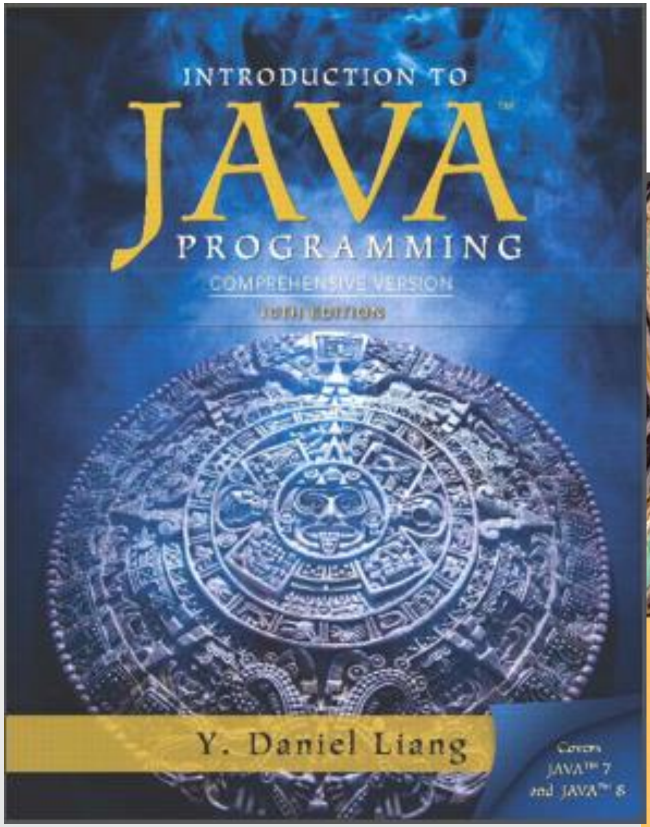
GCD(a, b) = GCD(b, a MOD b)

(Hint: What should the base case be?)

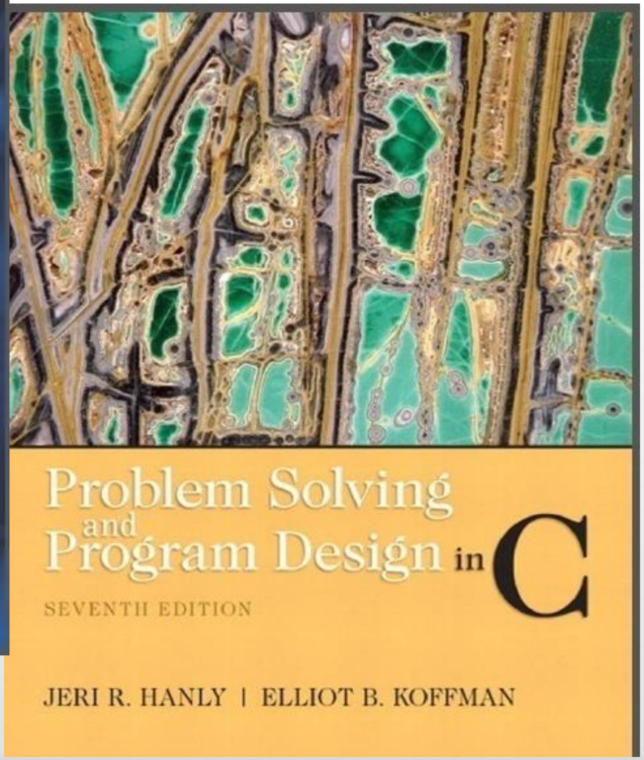
Question?

“Success is the sum of small efforts, repeated day in and day out.”
Robert Collier

Reference



INTRODUCTION TO
JAVA
PROGRAMMING
COMPREHENSIVE VERSION
Y. Daniel Liang



Problem Solving and Program Design in C
SEVENTH EDITION
JERI R. HANLY | ELLIOT B. KOFFMAN