Tema 4

Introducción a la Recursión

Introducción a la recursión

- 4.1. Conceptos básicos
- 4.2. Recursión lineal
- 4.3. Recursión múltiple

Objetivos

- Introducir el concepto de recursión.
- Utilizar correctamente la recursividad en el diseño de programas.
- Contrastar soluciones iterativas y recursivas.

4.1 Conceptos básicos

Formas de definir una función

directa

$$n! = n * (n-1) * \ldots * 1$$

recurrente

$$n! = \begin{cases} n * (n-1)! & si\ n > 0 \\ 1 & caso\ \ contrario \end{cases}$$

Conceptos básicos

$4! = 4 * 3!$ $\quad = 24$

$3! = 3 * 2!$ $\quad = 6$

$2! = 2 * 1!$ $\quad = 2$

$1! = 1 * 0!$ $\quad = 1$

$0! = 1$

$4! = 4 * 3 * 2 * 1 * 1 = 24$

Recurrencia y Recursividad

- Recurrencia:
  - una función aparece en su propia definición.
  - un problema se descompone en subproblemas del *mismo* tipo.
- Realización en Java:
  - subprogramas *recursivos.*
  - en el cuerpo del subprograma aparece una llamada a sí mismo.

Definiciones

- Definición:
  - Un método es *recursivo* si se llama a sí mismo, bien directamente o bien a través de otro método.
- Aplicación:
  - Forma natural de implementar relaciones recurrentes.
  - Técnica de repetición (alternativa al uso de bucles)

Recursión en Java

- Sintaxis:
  - Sintaxis habitual de las llamadas a métodos.
- Semántica:
  - Se deduce del mecanismo habitual de llamada a un método.

Recursión en Java

- Un método es recursivo si contiene llamadas o invocaciones a sí mismo.
- Un método recursivo tendría este aspecto

```
… metodoRecursivo (…) {
    …..
    metodoRecursivo (…);
   // llamada recursiva
   ….
}
```

- Este proceso se repite, hasta que se llegue a un caso base (una llamada que devuelve un resultado o no provoca una llamada recursiva).

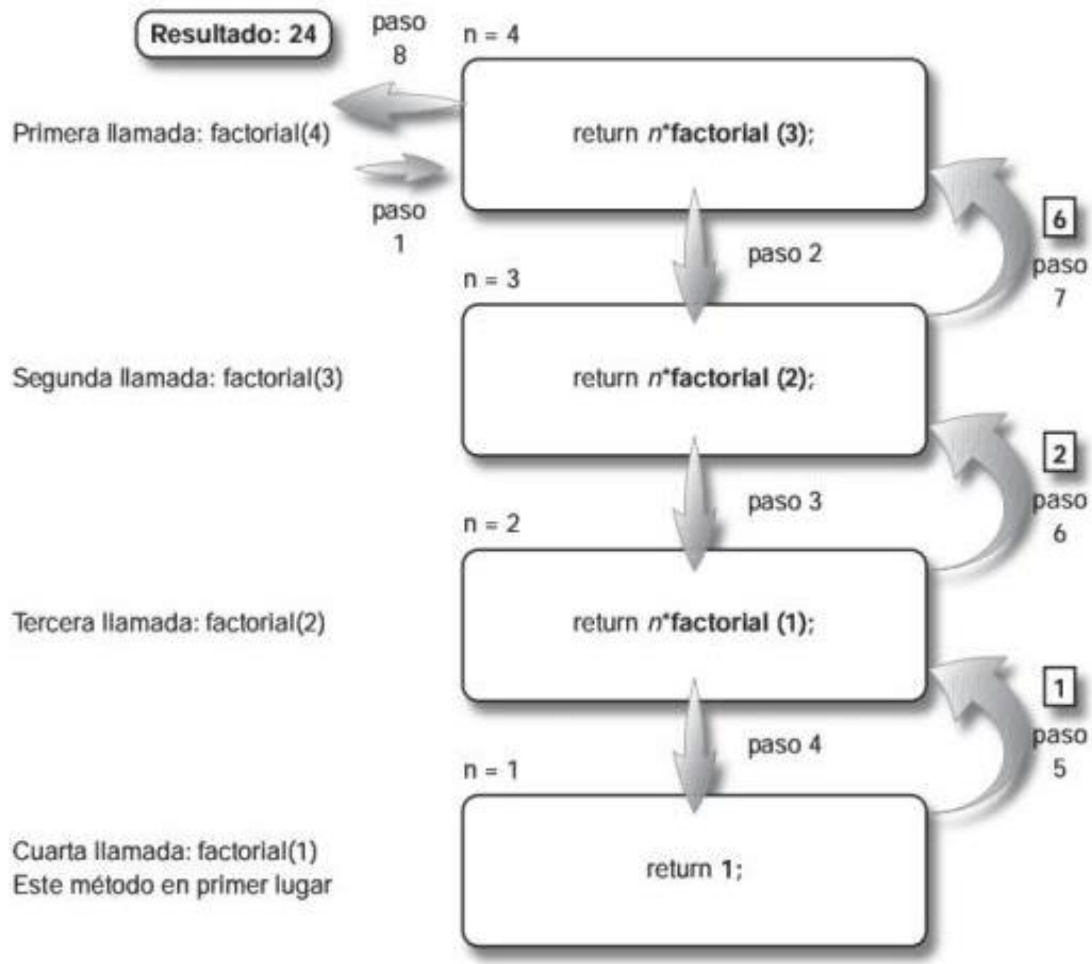
Factorial recursivo

```java
public class Recursion {

    public static void main(String[] args) {
        // TODO code application logic here
        int num;
        num=3;
        System.out.println("El factorial de " + num + " es " + factorial(num));

    }

    static int factorial(int n) {
        if (n > 1) {
            return factorial(n - 1) * n; // caso recursivo
        } else {
            return 1;
        }
    }
    // caso base
}
```

Salida:

```
run:
El factorial de 3 es 6
```

Proceso de llamada



Proceso de llamada

factorial (4) = 4 * factorial (3)

factorial (3) = 3 * factorial (2)

factorial (2) = 2 * factorial (1)

factorial (1) = 1 * factorial (0)

factorial (0) $\Rightarrow$ 1 (Caso Base)

factorial (1) $\Rightarrow$ 1*1 $\Rightarrow$ 1

factorial (2) $\Rightarrow$ 2*1 $\Rightarrow$ 2

factorial (3) $\Rightarrow$ 3*2 $\Rightarrow$ 6

factorial (4) $\Rightarrow$ 4*6 $\Rightarrow$ 24

Partes de un subprograma

- **Caso base**:
  - dados los parámetros de entrada, la solución del problema es “simple”.
  - no se generan llamadas recursivas, y se devuelve directamente una solución.
  - Ejemplo: 0! = 1
- **Caso recurrente**:
  - caso más complejo: *no* hay solución trivial.
  - se reduce a otro caso más simple.
  - Ejemplo: 4! = 4*3!

Recursión infinita

- **Recursión infinita**:
  - Se produce una sucesión infinita de llamadas.
  - El control pasa siempre al caso recurrente, nunca se llega al caso base.
  - Ejemplo:

    factorial(-1) produciría una recursión infinita.

Evitar errores comunes

- Evitar la recursión infinita:
  - Usar una *estructura de selección* (if o switch), para distinguir entre caso base y caso recurrente.
  - Asegurar que los parámetros de la llamada recursiva sean diferentes de los de entrada (condición necesaria para que “se acerquen” al caso base).
  - Comprobar que entre el caso base y los casos no base, se han cubierto todos los estados posibles.
- En los programas recursivos sencillos, *no* suele ser necesario usar *bucles.*
- *Cuando se diseña un algoritmo recursivo hay que identificar qué casos se pueden dar, y que solución se aplica a cada caso*

4.2 Recursión lineal

- Recursividad lineal:
  - cada llamada recursiva genera como máximo otra nueva llamada recursiva.
- Ejemplos:
  - Cálculo recursivo del factorial: *factorial.*
  - Versión recursiva del algoritmo de suma lenta: *SumaLentaRec.*

Ejemplo: Suma lenta recursiva

- Objetivo:
  - Calcular la suma de dos enteros de forma recursiva, utilizando solamente el incremento y decremento en uno.

- Definición recurrente de la suma lenta $+_{SL}$:

$$a +_{SL} b = \begin{cases} b & si \quad a = 0 \\ (a-1) +_{SL} (b+1) & si \quad a \neq 0 \end{cases}$$

Recursión por la cola

- Recursividad por la cola:
  - Caso especial de la recursividad lineal.
  - No se realizan operaciones con el resultado que devuelve una llamada recursiva.
  - El resultado es el que devuelve la última llamada.
- Ejemplos:
  - *factorial* NO es recursivo por la cola, porque se multiplica el resultado de la llamada recursiva por *num.*
  - *SumaLentaRec* SI es recursivo por la cola.

4.3 Recursión múltiple

- Recursividad múltiple (ó no lineal)
  - Alguna llamada genera dos o más nuevas llamadas recursivas.
- Ejemplos:
  - Números de Fibonacci.
  - Algoritmo recursivo para las Torres de Hanoi.

4.3 Recursión múltiple

- Sucesión de Fibonacci:

  $(fib_i)_{i \in N}$ = 1,1,2,3,5,8,13,21,34,…

  - $Fib_0 = 1$
  - $Fib_1 = 1$
  - $Fib_2 = Fib_0 + Fib_1$
  - $Fib_3 = Fib_1 + Fib_2$
  - …
  - $Fib_n = Fib_{n-2} + Fib_{n-1}$

Caso Base

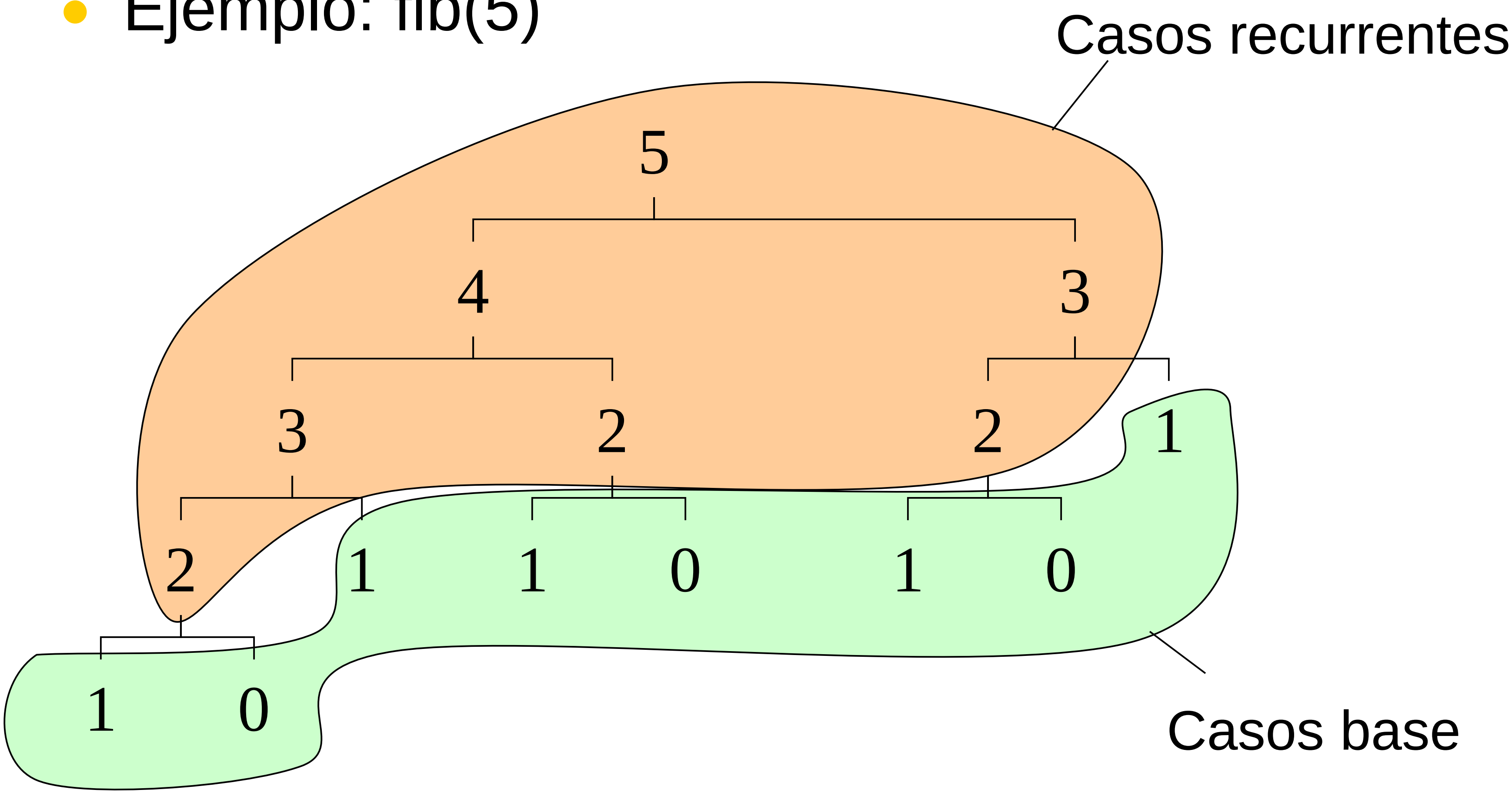
Ley de Recurrencia

4.3 Fibonacci codificación

```
static int fib(int n) {
    // siendo n un número entero no negativo
    if (n > 1) {
        return fib(n - 1) + fib(n - 2); // caso recursivo: para n>1
    } else {
        return n;
        // caso base: par n 00 0 n=1;
    }

}
```

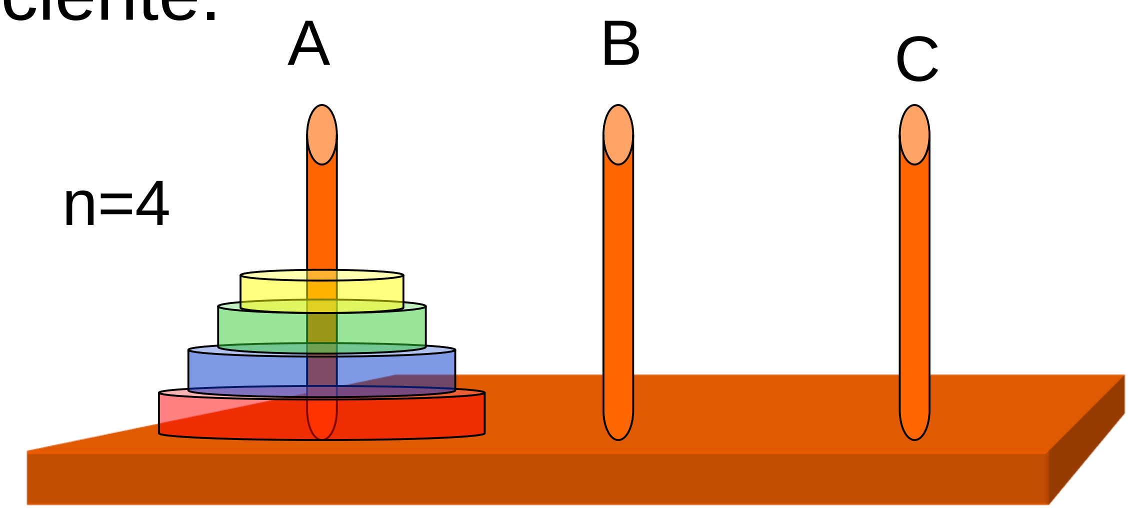
Nº de Fibonacci: Árbol de llamadas

- Ejemplo: fib(5)



Torres de Hanoi

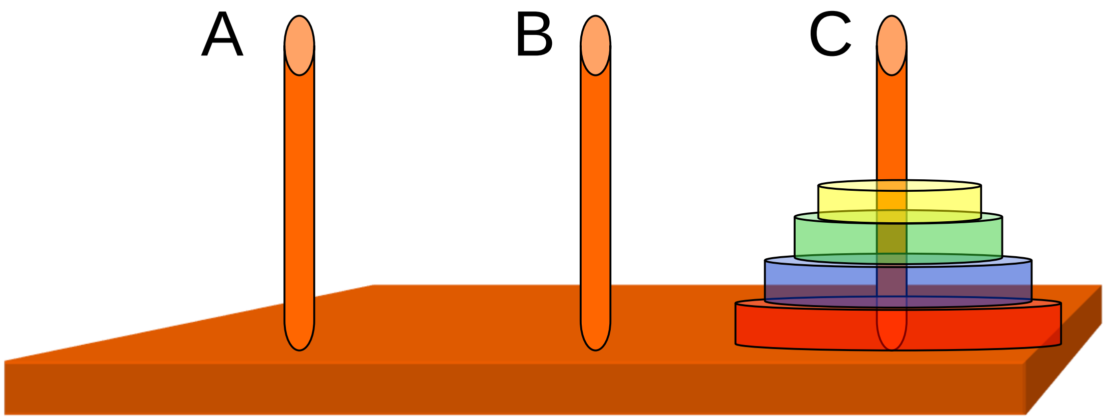
- Juego de sencilla solución recursiva.
- Situación inicial:
  - 3 agujas verticales *A*, *B* y *C*
  - en una de ellas hay *n* discos de tamaño creciente.



Torres de Hanoi

Objetivo:

Pasar los *n* discos en el mismo orden a otra aguja.



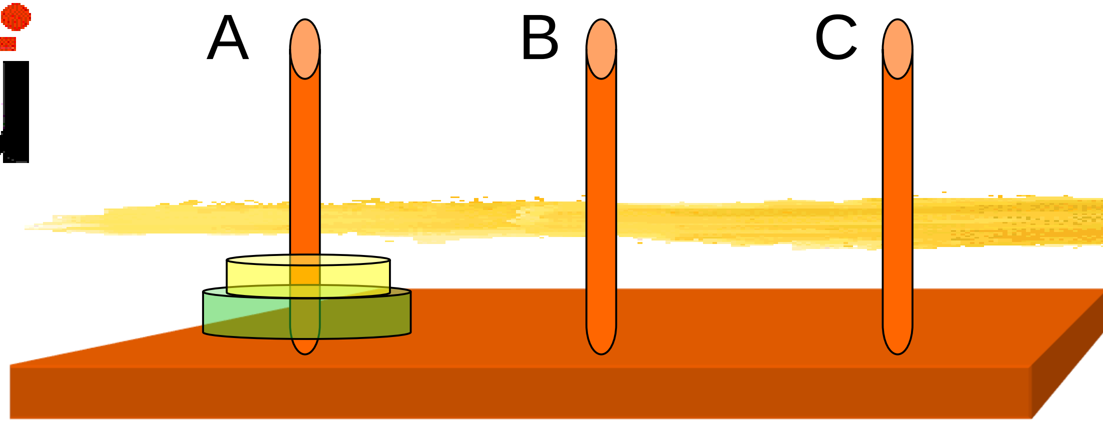
Restricciones:

- los discos se pasan de uno en uno.
- un disco NUNCA debe descansar sobre otro de menor tamaño.

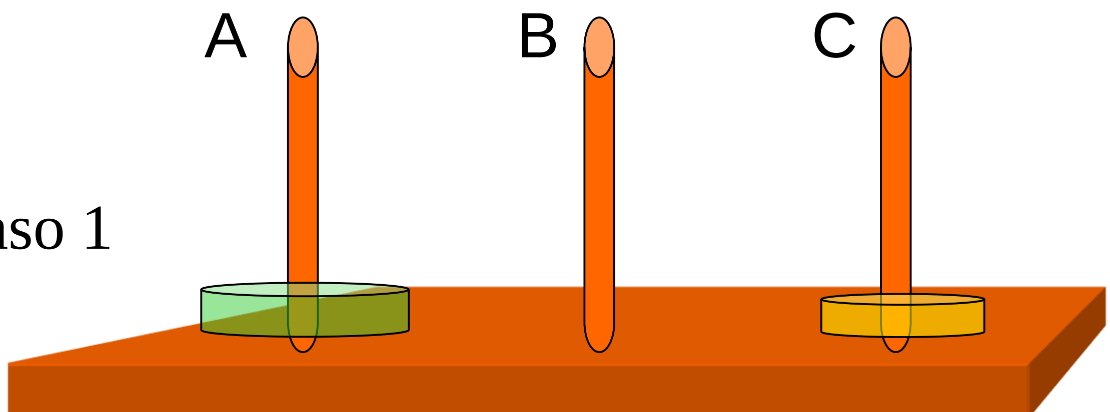
Torres de Hanoi: Algoritmo

- Caso n=1:
  - Pasar 1 disco de A $\rightarrow$ B
    - trivial

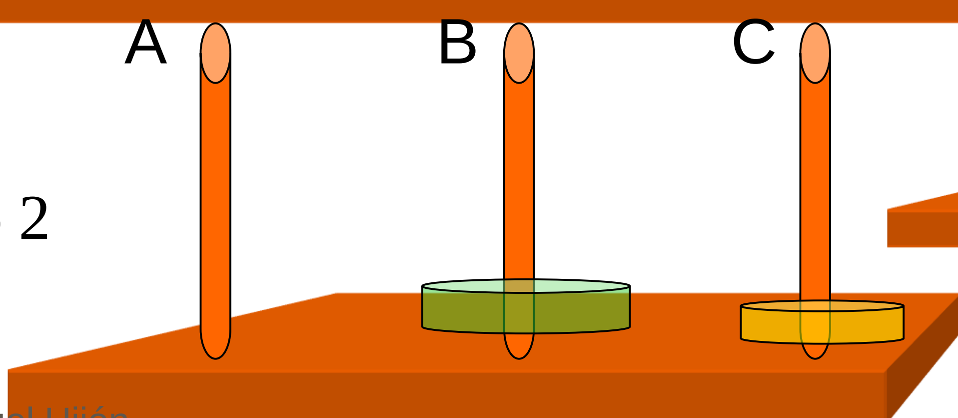
- Caso n=2:
  - Pasar 2 discos de A $\rightarrow$ B
    - mover disco de A $\rightarrow$ C
    - mover disco de A $\rightarrow$ B
    - mover disco de C $\rightarrow$ B



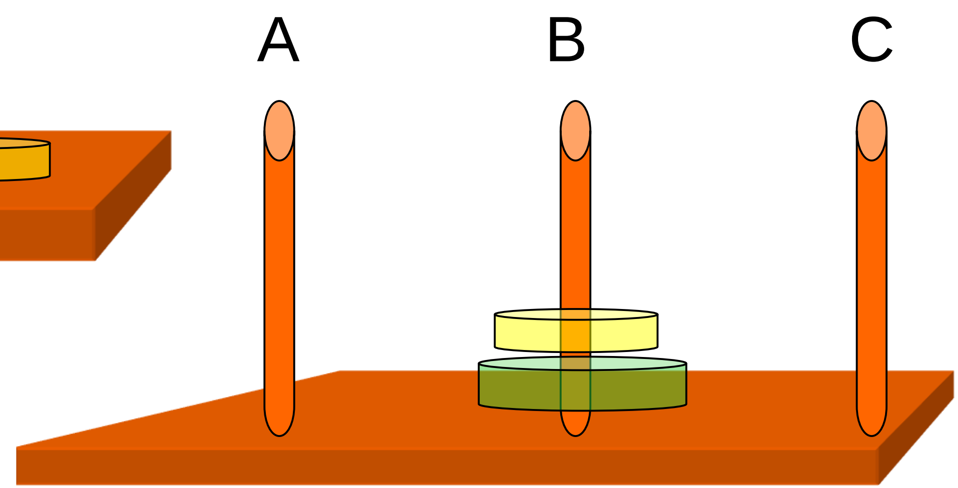
Paso 1



Paso 2



Paso 3



Torres de Hanoi: Algoritmo

- Caso n=3:
  - Pasar 3 discos de A → B

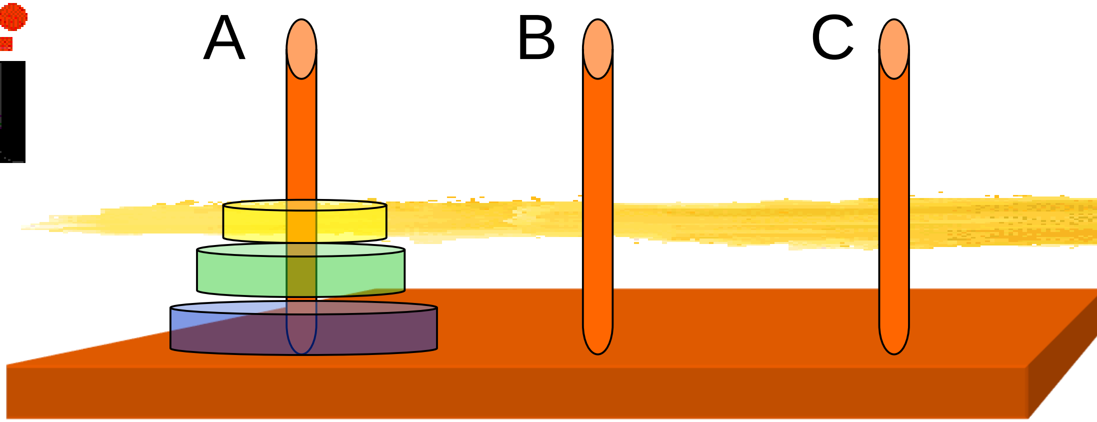
2 de A → C

mover A → B
mover A → C
mover B → C

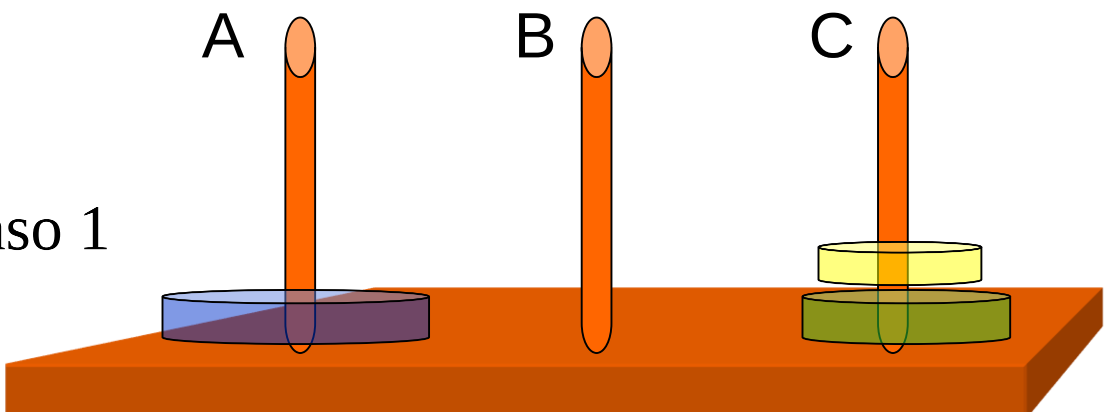
1 de A → B

2 de C → B

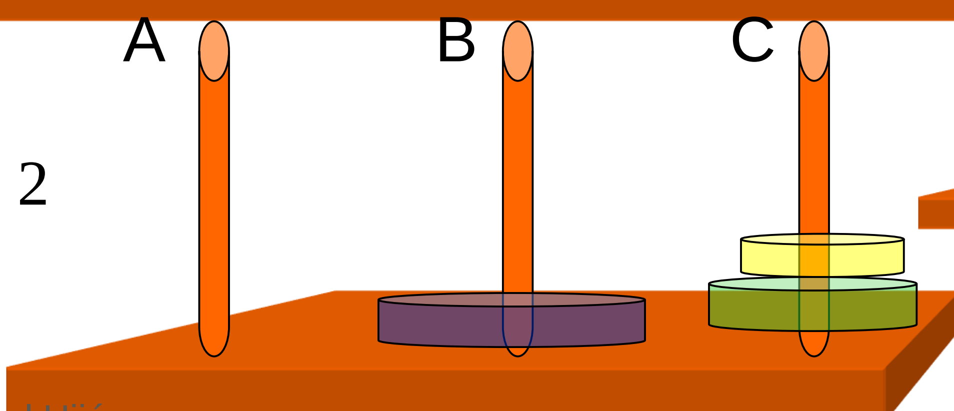
- mover C → A
- mover C → B
- mover A → B



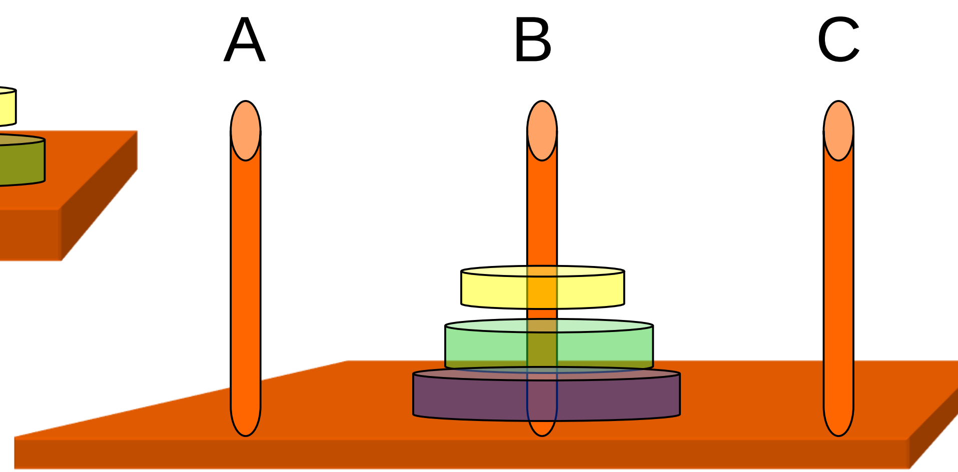
Paso 1



Paso 2



Paso 3



Torres de Hanoi: Algoritmo

- Caso general:
  - Pasar *n* discos de A $\rightarrow$ B
    - Pasar *n*-1 discos de A $\rightarrow$ C
    - Mover disco de A $\rightarrow$ B
    - Pasar *n*-1 discos de C $\rightarrow$ B

Torres de Hanoi: Algoritmo

MoverDiscos(4,'A','B','C')

MoverDiscos(3,'A','C','B')

MoverDiscos(2,'A','B','C')

MoverDiscos(1,'A','C','B')

MoverDiscos(0,'A','B','C')

Se pasa el disco 1 de A a C

MoverDiscos(0,'B','C','A')

Se pasa el disco 2 de A a B

MoverDiscos(1,'C','B','A')

MoverDiscos(0,' ...)

Se pasa el disco 1 de C a B ...

Torres de Hanoi: Traza

- Ejemplo de funcionamiento:
  - Llamada: MoverDiscos(4,’A’,’B’,’C’)
  - Resultado:

Se pasa el disco 1 de A a C
Se pasa el disco 2 de A a B
Se pasa el disco 1 de C a B
Se pasa el disco 3 de A a C
Se pasa el disco 1 de B a A
Se pasa el disco 2 de B a C
Se pasa el disco 1 de A a C
Se pasa el disco 4 de A a B

Se pasa el disco 1 de C a B
Se pasa el disco 2 de C a A
Se pasa el disco 1 de B a A
Se pasa el disco 3 de C a B
Se pasa el disco 1 de A a C
Se pasa el disco 2 de A a B
Se pasa el disco 1 de C a B

Iteración y Recursión

Repetición de bloques de instrucciones

Técnica	Iteración	Recursión
Realización	Bucles (while, for, do-while)	Subprogramas que se llaman a sí mismos

Iteración y Recursión

- Equivalencia de Iteración y Recursión:

  Cualquier cómputo recursivo puede expresarse de forma iterativa *y viceversa.*

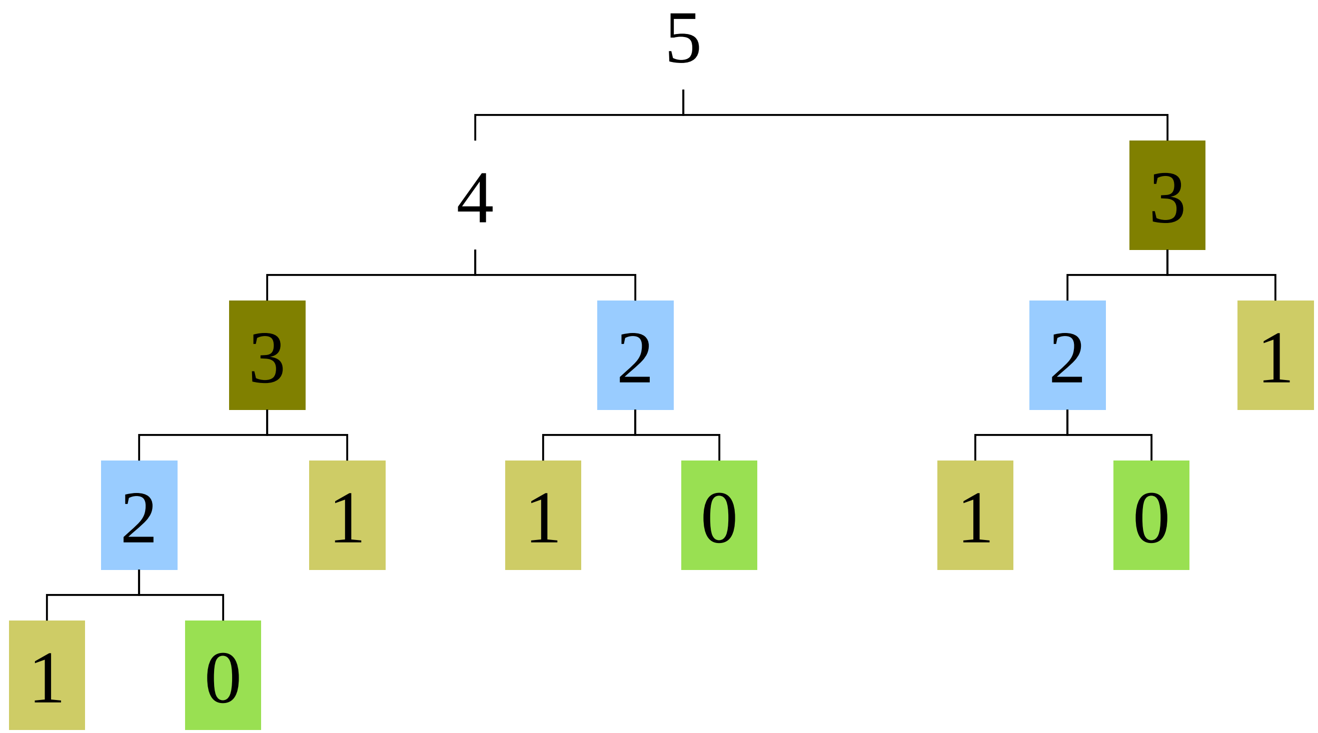
- Ejemplos:
  - Factorial: *fac* y *factorial.*
  - Suma lenta: *sumaLenta* y *sumaLentaRec.*
  - Nº de Fibonacci: *fib* y *fibIter.*

Claridad vs. Eficiencia

- Claridad:
  - muchos problemas se resuelven de forma “elegante” mediante recursión, requiriendo programas complejos y/o poco intuitivos en su versión iterativa.
  - Ejemplo: las Torres de Hanoi.
- Eficiencia:
  - hay que tener en cuenta también la complejidad añadida por la recursión.
  - Ejemplo: los números de Fibonacci.

Fibonacci: Llamadas repetidas

- Ejemplo: fib(5)



Recomendaciones técnicas

Utilizar recursividad:

- cuando clarifique el algoritmo y el programa que soluciona un problema.
- cuando no haya fuertes restricciones de memoria o tiempo de ejecución.

4.4. Recursión mutua

- Recursión simple (directa)
  - un subprograma llama a sí mismo.
- Recursión mutua (indirecta):
  - Definición de dos o más subprogramas basándose recíprocamente en ellos mismos.
  - La recursividad en el subprograma se produce *indirectamente*.
  - Un subprograma *A* llama a *B*, y *B* llama (directamente o indirectamente) a *A*.