Departamento de Ingeniería Informática
Grado en Ingeniería Informática
Elisa Guerrero Vázquez
Esther L. Silva Ramírez

Metodología de la Programación

Tema 1 – Teoría

RECURSIVIDAD

Contenidos

1.1.Introducción

1.2.Concepto de recursión a través de la inducción matemática

1.3. La recursividad

1.4. Tipos de recursividad

1.5. Transformaciones sobre algoritmos recursivos

1.6. Transformaciones sobre algoritmos iterativos

1.7. Las Torres de Hanoi

1. Introducción

- **Recursividad**: Generación de llamadas al propio subalgoritmo que se está definiendo.

- **Recursividad e Iteración**:
  - Ambos son mecanismos básicos para describir pasos que han de repetirse de un cierto número de veces con pequeñas variaciones.
  - Ambos tienen la misma potencia expresiva.

2. Concepto a través de la Inducción Matemática

- **Demostración por Inducción:**
  - Base de la Inducción: Se demuestra para 0 ó 1
  - Hipótesis de Inducción: Se supone para n
  - Paso Inductivo: Se demuestra para *n*+1
- **Diseño de una función recursiva es similar**
  - Cálculo del valor que devuelve en el caso base
  - Suponiendo que la función sabe calcular el resultado para n-1
  - Se escribe para n

3. La Recursividad

Definición de Factorial:

- $n! = n \times (n-1) \times (n-2) \times ... \times 1$

  $4! = 4 \times 3 \times 2 \times 1 = 24$

- $n! = n \times (n-1)!$ si $n>0, 0! = 1$

  $4! = 4 \times 3!$ $= 24$

  $3! = 3 \times 2!$ $= 6$

  $2! = 2 \times 1!$ $= 2$

  $1! = 1 \times 0!$ $=1$

  $0! = 1$

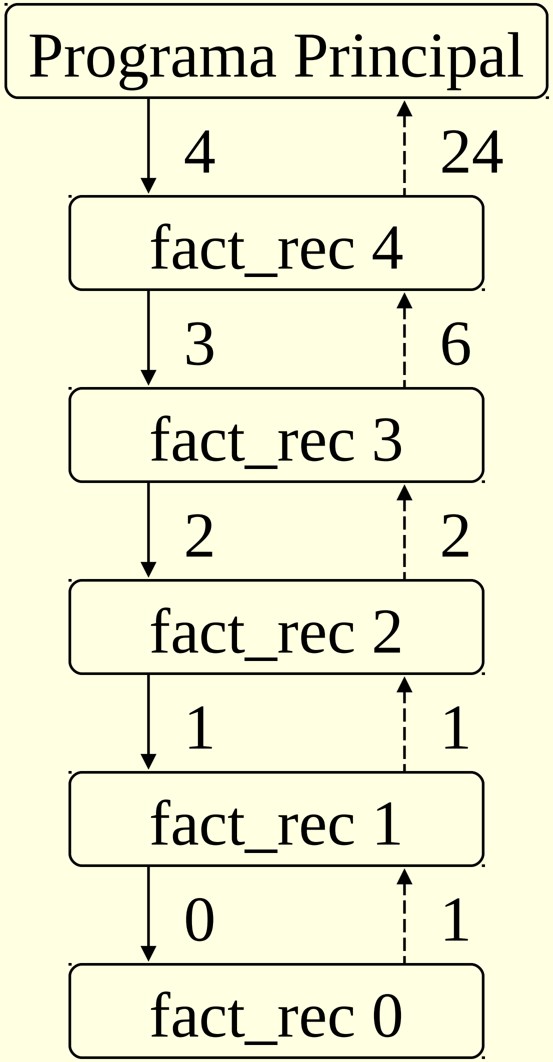
```
// Cabecera: entero factorial_iter (E entero: n)
// Precondición: recibe n, un número natural mayor o igual que 0.
// Postcondición: devuelve n! (el factorial de n)

entero función fact_iter(E entero: n)
var
    entero: i, prod
inicio
    prod ← 1
    desde i ← 1 hasta n hacer
        prod ← prod × i
    fin_desde
    devolver prod
fin_función
```

```
// Cabecera: entero factorial (E entero: n)
// Precondición: recibe n, un número natural mayor o igual que 0.
// Postcondición: devuelve n! (el factorial de n)

entero función fact_rec(E entero: n)
inicio
    si n = 0 entonces
        devolver 1
    si no
        devolver n × fact_rec(n-1)
    fin_si
fin_función
```

fact_rec(4)
4 * fact_rec(3)
4 * (3 * fact_rec(2))
4 * (3 * (2 * fact_rec(1)))
4 * (3 * (2 * (1 * fact_rec(0))))
4 * (3 * (2 * (1 * 1)))
4 * (3 * (2 * 1))
4 * (3 * 2)
4 * 6
24



→ Valor llamada
⇢ Valor devuelto

Procedimiento Recursivo:

- Caso Base $0! = 1$
- Caso General $n! = n \times (n-1)!$

Caso General debe reducir el “tamaño de la entrada”

4. Tipos de Recursividad

- Recursividad lineal:
  - Recursividad No Final
  - Recursividad Final
- Recursividad múltiple

Recursividad Lineal Final

factorial(4)
fact_recF(1, 1)
fact_recF(1, 2)
fact_recF(2, 3)
fact_recF(6, 4)
fact_recF(24, 5)
24

```
entero función factorial(E entero: n)
inicio
  devolver fact_recF(n, 1)
fin_función
```

```
entero función fact_recF(E entero: n, E entero: prod)
inicio
  si n=0 entonces
        devolver prod
  si_no
  devolver fact_recF(n-1, prod*n)
fin_función
```

Funciones Mutuamente Recursivas

lógico función num_par?(E entero: *n*)
inicio
    Si *n* = 0 **entonces**
        devolver verdadero
    si_no
        devolver num_impar?(*n*-1)
fin_función

lógico función num_impar?(E entero: *n*)
inicio
    si *n* = 0 **entonces**
        devolver falso
    si_no
        devolver num_par?(*n*-1)
fin_función

num_par?(88)
num_impar?(87)
num_par?(86)
num_impar?(85)
...
num_par?(2)
num_impar?(1)
num_par?(0)
verdadero

Recursividad Múltiple

Función de Fibonacci:

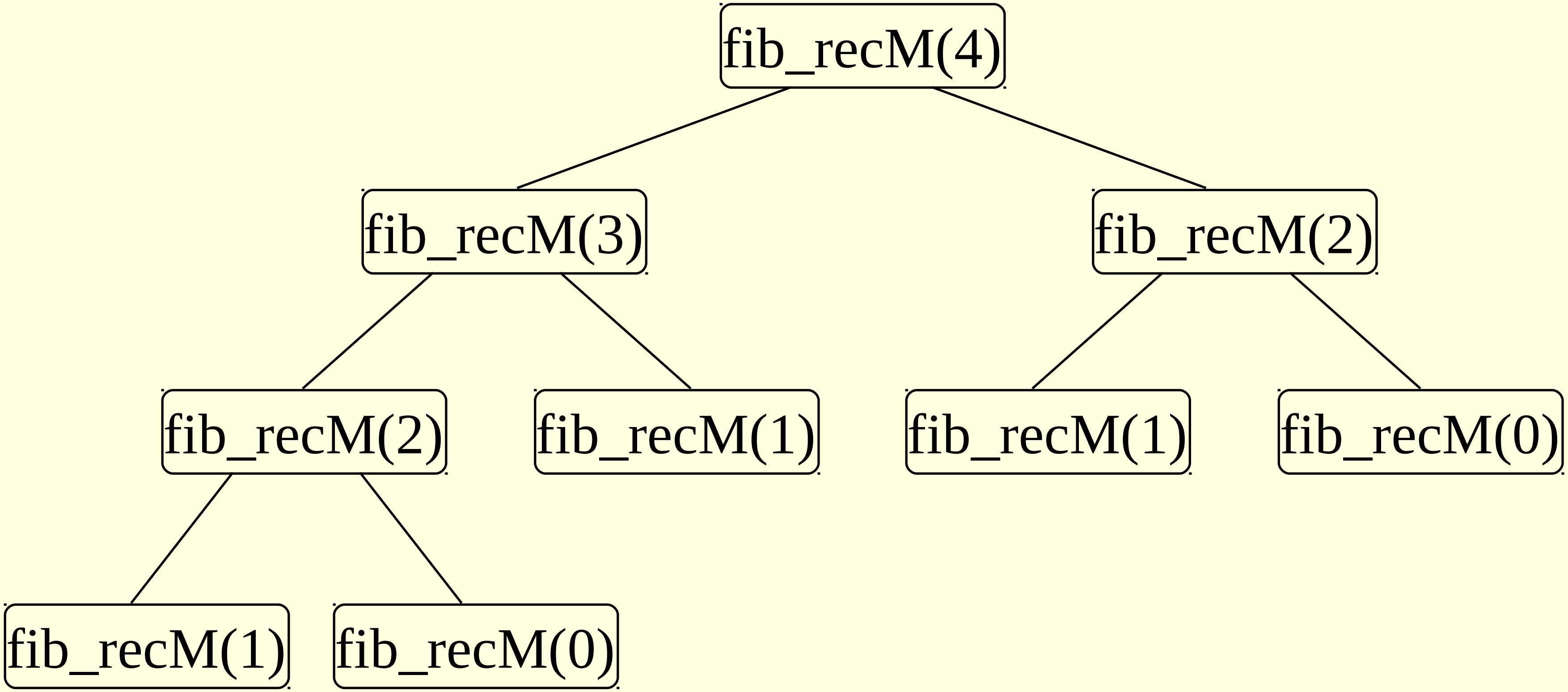
Fib(0) = 0
Fib(1) = 1
Fib(n) = Fib(n-1) + Fib(n-2)   Si $n > 1$

//Cabecera: entero fib_recM(E entero: *n*)
//**Precondición:** recibe un número natural *n* mayor o igual que 0
//**Postcondición:** devuelve el número de la serie de Fibonacci que ocupa la posición *n*

Recursividad Múltiple

```
entero función fib_recM(E entero: n)
inicio
    según_sea n hacer
        0: devolver 0
        1: devolver 1
        en_otro_caso: devolver fib_recM(n - 1) + fib_recM(n - 2)
fin_función
```

Árbol de llamadas



5. Transformaciones sobre alg. recursivos

- Técnicas de inmersión:
  - **Concepto**: Generalizar el algoritmo, con más parámetros de entrada y/o salida.

    Función original → sumergida

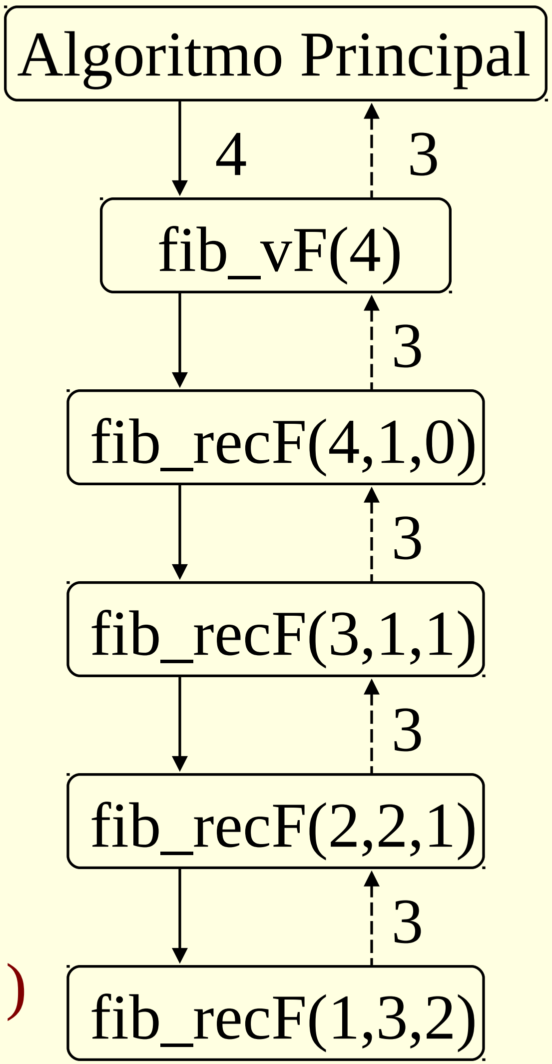
    Función que generaliza → inmersora
  - **Problema**: ¿Cómo generalizar?
  - **Objetivo**:
    - Obtener alg. recursivos más eficientes

Mejorar Eficiencia Fibonacci

//Cabecera: entero fib_vF(E entero: *n*)
//**Precondición:** recibe un número natural *n* mayor o igual que 0
//**Postcondición:** devuelve el número de la serie de Fibonacci que ocupa la posición *n*

//Cabecera: entero fib_recF(E entero: *n*,
E entero: *f1*, E entero: *f2*)
//**Precondición:** recibe un número natural *n* mayor o igual que 0
//**Postcondición:** devuelve el número de la serie de Fibonacci que ocupa la posición *n*, conteniendo en *f1* Fib(n) y en *f2* Fib(n-1). Si *n*=0 devuelve el valor 0.

entero **función** fib_vF(E entero: *n*)
inicio
    devolver fib_recF(*n*,1,0)
fin_función



entero **función** fib_recF(E entero: *n*,
        E entero: *f1*, E entero: *f2*)
inicio
    según_sea *n* **entonces**
        0: **devolver** 0
        1: **devolver** *f1*
        en_otro_caso:
            devolver fib_recF (*n*-1, *f1*+*f2*, *f1*)
fin_función

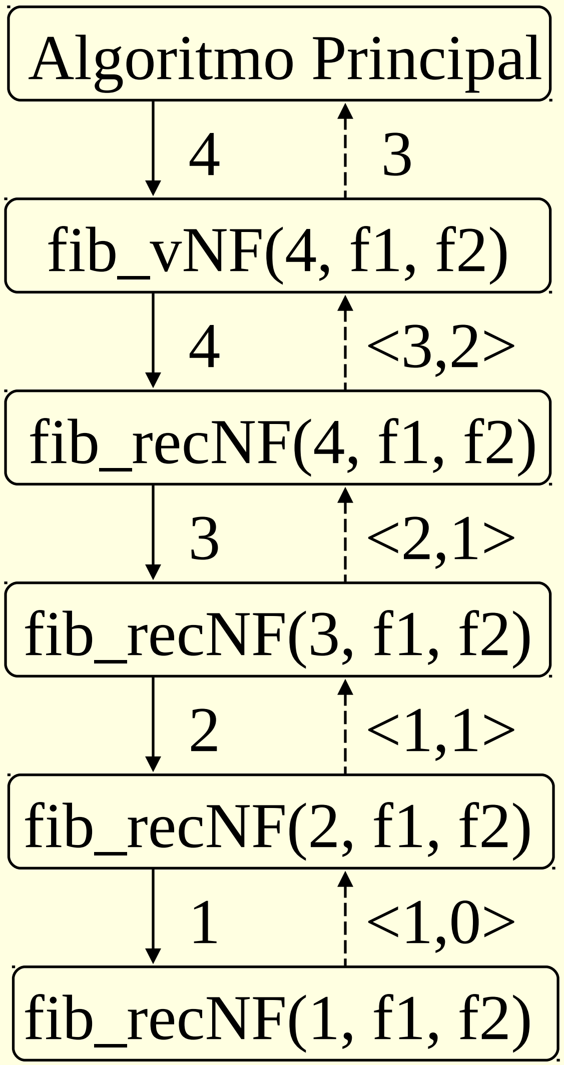
Mejorar Eficiencia Fibonacci

//Cabecera: entero fib_vNF(E entero: *n*)
//**Precondición:** recibe un número natural *n* mayor o igual que 0
//**Postcondición:** devuelve el número de la serie de Fibonacci que ocupa la posición *n*

//Cabecera: fib_recNF(E entero: *n*, S entero: *f1*, S entero: *f2*)
//**Precondición:** recibe un número natural *n* mayor o igual que 0
//**Postcondición:** devuelve en *f1* Fib(n) y en *f2* Fib(n-1). Si *n*=0 el valor de *f2* queda indeterminado.

entero **función** fib_vNF(*n*)
var entero: *f1, f2*
inicio
    fib_recNF(*n*, *f1*, *f2*)
    devolver *f1*
fin_función

procedimiento fib_recNF(E entero: *n*, S entero: *f1*, S entero: *f2*)
var entero: *aux*
inicio
    según_sea *n* **entonces**
        0: *f1* ← 0
        1: *f1* ← 1; *f2* ← 0
        en_otro_caso: fib_recNF(*n*-1, *f1*, *f2*)
            aux ← *f1*; *f1* ← *f1*+*f2*; *f2* ← *aux*
fin_procedimiento



5. 1.Transformaciones sobre alg. Recursivos: GDP

- Técnicas de desplegado/plegado:
  - **Concepto**: se realiza un inmersión incluyendo un parámetro donde se almacenan los resultados ya calculados.
  - **Problema**: ¿Cómo buscar la inmersión?
  - **Objetivo**:
    - convertir un alg. recursivo no final en un alg. recursivo final (+ eficiente)
    - Obtener alg. recursivos más sencillos

Función Recursiva Lineal No Final

tipo **función** f_rec(*x*)
inicio
    si caso_base? (*x*) **entonces**
        devolver sol(*x*)
    si_no
        devolver comb(f_rec(suc(*x*)), *x*)
fin_función

Función Recursiva Lineal Final

tipo **función** f_recFinal(*x*,*w*)
inicio
    si caso_base?(*x*) **entonces**
        devolver comb(sol(*x*),*w*)
    si_no
        devolver f_recFinal(suc(*x*), comb(*x*,*w*))
    fin_si
fin_función

a) **Generalización**:

Obtener la inmersión f_recFinal considerando la expresión del caso general de la función a transformar:

$$\mathrm{f\_rec}(x) = \mathrm{comb}(\ \mathrm{f\_rec}(\mathrm{suc}(x)),\ x)$$

Se propone una función más general que *f_rec*, añadiendo parámetros adicionales (variables de inmersión), obteniendo la función inmersora *f_recFinal* en términos de *f_rec*:

$$\mathrm{f\_recFinal}(x,w) = \mathrm{comb}(\mathrm{f\_rec}(x),\ w)$$

Si la función *comb* tiene elemento neutro w_0:

$$\mathrm{f\_recFinal}(x,\ w_0) = \mathrm{comb}(\mathrm{f\_rec}(x),\ w_0) = \mathrm{f\_rec}(x)$$

Se obtienen los valores con los que realizar la llamada inicial a la función recursiva final: $\mathrm{f\_recFinal}(x,\ w_0)$

b) Desplegado:

Se crea la función inmersora f_recFinal siguiendo el mismo análisis de casos que f_rec.

f_recFinal(*x*,*w*) = comb(f_rec(*x*), *w*)

```
tipo función f_rec(x)
inicio
   si caso_base? (x) entonces
      devolver sol(x)
   si_no
      devolver
      comb(f_rec(suc(x)), x)
   fin_si
fin_función
```

```
tipo función f_recFinal(x,w)
inicio
   si caso_base? (x) entonces
      devolver comb(sol(x),w)
   si_no
      devolver
      comb(comb(f_rec(suc(x)),x),w)
   fin_si
fin_función
```

- Si *comb* es asociativa, se puede reorganizar el caso general:

  comb(comb(f_rec(suc(x)),x),w) =
  comb(f_rec(suc(x)),comb(x,w))

Obtendríamos la definición de f_recFinal como sigue:

tipo **función** f_recFinal(*x*,*w*)
inicio
si caso_base? (*x*) **entonces**
devolver comb(sol(*x*),*w*)
si_no
devolver comb(f_rec(suc(*x*)),comb(*x*,*w*))
fin_si
fin_función

c) **Plegado**: f_recFinal(x,w) = comb(f_rec(x), w)

```
tipo función f_recFinal(x,w)
 inicio
   si caso_base? (x) entonces
     devolver comb(sol(x),w)
   si_no
     devolver
       comb(f_rec(suc(x)),comb(x,w))
   fin_si
 fin_función
```

```
tipo función f_recFinal(x,w)
 inicio
   si caso_base? (x)
 entonces
     devolver comb(sol(x),w)
   si_no
     devolver
       f_recFinal(suc(x),
                  comb(x,w))
   fin_si
 fin_función
```

Si *comb* es asociativa y tiene elemento neutro, podemos realizar esta transformación, de lo contrario pueden surgir problemas.

5.2. Transformación alg. recursivos en iterativos

- Motivos:
  - Lenguaje no soporta recursividad.
  - Reducir coste en tiempo y espacio.
- Considerar:
  - Pérdida de legibilidad.
  - Optimización recursividad final.
  - Problemas recursivos por naturaleza.
- Métodos:
  - Uso de una Pila, Rec. Lineal y Rec. Final

Función Recursiva Lineal

```
tipo función f_rec(x)
inicio
    si caso_base? (x) entonces
        devolver sol(x)
    si_no
        devolver comb( f_rec(suc(x)), x)
fin_función
```

Función Recursiva Final

```
tipo función f_recFinal(x)
inicio
    si caso_base? (x) entonces
        devolver sol(x)
    si_no
        devolver f_recFinal(suc(x))
fin_función
```

Transformación Recursividad Final

Función Recursiva Final

```
tipo función f_recFinal(x)
inicio
    si caso_base? (x) entonces
        devolver sol(x)
    si_no devolver f_recFinal(suc(x))
fin_función
```

Esquema Función Iterativa

```
tipo función f_iter(x)
inicio
    mientras ¬ caso_base? (x) hacer
        x ← suc (x)
    fin_mientras
    devolver sol(x)
fin_función
```

Ejemplo: Factorial Recursivo Final

Función Recursiva Final → Función Iterativa (Optimizada)

entero **función** factorial(E entero: *n*)
    devolver fact_recF(*n*, 1)
fin_función

entero **función** fact_recF(E entero: *n*, E entero: *prod*)
inicio
  si *n*=0 **entonces devolver** *prod*
  si_no devolver
    fact_recF(*n*-1,*prod*n)*
fin_función

entero **función** fact_iter(E entero: *n*, E entero: *prod*)
inicio
    mientras $n \neq 0$ **hacer**
        $prod \leftarrow prod * n$
        $n \leftarrow n-1$
    fin_mientras
    devolver *prod*
fin_función

Transformación Recursividad Lineal

Función Recursiva Lineal ⟶ Esquema Función Iterativa

tipo **función** f_rec(*x*)
inicio
  si caso_base? (*x*) **entonces**
    devolver sol(*x*)
  si_no
    devolver comb(f_rec(suc(*x*)), *x*)
fin_función

tipo **función** f_iter(*x*)
var *res*, *c*
inicio
  $c \leftarrow 0$
  mientras ¬ caso_base? (*x*) **hacer**
    $c \leftarrow c + 1$
    $x \leftarrow suc\ (x)$
  fin_mientras
  $res \leftarrow sol(x)$
  mientras $c \neq 0$ **hacer**
    $c \leftarrow c - 1$
    $x \leftarrow suc^{-1}\ (x)$
    $res \leftarrow comb\ (res, x)$
  fin_mientras
  devolver *res*
fin_función

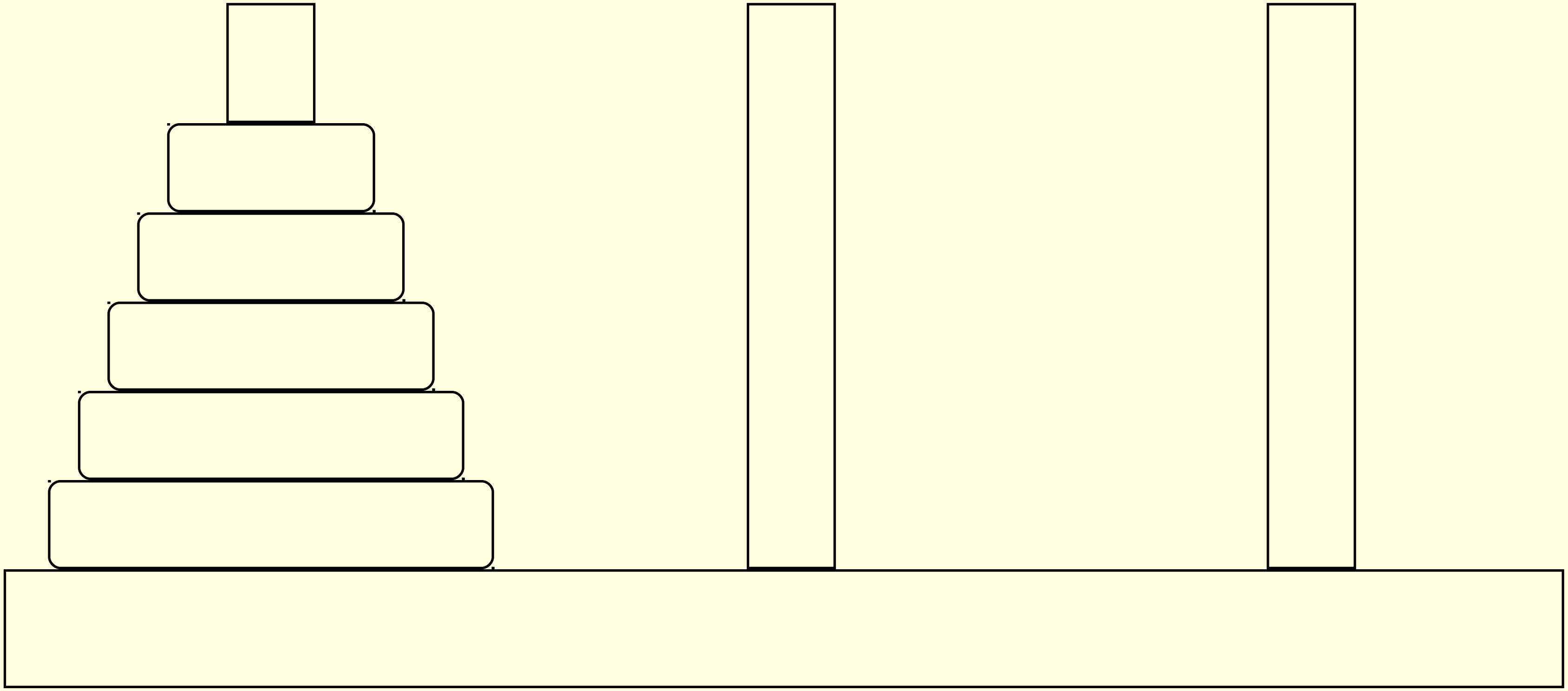
Ejemplo: Factorial Recursivo

Función Recursiva Lineal ⟶ Función Iterativa

```
entero función fact_rec(E entero: n)
inicio
   si n = 0 entonces
      devolver 1
   si_no
      devolver n × fact_rec(n-1)
fin_función
```

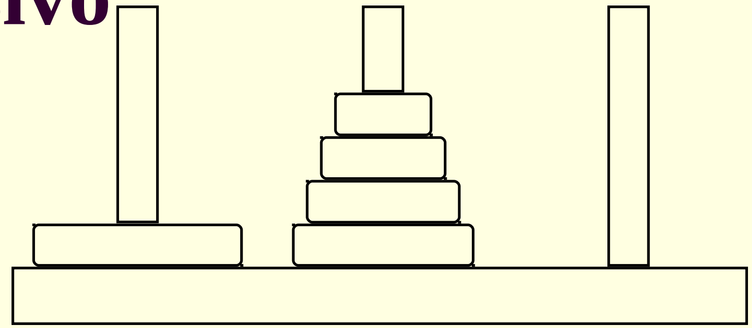
```
entero función fact_iter(E entero: n)
var entero: c, res
inicio
     c ← 0
     mientras n ≠ 0 hacer
        c ← c + 1
        n ← n – 1
     fin_mientras
     res ← 1
     mientras c ≠ 0 hacer
        c ← c – 1
        n ← n + 1
        res ← n × res
     fin_mientras
     devolver res
fin_función
```

6. Ejemplo: Las Torres de Hanoi

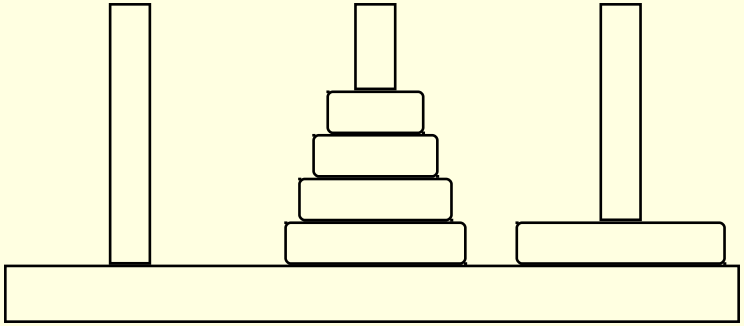


Planteamiento Recursivo

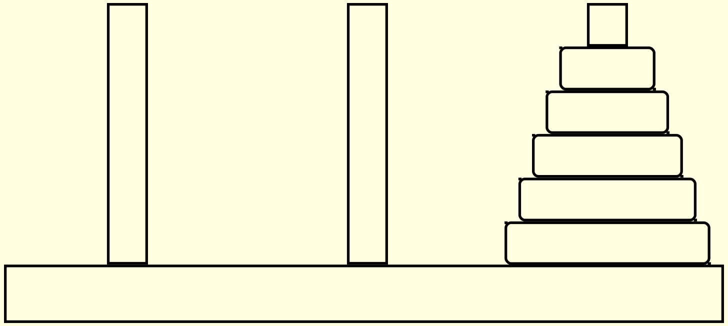
Mover n-1 discos de A a B



Mover disco de A a C



Mover n-1 discos de B a C



Especificación del procedimiento

//Cabecera: T_Hanoi (E entero: *n*,E caracter: *origen*, E caracter: *destino*, E caracter: *auxiliar*)

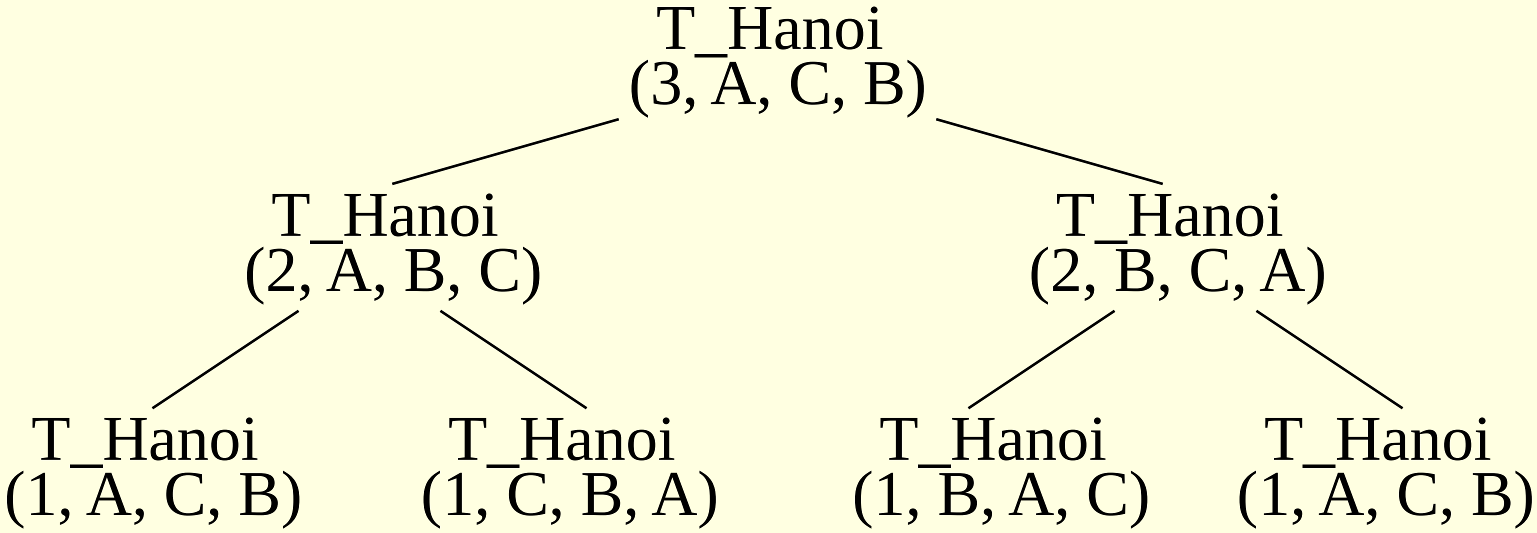
//Precondición: *n* es el número de discos ($n \geq 1$). Los parámetros de origen, destino y auxiliar se refieren respectivamente a los postes donde se encuentran los discos, donde deben ir los discos y el que se usa como poste auxiliar.

//Postcondición: Escribe la secuencia de movimientos para pasar los discos del poste origen al poste destino.

Algoritmo

```
procedimiento T_Hanoi (E entero: n,E caracter: origen,
                       E caracter: destino, E caracter: auxiliar)
inicio
   si  n = 1 entonces
       escribir("Mover disco de ", origen, " a ", destino)
   si_no
       T_Hanoi (n-1, origen, auxiliar, destino)
       escribir("Mover disco de ", origen, " a ", destino)
       T_Hanoi (n-1, auxiliar, destino , origen)
fin_procedimiento
```

Mover disco del poste A al C
Mover disco del poste A al B
Mover disco del poste C al B
Mover disco del poste A al C
Mover disco del poste B al A
Mover disco del poste B al C
Mover disco del poste A al C



Bibliografía

- Balcázar, J.L., *Programación Metódica*, McGraw--Hill, 1993.
- Castro, J., Cucker, F., Messeguer, X., Rubio, A., Solano, L., Valles, B., *Curso de Programación*, McGraw--Hill, 1993.
- Langsam, Y., Augenstein, M.J., Tenembaum, A.M., *Estructuras de Datos con C y C++.* 2"a Edición, Prentice Hall, 1997.
- Peña Marí, R., *Diseño de Programas. Formalismo y Abstracción*. 2ª Edición, Prentice Hall, 1998.
- Tremblay, J.P., Bunt, R.B., *Introducción a la Ciencia de las Computadoras. Enfoque algorítmico*, McGraw--Hill, 1988.
- Warford, J.S., *Computer Science*, DC Heath and Company, 1991.
- Wirth, N., *Algoritmos y Estructura de Datos*, Prentice Hall, 1987.