Tema 4
Introducción a la recursividad

Grado de Ingeniería de Computadores

Introducción a la Programación

Introducción a la recursividad

Objetivos

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

4.1 Conceptos básicos

Formas de definir una función

directa

recurrente

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

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

4.1 Conceptos básicos

$4! = 4 * 3!$ $= 24$

$3! = 3 * 2!$ $= 6$

$2! = 2 * 1!$ $= 2$

$1! = 1 * 0!$ $= 1$

$0! = 1$

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

Recurrencia y recursividad

- Recurrencia:
  - Una función aparece en su propia definición.
  - Un problema se descompone en subproblemas del *mismo* tipo.
- Realización en Pascal:
  - Subprogramas *recursivos.*
  - En el cuerpo del subprograma aparece una llamada a sí mismo.

Factorial recursivo. Ejemplo

```
FUNCTION FacRec(num:integer):integer;
{Pre: num≥0}
{Post: FacRec = num!}
BEGIN
   IF num=0 THEN
      FacRec := 1
   ELSE                                {num >0}
      FacRec := num * FacRec (num - 1)
END; {FacRec}
```

Factorial iterativo. Ejemplo

```
FUNCTION Factorial(num:integer):integer;
{Pre: num≥0}
{Post: Factorial = num!}
VAR
  i, productoAcumulado: integer;
BEGIN
  productoAcumulado := 1; {para 0! y 1!,
  caso base}
   FOR i := 2 TO num DO
     productoAcumulado :=
  productoAcumulado * i;
  Factorial := productoAcumulado
END; {Factorial}
```

Definiciones

- Definición:
  - Un subprograma es *recursivo* si se llama a sí mismo, bien directamente o bien a través de otro subprograma.

- Aplicación:
  - Es una forma natural de implementar relaciones recurrentes.
  - Se trata de una técnica de repetición (alternativa al uso de bucles).

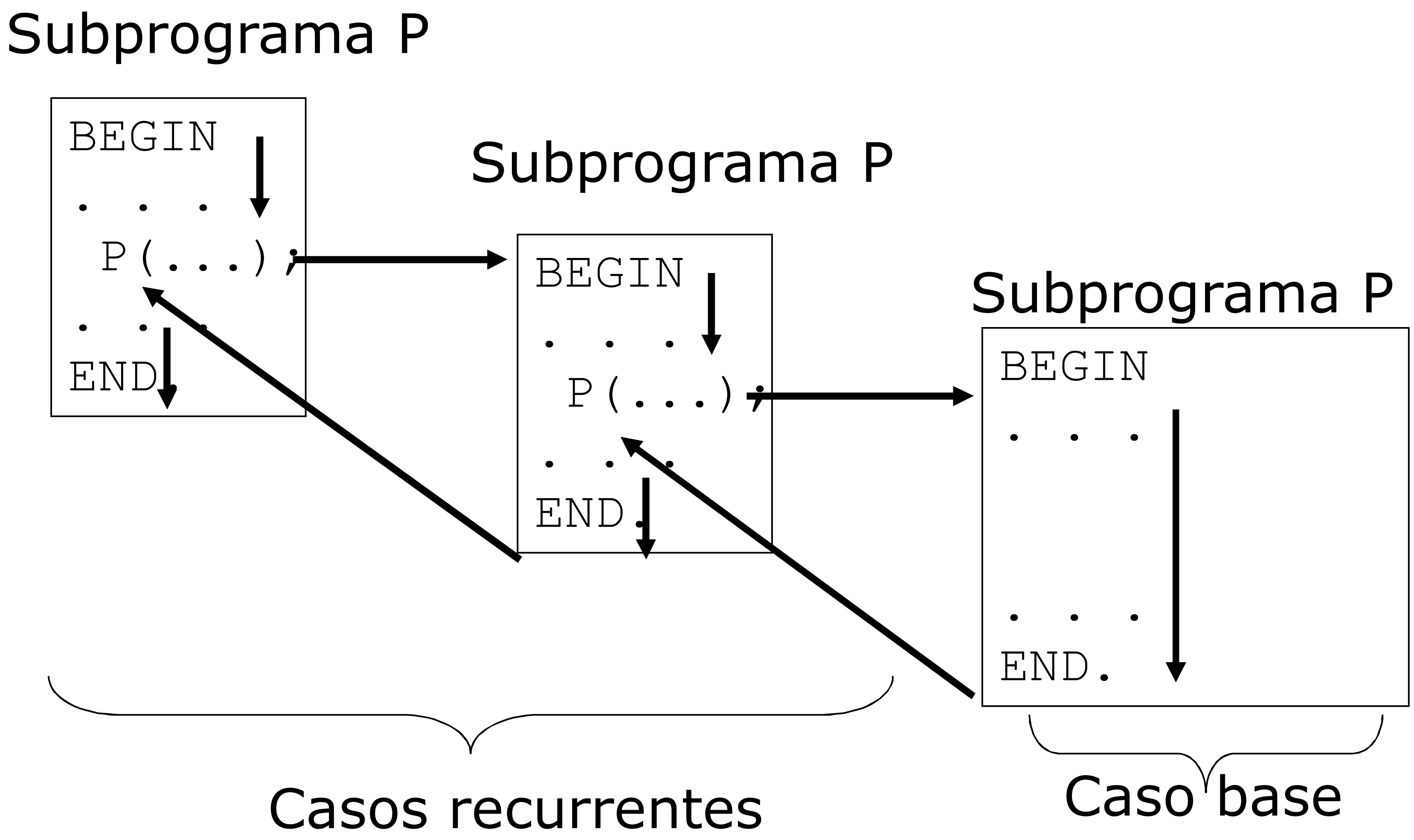
Recursividad en Pascal

- Sintaxis:
  - Sintaxis habitual de las llamadas a subprogramas.
  - No hace falta ninguna ampliación del lenguaje Pascal (excepto para la recursividad mutua, véase 4.4).
- Semántica:
  - Se deduce del mecanismo habitual de llamada a un subprograma.

Recursividad en Pascal

- Proceso de la llamada al subprograma recursivo
  - Se reserva espacio de memoria para almacenar los parámetros y demás objetos locales del subprograma.
  - Se reciben los parámetros y se cede el control de ejecución al subprograma que comienza a ejecutarse.
  - Cuando termina la ejecución, se libera el espacio reservado, los identificadores locales dejan de tener vigencia y se ejecuta la siguiente instrucción a la llamada.
- Este proceso se repite, hasta que se llegue a un caso base (una llamada que devuelve un resultado o no provoca una llamada recursiva).

Proceso de llamada



Proceso de llamada

FacRec (4) = 4 * FacRec (3)

FacRec (3) = 3 * FacRec (2)

FacRec (2) = 2 * FacRec (1)

FacRec (1) = 1 * FacRec (0)

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

FacRec (1) $\Rightarrow$1*1 $\Rightarrow$ 1

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

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

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

Partes de un subprograma recursivo

- **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 pues **no** hay solución trivial.
  - Se reduce a otro caso más simple.
  - Ejemplo: 4! = 4*3!

Recursividad infinita

- **Recursividad infinita**:
  - Se produce una sucesión infinita de llamadas.
  - El control pasa siempre al caso recurrente, nunca se llega al caso base.
- Ejemplo:
  - FacRec (-1) produciría una recursión infinita.

Evitar errores comunes

- Evitar la recursividad infinita:
  - Usar una *estructura de selección* (IF o CASE), 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).
- No olvidar asignar el resultado de la función recursiva a su nombre.
- En los programas recursivos sencillos, **no** suele ser necesario usar **bucles**.

4.2 Recursividad lineal

- Recursividad lineal:
  - Cada llamada recursiva genera como máximo otra nueva llamada recursiva.
- Ejemplos:
  - Cálculo recursivo del factorial: *FacRec.*
  - 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 $+_{SL}$:

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

Suma lenta recursiva. Ejemplo

```
FUNCTION
  SumaLentaRec(a,b:integer):integer;
{Pre: a=A y b=B y a≥0}
{Post: SumaLentaRec = A+B}
BEGIN
   IF a=0 THEN              {Caso base}
     SumaLentaRec := b
   ELSE                     {Caso recurrente}
     SumaLentaRec := SumaLentaRec( a-
  1, b+1)
END; {SumaLenta}
```

Suma lenta iterativa. Ejemplo

```
PROGRAM SumaLenta (input,output);
{Se suman dos enteros positivos, pasando unidades
  de uno a otro}
VAR
      a,b:integer;
BEGIN
  readln(a,b);
  WHILE a <> 0 DO BEGIN
      a:=a-1;
      b:=b+1
  END;  {while}
  writeln(b)
END. {SumaLenta}
```

Recursividad por la cola

- **Recursividad por la cola**:
  - Es un 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:
  - *FacRec* **NO** es recursivo por la cola, porque se multiplica el resultado de la llamada recursiva por *num.*
  - *SumaLentaRec* **SÍ** es recursivo por la cola.

4.3 Recursividad múltiple

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

4.3 Recursividad múltiple

Sucesión de Fibonacci, en matemáticas, es la sucesión de números en la que cada término es igual a la suma de los dos términos precedentes.
Por ejemplo: 1, 1, 2, 3, 5, 8, 13, 21, 34... y así sucesivamente.

- Sucesión de de Fibonacci:

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

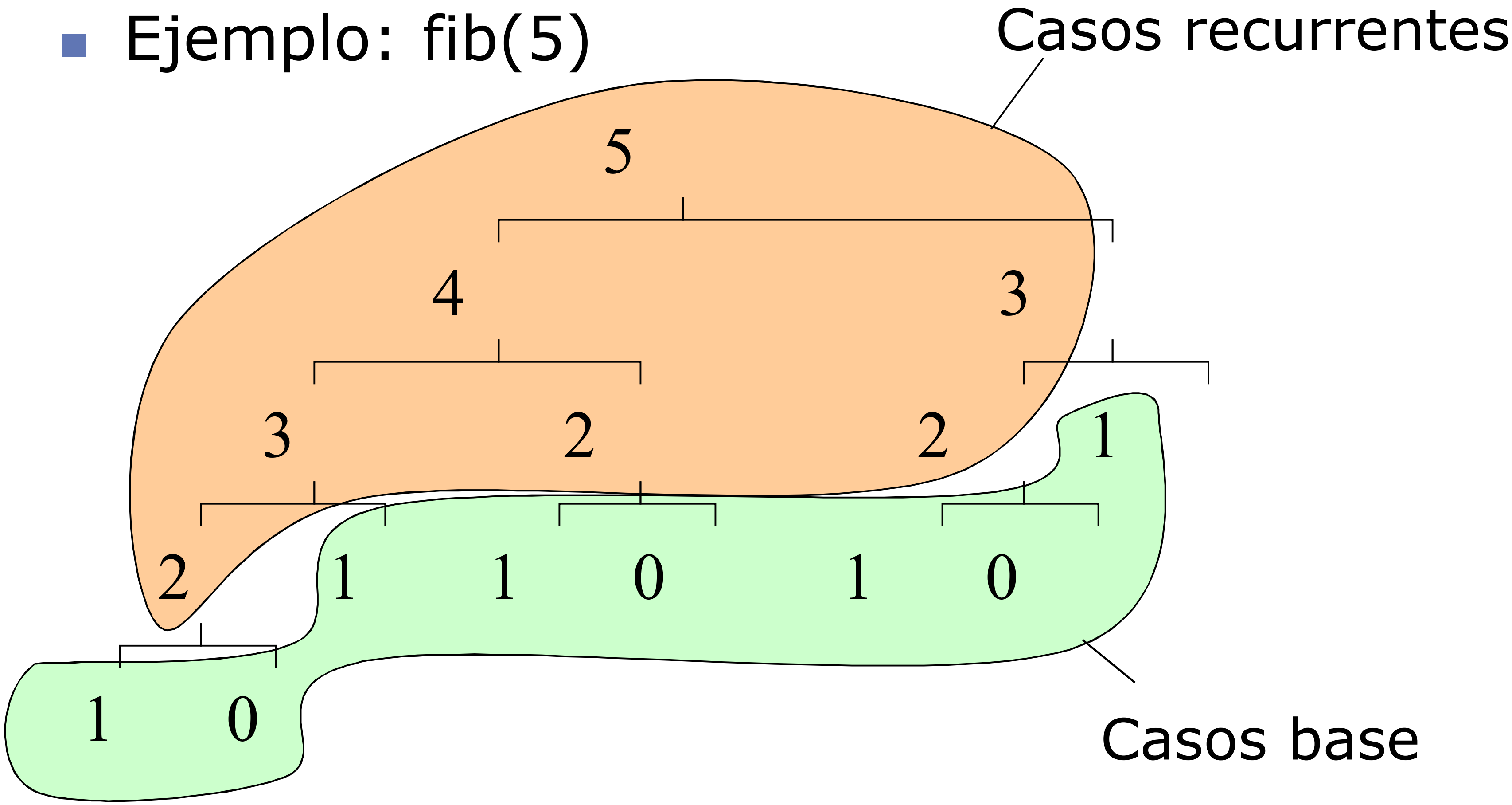
  - $Fib_0 = 1$
  - $Fib_1 = 1$
  - $Fib_2 = Fib_0 + Fib_1$
  - $Fib_3 = Fib_1 + Fib_2$
  - ...
  - $Fib_n = Fib_{n-2} + Fib_{n-1}$

Caso Base

Ley de recurrencia

Número de Fibonacci: Árbol de llamadas

- Ejemplo: fib(5)

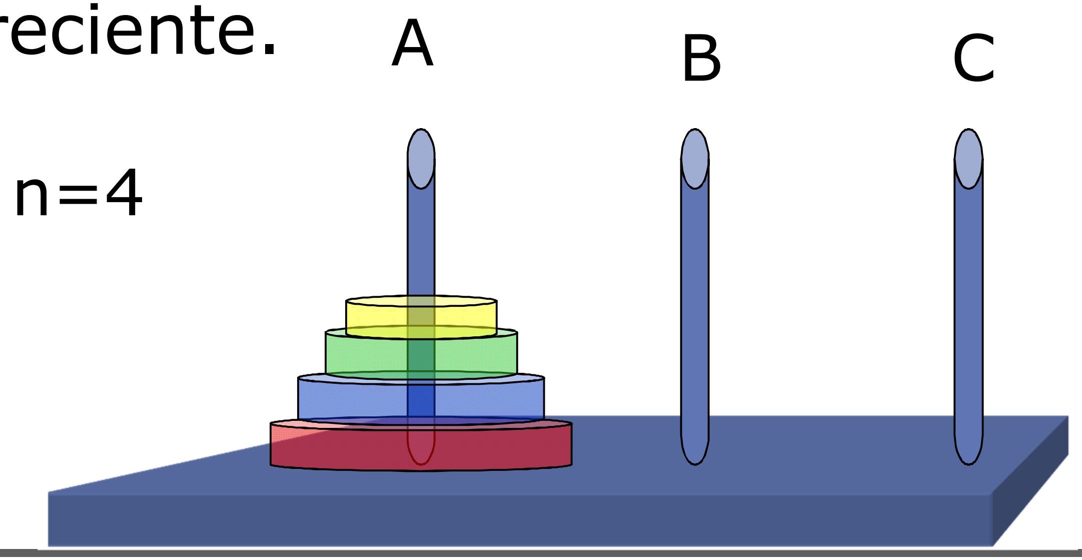


Versión recursiva de Fibonacci. Ejemplo

```
FUNCTION Fib(num:integer):integer;
{Pre: num≥0 }
{Post: Fib = fib_num }
BEGIN
      IF (num=0) OR (num=1) THEN
         Fib := 1
      ELSE
         Fib := Fib(num-1) +
  Fib(num-2)
END; {Fib}
```

Torres de Hanoi

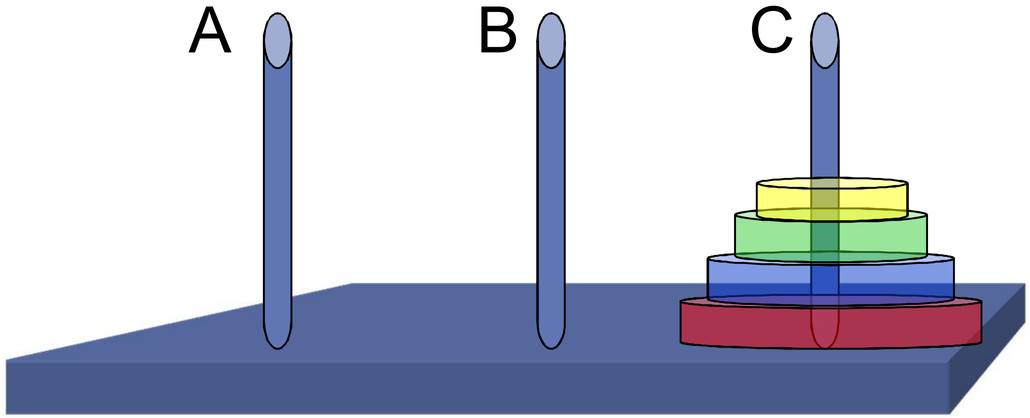
- Juego de sencilla solución recursiva.
- Situación inicial:
  - Tres agujas verticales *A*, *B* y *C*
  - En una de ellas hay *n* discos de tamaño creciente.



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.

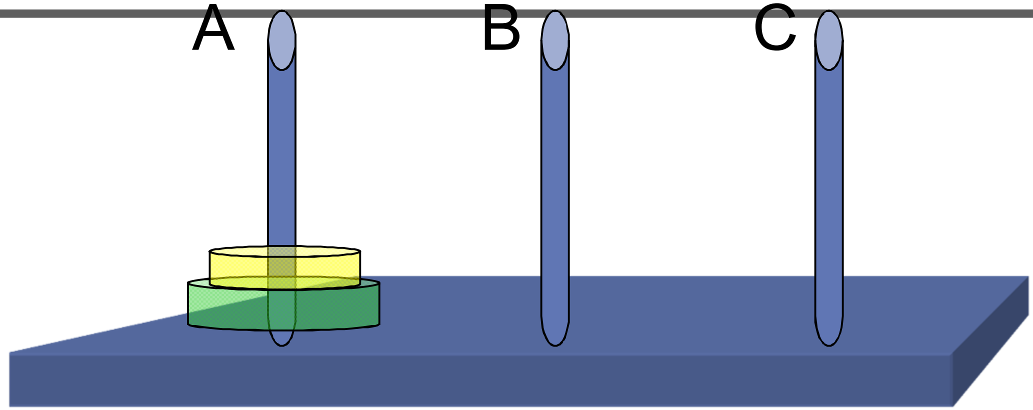
Torres de Hanoi: Algoritmo

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

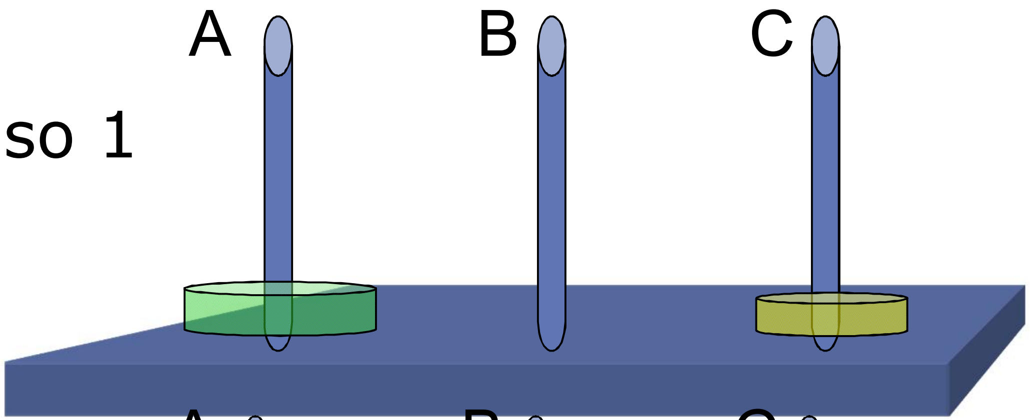
- Caso n=2:
  - Pasar 2 discos de A $\rightarrow$ B
    - Mover disco de A $\rightarrow$ C
    - Mover disco de A $\rightarrow$ B
    - Mover disco de C $\rightarrow$ B

Situación inicial

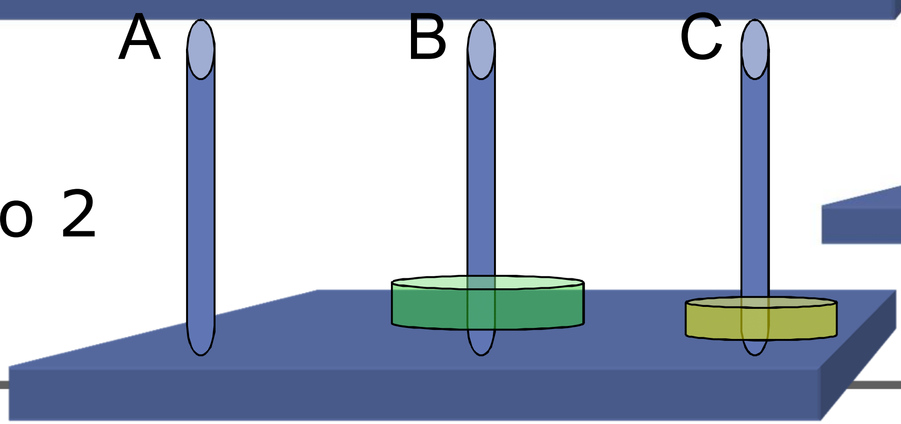
n = 2



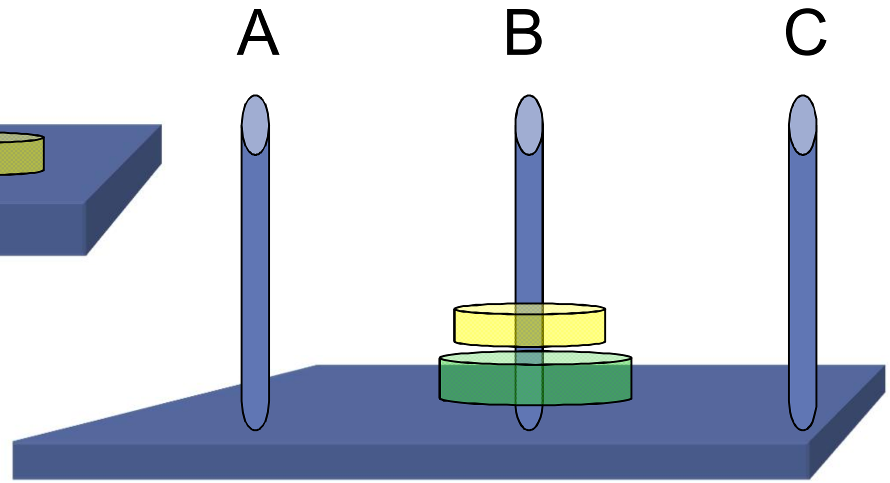
Paso 1



Paso 2



Paso 3



Torres de Hanoi: Algoritmo

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

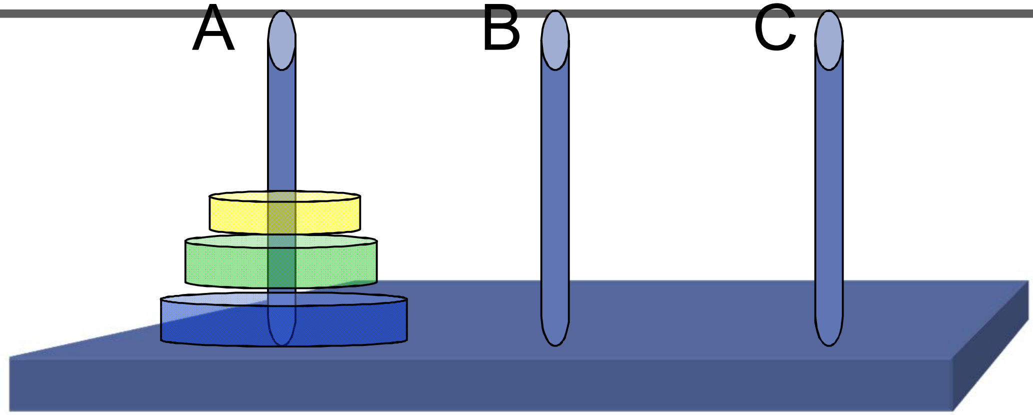
2 de A → C | 1 de A → B | 2 de C → B

mover A → B
mover A → C
mover B → C

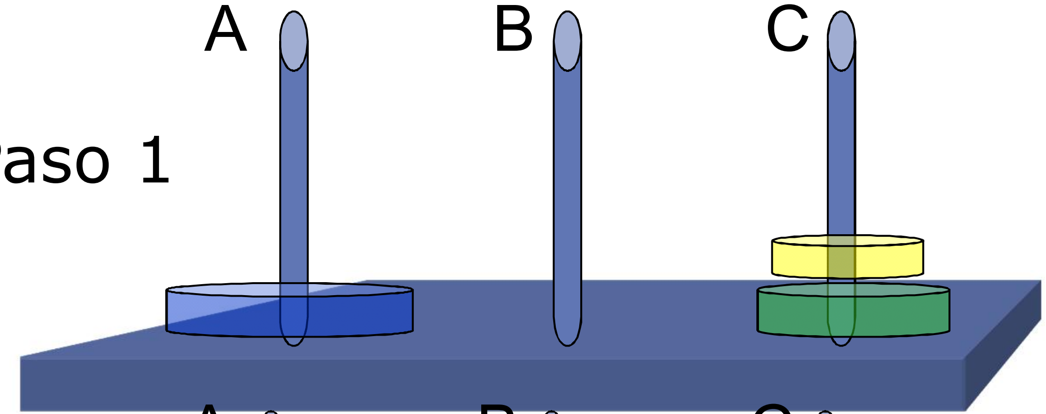
mover C → A
mover C → B
mover A → B

Situación inicial

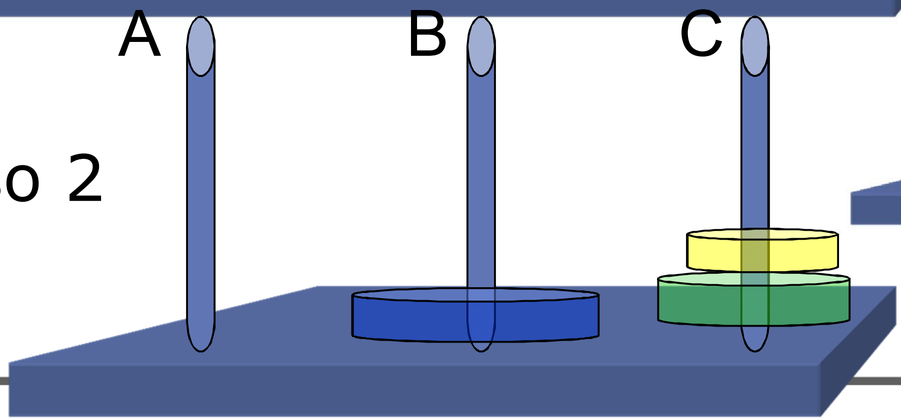
$n = 3$



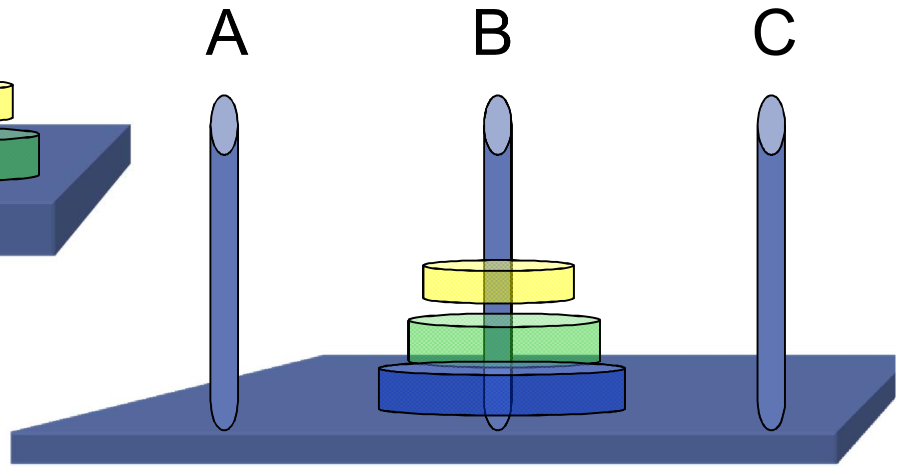
Paso 1



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. Ejemplo

```
PROCEDURE MoverDiscos (n : integer; origen,
  destino, auxiliar : char);
{ Pre: n > 0
  Post: output = [movimientos para pasar n discos de la
  aguja origen a la aguja destino]}
BEGIN
 IF n = 0 THEN     {Caso base}
    writeln
 ELSE BEGIN        {Caso recurrente}
    MoverDiscos(n-1, origen, auxiliar, destino);
    write('Pasar disco', n, 'de', origen, 'a',
  destino);
    MoverDiscos(n-1, auxiliar, destino, origen)
 END; {fin ELSE}
END; {fin MoverDiscos}
```

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 recursividad

Repetición de bloques de instrucciones

Técnica	Iteración	Recursividad
Realización	Bucles (WHILE, REPEAT, FOR)	Subprogramas que se llaman a sí mismos

Iteración y recursividad

- Equivalencia de iteración y recursividad:
  - Cualquier cómputo recursivo puede expresarse de forma iterativa y viceversa.

- Ejemplos:
  - Factorial: *factorial* y *facRec.*
  - Suma lenta: *sumaLenta* y *sumaLentaRec.*
  - Número de Fibonacci: *fib* y *fibIter.*

Versión iterativa de Fibonacci

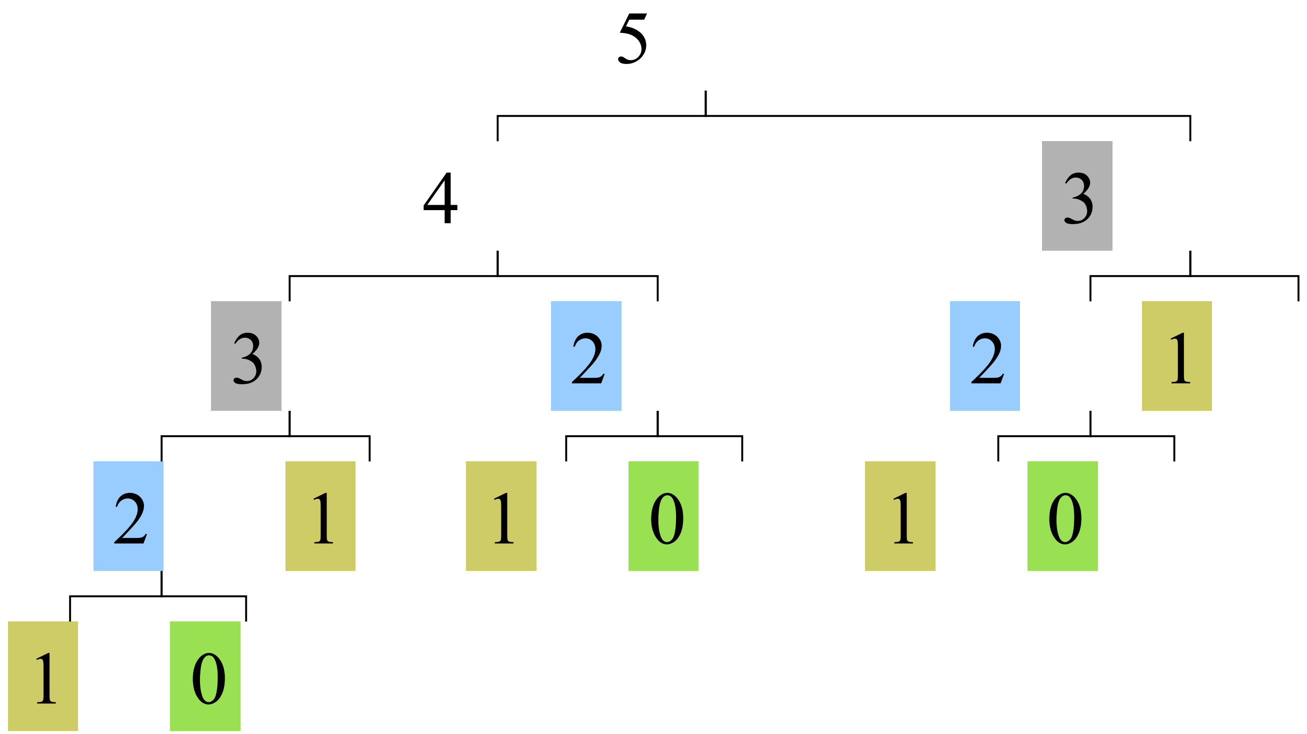
```
FUNCTION fibIter(n:integer):integer;
{Pre: n≥0
 Post: fibIter = fibn}
VAR a,a1,a2,i:integer;
BEGIN
     a:=1;
     a1:=1;
     FOR i:= 2 TO n DO BEGIN
          a2:=a1;
          a1:=a;
          a := a1+a2
     END;
     fibIter := a
END; {fibIter}
```

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. Recursividad mutua

- Recursividad simple (directa)
  - Un subprograma llama a sí mismo.
- Recursividad 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*.

4.4. Recursividad mutua

- Realización en Pascal: por medio de la subprogramación
- Problema:
  - Un identificador es conocido solo después de su declaración.
  - Al menos, un subprograma ha de llamar a otro antes de que éste último sea declarado.
- Solución:
  - **Predeclaración** del segundo subprograma con la palabra reservada **FORWARD**.

Definición en Pascal

```
PROCEDURE B (parámetros); FORWARD {Predeclaración
   de B}

PROCEDURE A (parámetros); {Declaración de A}
BEGIN {A}
   B();
   ...
END; {A}

PROCEDURE B (parámetros); {Declaración de B}
BEGIN {B}
   A():
   ...
END; {B}
...
```

EsPar / EsImpar. Ejemplo

- Escribir funciones para determinar la paridad de un número positivo utilizando recursividad mutua.
- EsPar (n):
  - TRUE si $n=0$
  - EsImpar ($n-1$) si n>0
- EsImpar (n):
  - FALSE si $n=0$
  - EsPar ($n-1$) si n>0

EsPar / EsImpar (I)

```
PROGRAM EsParEsImpar (input, output);
VAR n: integer;   {numero}

FUNCTION esImpar (n : integer) : boolean;
  FORWARD; {Predeclaración del identificador "esImpar"}

FUNCTION esPar(n : integer): boolean;
{Pre: n>=0}
{Post: esPar=true si n es par y esPar=false en caso contrario}
BEGIN
     IF n = 0 THEN
        esPar := true            {Caso base}
     ELSE
        esPar := esImpar (n-1) {Caso recurrente}
END; {esPar}
```

EsPar / EsImpar (II)

```
FUNCTION esImpar (n : integer): boolean;
{Pre: n >= 0}
{Post: esImpar=true si n es impar y esImpar=false en
  caso contrario}
BEGIN
   IF n = 0 THEN
     esImpar := false      {Caso base}
   ELSE
     esImpar := esPar(n-1){Caso recurrente}
END; {EsImpar}
```

EsPar / EsImpar (y III)

```
BEGIN {programa principal}
   write('Introduzca un número entero: ');
   readln(n);
   IF esPar(n) THEN
  {Aquí tambien es posible una condición de
  verificación del tipo not esImpar(n) }
     writeln('El número ',n,' es par')
   ELSE
     writeln('El número ',n,' es impar')
END.  {programa principal}
```

Recomendaciones técnicas

Evitar la recursión mutua:

- El grafo de llamadas puede resultar difícil de entender.
- Es preferible una estructura jerárquica de llamadas.

4.5. Aspectos formales

Aspectos formales de subprogramas recursivos

- Corrección (depuración y verificación formal).
- Complejidad.

Idea:

- Aplicar las técnicas generales para subprogramas al caso especial de subprogramas recursivos.

- Introducir nuevas técnicas sólo en los (pocos) aspectos en los que sea imprescindible.

4.5.1 Corrección de subprogramas recursivos

- Depuración:
  - Depuración de la **llamada**
    - Depurar la llamada al subprograma recursivo como una sola instrucción/ expresión
    - TurboPascal: Menu “Run” / Opción “Step Over”
  - Depuración del **cuerpo**
    - Depurar las instrucciones del cuerpo del subprograma recursivo una por una.
    - TurboPascal: Menu “Run” / Opción “Trace Into”

ascension.lovillo@urjc.es