Recursividad

¿Qué es la recursividad?

- *La recursividad es un concepto fundamental en matemáticas y en computación.*
- *Es una alternativa diferente para implementar estructuras de repetición (ciclos). Los módulos se hacen llamadas recursivas.*
- *Se puede usar en toda situación en la cual la solución pueda ser expresada como una secuencia de movimientos, pasos o transformaciones gobernadas por un conjunto de reglas no ambiguas.*

Función recursiva

Las funciones recursivas se componen de:

- ***Caso base***: una solución simple para un caso particular (puede haber más de un caso base). La secuenciación, iteración condicional y selección son estructuras válidas de control que pueden ser consideradas como enunciados.

NOTA: ***Regla recursiva*** Las estructuras de control que se pueden formar combinando de manera válida la secuenciación, iteración condicional y selección también son válidos.

Función recursiva

- ***Caso recursivo***: una solución que involucra volver a utilizar la función original, con parámetros que se acercan más al caso base. Los pasos que sigue el caso recursivo son los siguientes:
  1. La función se llama a sí misma.
  2. El problema se resuelve, resolviendo el mismo problema pero de tamaño menor.
  3. La manera en la cual el tamaño del problema disminuye asegura que el caso base eventualmente se alcanzará.

Ejemplo: factorial

Escribe un programa que calcule el factorial (!) de un entero no negativo. Ejemplos de factoriales:

- 0! = 1
- 1! = 1
- 2! = 2 → 2! = 2 * 1!
- 3! = 6 → 3! = 3 * 2!
- 4! = 24 → 4! = 4 * 3!
- 5! = 120 → 5! = 5 * 4!

Ejemplo: factorial (iterativo)

int factorial (int n)

<u>comienza</u>

fact ← 1

<u>para</u> i ← 1 <u>hasta</u> n

fact ← i * fact

<u>regresa</u> fact

<u>termina</u>

```
int factorial (int n) {
  int fact = 1;
  for  (int i = 1; i <= n; i++)
 fact *= i;
  return fact;
 }
```

Ejemplo: factorial (recursivo)

int factorial (int n)

<u>comienza</u>

    <u>si</u> n = 0 <u>entonces</u>

        <u>regresa</u> 1

    <u>otro</u>

        <u>regresa</u> factorial (n-1)*n

<u>termina</u>

```
int factorial (int n) {
  if n == 0 return 1;
  else
  return factorial (n-1) * n;
}
```

Ejemplo:

- A continuación se puede ver la secuencia de factoriales.
  - **0! = 1**
  - **1! = 1** = 1 * 1 = 1 * 0!
  - **2! = 2** = 2 * 1 = 2 * 1!
  - **3! = 6** = 3 * 2 = 3 * 2!
  - **4! = 24** = 4 * 6 = 4 * 3!
  - **5! = 120** = 5 * 24 = 5 * 4!
  - **...**
  - **N! =** = N * (N – 1)!

Solución

Aquí podemos ver la secuencia que toma el factorial

$$
N! = \begin{cases} 1 & \text{si } N = 0 \ \ \text{(base)} \\ N * (N-1)! & \text{si } N > 0 \ \ \text{(recursión)} \end{cases}
$$

Un razonamiento recursivo tiene dos partes: la base y la regla recursiva de construcción. La base no es recursiva y es el punto tanto de partida como de terminación de la definición.

Solución Recursiva

Dado un entero no negativo x, regresar el factorial de x fact:

Entrada n entero no negativo,

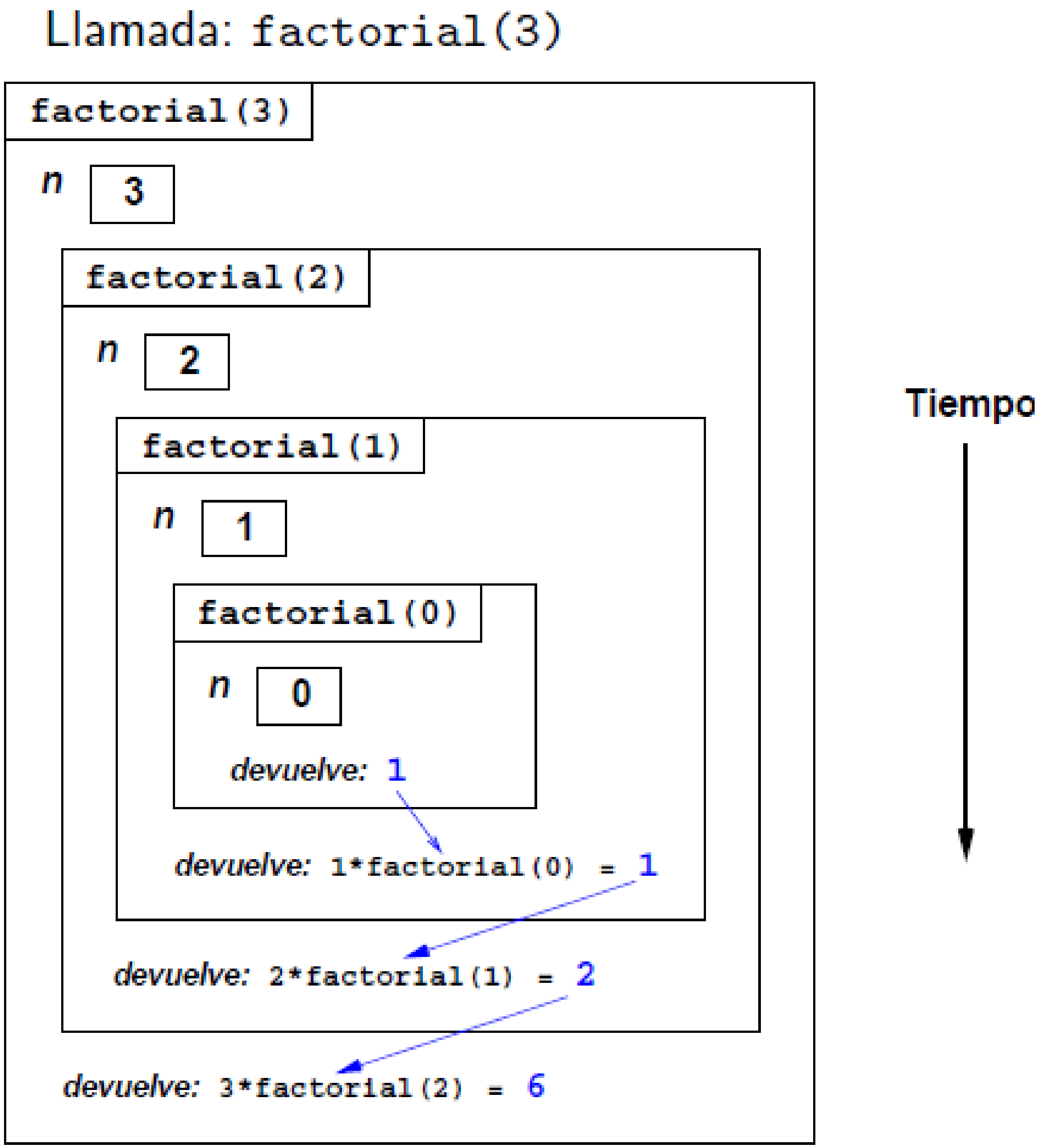
Salida:entero.

```
int fact (int n)
{
  if (n == 0)
    return 1;
  else
    return  fact(n – 1) * n ;
}
```

Es importante determinar un caso base, es decir un punto en el cual existe una condición por la cual no se requiera volver a llamar a la misma función.

Traza de algoritmos recursivos:

Se representan en cascada cada una de las llamadas al módulo recursivo, así como sus respectivas zonas de memoria y los valores que devuelven.



¿Por qué escribir programas recursivos?

- Son más cercanos a la descripción matemática.
- Generalmente más fáciles de analizar
- Se adaptan mejor a las estructuras de datos recursivas.
- Los algoritmos recursivos ofrecen soluciones estructuradas, modulares y elegantemente simples.

¿Cómo escribir una función en forma recursiva?

<*tipo_de_regreso*><nom_fnc> (<param>){

[declaración de variables]

[condición de salida]

[instrucciones]

[llamada a <nom_fnc> (<param>)]

return <resultado>

}

Cálculo de la potencia

$$x^n = \begin{cases} 1 & \text{si } n = 0 \\ x \cdot x^{n-1} & \text{si } n > 0 \end{cases}$$

```
int potencia(int base, int expo){
  if (expo==0)
    return 1;
  else
    return base * potencia(base,expo-1);
}
```

La suma de forma recursiva

$$suma(a,b) = \begin{cases} a & \text{si } b = 0 \\ 1 + suma(a, b-1) & \text{si } b > 0 \end{cases}$$

```
int suma(int a, int b){
  if (b==0)
    return a;
  else
    return 1+suma(a,b-1);
}
```

3. El producto de forma recursiva

$$producto(a,b) = \begin{cases} 0 & \text{si } b = 0 \\ a + producto(a, b-1) & \text{si } b > 0 \end{cases}$$

```
int producto(int a, int b){
  if (b==0)
    return 0;
  else
    return a+producto(a,b-1);
}
```

Ejercicio

Considere la siguiente ecuación recurrente:

$$a_n = a_{n-1} + 2^n$$
$$a_0 = 1$$

Escribe el algoritmo de la solución.

¿Cuándo usar recursividad?

- Para simplificar el código.
- Cuando la estructura de datos es recursiva ejemplo: árboles.

¿Cuándo no usar recursividad?

- Cuando los métodos usen arreglos largos.
- Cuando el método cambia de manera impredecible de campos.
- Cuando las iteraciones sean la mejor opción.

Algunas Definiciones.

- Cuando una función incluye una llamada a sí misma se conoce como **recursión directa.**

Algunas Definiciones.

- Cuando una función llama a otra función y esta causa que la función original sea invocada, se conoce como **recursión indirecta.**

NOTA: Cuando una función recursiva se llama recursivamente a si misma varias veces, para cada llamada se crean *copias independientes* de las variables declaradas en el procedimiento.

Recursión vs. iteración

Repetición

Iteración: ciclo explícito

Recursión: repetidas invocaciones al método

Terminación

Iteración: el ciclo termina o la condición del ciclo falla

Recursión: se reconoce el caso base

En ambos casos podemos tener ciclos infinitos:

Considera que resulta más positivo para cada problema, la elección entre eficiencia (iteración) o una buena ingeniería de software, la recursión resulta normalmente más natural.

Otros Ejemplos de recursividad:

- Inversión de una cadena

```
char invierte (char cadena, int limIzq, int
   limDer)
   si limDer = limIzq entonces   regresa cadena
   sino   regresa invierte (cadena, limDer,
   limIzq+1) + cadena [limIzq]
  fin
```

Otros Ejemplo de recursividad:

Palíndromos

Un palíndromo es una cadena que se lee (se escribe, en este caso) igual de izquierda a derecha que de derecha a izquierda. Escribir una función que determine cuando una cadena es o no un palíndromo.

Solución

```
bool palindrome (Cad c, int limIzq, int limDer)
     si  limIzq > limDer   entonces
      regresa verdadero
     sino
      si  c [limIzq] = c [limDer] entonces
          regresa palindrome (c, limIzq+1, limDer-1)
      sino regresa falso
     fin
```

Ejemplo: Serie de Fibonacci

Valores: 0, 1, 1, 2, 3, 5, 8...

Cada término de la serie suma los 2 anteriores. Fórmula recursiva

fib(n) = fib (n - 1) + fib (n - 2)

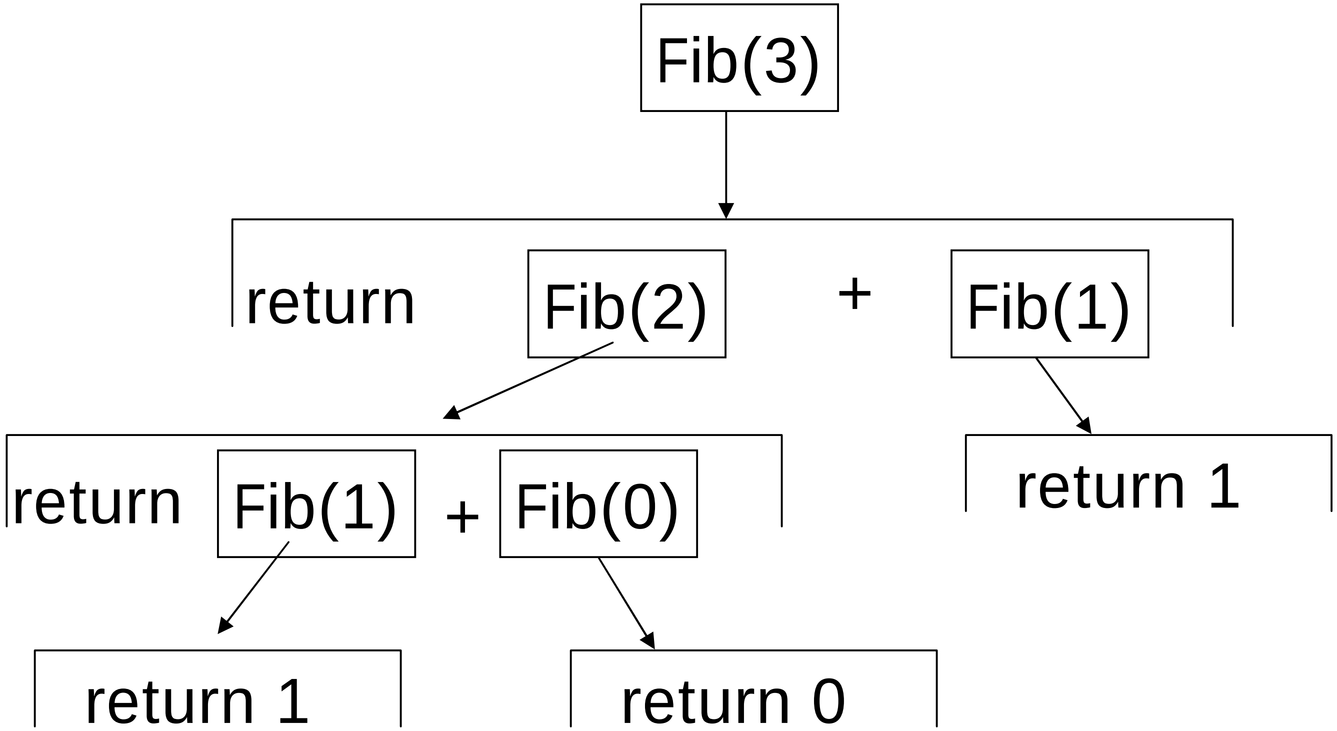
Caso base: Fib (0)=0; Fib (1)=1

Caso recursivo: Fib (i) = Fib (i -1) + Fib(i -2)

```
int fib(int n){
   if (n <= 1) return n;        //condición base
    else
 return fib(n-1)+fib(n-2);    //condición recursiva
}
```

Ejemplo: **Serie de Fibonacci**

Traza del cálculo recursivo



Trampas sutiles: Código ineficiente.

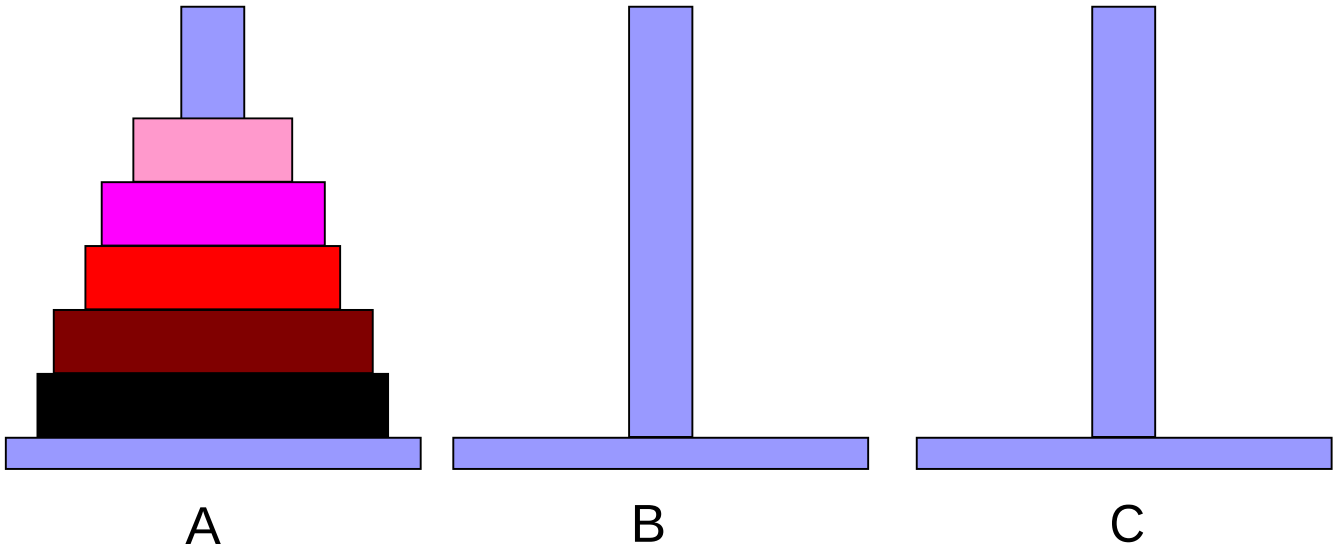
```
int fib (int n)
{
   if  (n < 2)
       return 1;
   else
       return fib (n-2) +
                 fib ( n-1);
}
```

fib (100) toma mucho tiempo en dar el resultado

```
int fib  (int n)
{
   int f1 = 1, f2 = 1, nuevo;
   while (n > 2)
   {
      nuevo = f1 + f2;
      f1 = f2;   f2 = nuevo;
      n--;
   }
   return f2;
}
```

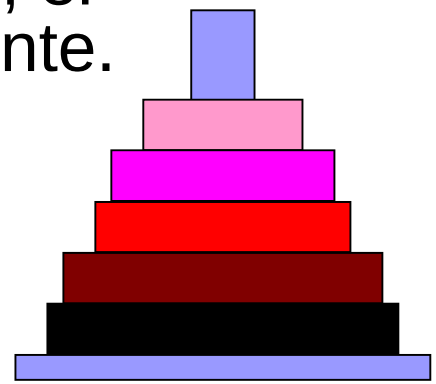
fib (100) toma tan sólo unos microsegundos en dar el resultado

Un ejemplo clásico de recursividad: Torres de Hanoi



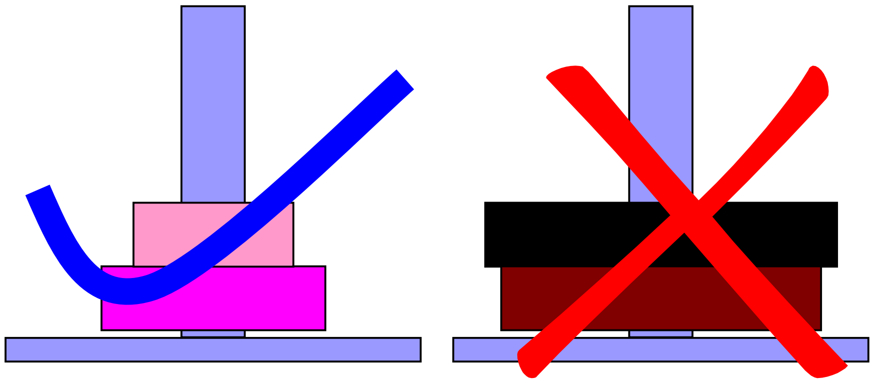
Torres de Hanoi

- Tenemos tres astas A, B y C, y un conjunto de cinco aros, todos de distintos tamaños.
- El enigma comienza con todos los aros colocados en el asta A de tal forma que ninguno de ellos debe estar sobre uno más pequeño a él; es decir, están apilados, uno sobre el otro, con el más grande hasta abajo, encima de él, el siguiente en tamaño y así sucesivamente.



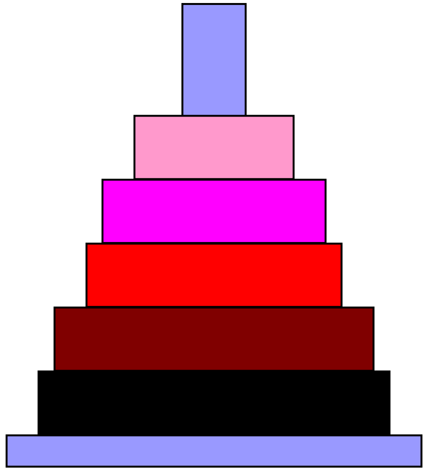
Torres de Hanoi

- El propósito del enigma es lograr apilar los cincos aros, en el mismo orden, pero en el hasta C.
- Una restricción es que durante el proceso, puedes colocar los aros en cualquier asta, pero debe apegarse a las siguientes reglas:
  - Solo puede mover el aro superior de cualquiera de las astas.
  - Un aro más grande nunca puede estar encima de uno más pequeño.

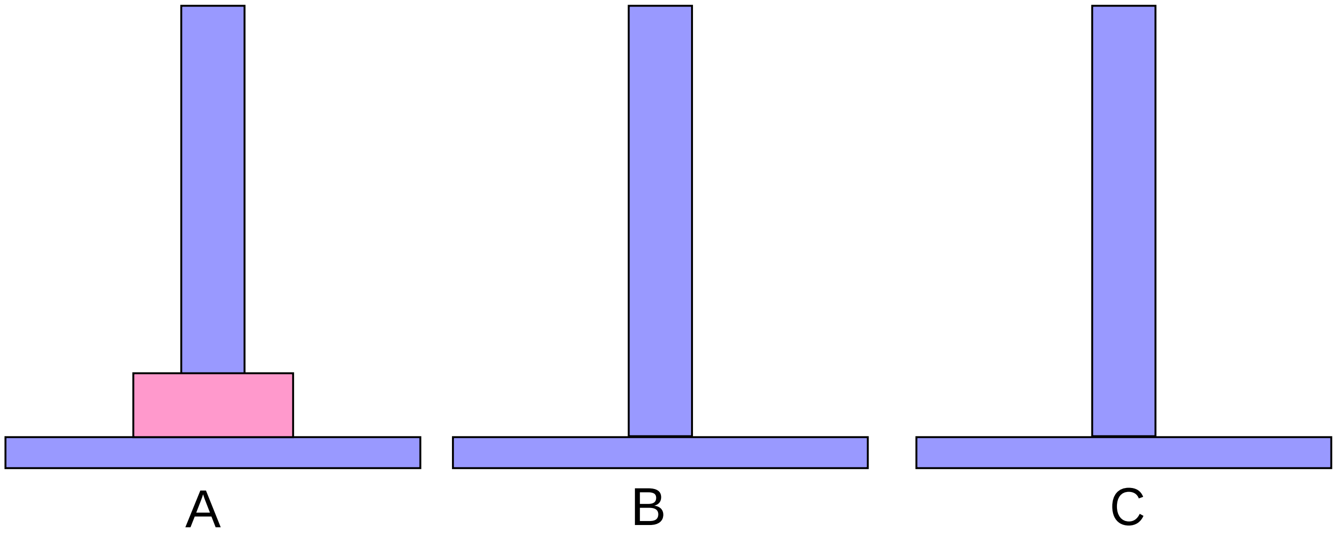


¿Cómo resolvemos el problema?

- Para encontrar cómo se resolvería este problema, debemos ver cómo se resolvería cada caso.

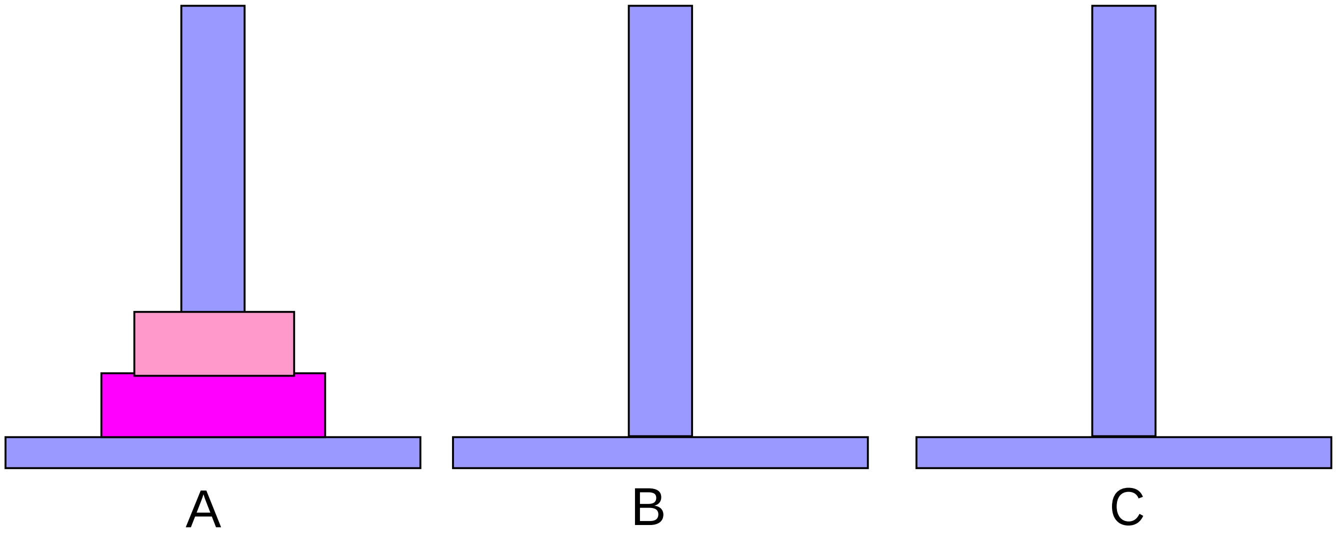


¿Cómo se resolvería el caso en que hubiese un aro?



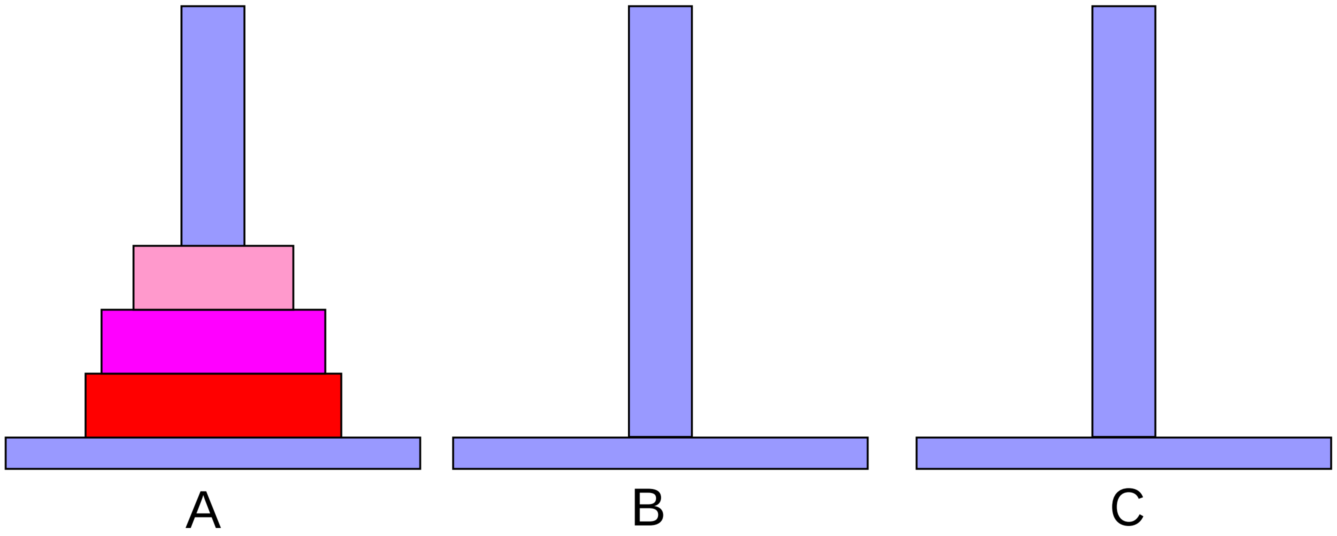
Pasando directamente el aro de A a C.

¿Cómo se resolvería el caso en que hubiera 2 aros?



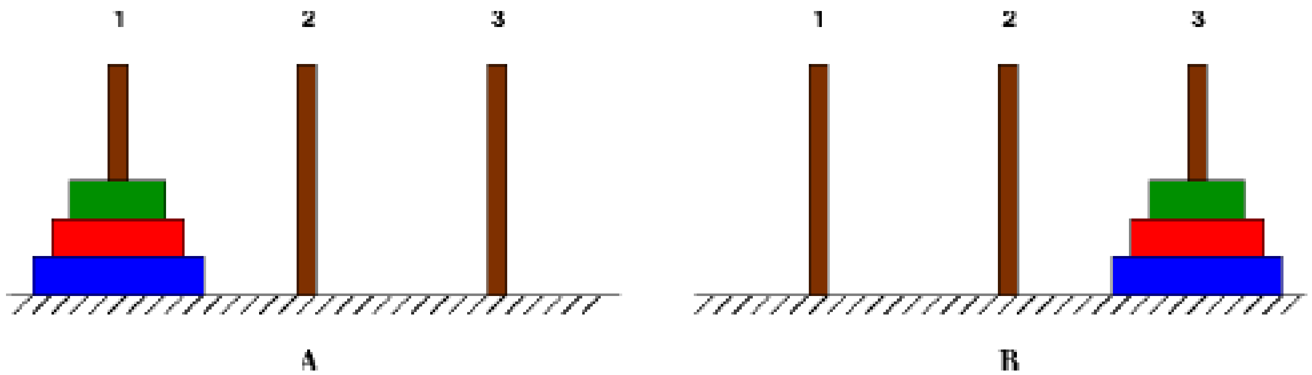
Colocando el más pequeño en el asta B, pasando el grande a el asta C y después moviendo el que está en B a C.

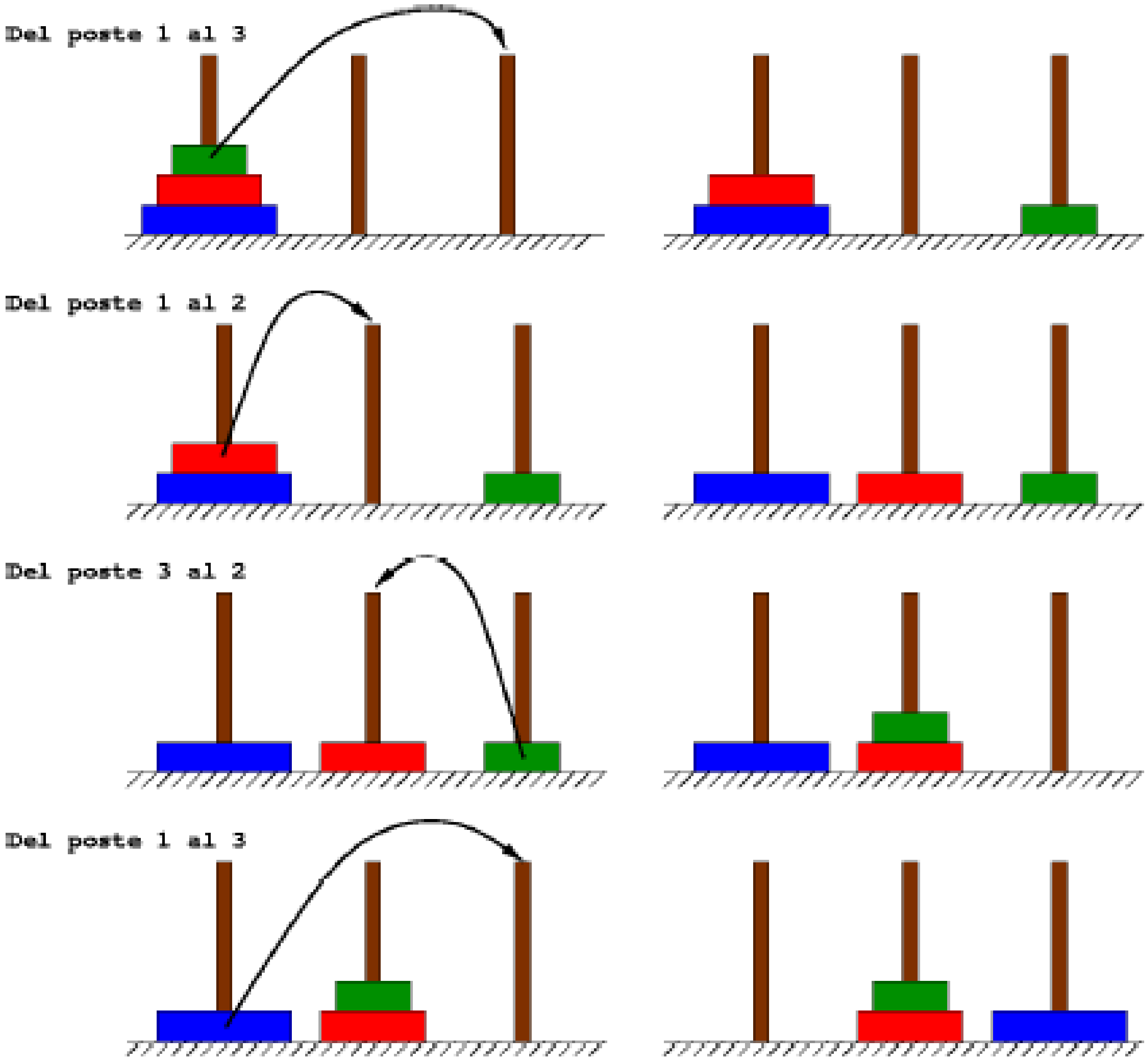
¿Cómo se resolvería el caso de 3 aros?

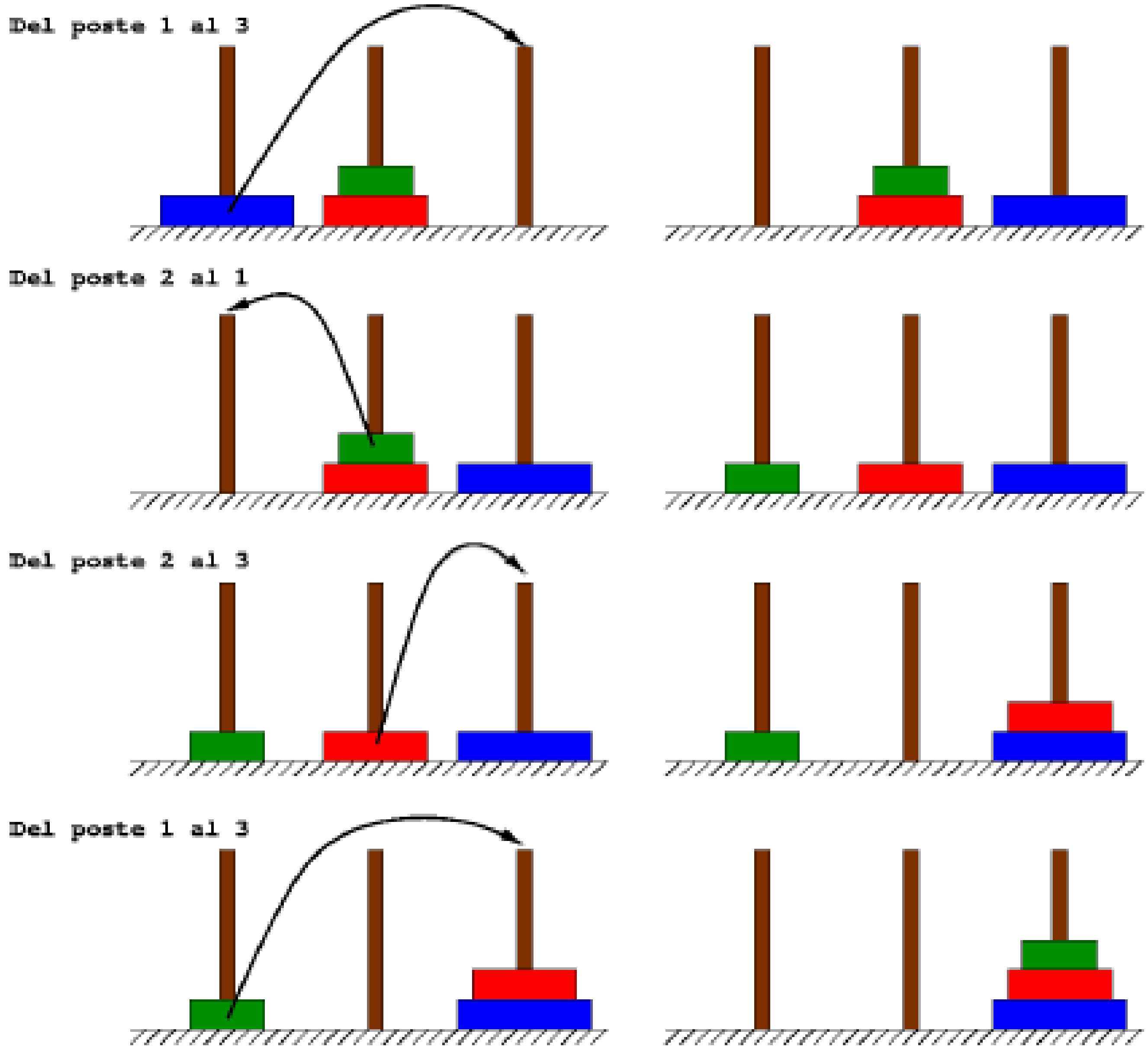


Resolviendo el problema de las Torres de Hanoi

- Entonces, por lo que hemos podido ver, el programa podría definirse de la siguiente manera:
  - Si es un solo disco, lo movemos de A a C.
  - En otro caso, suponiendo que n es la cantidad de aros que hay que mover
    - Movemos los n-1 aros superiores - es decir, sin contar el más grande- de A a B (utilizando a C como auxiliar).
    - Movemos el último aro (el más grande) de A a C.
    - Movemos los aros que quedaron en B a C (utilizando a A como auxiliar).


1
2
3
1
2
3
A
B


Del poste 1 al 3
Del poste 1 al 2
Del poste 3 al 2
Del poste 1 al 3


Del poste 1 al 3
Del poste 2 al 1
Del poste 2 al 3
Del poste 1 al 3

Numero de discos: 3

- **Del poste 1 al 3**
- **Del poste 1 al 2**
- **Del poste 3 al 2**
- **Del poste 1 al 3**
- **Del poste 2 al 1**
- **Del poste 2 al 3**
- **Del poste 1 al 3**

```c
#include <stdio.h>
#include <stdlib.h>

void hanoi (int n, int inic, int tmp, int final);

main ()
{
int n; // Numero de discos a mover
printf( "Numero de discos: ");
scanf("%d",&n);
hanoi (n, 1, 2, 3); // mover "n" discos del 1 al
3 usando el 2 como temporal.
return 0;
}
```

```c
void hanoi (int n, int inic, int tmp, int final)
{
if (n > 0) {
// Mover n-1 discos de "inic" a "tmp".
// El temporal es "final".
hanoi (n-1, inic, final, tmp);
// Mover el que queda en "inic"a "final"
printf("Del poste %d al %d\n«, inic,final);
// Mover n-1 discos de "tmp" a "final".
// El temporal es "inic".
hanoi (n-1, tmp, inic, final);
}
}
```

Fin