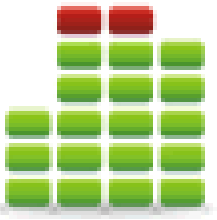
Instituto Politécnico Nacional

Escuela Superior de Cómputo



Estructuras de Datos

Tema 06: Recursividad

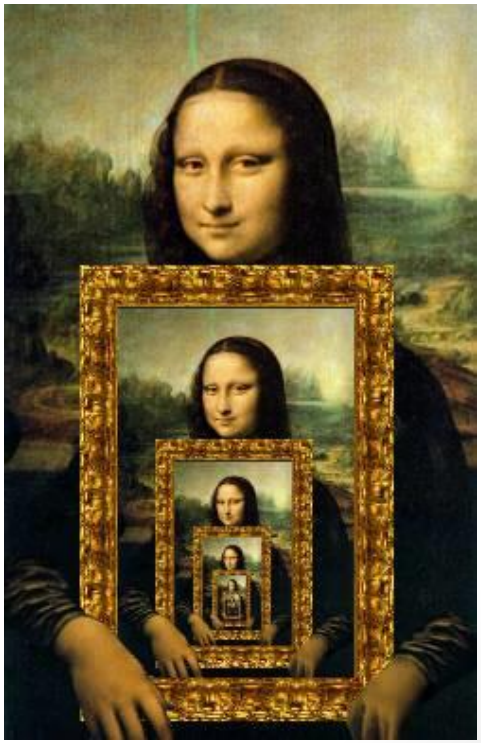
M. en C. Edgardo Adrián Franco Martínez
http://www.eafranco.com
edfrancom@ipn.mx
@edfrancom edgardoadrianfrancom

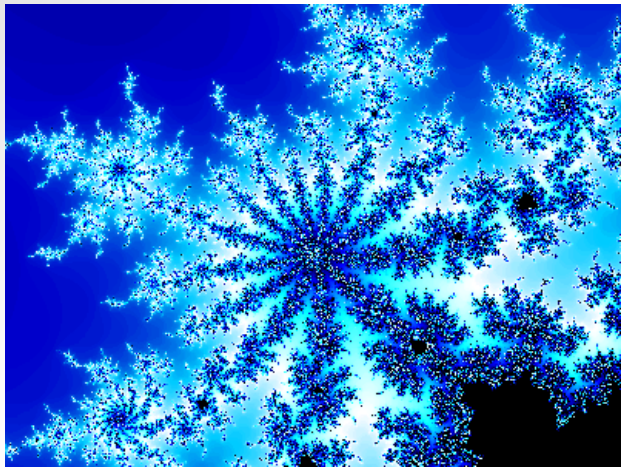
Contenido

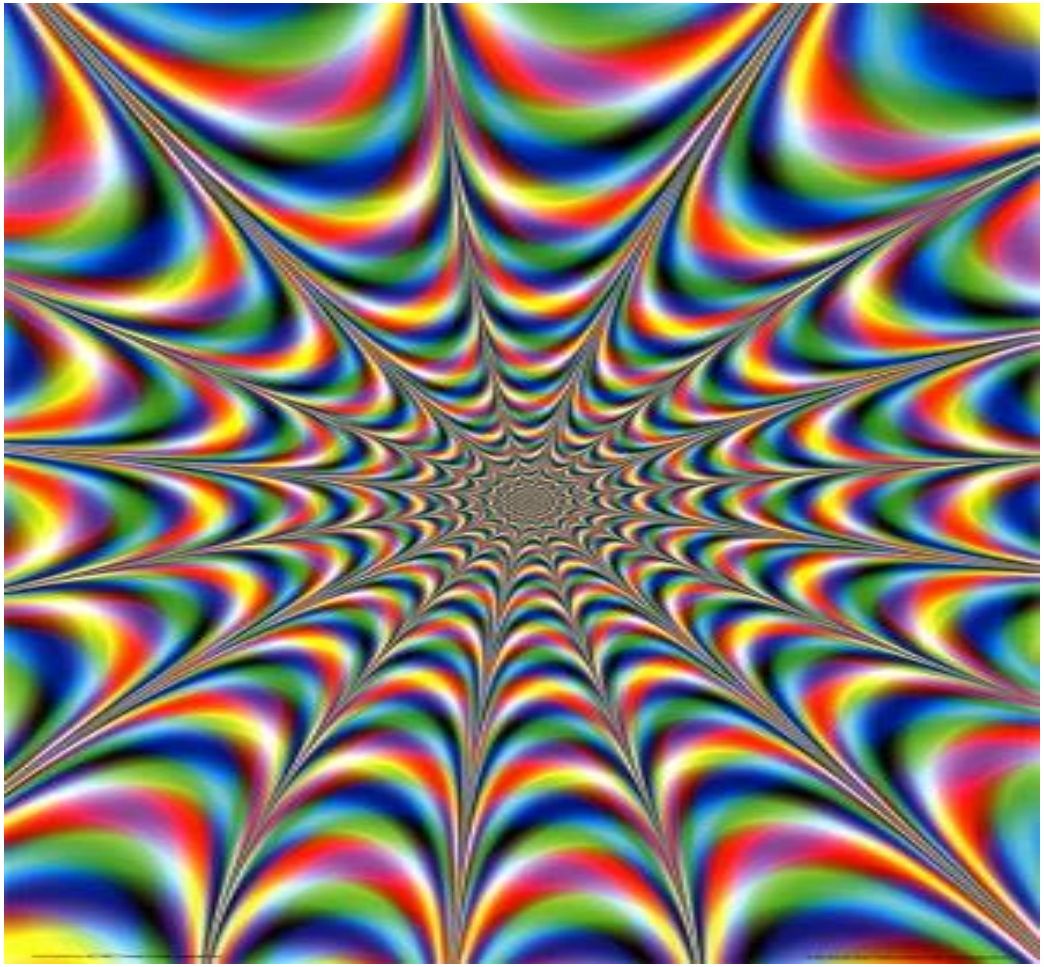
- Recursión
- Recursividad
- Programación recursiva
- Tipos de recursión
- Ejemplo: Factorial
- ¿Por qué escribir programas recursivos?
- ¿Cómo escribir una función en forma recursiva?
- ¿Qué pasa si se hace una llamada recursiva que no termina?
- ¿Cuándo usar recursividad?
- ¿Cuándo NO usar recursividad?
- Recursión vs. Iteración

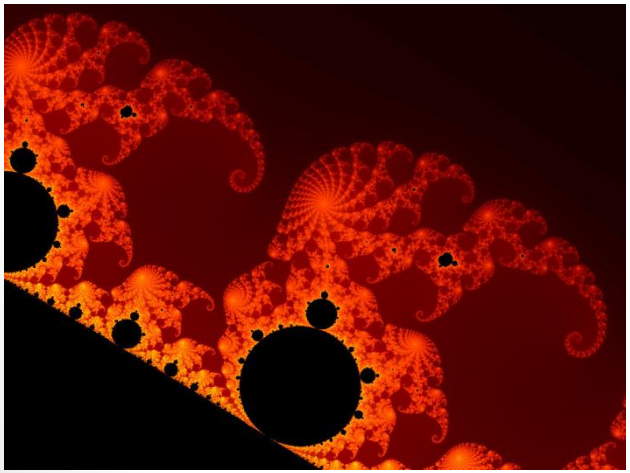
Recursión

- La **recursión** o **recursividad** es un concepto amplio, con muchas variantes, y difícil de precisar con pocas palabras.









- La Matrushka es una artesanía tradicional rusa. Es una muñeca de madera que contiene otra muñeca más pequeña dentro de sí. Esta muñeca, también contiene otra muñeca dentro. Y así, una dentro de otra.

Recursividad

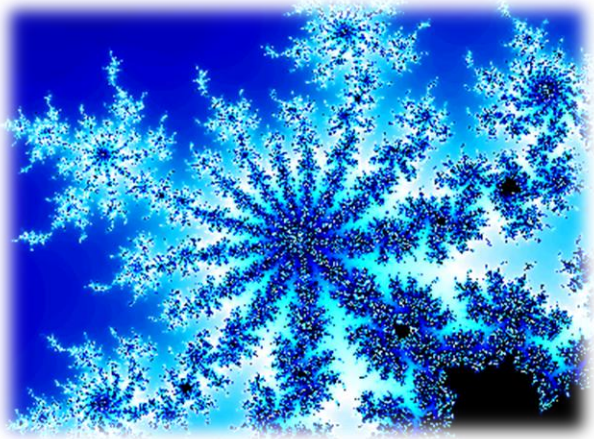
- La **recursividad** es un concepto fundamental en matemáticas y en computación.

- Es una **alternativa** diferente **para implementar** estructuras de repetición ***(iteración).***

- 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.**

- La recursividad es un recurso muy poderoso que **permite expresar soluciones simples y naturales a ciertos tipos de problemas**. Es importante considerar que no todos los problemas son naturalmente recursivos.

- Un **objeto recursivo** es aquel que **aparece en la definición de si mismo**, así como el que *se llama a sí mismo*.



Programación recursiva

- Las funciones recursivas se componen de:
  - **Paso base:** una solución simple para un caso particular *(puede haber más de un paso base)*. La secuenciación, iteración condicional y selección son estructuras válidas de control que pueden ser consideradas como enunciados.
  - **Paso recursivo**: una solución que involucra volver a utilizar la función original, con parámetros que se acercan más al paso base. Los pasos que sigue el paso recursivo son los siguientes:
    - El procedimiento se llama a sí mismo.
    - El problema se resuelve, resolviendo el mismo problema pero de tamaño menor.
    - La manera en la cual el tamaño del problema disminuye asegura que el paso base eventualmente se alcanzará.

- Asimismo, puede definirse un programa en términos recursivos, como una serie de pasos básicos, o **pasos base** *(condición de parada)*, y uno más **pasos recursivo**, donde vuelve a llamarse al programa.

- En un programa recursivo, esta serie de casos recursivos debe ser finita, terminando con al menos un paso base.

- Es decir, a cada paso recursivo se reduce el número de pasos que hay que dar para terminar, llegando un momento en el que no se verifica la condición de paso a la recursividad.

- Ni el **paso base** ni el **paso recursivo** son necesariamente únicos.

Tipos de Recursión

- Por otra parte, la recursividad también puede ser indirecta, si tenemos un procedimiento P que llama a otro Q y éste a su vez llama a P o directa si P llama a P.
  - ***Directa**: el programa o subprograma se llama directamente a si mismo.*

```
int funcion (int numero)
{
        if(numero==0)
                return 0;
        else
                funcion(numero-1);

        return numero;
}
```

- ***Indirecta**: el subprograma llama a otro subprograma, y éste, en algún momento, llama nuevamente al primero.*

```
int funcion_uno (int numero)
{
        for (i=0;i<numero;i++)
                printf("\nHola %d",numero);
        numero =funcion_dos(numero-1);    //Función 2
        return numero;
}
```

```
int funcion_dos
        if(numero==0) //Paso base
                return 0;
        else
                funcion_uno(numero-1);      //Paso
recursivo (Recursividad Indirecta)
        return numero;
}
```


Recursividad (Ejemplo: Factorial)

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

  - El factorial de n esta dado por:

$$n! = \prod_{i=1}^{n} i$$

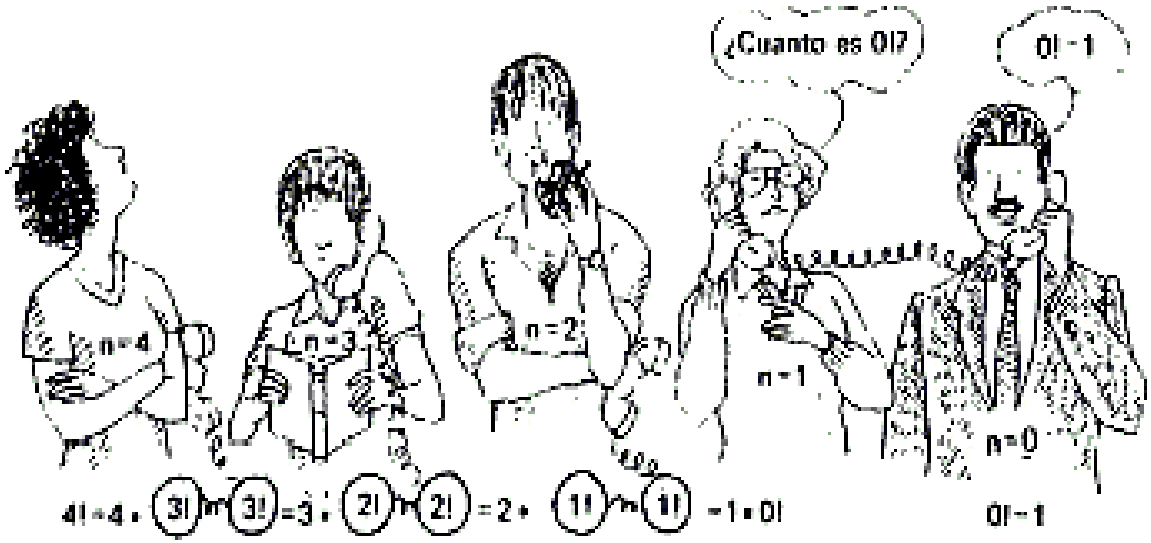
$$n! = 1\ x\ 2\ x\ 3\ x\ 4\ x\ \ldots\ x\ (n-1)\ x\ n$$

- Definición recursiva del factorial:

$$n! = \begin{cases} 1 & \rightarrow si\ n = 0 \\ (n-1)!\, n & \rightarrow \sin n \geq 1 \end{cases}$$

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


¿Cuánto es 3!?
3! = 3 • 2!
n = 4
n = 3
4! = 4 • 3!
¿Cuánto es 2!?
2! = 2 • 1!
A propósito, ¿qué vas a hacer esta noche?
n = 4
n = 3
n = 2
4! = 4 • 3! 3! = 3 • 2!
¡Hola mamá!, ¿cuánto es 1!?
1! = 1 • 0!
n = 4
n = 3
n = 2
n = 1
4! = 4 • 3! 3! = 3 • 2! 2! = 2 • 1!


¿Cuanto es 0!?
0! = 1
n=4
n=3
n=2
n=1
n=0
4! = 4 • 3! = 3 • 2! = 2 • 1! = 1 • 0!
0! = 1


3! es 6.
2! es 2.
1! es 1.
0! es 1.
n=3
n=2
n=1
n=0
3! = 3 • 2! =
3 • 2 = 6
2! = 2 • 1! =
2 • 1 = 2
1! = 1 • 0! =
1 • 1 = 1
0! = 1

- 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)!

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.

Recursividad (Ejemplo Factorial "Solución")

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

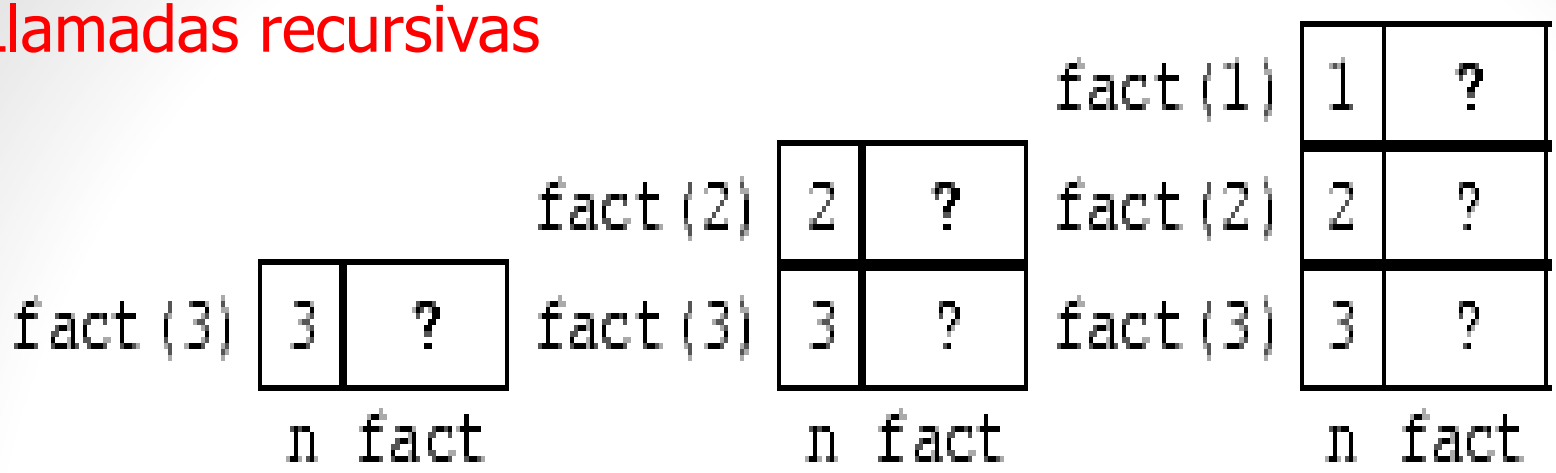
//Entrada: n entero no negativo.

//Salida: entero.

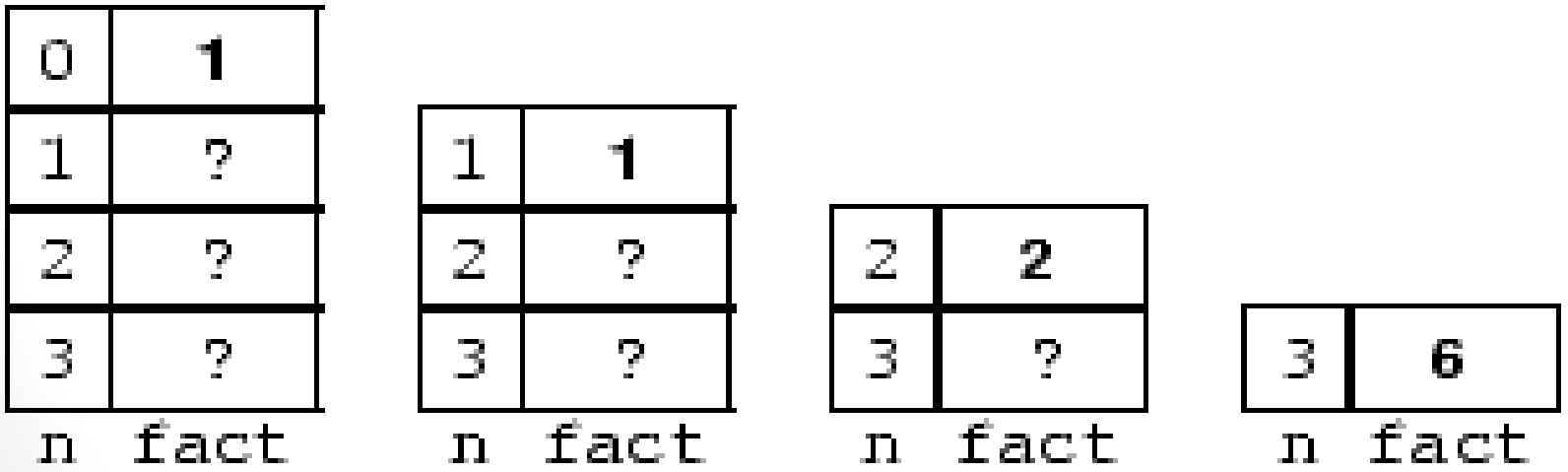
```
int fact (int n)
{
   if (n == 0)
     return 1;
   else
     return  fact(n – 1) * n ;
}
```

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.

Llamadas recursivas



Resultados de las llamadas recursivas



¿Por qué escribir programas recursivos?

- Son mas cercanos a la descripción matemática.
- Generalmente mas fáciles de analizar
- Se adaptan mejor a las estructuras de datos recursivas (*arboles, listas, etc.*).
- Los algoritmos recursivos ofrecen soluciones estructuradas, **modulares y elegantemente simples**.

¿Cómo escribir una función en forma recursiva?

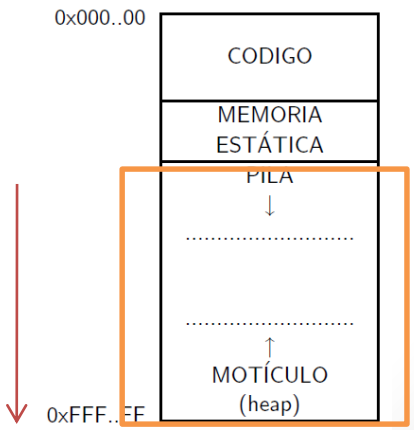
```
<tipo_de_regreso><nom_fnc> (<param>)
{
   [declaración de variables]
   [condición de salida]
   [instrucciones]
   [llamada a <nom_fnc> (<param>)]
   return <resultado>
 }
```

¿Qué pasa si se hace una llamada recursiva que no termina?

- Cada llamada recursiva almacena los parámetros que se pasaron al procedimiento, y otras variables necesarias para el correcto funcionamiento del programa. Por lo tanto si se produce una llamada recursiva infinita, esto es, que no termina nunca, llega un momento en que no quedará memoria para almacenar más datos, y en ese momento se abortará la ejecución del programa. Algunos S.O. modernos detectan la recursividad infinita de manera muy anticipada y evitan la ejecución de esta.

- Para probar esto se puede intentar hacer esta llamada en el programa factorial definido anteriormente:

factorial(-1);



Ejercicio de clase

- Considere la siguiente ecuación recurrente:

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

$a_0 = 1$

  - Diseña un algoritmo computacional recursivo que de la solución.

¿Cuándo usar recursividad?

- Para empezar, algunos lenguajes de programación no admiten el uso de recursividad, como por ejemplo el ensamblador o el FORTRAN. Es obvio que en ese caso se requerirá una solución no recursiva (iterativa).

- Para simplificar el código.

- **Cuando el problema tiene por definición una solución recursiva.**

- Cuando es necesario navegar en una estructura de datos recursiva ejemplo: **listas y árboles**.

¿Cuándo NO usar recursividad?

- **No se debe utilizar cuando la solución iterativa sea clara a simple vista**. Sin embargo, en otros casos, obtener una solución iterativa es mucho más complicado que una solución recursiva, y es entonces cuando se puede plantear la duda de si merece la pena **transformar la solución recursiva en otra iterativa.**

- Cuando los métodos usen arreglos o estructuras de datos con un **gran número** de **elementos (>miles)**.

- Cuando el método cambia de manera impredecible de campos.

- Cuando las iteraciones sean la mejor opción

Recursión vs. iteración

- **Repetición**
  - Iteración: ciclo explícito
  - Recursión: repetidas invocaciones a método
- **Terminación**
  - Iteración: el ciclo termina o la condición del ciclo falla
  - Recursión: se reconoce el paso base
- **En ambos casos podemos tener ciclos infinitos**
  - Considerar que resulta más positivo para cada problema la elección entre eficiencia **(iteración)** o una **buena ingeniería de software**, La recursión resulta normalmente más natural.