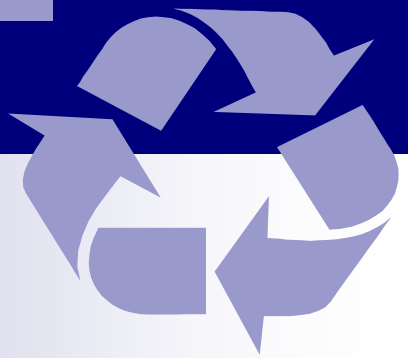
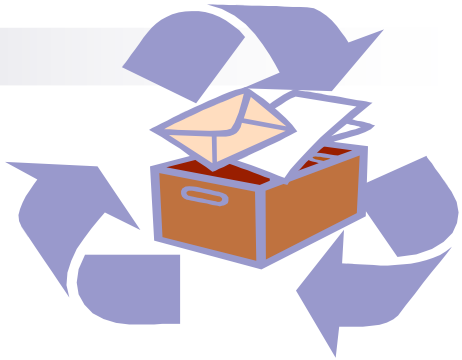
Recursividad



Ejemplo Matrushka

- 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.

¿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).
- Un caso base se encuentra identificando las condiciones de parada del algoritmo no recursivo.
- Una función recursiva tiene estructuras de control que se pueden formar combinando de manera válida la secuenciación

Función recursiva

- Algoritmo recursivo: una solución que involucra volver a utilizar la función original, con parámetros que se acercan más al caso base.
- Los pasos que se siguen para crear una función recursiva son los siguientes:
  1. La función se llama a sí mismo
  2. El problema se resuelve, con la llamada a la función con argumentos que se acerquen a el caso base y se apila
  3. Terminará la ejecución al llegar al caso base y se retornará las llamadas a la función anteriores ya apiladas

Ejemplo: factorial (iterativo)

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

Ejemplo:

- A continuaciòn 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.

¿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.
- Los algoritmos recursivos ofrecen soluciones estructuradas, modulares y elegantemente simples.

¿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 un procedimiento incluye una llamada a sí mismo se conoce como **recursión directa.**

Algunas Definiciones.

- Cuando un procedimiento llama a otro procedimiento y éste causa que el procedimiento original sea invocado, se conoce como **recursión indirecta.**

NOTA: Cuando un procedimiento recursivo se llama recursivamente a si mismo 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 a 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

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.

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
}
```

Trampas sutiles: Código ineficiente.

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

fib (100) toma 50 años 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

Serie fibonacci Iteración vs recursión