TEMA 3

RECURSIVIDAD

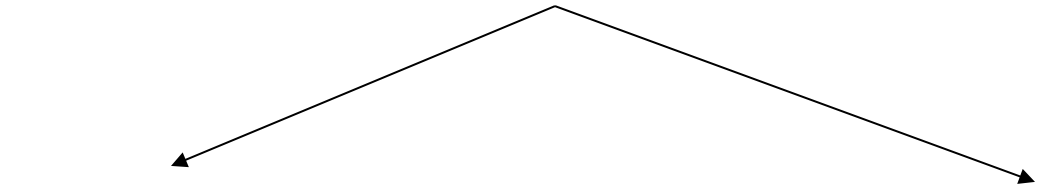
RECURSIVIDAD

- INDICE
  - ¿QUÉ ES LA RECURSIVIDAD?
  - IMPLEMENTACIÓN
  - UTILIDAD DE LA RECURSIVIDAD

¿QUÉ ES LA RECURSIVIDAD?

- Definición de un objeto de forma que lo que se define está incluido en la definición.

Formas de definir una función



directa

recurrente

$$n! = n * (n\text{-}1) * \ldots * 1$$

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

¿QUÉ ES LA RECURSIVIDAD?

- Recursividad o recursión:
  - una función aparece en su propia definición.
  - un problema se descompone en subproblemas **del *mismo* tipo**.
- Realización en programación:
  - subprogramas *recursivos.*
  - en el cuerpo del subprograma aparece una llamada a sí mismo.

¿QUÉ ES LA RECURSIVIDAD?

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

- Aplicación:
  - Forma natural de implementar relaciones recurrentes.
  - Técnica de repetición (alternativa al uso de bucles).

PARTES DE UN SUBPROGRAMA RECURSIVO

- **Caso base**:
  - dados los parámetros de entrada, la solución del problema es “simple”.
  - Solución conocida NO recursiva del caso más sencillo.
  - no se generan llamadas recursivas, y se devuelve directamente una solución. CONDICIÓN DE CORTE.
  - **Siempre debe haber al menos un caso base.**
    - Ejemplo: 0! = 1
- **Caso recurrente**:
  - caso más complejo: *no* hay solución trivial.
  - se reduce a otro caso más simple de la misma naturaleza.
  - se debe ir acercando al caso base = cada llamada recursiva se realiza con un dato más pequeño
    - Ejemplo: 4! = 4*3!

SUBPROGRAMA RECURSIVO

- **Recursión infinita**:
  - Se produce cuando se realiza una sucesión infinita de llamadas.
  - Es debida a que el control pasa siempre al caso recurrente, por lo que nunca se llega al caso base.
    - Ejemplo: (-1)! produciría una recursión infinita.

SUBPROGRAMA RECURSIVO

- Evitar la recursión infinita:
  - Usar una *estructura de selección* (`Si` o `Segun`), 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).

SUBPROGRAMA RECURSIVO

- **Recursión directa:**
  - La función se llama a sí misma.
- **Recursión indirecta:**
  - La función A llama a la función B, y ésta llama a A.

IMPLEMENTACIÓN

- No hace falta ninguna ampliación del lenguaje.
  - Ejemplo:

```
Funcion fact <- factorial (n)
    Si n=0 Entonces
          fact <- 1;
    SiNo
          fact <- (n*factorial(n-1))
    FinSi
FinFuncion
```

Emplear un procedimiento o función que hace referencia a sí mismo dentro de la propia definición.

IMPLEMENTACIÓN

- Llamadas al procedimiento para el cálculo de factorial, para n = 5

Nivel de Recursión	*N*	devuelve	recibe	resultado
1	5	5 * Factorial(4)	24	120
2	4	4 * Factorial(3)	6	24
3	3	3 * Factorial(2)	2	6
4	2	2 * Factorial(1)	1	2
5	1	1 * Factorial(0)	1	1
6	0	1	-	1

UTILIDAD DE LA RECURSIVIDAD

- **Ventajas**

  - Da lugar a algoritmos simples que solucionan problemas complejos. Es más natural y sencillo de escribir.

- **Inconvenientes**

  - Los algoritmos recursivos suelen ser difíciles de entender.

  - Son más costosos en tiempo y espacio (registro = memoria) que sus versiones iterativas. Coste debido a llamadas a subprogramas, creación de variables dinámicamente en la pila, duplicación de variables, etc..

UTILIDAD DE LA RECURSIVIDAD

- Un algoritmo recursivo ejecuta cálculos repetidas veces mediante llamadas consecutivas a sí mismo.
- La recursividad es una **alternativa a la utilización de estructuras iterativas.**
- En general, si la solución no recursiva no es mucho más larga que la recursiva, usar la NO recursiva.
- La iteración y la recursividad tienen el mismo objetivo: ejecutar un bloque *n* veces y que con una condición de fin adecuada lo termine.

UTILIDAD DE LA RECURSIVIDAD

- No vale la pena que un programador invierta tiempo y esfuerzo desarrollando una complicada solución iterativa para un problema que tiene una solución recursiva sencilla.
- Tampoco vale la pena implementar soluciones recursivas para problemas que pueden solucionarse fácilmente de manera iterativa.