UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escola Tècnica Superior d'Enginyeria Informàtica

etsinf

Tema 1. Recursividad

Programación (PRG)

Jorge González Mollá

Departamento de Sistemas Informáticos y Computación

Índice

Definición

- El término recursivo se aplica a cualquier entidad cuya definición se realice en función de sí misma. Por ejemplo, una función matemática es recursiva si su resolución requiere la solución previa de la misma función para un caso más sencillo. Veámoslo con ayuda de la función factorial:

$$n! = \begin{cases} 1 & n = 0 \\ \prod_{i=1}^{n} i & n > 0 \end{cases}$$

DEFINICIÓN ITERATIVA (NO RECURSIVA)

$$n! = \begin{cases} 1 & n = 0 \\ n \cdot (n-1)! & n > 0 \end{cases}$$

DEFINICIÓN RECURSIVA

- En recursividad, siempre hay 2 situaciones distintas que hay que distinguir:
  - Caso base: el problema a resolver es lo suficientemente pequeño como para presentar una solución trivial sin características recursivas
  - Caso general: en caso contrario, la solución al problema principal requiere obtener previamente la solución de una instancia más simple

Programación

- En programación, el concepto de recursividad se aplicaría a los algoritmos, cuya implementación en Java resulta fácil mediante métodos recursivos. Un método es recursivo si se invoca a sí mismo:

```
/** n>=0 */
public static int factorial(int n) {
  if (n==0)                        // Condición     del caso base
     return 1;                     // Instrucciones del caso base
  else /* n>0 */                   // Condición     del caso general
     return n * factorial(n-1);    // Instrucciones del caso general
}
```

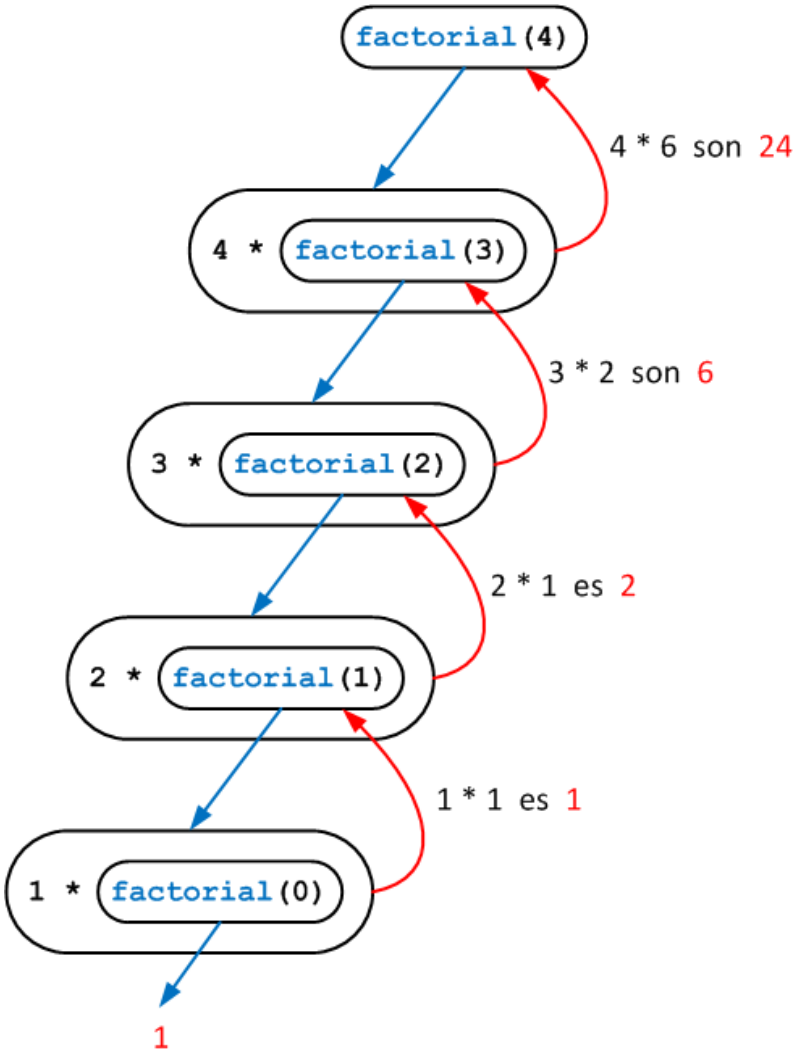
- El bloque principal del código es una estructura de selección condicional que permita distinguir entre el caso base y el caso general.
- Las llamadas recursivas que se invocan de forma progresiva deben ser estrictamente más simples, es decir, cada vez más cercanas a su caso base.

Taxonomía

- Recursividad lineal: el caso general describe una única llamada recursiva en tiempo de ejecución, de este modo se crea una secuencia de llamadas.
  - Final: el resultado del método es exactamente el mismo que el de la llamada recursiva; es decir, la solución del caso más sencillo es a su vez la del caso más complejo, por tanto no hay combinación de resultados.
  - No final: el resultado de la llamada recursiva se utiliza como un paso previo para poder calcular el resultado de la llamada de la cabecera, cuyo valor final será potencialmente distinto al de la llamada recursiva.
- Recursividad múltiple: en el caso general hay 2 o más llamadas recursivas que se invocan en tiempo de ejecución, creando así un árbol de llamadas.

Taxonomía

- Por ejemplo, el método `factorial(n)` tiene recursividad lineal (se hace una única llamada recursiva a `factorial(n-1)` en tiempo de ejecución) no final (el resultado de la llamada recursiva se multiplica por el valor de n para calcular el resultado a devolverse).
- Para `factorial(4)` se genera la secuencia de llamadas de la figura.



Diseño

1. Declarar la cabecera del método, indicando las precondiciones de entrada del algoritmo
2. Explicitar el caso base y el caso general, estableciendo para cada caso las instrucciones pertinentes para resolver cada problema por separado. Comprobar que se cubren todos las situaciones posibles, es decir, que ante cualquier posible entrada del problema, el algoritmo ejecuta o bien el caso base o bien el caso general.
3. Determinar que en cada nueva llamada recursiva se cumple que:
   - El nuevo subproblema a resolver es estrictamente más cercano al caso base que el problema anterior, por lo que a partir de cualquier caso general planteado inicialmente, eventualmente se alcanzará el caso base (terminación del algoritmo).
   - Los datos de entrada de la nueva llamada recursiva siguen cumpliendo las precondiciones de entrada del algoritmo.
4. Verificar la corrección del algoritmo, es decir, que para cualquier entrada, la solución del algoritmo sea correcta (suele requerir una demostración matemática por inducción sobre algún parámetro del algoritmo).

Ejemplo: Potencia n-ésima

1. Dados un número real $a \neq 0$, y un entero $n \geq 0$, obtener la potencia a^n

```
/** a!=0 y n>=0 */
public static double potencia(double a, int n)
```

2. Caso base: Si $n=0$, entonces $a^n = 1$

   Caso general: Si $n>0$, entonces $a^n = a * a^{n-1}$

```
/** n>=0 y a!=0 */
public static double potencia(double a, int n) {
        if (n==0) return 1;
        else return a * potencia(a, n-1);
}
```

recursividad lineal no final

3. En el caso general, en cada llamada el valor del segundo parámetro decrece en 1 unidad; así, para cualquier posible valor inicial de $n \geq 0$, eventualmente éste llegará a 0 alcanzando por consiguiente el caso base, finalizando así el algoritmo. Además, las precondiciones $a \neq 0$ y $n \geq 0$ siguen satisfaciéndose siempre en cada una de las llamadas generadas.

Ejemplo: Resto de la división entera

1. Dados dos números enteros $a \geq 0$ y $b > 0$, calcular el resto de su división entera a/b.

```
/** a>=0 y b>0 */
public static int resto(int a, int b)
```

2. Caso base: Si $a < b$, entonces el resultado es a

   Caso general: En otro caso, el resultado es el `resto` de $(a-b)/b$

```
/** a>=0 y b>0 */
public static int resto(int a, int b) {
        if (a<b) return a;
        else return resto(a-b,b);
}
```

recursividad lineal final

3. En el caso general, en cada llamada el valor del primer parámetro va decreciendo; en algún momento será inferior al segundo parámetro, alcanzando el caso base y finalizando así el algoritmo. Además, las precondiciones $a \geq 0$ y $b > 0$ se siguen cumpliendo llamada tras llamada.

Ejemplo: Algoritmo de Euclides I

1. Dados dos números enteros $a>0$ y $b>0$, calcular su máximo común divisor (m.c.d.) mediante el algoritmo de Euclides.

```
/** a>0 y b>0 */
public static int mcd(int a, int b)
```

2. Caso base: Si $a=b$, el resultado es b
   Caso general: Si $a>b$, el resultado es el mcd entre $a-b$ y b
   Si $a<b$, el resultado es el mcd entre a y $b-a$

```
/** a>0 y b>0 */
public static int mcd(int a, int b) {
        if (a==b) return b;
        else if (a>b) return mcd(a-b,b);
             else return mcd(a,b-a);
}
```

recursividad lineal final

3. En el caso general, en cada llamada el valor del primer o del segundo parámetro van decreciendo; en algún momento llegarán a ser idénticos, por consiguiente se alcanzará el caso base, finalizando así el algoritmo. Además, llamada tras llamada se cumplen las precondiciones $a>0$ y $b>0$.

Ejemplo: Algoritmo de Euclides II

1. Dados dos números enteros a>0 y b>0, calcular su máximo común divisor (m.c.d.) mediante el algoritmo de Euclides.

```
/** a>0 y b>0 */
public static int mcd(int a, int b)
```

2. Caso base: Si el resto de a/b es 0, el resultado es b

   Caso general: En otro caso, el resultado es el mcd de b y el resto de a/b

```
/** a>0 y b>0 */
public static int euclides(int a, int b) {
        if (a%b==0) return b;
        else return euclides(b,a%b);
}
```

recursividad lineal final

3. En el caso general, en cada llamada el valor del segundo parámetro va decreciendo; eventualmente éste llegará a ser el m.c.d. de los valores originales, alcanzando así el caso base y finalizando el algoritmo. Además, llamada tras llamada las precondiciones a>0 y b>0 se cumplen.

Ejemplo: Sucesión de Fibonacci

1. Calcular el término n-ésimo de la sucesión de Fibonacci, asumiendo que el término 0-ésimo es el primer término de la sucesión 0, 1, 1, 2, 3, 5, …

```
/** n>=0 */
public static int fibonacci(int n)
```

2. Caso base: Si $n \leq 1$, dicho término es el propio valor de n

   Caso general: Si n>1, el resultado es la suma de los 2 términos inmediatamente anteriores, `fibonacci(n-1)` y `fibonacci(n-2)`

```
/** n>=0 */
public static int fibonacci(int n) {
   if (n<=1) return n;
   else return fibonacci(n-1) + fibonacci(n-2);
}
```

recursividad múltiple

3. En el caso general, en cada llamada se va tendiendo a términos inferiores de la sucesión; en algún momento se cumplirá la condición del caso base, finalizando así el algoritmo. Además, en cada llamada se satisface $n>=0$.

Ejemplo: Sucesión de Fibonacci

- El método `fibonacci` es un método recursivo múltiple (en el caso general se realizan 2 llamadas recursivas en tiempo de ejecución, cuyos resultados particulares se suman para componer el resultado del método).
- Para `fibonacci(4)` se genera el árbol de llamadas de la figura: