Teoría de la Recursividad
Capítulo 2: Primeras nociones sobre recursividad

En Matemáticas, algunas definiciones se expresan de forma "recurrente". Un caso típico es la conocida función factorial. Tradicionalmente, se define como factorial de un número natural n (n>1) y se simboliza n!, al producto de los n primeros números naturales (a partir de 1):

$$n! = 1*2*3*...*n \quad : \quad n>1$$

Distintas consideraciones llevaron posteriormente a definir dos casos no comprendidos en la definición anterior. Por convenio, se acordó que $0! = 1$ y $1! = 1$.

La definición "recurrente" es más "compacta", contempla todos los casos y evita la ambigüedad de los puntos suspensivos:

$$n! \begin{cases} 0 & n = 0 \\ n * (n-1)! & n > 0 \end{cases} \qquad [R1]$$

Aunque una definición de este tipo parece llevarnos a un "círculo vicioso", pues factorial de un número se expresa a partir de factorial de otro número, pronto nos habituamos a interpretarla como un proceso de cálculo que va "descendiendo" en complejidad, hasta llegar al caso más simple (0!), definido directamente:

Ej.: $3! = 3*(2!) = 3 * 2*(1!) = 3 * 2 * 1*(0!) = 3*2*1*1 = 6$

Observemos que, aplicando repetidamente la MISMA definición recurrente, nos encontramos en cada caso con una nueva situación, y la regla R1 es capaz de responder adecuadamente a cada una de esas variantes, para darnos finalmente el resultado deseado.

En algunos casos resulta prácticamente inevitable una definición de este tipo. Así, la sucesión:

$\{Xn\}$: 1, 1, 2, 3, 5, 8, 13, 21, 34,...

llamada “de Fibonacci”, se comprende claramente con la definición

$$Fib(n) \begin{cases} 1 & n \leq 1 \\ Fib(n-1) + Fib(n-2) & n > 1. \end{cases} \qquad [R2]$$

Criterios “recurrentes” se aplican a menudo en otras reas de trabajo de las matemáticas, a veces sin que tomemos plena conciencia de ello. Un ejemplo en este aspecto, es el conocido conjunto de reglas sobre el orden de operaciones para evaluar una expresión algebraica:

1ro. - Expresiones entre paréntesis
2do. - Potencias [R3]
3ro. - Multiplicaciones y divisiones
4to. - Adiciones y sustracciones.

Al evaluar una expresión que contenga varios paréntesis anidados, realizamos también un proceso recurrente, mediante el cual reiteradamente debemos aplicar el MISMO grupo de reglas.

Ejemplo concreto: ¿En qué, orden debe evaluarse la expresión:

(((2 + 3 * 4) -1) * 5 - (6 / 2 + 4)) * (3 + 1) ?

R/

1- Recorremos de izquierda a derecha la expresión, hasta encontrar una posibilidad de aplicar R3: el primer paréntesis interior.
2- En ese paréntesis: (2 + 3 * 4) aplicando nuevamente R3 se obtiene (2 + 3 * 4) = 3*4 + 2 = 14.
3- Se sustituye en la operación inicial y se elimina ese paréntesis: ((14 -1) * 5 - (6 / 2 + 4)) / (3 + 1)
4- Ahora, el primer paréntesis interior que se cierra es (14-1) = 13, y, efectuando y eliminando queda (13 * 5 - (6 / 2 + 4)) / (3 + 1).
5- Se recorre la expresión hasta encontrar un caso en que pueda aplicarse R3 otra vez. Se encuentra (6 / 2 + 4) En ,l, mediante nueva aplicación de R3, resulta 6/2 + 4 = 7.
6- Queda: (13 * 5 – 7) / (3 + 1).
7- Se encuentra ahora (13 * 5 - 7). Nueva aplicación de R3 conduce ahora a (13 * 5ý - 7) = 5 * 13 - 7 = 25*13 - 7 = 325 - 7 = 318.
8- Sustituyendo, la expresión queda 318 / (3 + 1).
9- Efectuando el último paréntesis (otra vez R3) resulta 318 / 4 = 79.5

Reiteradamente hubo que aplicar el orden de operaciones según el MISMO conjunto de reglas R3. Obsérvese que EL MISMO grupo de reglas, aplicado cada vez que hizo falta, llevó a realizar OPERACIONES DIFERENTES, según la situación particular de la expresión en cada etapa de trabajo. En cierto modo, con un procedimiento recursivo ocurrir algo parecido:

- UN MISMO procedimiento, aplicado a etapas o momentos diferentes, va a originar la selección del elemento (u operación) apropiado a la situación específica de dicha etapa. Volviendo a aplicar EL MISMO procedimiento a la nueva situación creada, (nueva etapa) se hará una nueva selección apropiada. Y así, aplicándolo sucesivamente, se podrá obtener una sucesión de operaciones que se ajustan estrictamente a las reglas que se habían fijado. (Desde luego, esto ocurrirá si el procedimiento se ha programado correctamente, previendo todas las incidencias del proceso).

- Si, adicionalmente, se logra darle FLEXIBILIDAD al procedimiento, de modo que el mismo pueda ir PROBANDO en cada etapa con diferentes variantes aceptables, el mismo podrá ir haciendo sucesivamente DISTINTAS elecciones en cada etapa (como lo hace, por ejemplo, un ciclo `for`...) y por tanto, en el conjunto de todas las etapas, se irá logrando TODAS o al menos LAS MEJORES combinaciones posibles. De entre ellas, será bastante fácil ir guardando la mejor hasta el momento, hasta concluir con la mejor de todas, según el problema que se haya propuesto.

En los pocos ejemplos anteriores, es posible apreciar algunas características que posteriormente también estarán presentes en el análisis y solución de problemas de programación con aplicación de técnicas recursivas: Así, vemos que en los procesos de cálculo anteriores, hay que ir solucionando gradualmente el problema, mediante "pasos" (o "niveles") que nos aproximan cada vez más a la solución; aparecen en el proceso "resultados intermedios", que deben quedar provisionalmente "en espera", o transmitirse de una etapa de trabajo a otra; y algunas más que veremos oportunamente.

Por lo visto anteriormente, resulta muy natural que, cuando los autores de un lenguaje de programación debieron incluir entre sus posibilidades el tratamiento por computadora de procesos recurrentes, lo implementaran de la manera más natural: se admitió que un "subprograma" (Método) pueda "llamarse" a SI MISMO reiteradamente, desencadenando una sucesión de acciones y cálculos "en niveles", hasta alcanzar en uno de ellos la solución. En computación, este mecanismo se conoce con el nombre de **Recursividad**, o en ocasiones, **Recursión**.

En realidad, (como se sugirió antes al hablar de "flexibilidad"), el proceso suele ser mucho más rico que una simple sucesión "lineal" de etapas cada vez más próximas al objetivo: Puede haber "tanteos" del camino a seguir (algo así como "si escojo esta variante vamos a ver que ocurre en las etapas siguientes"), y controles apropiados que faciliten "volver atrás", cuando sea evidente que la combinación actual de pasos no es adecuada, o para explorar nuevas combinaciones. Y son precisamente todas estas posibilidades las que transforman a la recursividad en una poderosa herramienta de programación, capaz de adaptarse a muy variadas situaciones, y, en muchos casos, más sencilla de plantear que otras vías, con el consiguiente ahorro de esfuerzo y de tiempo para el programador (aunque con más trabajo para la máquina...).

Vale la pena, después de analizadas estas ideas, observar y comparar dos funciones que realizan el mismo cálculo, una de ellas no recursiva, la otra, aplicando recursividad: se trata de la ya mencionada función factorial. En ambas versiones, se supone que el valor de n es correcto, es decir, un número natural (no se programan acciones para el caso de que hubiera un error en el valor dado: n).

Versión no-recursiva:

```
public static int Fact(int n)
{
   int p=1;
   for(int i=2; i<=n; i++)
         p*=i;
   return p;
}
```

Versión recursiva:

```
public static int Fact(int n)        //  Version A
{
      if (n==0 || n==1) return 1;
      return n*Fact(n-1);
}

public static int Fact(int n)        //  Version B
{
      return (n==0)? 1 : n*Fact(n-1);
}
```

No es nuestra intención explicar en este momento el funcionamiento de la versión recursiva. Para ello, debemos esperar aún por algunos detalles que se explican en el siguiente capítulo.
Pero sí, podemos formular algunas observaciones:

1. En cuanto a "complejidad" de programación, los ejemplos muestran que ambas formas están bastante balanceadas. Un procedimiento o función recursivos no tienen por qué, ser complicados.
2. Es característico de la versión recursiva que ella se invoca a sí misma. Aquí, se puede observar en la línea "`return`...".
   (NOTA: función se le llama a un método cuando este devuelve algo)
3. No tenga ninguna duda: Las versiones "no recursivas" (a veces se dice "iterativas", si contienen ciclos) surgen en nuestra mente de manera más natural y requieren, generalmente, un análisis más sencillo, y por ello, siempre que podamos encontrar soluciones no-recursivas, debemos preferirlas. Pero muchas veces no hay (o no vemos) una VIA DIRECTA para resolver el problema, y no queda más remedio que realizar muchos "tanteos". Ahí es donde debemos aplicar la solución recursiva.

Con este primer y muy elemental acercamiento a la noción de recursividad, pasamos a los capítulos siguientes, donde tratamos otros temas imprescindibles para profundizar el concepto. El mismo se expone con más detalle en el Capítulo 4.