CAPÍTULO 6

RECURSIVIDAD (RECURRENCIA)

> "**Para entender la recursividad primero hay que entender la recursividad.**"

Tratamos ahora de un concepto de gran utilidad para la programación: la recursividad. La recursividad es un comportamiento presente en la misma naturaleza de muchos problemas para los que buscamos soluciones informáticas. En este capítulo queremos presentar el concepto de recursividad o de recurrencia, y mostrar luego las etapas habituales para diseñar algoritmos recursivos. Presentamos también el concepto de inducción, que trae de la mano al de recurrencia y que permite en muchos casos llegar a una correcta definición de un algoritmo recurrente o recursivo.

Para la redacción de este capítulo se ha empleado la siguiente bibliografía:

- "**Diseño y verificación de algoritmos. Programas recursivos.**" Francisco Perales López

Edita: Universitat de les Illes Balears. Palma 1998.
Col·lecció Materials Didáctics, n. 56.
Capítulo 3: "**Diseño del algoritmos recursivos**".

El concepto de recursividad.

La palabra "recursividad" no aparece en el diccionario de la Real Academia. Al menos en su 22º edición del año 2001. Tampoco aparece en el Diccionario de María Moliner.

La palabra castellana que se emplea para tratar de lo que se va a hablar en este capítulo es la de **Recurrencia**. El diccionario de la RAE define así esta palabra: ***1.*** *f. Cualidad de recurrente.* ***2.*** *f. Mat. Propiedad de aquellas secuencias en las que cualquier término se puede calcular conociendo los precedentes*.

De todas formas se ha mantenido el término "recursividad" en el título del capítulo porque esta palabra sí se emplea de forma habitual en el lenguaje informático y aparece en cualquier referencia o manual que verse sobre algoritmia. Aquí emplearemos ambas palabras de forma indistinta.

La recursividad está presente en muchos sistemas del mundo real. Y por eso, porque muchos problemas que queremos afrontar tiene una esencia recursiva, la recursividad es una herramienta conveniente y muy útil para diseñar modelos informáticos de la realidad que deseamos modelizar.

Se dice que un **sistema** es **recursivo** cuando está parcial o completamente definido en términos de sí mismo.

Quizá todos nos hemos planteado alguna vez una imagen recursiva: una fotografía que muestra a una persona que sostiene entre sus manos esa fotografía, que muestra a esa persona que sostiene entre sus manos esa fotografía, que muestra a esa persona que sostiene entre sus manos esa fotografía, que muestra…

Una imagen recursiva podemos definirla, por ejemplo, de la siguiente manera. Se toma un cuadrado, se divide en nueve cuadrados y se elimina el cuadro central. Con cada uno de los ocho cuadrados restantes se realiza la misma operación: dividir en nueve cuadrados y eliminar el central. Y con cada uno de ellos de nuevo lo mismo, y lo mismo con cada uno de los cuadrados que quedan, y así hasta el infinito.

Por ejemplo, en la figura 6.1. se representa este dibujo de definición recurrente. Este cuadro puede parecer inútil. No se trata ahora de explicar sus muchas virtudes, sino de presentar el concepto de recursividad. El cuadro representa el conocido conjunto de Cantor en el plano. Basta ponerse en Google y buscar "conjunto de Cantor" para lograr encontrar dibujos y aplicaciones prácticas de este conjunto. Se pueden seguir buscando representaciones gráficas de curvas cuya definición es recurrente: la curva de Peano, la de Hilbert, la de Hock,… o de los fractales. Pero ese es otro asunto.

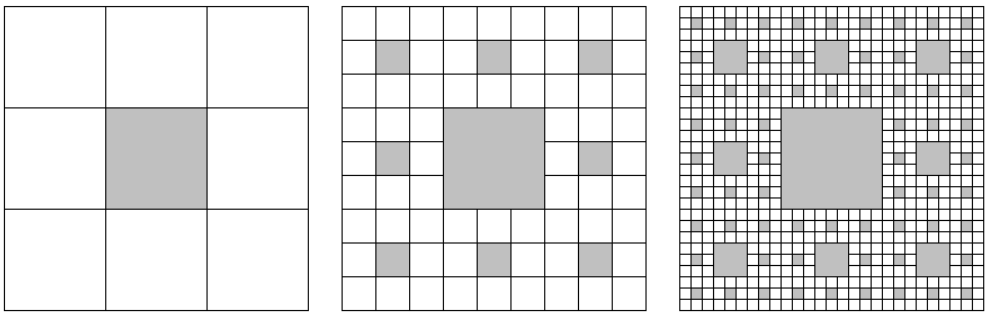


Figura 6.1.: Conjunto de Cantor

Pero además de poder mostrar con este concepto algunas curiosidades aparentemente inútiles (sólo "aparentemente"), también podemos presentar otros ejemplos de recurrencia. Hay muchos problemas matemáticos que se resuelven mediante técnicas recursivas. Podemos acudir a muchos de los problemas planteados en los capítulos anteriores.

El **algoritmo de Euclides** es recurrente. El máximo común divisor de dos números m y n ($mcd(m,n)$) se puede definir como el máximo común divisor del segundo (de n) y del resto de dividir el primero con el segundo: $mcd(m,n) = mcd(n, m\ mod\ n)$. La secuencia se debe parar cuando se alcanza un valor de segundo entero igual a cero: de lo contrario en la siguiente vuelta se realizaría una división por cero.

Él **cálculo del factorial**: podemos definir el factorial de un entero positivo n ($n!$) como el producto de n con el factorial de $n - 1$: $n! = n \cdot (n - 1)$. Y de nuevo esta definición deberá tener un límite: el momento en el que se llega al valor cero: el factorial de cero es, por definición, igual a uno.

La búsqueda de un **término de la serie de Fibonacci**: Podemos definir la serie de Fibonacci como aquella cuyo término n es igual a la suma de los términos $(n - 1)$ y $(n - 2)$: $Fib_n = Fib_{n-1} + Fib_{n-2}$. Y de nuevo se tiene el límite cuando $n \leq 2$ donde el valor del término es la unidad.

Cálculo de una **fila del triángulo de Tartaglia**. Es conocida la propiedad del triángulo de Tartaglia según la cual, si calculamos el valor de cualquier elemento situado en el primer o último lugar de cada fila, el valor de ese elemento es igual a uno; y el valor de cualquier otro elemento del triángulo resulta igual a la suma de los elementos de su fila anterior que están encima de la posición que deseamos calcular. Es decir:

$t_{i,0} = 1$; $t_{i,i} = 1$; $t_{i,j\,(j \neq 0, j \neq i)} = t_{i-1,j-1} + t_{i-1,j}$.

donde $t_{i,j} = \binom{i}{j}$, donde siempre se verifica que $0 \leq j \leq i$.

Y así tenemos que para calcular una fila del triángulo de Tartaglia no es necesario acudir al cálculo de binomios ni de factoriales, y basta con conocer la fila anterior, que se calcula si se conoce la fila anterior, que se calcula si se conoce la fila anterior… hasta llegar a las dos primeras filas, que están formadas todo por valores uno. Tenemos, por tanto, y de nuevo, un camino recurrente hecho a base de simples sumas para

encontrar la solución que antes habíamos buscado mediante el cálculo de tres factoriales, dos productos y un cociente para cada elemento.

En todos los casos hemos visto un camino para definir un procedimiento recurrente que llega a la solución deseada. En los temas anteriores no hemos llegado a soluciones recurrentes, sino que únicamente hemos hecho uso de estructuras de control iterativas. Ahora, en todas estas soluciones, podemos destacar dos características básicas de todo procedimiento recurrente o recursivo:

1. Lo indicaba el diccionario de la RAE: Propiedad de aquellas secuencias en las que cualquier término se puede calcular conociendo los precedentes. La recurrencia se puede usar en aquellos **sistemas donde sepamos llegar a un valor a partir de algunos valores precedentes**. Es la propiedad que implica recurrencia.
2. Es imprescindible que, además, haya unos **valores iniciales preestablecidos** hacia los que se converge: unos valores que no se definen por recurrencia. En el algoritmo de Euclides se para el proceso en cuanto se llega a dos valores tales que su módulo es cero. En el algoritmo del cálculo del factorial siempre se llega a un valor mínimo establecido en el cero: $0! = 1$. En el algoritmo de Fibonacci tenemos que los dos primeros elementos de la sucesión son unos. Y para el triángulo de Tartaglia tenemos que todos los extremos de todas las filas son también iguales a uno.

La recursividad es una herramienta de diseño de algoritmos aceptada en la mayoría de los lenguajes de programación. Esos lenguajes permiten la creación de un procedimiento o función que hace referencia a sí mismo dentro de la propia definición. Eso supone que al invocar a una función recursiva, ésta genera a su vez una o varias nuevas llamadas a ella misma, cada una de las cuales genera a su vez una o varias llamadas a la misma función, y así sucesivamente. Si la definición está bien hecha, y los parámetros de entrada son adecuados, la cadena de

invocaciones termina felizmente en alguna o varias llamadas que no generan nuevas invocaciones. Esas llamadas finales terminan su ejecución y devuelven el control a la llamada anterior, que finaliza su ejecución y devuelve el control a la llamada anterior, y así retornando el camino de llamadas emprendido, se llega a la llamada inicial y se termina el proceso.

Cuando un procedimiento recursivo se invoca por primera vez decimos que su **profundidad de recursión** es 1 y el procedimiento se ejecuta a nivel 1. Si la profundidad de recursión de un procedimiento es N y de nuevo se llama a sí mismo, entonces desciende un nivel de recursión y pasa al nivel $N + 1$. Cuando el procedimiento regresa a la instrucción de donde fue llamado asciendo un nivel de recursión.

Si una función o procedimiento P contiene una referencia explícita a sí mismo, entonces decimos que tenemos un procedimiento **recursivo directo**. Si P contiene una referencia a otro procedimiento Q el cual tiene (directa o indirectamente) una referencia a P, entonces decimos que P es un procedimiento **recursivo indirecto**.

Veamos el ejemplo de una función recursiva que calcula el factorial de un entero que recibe como parámetro de entrada:

Función $factorial$ (N Entero) → Entero.
1. **Si** $N = 0$ **Entonces** $factorial \leftarrow 1$
   Sino (es decir, $N \neq 0$) **Entonces** $factorial(N) \leftarrow N \cdot factorial(N-1)$
2. **Fin**.

Supongamos que invocamos al procedimiento con el valor $N = 5$. La lista de llamadas que se produce queda recogida en la Tabla 6.1. Hay un total de seis llamadas al procedimiento definido para el cálculo del factorial, hasta que se le invoca con el parámetro $N = 0$, que es el único que en lugar de devolver un valor calculado con una nueva llamada, lo que hace es devolver directamente el valor 1.

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

Tabla 6.1.: Llamadas al procedimiento definido para el cálculo del Factorial para el caso de $N = 5$.

La demostración por inducción o recurrencia.

Vamos a ver a continuación del principio de demostración por recurrencia o inducción sobre los números naturales. Es conveniente tratar de ello fuera del contexto informático, porque esta forma de demostración muestra un modo de razonamiento esencial para el correcto dominio de la programación recursiva.

La demostración por recurrencia se emplea para establecer que una determinada propiedad P es cierta para todo número natural. Existen distintas formas de plantear la demostración por inducción. Aquí recogemos la de uso más habitual.

Se demuestra que P es cierto para todo natural a partir de los siguientes dos pasos de razonamiento:

a) **Base** de la inducción o recurrencia: Demostrar que P es cierto para el entero 0.

b) **Recurrencia**: se demuestra que la propiedad es **hereditaria**. Para ello se establece una **hipótesis de inducción**: se supone que P es cierto para el natural n, y con base a eso se debe demostrar que entonces es cierto para $n + 1$.

Ejemplos de esta forma de demostración hay muchos. Presentamos aquí uno: supongamos que queremos demostrar que

$$\sum_{j=0}^{n} 2^j = 2^{n+1} - 1$$

Podemos demostrarlo con los dos pasos antes indicados:

a) Es cierto para $n = 0$: $\sum_{j=0}^{0} 2^j = 2^0 = 1 = 2^1 - 1$.

b) Supongamos que es cierto para un determinado valor k: $\sum_{j=0}^{k} 2^j = 2^{k+1} - 1$.

c) Entonces, demostrar que se cumple para $k + 1$:

$$\sum_{j=0}^{k+1} 2^j = \sum_{j=0}^{k} 2^j + 2^{k+1} = (2^{k+1} - 1) + 2^{k+1} = 2 \cdot 2^{k+1} - 1 = 2^{k+2} - 1$$

cqd.

Definición recurrente de un conjunto.

Así como se pueden demostrar numerosas propiedades de los enteros mediante la inducción o recurrencia, también se pueden definir conjuntos.

La noción de conjunto juega un papel importante en la programación y es, por tanto, esencial saber definir conjuntos. La inducción o recurrencia aporta un método para definir muy interesante, pues permite establecer reglas de construcción de los conjuntos: permite por tanto dar una definición de conjuntos infinitos de forma inequívoca, no por extensión, ni por descripción de sus propiedades, sino explicitando un modo de construir ese conjunto.

La definición recurrente de un conjunto muestra el modo de construir cualquier elemento del mismo a partir de ciertos elementos (ya construidos y llamados elementos base), y de ciertas reglas específicas de construcción.

La definición recurrente de conjuntos contiene dos partes:

a) **Base**: se describen (por comprensión o por extensión) los elementos del conjunto que servirán para crear los otros con la ayuda de las reglas de recurrencia señaladas.

b) **Recurrencia**: se ofrece una formulación recurrente de las reglas que permiten crear un elemento a partir de elementos ya creados.

Por ejemplo, para definir el conjunto de los naturales, basta decir:

a) Base: 1 es un número natural.

b) Recurrencia: El siguiente de un número natural es un natural. Es decir, si n es natural, entonces $n + 1$ es natural.

Otro ejemplo: si queremos definir el conjunto de los enteros impares, bastará la siguiente definición:

a) Base: el entero 1 es impar.

b) Recurrencia: Sea x impar. Entonces $x + 2$ también es impar.

Demostración recurrente de propiedades sobre conjuntos definidos recurrentemente.

Por último, una vez tenemos visto como definir un conjunto mediante recurrencia, y una vez tenemos visto el método de demostración recurrente, es fácil ofrecer ahora cómo establecer y demostrar distintas propiedades sobre esos conjuntos de definición recurrente:

a) **Base**: demostrar que la propiedad P es cierta para todos los elementos de la base de definición del conjunto.

b) **Recurrencia**: La hipótesis de recurrencia consiste en suponer que los elementos que sirven para la creación de un elemento e del conjunto verifican la propiedad P. La demostración de herencia consiste en demostrar que e también verifica P, a partir de la hipótesis de inducción y de las propiedades de los elementos.

Por ejemplo, supongamos que queremos demostrar que el i –ésimo impar mayor que cero (lo llamaremos x_i) tiene el valor $2 \cdot i - 1$.

Para demostrarlo, primero verificamos que efectivamente se cumple así en los elementos de la base: el número 1. Efectivamente $x_1 = 1$ y también $2 \cdot 1 - 1 = 1$.

Y acudimos ahora a la recurrencia: Suponiendo que $x_i = 2 \cdot i - 1$, ¿es cierto que $x_{i+1} = 2 \cdot (i + 1) - 1$? Teniendo en cuenta que $x_{i+1} = x_i + 2$ (así hemos descrito, por construcción de recurrencia, el conjunto de los impares), es inmediata la demostración, sin más que sustituir el valor de x_i: $x_{i+1} = x_i + 2 = 2 \cdot i - 1 + 2 = 2 \cdot (i + 1) - 1$; sin más que sacando factor común al dos que multiplica a i y a 1.

Todo lo visto en este apartado del capítulo nos permitirá tener unas pautas para la construcción de algoritmos o programas recursivos o recurrentes.

Etapas en la construcción de algoritmos recursivos.

Un algoritmo recursivo requiere, como primera providencia, una definición más o menos matemática del problema que se está abordando. Y además esta especificación matemática del problema debe ofrecer las ecuaciones de recurrencia. Para llegar a esa descripción nos basamos en la definición recurrente del dominio.

Una vez tenemos las relaciones de recurrencia, hay que transcribir esas relaciones a notación algorítmica.

Veamos con un ejemplo cómo podemos estudiar un problema de forma recursiva y pasar por distintas etapas hasta llegar a la construcción de la correspondiente función recursiva. El ejemplo es el de **la calle adoquinada y la rayuela**:

Se considera una calle adoquinada. Cada adoquín está numerado a partir de 0:

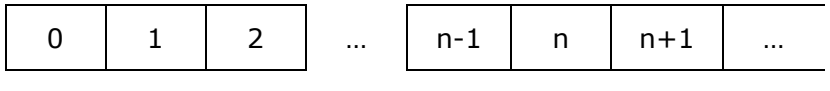


Figura 6.2.: Adoquines de la rayuela.

En esa calle hay niños jugando a la rayuela: se desplazan de un adoquín al siguiente, pero también pueden saltar un adoquín. El juego está, pues, basado en estos dos desplazamientos elementales (ver figura 6.3.)

Al principio del juego, los niños se encuentran en el adoquín 0. *El problema consiste en calcular el número de caminos posibles hasta alcanzar un adoquín n prefijado como objetivo*.

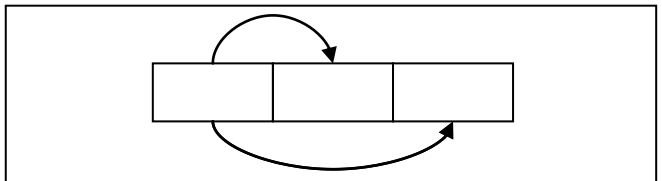


Figura 6.3.: Movimientos de la rayuela.

Para poder realizar una definición recurrente de nuestro problema, debemos seguir los siguientes pasos:

a) En primer lugar debemos establecer cuál es el **perfil matemático** de la función que estamos buscando. A esta función la llamaremos "*Caminos*", cuyo **parámetro de entrada** es un natural *n*, y la **salida** en otro natural que indica el número de caminos posibles.

b) En segundo lugar, necesitamos saber cuál será el **enunciado de la función** (enunciado en comprensión): "$Caminos(n)$ *es el número de caminos posibles que permiten llegar al adoquín* n *a partir del adoquín 0, pasando de un adoquín al siguiente, o saltando uno*."

c) El siguiente paso es determinar cuáles son los **casos más simples** en los que podemos llamar a nuestra función *Caminos*:

   Al adoquín 1 se llega de una sola manera: $Caminos(1) = 1$.

Al adoquín 2 se llega de dos maneras: $Caminos(2) = 2$.

El caso del adoquín 3 ya supone un estudio más complicado.

d) Una vez visto como se describe la función y cuáles son los casos más elementales, queda descubrir cómo **describir el caso general**. Para ello debemos basarnos en los desplazamientos elementales. Estudiamos tres adoquines consecutivos: para alcanzar el adoquín n se puede provenir del adoquín $n - 1$, o bien del adoquín $n - 2$.

Se divide el conjunto de caminos que llevan al adoquín n en dos subconjuntos disjuntos:

- Los caminos que pasan por el adoquín $n - 1$.
- Los caminos que NO pasan por el adoquín $n - 1$.

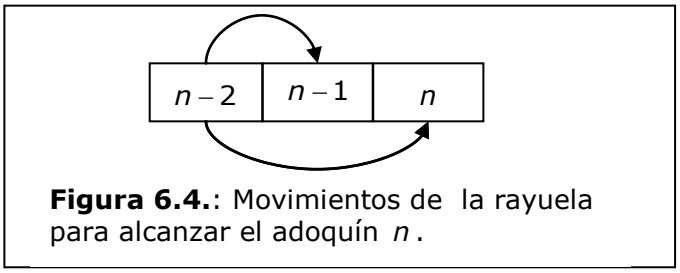


Figura 6.4.: Movimientos de la rayuela para alcanzar el adoquín n.

Todo camino a n que pasa por $n - 1$ está creado a partir de un camino que llega a $n - 1$ al que se la añade el trayecto $(n - 1, n)$. Todo camino que no pasa por $n - 1$ está creado a partir de uno que llega hasta el adoquín $n - 2$ al que se le añade el trayecto $(n - 2, n)$.

Podemos, por tanto, afirmar que los caminos que llegan desde 0 hasta n son: $Caminos(n) = Caminos(n - 1) + Caminos(n - 2)$.

e) Ya tenemos un enunciado recursivo:

$Caminos(1) = 1$.

$Caminos(2) = 2$.

$Caminos(n) = Caminos(n - 1) + Caminos(n - 2)$, $\forall n \geq 3$.

Es evidente, por exigencias del mismo enunciado del problema, que el valor de n debe ser siempre mayor que cero y que no se debe invocar a la función *Caminos* con parámetros negativos. Y así lo respeta nuestra definición recursiva, que toma una valor base no calculado para $n = 1$ y $n = 2$, y solamente acude a la definición recursiva a partir de $n = 3$, donde se invoca a la función $Caminos()$ para los valores $n - 1 = 2$ y $n - 2 = 1$, ambos mayores que cero.

Además podemos **comprobar que la recurrencia termina**: todo valor del dominio inicial provoca un cálculo finito. Vamos a demostrarlo por inducción:

Base: $Caminos(1)$ y $Caminos(2)$ terminan ya que no generan ninguna nueva llamada recursiva.

Recurrencia: Supongamos que $Caminos(n)$ (para valores de $n \geq 3$) termina (hipótesis de recurrencia). Eso implica que $Caminos(n - 1)$ y $Caminos(n - 2)$ también terminan. Entonces, la llamada $Caminos(n + 1)$ genera el cálculo de $Caminos(n)$ y de $Caminos(n - 1)$, que ambos terminan, por la hipótesis de recurrencia.

Conclusión: $Caminos(n)$ termina para $n \geq 3$.

Con todo lo hecho, hemos llegado a una correcta definición de la función o método diseñado parar calcular de forma recurrente el número de caminos posibles para llegar del adoquín cero al adoquín n.

Así, la definición de la función $Caminos()$ podría tener el siguiente aspecto:

Función $Caminos$ (n Entero) → Entero
Constantes
∅
Variables
∅
Acciones:
1. **Si** $n < 3$ **Entonces** $Caminos(n) \leftarrow n$.
Sino $Caminos(n) \leftarrow Caminos(n-1) + Caminos(n-2)$.
2. **FIN**.

Recapitulamos: las etapas seguidas para la elaboración de la función sustentada por un algoritmo recursivo pueden quedar resumidas en las siguientes:

a) **Especificación matemática**. Estudio del perfil de la función.

b) **Enunciado en comprensión** de la función. Un enunciado intuitivo que permita seguir el razonamiento.

c) Examen de los **casos particulares simples**. Entendemos por casos simples aquellos para los cuales el cálculo se formula de manera analítica. Son los casos que constituirán la base de la recurrencia.

d) Puesta de relieve de un **caso general** por descubrimiento de una **fórmula que lo relaciona con el mismo problema basado en una información más pequeña, más próxima al caso de las bases**. Este paso es el encargado de buscar la recurrencia: hay que expresar el valor de la función en un punto del dominio a partir de uno o varios puntos del dominio.

e) **Demostración de la recurrencia**. Ese paso último requiere a su vez de tres subpasos:

   e.1. Comprobación de que el valor de la función puede ser calculado correctamente en cualquier punto o valor del dominio únicamente mediante la aplicación de las relaciones facilitadas.

   e.2. Comprobación de la coherencia de las llamadas de la función recursiva.

   e.3. Comprobación de la terminación de la función.

Árbol de recursión.

Un árbol de recursión sirve para representar las sucesivas llamadas recursivas que un programa recursivo puede generar. Es un concepto

simple, pero de gran utilidad práctica para analizar el comportamiento de los algoritmos recursivos diseñados.

Para construir un árbol, se representa cada llamada a la función como un nodo (que dibujaremos en forma de un círculo). Arriba se dibuja el nodo inicial con el que es invocada por primera vez la función recurrente. Más allá (hacia abajo) se van insertando las llamadas que recibe la función recursivamente. En cada nodo del árbol (que se corresponde con una llamada) se le adjunta una etiqueta con el valor o valores de los parámetros de entrada.

Por ejemplo, el árbol de recursión de la función Factorial tiene el aspecto como el recogido en la figura 6.5.

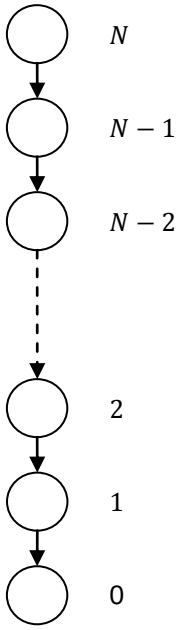


Figura 6.5.: Árbol de recursión de la función Factorial.

Para la rayuela, el árbol queda recogido en la Figura 6.6. Hemos tomado el caso en que $N = 5$.

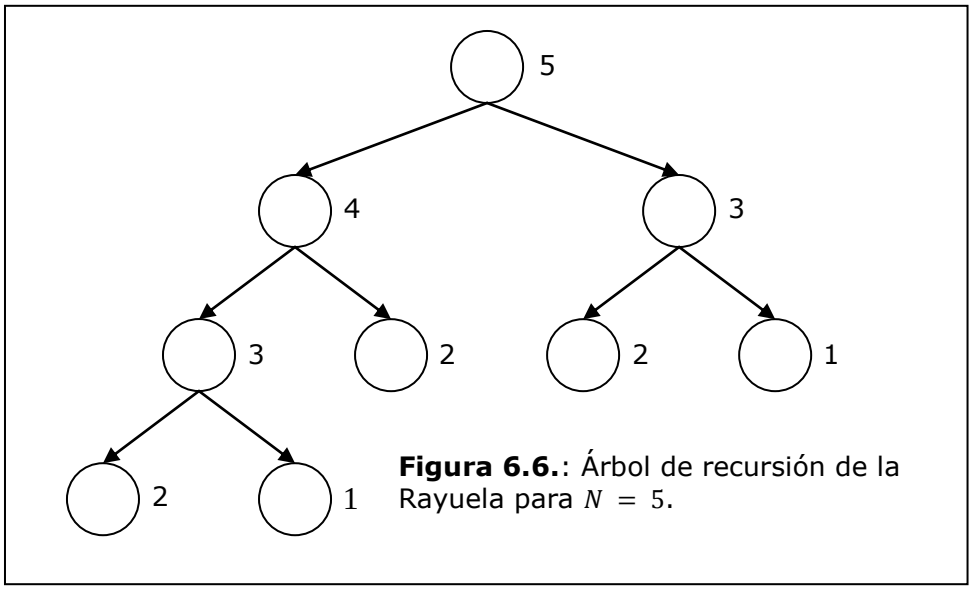


Figura 6.6.: Árbol de recursión de la Rayuela para $N = 5$.

Recursión e iteración.

El concepto de iteración ya quedó visto en el capítulo 4, al hablar de los algoritmos y de sus modos de representación. Y al inicio de éste capítulo hemos visto que todos los algoritmos que en el capítulo 4 resolvimos mediante la iteración, también pueden ser solventados mediante la recursividad o recurrencia.

La cuestión a plantear es entonces: ¿qué es mejor: recursividad, o estructuras iterativas? Para responder a esta cuestión, un factor importante que cabe plantearse es cuál de las dos formas requiere menor espacio de almacenamiento en memoria. Y otro, cuál de las dos formas resuelve el problema con menor tiempo.

Veamos el algoritmo del cálculo del factorial, expresado como función recurrente (**Función** FactorialR()) y expresado como función con estructura iterativa (**Función** FactorialI())

Función $FactorialR$ (N Entero) → Entero
Constantes
∅
Variables
∅

Acciones:
1. Si $N = 0$ Entonces $FactorialR(0) \leftarrow 1$.
2. Sino, Entonces $FactorialR(N) \leftarrow N \cdot FactorialR(N-1)$.
3. **FIN**.

Función $FacorialI$ (N Entero) → Entero.
Constantes
∅
Variables
Aux, I, Enteros
Acciones:
1. **Si** $N = 0$, **Entonces** $FactorialI(0) = 1$.
   Sino, **Entonces**:
   1.1. $aux \leftarrow 1$.
   1.2. **Para** $I \leftarrow 1$ **Hasta** N **Repetir**: $aux \leftarrow aux \times I$.
   1.3. $FactorialI \leftarrow aux$.
2. **Fin**.

Aparentemente, parece que la función recursiva exige menos memoria, pues de hecho no requiere el uso de ninguna variable local (es decir, variable propia de la función) mientras que la función iterativa requiere de dos variables.

Pero en realidad el programa recursivo va almacenando en una pila de memoria los N números: N, $N-1$, $N-2$, … 2, 1, que son los parámetros de llamada antes de cada recursión. A medida que vaya saliendo de las sucesivas llamadas, multiplicará esos números en el mismo orden en que lo hace el programa iterativo. Así pues, el programa recurrente requiere de mayor almacenamiento. Además, también tardará más tiempo en ejecutarse, porque debe almacenar en memoria y recuperar luego de ella todos los números, realizar todas las multiplicaciones, y realizar las distintas llamadas a la función.

Otro ejemplo para estudiar el comportamiento de una función definida de forma recursiva o de forma iterativa lo tenemos con los números de Fibonacci. Mostramos de nuevo la definición recursiva (**Función** FibonacciR()) y la definición iterativa (**Función** FibonacciI()).

Función $FibonacciR$ (N Entero) → Entero
Constantes
∅
Variables

∅

Acciones:

1. **Si** $N \leq 2$ **Entonces** $FibonacciR(N) \leftarrow 1$.
   Si no, **Entonces**,
   $FibonacciR(N) \leftarrow FibonacciR(N-1) + FibonacciR(N-2)$.
2. **Fin**.

Función $FibonacciI$ (N Entero) → Entero

Constantes

∅

Variables

A, B, C e I, Enteros.

Acciones:

1. **Si** $N \leq 2$ **Entonces** $FibonacciR(N) \leftarrow 1$.
2. **Si no**, **Entonces**,
   2.1. $A \leftarrow 1$.
   2.2. $B \leftarrow 1$
   2.3. **Para** $I \leftarrow 3$ **Hasta** N, **Hacer**:
      2.3.1. $C \leftarrow A + B$
      2.3.2. $A \leftarrow B$
      2.3.3. $B \leftarrow C$
   2.4. $FibonacciI \leftarrow C$
3. **FIN**.

El árbol de recursividad para la función $FibonacciR()$, para $N = 5$ queda de forma idéntica al árbol antes presentado en la Figura 6.6. para el problema de la Rayuela. Y así observamos que para el cálculo del valor de $FibonacciR(5)$ hemos calculado una vez el valor de $FibonacciR(4)$, dos veces el valor de $FibonacciR(3)$, tres veces el de $FibonacciR(2)$ y dos veces el de $FibonacciR(1)$. Así pues, con el árbol de recursión vemos que el programa recursivo repite innecesariamente los mismos cálculos una y otra vez. Y el tiempo que tarda la función recursiva en calcular $FibonacciR(N)$ aumenta de forma exponencial a medida que aumenta N.

En cambio, la función iterativa usa un tiempo de orden lineal (aumenta linealmente con el incremento del valor de N), de forma que la diferencia de tiempos entre una y otra forma de implementación es notable.

La recursión siempre puede reemplazarse con la iteración y las pilas de almacenamiento de datos. La pregunta es, entonces: ¿Cuándo usamos recursión, y cuándo usamos iteración?

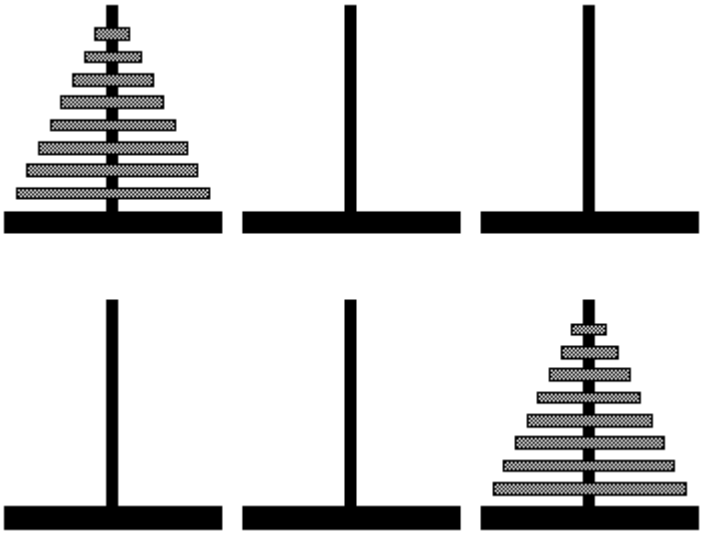
Si el árbol de recursión tiene muchas ramas, con poca duplicación de tareas, entonces el método más adecuado es la recursión, siempre que además el programa resultante sea más claro, sencillo y fácil de obtener.

Los algoritmos recursivos son muy apropiados cuando el problema a resolver, la función a calcular o la estructura de datos a procesar están definidos en términos recursivos. La recursión debe tenerse en cuenta como una técnica que nos permite diseñar algoritmos de una forma sencilla, clara y elegante. No se debe utilizar por tanto cuando dificulte la comprensión el algoritmo.

Ejercicio: las torres de Hanoi.

Dice la leyenda que, al crear el mundo, Dios situó sobre la Tierra tres varillas de diamante y sesenta y cuatro discos de oro. Los discos eran todos de diferente tamaño, e inicialmente fueron colocados en orden decreciente de diámetros sobre la primera de las varillas. También creó Dios un monasterio cuyos monjes tenían la tarea de trasladar todos los discos desde la primera varilla a la tercera. La única operación permitida a los monjes para llevar a cabo su tarea era mover un único disco cada vez, de una varilla a otra cualquiera, con la condición de que no se podía situar encima de un disco otro de diámetro mayor. La leyenda decía también que cuando los monjes terminasen su tarea, el mundo llegaría a su fin.

La leyenda no es cierta: es fruto de la imaginación de un matemático del siglo XVIII, que inventó el juego de las Torres de Hanoi (así se llama) y que diseñó así su propia campaña de publicidad y marketing que le permitiera lanzar su invento al mercado. La invención resultó, sin embargo, efectiva: ha perdurado hasta nuestros días, el juego es conocido en todo el mundo,… y nos ha dejado una pregunta: si la leyenda fuera cierta… ¿cuándo sería el fin del mundo?



El mínimo número de movimientos que se necesitan para resolver este problema es de $2^{64} - 1$. Si los monjes hicieran un movimiento por segundo, y no parasen en ningún instante hasta completar la total ejecución de la tarea de traslado, los discos estarían en la tercera varilla en poco menos de 585 mil millones de años. Teniendo en cuenta que la tierra tiene sólo unos cinco mil millones, o que el universo está entre quince y veinte mil millones de años, está claro que o los monjes se dan un poco más de prisa, o tenemos universo y mundo para rato.

Se puede generalizar el problema de Hanoi variando el número de discos: desde un mínimo de 3 hasta el máximo que se quiera.

El problema de Hanoi es curioso, y su formulación elegante. Y su solución es muy rápida de calcular. La pega no está en la complicación del procedimiento (que no es complicado), sino en que a medida que aumenta el número de discos crece exponencialmente el número de pasos.

Existe una versión recursiva eficiente para hallar el problema de las torres de Hanoi para pocos discos; pero el coste de tiempo y de memoria se incrementa excesivamente al aumentar el número de discos.

La solución recursiva es llamativamente sencilla de implementar y de construir. Y de una notable belleza, creo yo.

Si numeramos los discos desde 1 hasta N, y si llamamos X al poste donde quedan inicialmente colocado los discos, Z al poste de destino de los discos, e Y el tercer poste intermedio, el algoritmo recurrente que resuelve el problema de las torres de Hanoi quedaría de la siguiente manera:

1. **Si** N = 1, **Entonces** Trasladar disco de X a Z.
   Sino
   1.1. Trasladar 1 .. N – 1 de X z Y, (auxiliar: Z).
   1.2. Trasladar N de X a Z.
   1.3. Trasladar 1 .. N de Y a Z (auxiliar: X)
2. **Fin**.

Podemos definir una función que muestre por pantalla los movimientos necesarios para hacer el traslado de la torre de un poste a otro. A esa función la llamaremos $Hanoi$, y recibe como parámetros los siguientes valores:

a) una variable entera que indica el disco más pequeño de la pila que se quiere trasladar.

b) una variable entera que indica el disco más grande de la pila que se desea trasladar. (Hay que tener en cuenta que con el algoritmo que hemos definido siempre se trasladan discos de tamaños consecutivos.)

c) Una variable que indique el poste donde están los discos que trasladamos.

d) Una variable que indica el poste a donde se dirigen los discos.

Un modo de **identificar los postes** y de tener en todo momento identificado el poste auxiliar con variables enteras es el siguiente: Llamaremos al poste origen poste i, y al poste destino poste j. Y exigimos a las dos variables i, j las siguientes restricciones: $i, j \in [1..3]$, e $i \neq j$.

Y entonces siempre tendremos que el poste auxiliar vendrá siempre definido con la expresión $6 - i - j$, como puede verse en la Tabla 6.2.

i	j	$6 - i - j$
1	2	3
1	3	2
2	1	3
2	3	1
3	1	2
3	2	1

Tabla 6.2.: Postes de las torres de Hanoi.

La función $Hanoi$ queda entonces de la siguiente forma:

Función $Hanoi$ ($D1$, $D2$, i, j, Enteros)
Constantes
∅
Variables
∅
Acciones:
1. **Si** $D1 = D2$, **Entonces** [Mostrar] Mover disco D1 de i a j.
2. **Sino**
   2.1. $Hanoi(D1, D2 - 1, i, 6 - i - j)$.
   2.2. $Hanoi(D2, D2, i, j)$.
   2.3. $Hanoi(D1, D2 - 1, 6 - i - j, j)$.
3. **FIN**.

La forma que adquiere esta función implementada en C es tan sencilla como lo que se muestra a continuación:

```
typedef unsigned short int usi;
void Hanoi(usi D1, usi D2, usi i, usi j)
{
      if(D1 == D2)
            printf("Disco %hu de %hu a %hu\n",D1, i, j);
      else
      {
            Hanoi(D1, D2 - 1, i, 6 - i - j);
            Hanoi(D2, D2, i, j);
            Hanoi(D1, D2 - 1, 6 - i - j, j);
      }
}
```

Y si se invoca esta función inicialmente con la sentencia

```
Hanoi(1, discos, 1, 3);
```

donde discos es el número de discos que tiene la torre de Hanoi, la ejecución de esta función muestra todos los movimientos que debe hacerse con los anillos para pasarlos desde el poste 1 al poste 3.

Por ejemplo, si tomamos el valor *discos = 4* tendremos la siguiente salida:

```
Disco 1 de 1 a 2
Disco 2 de 1 a 3
Disco 1 de 2 a 3
Disco 3 de 1 a 2
Disco 1 de 3 a 1
Disco 2 de 3 a 2
Disco 1 de 1 a 2
Disco 4 de 1 a 3
Disco 1 de 2 a 3
Disco 2 de 2 a 1
Disco 1 de 3 a 1
Disco 3 de 2 a 3
Disco 1 de 1 a 2
Disco 2 de 1 a 3
Disco 1 de 2 a 3
```

En lenguaje Java, una aplicación que desarrolla completamente este algoritmo podría tener la siguiente forma:

```
public class TorresHanoi {

      public static void Hanoi(int D1, int D2, int i, int j)
      {
            if(D1 == D2)
            {
                  System.out.print("Disco " + D1 + " de ");
                  System.out.println(i + " a "+ j);
            }
            else
            {
                  Hanoi(D1 , D2 - 1 , i , 6 - i - j);
                  Hanoi(D2 , D2 , i , j);
                  Hanoi(D1 , D2 - 1 , 6 - i - j , j);
            }
      }

      public static void main(String[] args) {
            int discos;
            System.out.println("número de discos ... ");
            discos = Teclado.readInt();
```

```
            Hanoi(1, discos, 1, 3);
        }
}
```

Donde la clase *Teclado* está definida para dar entrada valores desde el teclado.

Para terminar esta presentación de las torres de Hanoi queda pendiente una cosa: demostrar que, efectivamente, el número de movimientos para desplazar todos los discos de un poste a otro con las condiciones impuestas en el juego es igual a dos elevado al número de discos, menos uno.

Eso se demuestra fácilmente por inducción (como no):

Base: Se verifica para $N = 1$: $M(1) = 2^{N-1} - 1 = 1$: efectivamente, el mínimo número de movimientos es uno.

Recurrencia: Supongamos que se cumple para N: $M(N) = 2^N - 1$. Entonces si tenemos $N + 1$ discos, lo que hacemos es desplazar dos veces los N primeros discos y entre medias una vez el disco $N + 1$, es decir, tenemos $M(N + 1) = 2 \cdot M(N) + 1 = 2 \cdot (2^N - 1) + 1 = 2^{N+1} - 1$ (c.q.d.)

Recapitulación.

Hemos visto el modelo recursivo para plantear y solventar problemas. Esta forma de trabajo es especialmente útil cuando pretendemos solventar cuestiones basadas en propiedades definidas sobre el conjunto de enteros o un conjunto numerable donde se pueda establecer una correspondencia entre los enteros y los elementos de ese conjunto.

Hemos visto el modo en que se definen por recurrencia una serie de conjuntos, cómo se demuestran propiedades para estos conjuntos basándonos en la demostración por inducción, y cómo se llega así a replantear muchos problemas, alcanzando una formulación muy elegante y sencilla: la forma recursiva.

La recursividad ofrece habitualmente algoritmos muy sencillos de implementar y de interpretar. La principal limitación que presenta esta forma de resolver problemas es que habitualmente supone un elevado coste de tiempo de computación y de capacidad de almacenamiento de memoria.

Hemos descrito el camino a seguir para alcanzar una solución recurrente ante un problema que nos planteamos. Los ejercicios que planteamos a continuación pueden servir para seguir profundizando en ese itinerario por el que se llega a las definiciones recurrentes.

Ejercicios.

- ***Definir la operación potencia de forma recursiva. Escribir el pseudocódigo de una función que pudiera realizar ese cálculo.***
- ***Definir la operación suma de enteros de forma recursiva. Téngase en cuenta, por ejemplo, que si pretendemos sumar dos enteros a y b, podemos tomar: $suma(a,b) = a$ si b es igual a cero; y $suma(a,b)$ es $1 + suma(a,b-1)$ en otro caso. Escribir el pseudocódigo de una función que pudiera realizar esa operación.***

  Definir la operación producto de forma recursiva. Téngase en cuenta, por ejemplo, que si pretendemos multiplicar dos enteros a y b, podemos tomar: $producto(a,b) = 0$ si b es igual a cero; y $producto(a,b)$ es $a + producto(a,b-1)$ en otro caso. Escribir el pseudocódigo de una función que pudiera realizar ese cálculo.
- ***Definir una función que devuelva el mayor de una lista de números. El algoritmo no puede comparar en ningún momento más de dos números para decidir el mayor de entre todos los de la lista.***

  Vamos a definir una función, que llamaremos sencillamente M, y que recibe como parámetros una lista de enteros y un valor entero que

indica la cantidad de elementos de esa lista. La función devuelve un entero: el mayor de entre todos los valores de la lista.

La limitación impuesta en el enunciado nos indica la **base** de nuestro algoritmo de recurrencia: Si la lista tiene un solo elemento, ese elemento es el que devuelve la función; si la lista tiene dos elementos, la función devuelve el mayor de ellos. Si la lista tiene más de dos elementos, entonces la función no puede contestar a la cuestión, porque en ningún momento podemos comparar más de dos elementos.

Supongamos que tenemos n elementos en la lista. Entonces está claro que el mayor de entre ellos es igual al mayor entre el mayor de los $n/2$ primeros y el mayor entre los $n/2$ segundos. Esta es la base para definir nuestro proceso de **recurrencia**:

Podemos entonces plantear el algoritmo de la siguiente manera:

Función M ($x_1, \dots, x_n$ Enteros, n Entero) → Entero
Constantes
∅
Variables
M, x e y, Enteros.
Acciones:
1. **Si** n = 1 **Entonces** $M \leftarrow x_1$.
Si no, **Entonces**
1.1. $m = \lfloor n/2 \rfloor$.
1.2. $x \leftarrow M(x_1, \dots, x_m, m)$.
1.3. $y \leftarrow M(x_{m+1}, \dots, x_n, n-m)$.
1.4. **Si** $x > y$ **Entonces** $M \leftarrow x$.
Si no [es decir, $x \leq y$] **Entonces** $M \leftarrow y$.
2. **Fin**.

Que en código C (o en Java: tiene el mismo aspecto) queda una función como la que sigue:

```
short M(short *v, short n)
{
	short m;
	short x, y;

	if(n == 1) return *v;
	m = n / 2;
	x = M(v , m);
	y = M(v + m , n - m);
```

```
        return x > y ? x: y;
}
```

Y la llamada a esta función o al método es de la forma *M(v, dim);* donde *v* es una variable de tipo vector o array de enteros y donde *dim* es el número de elementos del vector *v*. En Java no sería necesario el paso de este segundo parámetro, que es igual a *v.length*. (Como ya se ha dicho en capítulos anteriores, el código aquí recogido no es parte sustancial del contenido de este manual: no es menester en estos momentos comprender todo lo referente al código en Java o en C: se presenta aquí para mostrar la forma que tiene un programa que implementa nuestros algoritmos.)

- ***Definir una función que muestre, mediante un proceso recurrente, el código binario de un entero dado. Convendrá recordar lo explicado en capítulos anteriores sobre codificación binaria.***

  Para encontrar este proceso es sencillo buscar los casos más sencillos, que forman la **Base** de la recurrencia: El código binario de $(0)_{10}$ es 0; el código binario de $(1)_{10}$ es 1.

  Para el proceso de recurrencia baste con decir que si el número es mayor que 1, entonces ya necesita de al menos dos dígitos binarios; y si el número es mayor que 3 entonces requiere ya de al menos tres dígitos. Y, en general, si el número es mayor que $2^n - 1$ entonces el número requiere, para su codificación binaria, de al menos n dígitos.

  Y, por lo indicado en los capítulos sobre codificación numérica y sobre codificación interna de la información, sabemos que podemos concatenar dígitos a fuerza de dividir por dos sucesivas veces hasta llegar al valor 0 ó 1.

  Podemos entonces definir el siguiente algoritmo recursivo para obtener el código binario de un entero:

  Función $Binario$ ($n > 0$ Entero en base 10) → Muestra Código binario
  Constantes
  ∅

Variables

Ø

Acciones:

1. **Si** $n = 1$ **Entonces** [Mostrar dígito] **Mostrar** n.
   Si no [Es decir, $n > 1$] **Entonces**
   1.1. $Binario(n/2)$.
   1.2. [Mostrar digito] **Mostrar** $n \, mod \, 2$.
2. **Fin**.

Cuyo código en C tiene el siguiente aspecto:

```
void Binario(unsigned long n)
{
      if(n == 1)
         printf("%ld",n);      // Java: System.out.print(n);
      else
      {
         Binario(n/2);
         printf("%ld",n % 2); // Java: System.out.print(n%2);
      }
}
```