



Escuela Superior de Ingeniería Informática

DISEÑO Y ANÁLISIS DE ALGORITMOS

Metodología para el desarrollo de algoritmos recursivos finales (por cola)

Índice

1. Introducción	2
2. Metodología para diseñar algoritmos recursivos por cola	3
2.1. Primer ejemplo: factorial	3
2.2. Descripción formal de la metodología	5
2.3. La función factorial, paso a paso	5
3. Más ejemplos	7
3.1. Suma de los primeros x números naturales	7
3.2. Multiplicación mediante sumas repetidas	8
3.3. Determinar si un número natural contiene un dígito impar	8
3.4. Suma de los elementos de un vector	10
4. Ejemplos más complicados	11
4.1. Elección de una generalización adecuada: factorial	11
4.2. La inclusión de varios parámetros adicionales: cambio de base	12
4.3. Método con funciones que admiten recursividad múltiple: números de Fibonacci	13
4.4. Metodología aplicada cuando se desconoce una fórmula para la función: paridad de un número	16

```

int fact_lineal(int x){
    if(x==0)
        return 1;
    else
        return x*fact_lineal(x-1);
}

int f_final(int x, int y){
    if(n==0)
        return y;
    else
        return f_final(x-1,x*y);
}

int fact_final(int x){
    return f_final(x,1);
}

int fact_iterativo(int x){
    int y=1;
    while(x>0){
        y = y*x;
        x = x-1;
    }
    return y;
}

```

fact_lineal(4)

x	resultado
4	24
3	6
2	2
1	1
0	1

(a)

f_final(4,1)

x	y	resultado
4	1	24
3	4	24
2	12	24
1	24	24
0	24	24

(b)

fact_iterativo(4)

x	y	resultado
4	1	-
3	4	-
2	12	-
1	24	-
0	24	24

(c)

Figura 1: Función factorial. Implementación recursiva lineal no-final (a), final o por cola (b), e iterativa (c).

1. Introducción

Existe una relación estrecha entre la recursividad final (también denominada “por cola”) y la iteración. Resulta relativamente sencillo transformar un algoritmo recursivo final a su equivalente iterativo, y esto lo aprovechan algunos compiladores para generar código algo más eficiente.

A la hora de diseñar algoritmos recursivos por cola, la estrecha relación con la iteración también es clara. Al ser habitual el uso de determinados parámetros como variables acumuladoras, la forma de pensar sobre estos algoritmos está asociada a los valores que van tomando las variables, y en la secuencia en la que se llevan a cabo las instrucciones del programa. En otras palabras, el diseño de algoritmos recursivos por cola puede estar más relacionado con el paradigma imperativo (el cual describe la programación en términos del estado del programa y sentencias que cambian dicho estado), que en el declarativo o funcional (donde las sentencias principalmente describen el problema que se quiere solucionar).

Esto se ilustra en la Fig. 1, donde se puede apreciar como el parámetro y de la función `f_final` va acumulando un resultado parcial, y realiza el mismo propósito que la variable `y` en la función `fact_iterativo`. Por tanto, si analizamos los valores de `x` e `y` en ambos algoritmos tras realizar una llamada recursiva o una iteración del bucle, éstos coinciden, como se aprecia en las respectivas tablas.

2. Metodología para diseñar algoritmos recursivos por cola

A continuación se describe una metodología para desarrollar algoritmos recursivos por cola utilizando:

- El paradigma declarativo
- El concepto de generalización de funciones

El método pretende que el diseñador evite pensar de manera imperativa.

2.1. Primer ejemplo: factorial

El siguiente apartado describe formalmente la metodología. No obstante, empezamos viendo un primer ejemplo. Considérese la función factorial:

$$g(x) = x! \quad g(0) = 1$$

Una definición recursiva natural sería $g(x) = x \cdot g(x - 1)$. Sin embargo, ésta conduciría a un algoritmo recursivo no-final, dado el producto que aparece en la fórmula.

Por tanto, para usar la recursividad por cola es necesario definir, en primer lugar, una nueva función recursiva $f(x, y)$, **añadiendo un nuevo parámetro y** . La función f , por tanto, se puede ver como una **generalización** de g , que tendrá su propia expresión. Existen varias generalizaciones posibles, por ejemplo:

$$y \cdot x! \quad \frac{x!}{y} \quad (x!)^y \quad y + x!$$

Donde sería posible recuperar el valor del factorial de x tomando $y = 1$ en las tres primeras generalizaciones, e $y = 0$ en la última. Naturalmente es conveniente que vuelva a aparecer $x!$ en la expresión, y es necesario incorporar y de alguna manera. En este caso, como el factorial involucra productos podemos elegir la primera, por lo que consideramos:

$$f(x, y) = y \cdot x! \tag{1}$$

A continuación, para aplicar la recursividad, es necesario considerar uno o varios **casos base**, y una **simplificación de la función hacia dichos casos base**. El caso trivial natural para la función factorial es $0! = 1$. Por tanto, según (1) tenemos como caso base:

$$f(0, y) = y \tag{2}$$

Por otro lado, nos acercáramos al caso base restando una unidad al parámetro x en la expresión recursiva. Por tanto, la simplificación de $f(x, y)$ hacia el caso base sería $f(x - 1, z)$, donde ahora **desconocemos el valor del segundo parámetro**.

Finalmente, al tratarse de una función recursiva por cola, la expresión recursiva consiste en llamar de nuevo a la propia función con otros parámetros, sin realizar ninguna operación adicional. Es decir, tendríamos:

$$f(x, y) = f(x - 1, z) \tag{3}$$

Aplicando la definición de nuestra función general (1) tenemos:

$$f(x, y) = y \cdot x! = z \cdot (x - 1)! = f(x - 1, z)$$

de donde podemos despejar el valor que debería tener el parámetro adicional z de la igualdad central, que en este caso es:

$$z = x \cdot y \tag{4}$$

Este proceso se puede explicar mediante el siguiente diagrama gráfico:

$f(x, y)$	→	$y \cdot x!$	
			⇒ $z = y \cdot x$
$f(x - 1, z)$	→	$z \cdot (x - 1)!$	

Por último, sustituyendo (4) en (3), y considerando el caso base (2), la función recursiva por cola resultante es

$$f(x, y) = \begin{cases} y & \text{si } x = 0 \\ f(x - 1, x \cdot y) & \text{si } x > 0 \end{cases}$$

donde $x! = g(x) = f(x, 1)$, que es el algoritmo descrito en la Fig. 1(b), y que resulta fácil pasarlo a una versión iterativa (c). Es importante ver que el parámetro y adicional aparece como consecuencia de aplicar el concepto de generalización, y no como una variable acumuladora.

2.2. Descripción formal de la metodología

La metodología para diseñar una función recursiva final o por cola f , que es una generalización de alguna función g , consiste en los siguientes pasos:

1. Obtener una fórmula matemática o lógica de $g(X)$, donde X es un conjunto de parámetros. Por otro lado, sea $Y = \emptyset$.
2. Añadir un parámetro al conjunto Y , y proponer una nueva función $f(X, Y)$, que sea una generalización de $g(X)$, incorporando los parámetros de Y a la fórmula de para g .
3. Establecer los casos base para la función f , en función de los casos base de g .
4. Escoger una simplificación \tilde{X} de X hacia los casos base de g .
5. Formar la ecuación $f(X, Y) = f(\tilde{X}, Z)$, donde Z reemplaza a Y y es desconocida.
6. Despejar el valor de Z para que se cumpla la igualdad. Si no es posible encontrar dicho Z volver a los pasos 1, 2, ó 3.
7. Habiendo obtenido Z , definir la función recursiva final incorporando los casos base y el recursivo.
8. Establecer los valores de los parámetros en \hat{Y} de tal forma que $g(X) = f(X, \hat{Y})$.

2.3. La función factorial, paso a paso

A continuación se presenta el uso de la metodología tal y como se ha descrito en el punto 2.2 con la función factorial:

1. La función factorial sólo tiene un parámetro, con lo cual $X = \{x\}$. Por otro lado. $Y = \emptyset$. La función g factorial puede definirse matemáticamente de varias maneras:

$$g(x) = x!$$

o

$$g(x) = \prod_{i=1}^x i!$$

o simplemente:

$$g(x) = x \cdot (x - 1) \cdot \dots \cdot 2 \cdot 1$$

Podríamos elegir cualquiera de las tres. La primera es compacta y clara, con lo cual la escogemos. En este momento no es necesario tener en cuenta la definición de los casos base.

2. Añadimos un nuevo parámetro y al conjunto Y , y proponemos una generalización de $g(X)$, añadiendo y a la fórmula:

$$f(X, Y) = f(x, y) = y \cdot x!$$

3. El caso base para $g(x)$ es $g(0) = 1$, es decir, cuando $x = 0$, con lo cual el caso base para f es

$$f(0, y) = y$$

4. Escogemos una simplificación \tilde{X} de X hacia los casos base de g . En este caso, el único parámetro de X es x , donde tomando el valor $x - 1$ nos acercamos al caso base.

5. Formamos la ecuación $f(X, Y) = f(\tilde{X}, Z)$:

$$f(x, y) = y \cdot x! = z \cdot (x - 1)! = f(x - 1, z)$$

6. Despejamos el valor de z

$$z = x \cdot y$$

Todo este proceso se puede visualizar de la siguiente manera:

$f(x, y)$	\longrightarrow	$y \cdot x!$	
			$\Rightarrow z = y \cdot x$
$f(x - 1, z)$	\longrightarrow	$z \cdot (x - 1)!$	

Como hemos podido despejar z continuamos al siguiente punto.

7. Definimos la función recursiva final incorporando los casos base y el recursivo:

$$f(x, y) = \begin{cases} y & \text{si } x = 0 \\ f(x - 1, x \cdot y) & \text{si } x > 0 \end{cases}$$

8. Finalmente, establecemos el valor del parámetro introducido \hat{y} en la función general para poder calcular el factorial. En este caso, $\hat{y} = 1$, quedando:

$$x! = g(x) = f(x, 1)$$

3. Más ejemplos

En esta sección se ilustra el uso de la metodología para diseñar funciones recursivas finales que tradicionalmente se cubren en cursos de programación.

3.1. Suma de los primeros x números naturales

Este ejemplo es muy similar al del factorial, donde aparece una generalización diferente. En este caso, $g(x)$ se puede escribir como

$$g(x) = \sum_{i=1}^x i$$

Como la fórmula contiene un sumatorio elegimos la siguiente generalización:

$$f(x, y) = y + \sum_{i=1}^x i$$

El caso base para este ejemplo es $g(0) = 0$. Por tanto, la simplificación hacia el caso base debe reducir argumento. De esta manera, podemos decrementar el valor de x en una unidad ($x - 1$), y la llamada recursiva sería $f(x, y) = f(x - 1, z)$ con parámetro z desconocido. El proceso para hallar la expresión recursiva es:

$$\begin{array}{rcl} f(x, y) & \longrightarrow & y + \sum_{i=1}^x i \\ \parallel & & \parallel \\ f(x - 1, z) & \longrightarrow & z + \sum_{i=1}^{x-1} i \end{array} \quad \Rightarrow z = y + x$$

Donde es fácil despejar el valor de $z = y + x$. Finalmente, junto con el caso base el algoritmo recursivo final es:

$$f(x, y) = \begin{cases} y & \text{si } x = 0 \\ f(x - 1, x + y) & \text{si } x > 0 \end{cases}$$

donde $\sum_{i=1}^x i = g(x) = f(x, 0)$.

3.2. Multiplicación mediante sumas repetidas

Considérese $g(a, b) = a \cdot b = \sum_{i=1}^b a$, donde $b \in \mathbb{N}$. Aquí la función también contiene un sumatorio, por lo que podemos elegir como generalización $f(a, b, y) = y + \sum_{i=1}^b a$. Una simplificación natural hacia el caso base consiste en decrementar el valor de b en una unidad. Por tanto, el proceso es:

$$\begin{array}{ccc} f(a, b, y) & \longrightarrow & y + \sum_{i=1}^b a \\ \parallel & & \parallel \\ f(a, b-1, z) & \longrightarrow & y + \sum_{i=1}^{b-1} a \end{array} \Rightarrow z = y + a$$

y el algoritmo se puede obtener fácilmente incorporando el caso base:

$$f(a, b, y) = \begin{cases} y & \text{si } b = 0 \\ f(a, b-1, y+a) & \text{si } b > 0 \end{cases}$$

donde $a \cdot b = g(a, b) = f(a, b, 0)$

3.3. Determinar si un número natural contiene un dígito impar

Considérese un número natural x expresado en base 10, es decir, $x = (d_m d_{m-1} \cdots d_1 d_0)_{10} = \sum_{i=0}^m d_i \cdot 10^i$, donde $d_m \neq 0$ (evidentemente, el número de dígitos en base 10 de x es $m+1$). Además, d_i representa al dígito i -ésimo en base 10, donde d_0 corresponde a las unidades (es el menos significativo). Por otro lado, sean *div* y *mod* operaciones que calculan el cociente y el resto, respectivamente, de la división entera. El primer paso consiste en hallar una expresión para la función:

$$g(x) = \bigvee_{i=0}^m \text{odd}(d_i) = \text{odd}(d_m) \vee \text{odd}(d_{m-1}) \vee \cdots \vee \text{odd}(d_0)$$

donde $\text{odd}(d)$ es la función Booleana que se evalúa como cierta si el dígito d es impar.

La función general podría ser:

$$f(x, y) = y \vee \bigvee_{i=0}^m \text{odd}(d_i)$$

donde y es una variable Booleana. Podemos definir el caso base cuando x contiene un dígito, aunque para esta función podemos asumir que el caso base es $g(0) = \text{FALSE}$. Por tanto, la simplificación natural consiste en eliminar

algún dígito del parámetro original. Como resulta mucho más sencillo eliminar el dígito menos significativo (realizando una división entera por 10) que el más significativo, elegimos la simplificación $x \text{ div } 10$. De esta manera, además, podemos usar la operación $x \text{ mod } 10$ para recuperar el valor del dígito menos significativo. Finalmente, tenemos el siguiente proceso:

$$\begin{array}{ccc} f(x, y) & \longrightarrow & y \vee \bigvee_{i=0}^m \text{odd}(d_i) \\ \parallel & & \parallel \\ f(x \text{ div } 10, z) & \longrightarrow & z \vee \bigvee_{i=0}^{m'} \text{odd}(d'_i) \end{array} \quad \Rightarrow z = ??$$

Donde $x \text{ div } 10 = \sum_{i=0}^{m'} d'_i \cdot 10^i$, con $d'_{m'} \neq 0$. En este momento, es difícil despejar z ya que no hemos definido la correspondencia entre los dígitos d y d' , ni entre el número de dígitos m y m' . Sin embargo, es fácil ver que:

$$m - 1 = m' \quad \text{y} \quad d_{i+1} = d'_i$$

Con lo cual, sustituyendo obtenemos:

$$\begin{array}{ccc} f(x, y) & \longrightarrow & y \vee \bigvee_{i=0}^m \text{odd}(d_i) \\ \parallel & & \parallel \\ f(x \text{ div } 10, z) & \longrightarrow & z \vee \bigvee_{i=1}^m \text{odd}(d_i) \end{array} \quad \Rightarrow z = y \vee \text{odd}(d_0)$$

Por lo que el algoritmo final es:

$$f(x, y) = \begin{cases} y & \text{si } x = 0 \\ f(x \text{ div } 10, y \vee \text{odd}(x \text{ mod } 10)) & \text{si } x > 0 \end{cases}$$

donde $g(x) = f(x, FALSE)$.

3.4. Suma de los elementos de un vector

En este ejemplo parte de la dificultad reside en especificar la función que queremos desarrollar. Dado un vector v (por ejemplo, de números reales) de longitud n , la función debe admitir ambos parámetros:

$$g(v, n) = \sum_{i=1}^n v[i]$$

La fórmula contiene un sumatorio, con lo cual parece razonable utilizar la siguiente generalización:

$$f(v, n, a) = a + \sum_{i=1}^n v[i]$$

Como casos base podemos considerar $g(v, 0) = 0$ o $g(v, 1) = v[1]$. Por tanto, tiene sentido que el parámetro asociado a la longitud del vector se decremente. Así, tenemos:

$$\begin{array}{lcl} f(v, n, a) & \longrightarrow & a + \sum_{i=1}^n v[i] \\ \parallel & & \parallel \\ f(v, n-1, z) & \longrightarrow & z + \sum_{i=1}^{n-1} v[i] \end{array} \Rightarrow z = a + v[n]$$

de donde resulta sencillo despejar el valor de z . El algoritmo final es:

$$f(v, n, a) = \begin{cases} a & \text{si } n = 0 \\ f(v, n-1, a + v[n]) & \text{si } n > 0 \end{cases}$$

donde $g(v, n) = f(v, n, 0)$

4. Ejemplos más complicados

En este apartado se analizan particularidades del método como:

- La elección de una generalización adecuada
- La inclusión de varios parámetros adicionales
- Su uso con funciones que admiten la recursividad múltiple
- Su uso cuando se desconoce una fórmula para la función

4.1. Elección de una generalización adecuada: factorial

La elección de una función general es importante para diseñar una función recursiva eficiente. Veamos qué ocurre si elegimos la siguiente generalización para el caso del factorial:

$$f(x, y) = y + x!$$

En ese caso, volviendo a tomar $x - 1$ como la simplificación hacia el caso base, obtenemos:

$$\begin{array}{ccc} f(x, y) & \longrightarrow & y + x! \\ \parallel & & \parallel \\ f(x - 1, z) & \longrightarrow & z + (x - 1)! \end{array}$$

de donde podemos despejar z de la siguiente manera

$$\begin{aligned} z &= y + x! - (x - 1)! = y + (x - 1) \cdot (x - 1)! = \\ &= y + (x - 1) \cdot f(x - 1, 0) \end{aligned}$$

Nótese que es necesario introducir una llamada a la función general (aunque en este caso con el parámetro $x - 1$) en el propio parámetro z . El resultado es un algoritmo correcto que utiliza recursividad anidada (que generará un árbol de recursión binario, y por lo tanto será ineficiente al tener una complejidad exponencial):

$$f(x, y) = \begin{cases} 1 + y & \text{si } x = 0 \\ f(x - 1, y + (x - 1) \cdot f(x - 1, 0)) & \text{si } x > 0 \end{cases}$$

donde $n! = g(x) = f(x, 0)$.

4.1.1. Conclusiones sobre la elección de la función general

En general, sumar una variable cuando la fórmula original involucra un productorio, o multiplicar por una variable cuando la fórmula contiene un sumatorio, conduce a una solución con recursividad anidada. Para el factorial, la generalización $(x!)^y$ también lleva a la recursividad anidada.

Por último, el tipo del parámetro adicional también debe tenerse en cuenta a la hora de implementar la función. Si hubiésemos escogido la generalización $f(x, y) = x!/y$, entonces la regla recursiva sería $f(x, y) = f(x - 1, y/x)$, y el segundo parámetro tendría que ser real en lugar de entero.

4.2. La inclusión de varios parámetros adicionales: cambio de base

En los ejemplos anteriores sólo se añadía un parámetro a las funciones generales. En este caso habrá que incluir dos parámetros (será necesario realizar dos iteraciones del bucle descrito en la metodología).

Dado un entero x (expresado en base 10), y una nueva base $2 \leq b \leq 10$, el objetivo es diseñar una función que devuelva un nuevo entero que represente a x en dicha base b . La fórmula para la función es:

$$g(x, b) = \sum_{i=0}^n (R_i \cdot 10^i)$$

donde n es el número de dígitos que tendría el nuevo número en base b , y los coeficientes R_i pueden calcularse mediante las siguientes relaciones de recurrencia:

$$R_i = \begin{cases} x \bmod b & \text{si } i = 0 \\ Q_{i-1} \bmod b & \text{si } i > 0 \end{cases}$$

$$Q_i = \begin{cases} x \operatorname{div} b & \text{si } i = 0 \\ Q_{i-1} \operatorname{div} b & \text{si } i > 0 \end{cases}$$

Como la función contiene un sumatorio, una posible generalización podría ser:

$$f(x, b, s) = s + \sum_{i=0}^n (R_i \cdot 10^i)$$

Además, una simplificación lógica hacia el caso base es $x \operatorname{div} b$. En este escenario tendríamos:

$$\begin{array}{ccc} f(x, b, s) & \longrightarrow & s + \sum_{i=0}^n (R_i \cdot 10^i) \\ \parallel & & \parallel \\ f(x \operatorname{div} b, b, z) & \longrightarrow & z + \sum_{i=1}^n (R_i \cdot 10^{i-1}) \end{array}$$

Sin embargo, despejar z es complicado. Por tanto, es necesario volver a algún paso anterior de la metodología. Asumiendo que la fórmula para g es correcta, podemos volver al paso 2 e incorporar un nuevo parámetro para formular una segunda función general h . En este ejemplo, se puede apreciar como el despeje de z se habría simplificado si hubiésemos podido multiplicar el sumatorio de abajo por 10. Por tanto, podemos incluir un nuevo parámetro que multiplique al sumatorio, lo cual resulta en:

$$h(x, b, s, p) = s + p \cdot \sum_{i=0}^n (R_i \cdot 10^i)$$

De esta manera, tenemos:

$$\begin{array}{ccc} h(x, b, s, p) & \longrightarrow & s + p \cdot \sum_{i=0}^n (R_i \cdot 10^i) \\ \parallel & & \parallel \\ h(x \operatorname{div} b, b, z, y) & \longrightarrow & z + y \cdot \sum_{i=1}^n (R_i \cdot 10^{i-1}) \end{array}$$

de donde es fácil ver que las expresiones de la derecha son iguales si $z = s + pR_0 = s + p(x \operatorname{mod} b)$, e $y = 10p$. Finalmente, el algoritmo recursivo final es:

$$h(x, b, s, p) = \begin{cases} s & \text{si } x = 0 \\ h(x \operatorname{div} b, b, s + p(x \operatorname{mod} b), 10p) & \text{si } x > 0 \end{cases}$$

donde $g(x, b) = h(x, b, 0, 1)$.

4.3. Método con funciones que admiten recursividad múltiple: números de Fibonacci

El n -ésimo número de Fibonacci $F_n = F_{n-1} + F_{n-2}$, con $F_0 = 0$ y $F_1 = 1$ se puede calcular mediante un algoritmo iterativo sencillo de complejidad $\Theta(n)$ ¹, o un algoritmo recursivo por cola que calcule sucesivos números de Fibonacci y almacene sus valores en dos parámetros, cuyos valores iniciales determinarán el resultado de la función.

La elección de una generalización para esta función es más compleja, por lo que vamos a analizar varias posibilidades.

¹Existen algoritmos para hallar números de Fibonacci con complejidad $\Theta(\log n)$.

4.3.1. Primera generalización

Aplicando la metodología no resulta complicado hallar un algoritmo recursivo final correcto, aunque ineficiente. Si $g(n) = F_n$, podemos considerar la generalización:

$$f(n, y) = y + F_n$$

Simplificando hacia los caso base tenemos:

$$\begin{array}{ccc} f(n, y) & \longrightarrow & y + F_n \\ \parallel & & \parallel \\ f(n-1, z) & \longrightarrow & z + F_{n-1} \end{array} \Rightarrow z = y + F_{n-2}$$

Dado que $F_{n-2} = f(n-2, 0)$, el algoritmo resultante es:

$$f(n, y) = \begin{cases} 1 + y & \text{si } n = 1, 2 \\ f(n-1, y + f(n-2, 0)) & \text{si } n \geq 3 \end{cases}$$

donde $F_n = g(n) = f(n, 0)$.

Es importante destacar que el algoritmo anterior utiliza recursividad anidada y es ineficiente, ya que tiene la misma complejidad exponencial que el tradicional basado en recursividad múltiple: $\Theta(\phi^n)$.

4.3.2. Segunda generalización

Dado que $F_n = F_{n-1} + F_{n-2}$, podríamos proponer la siguiente generalización, introduciendo dos parámetros:

$$f(n, a, b) = aF_{n-1} + bF_{n-2}$$

Haciendo una simplificación hacia el caso base obtendríamos $f(n, a, b) = f(n-1, x, y) = xF_{n-2} + yF_{n-3}$. Sin embargo, como la expresión contiene dos variables (x e y) no es posible hallar su valor. Necesitaríamos formar una segunda ecuación y resolver un sistema de dos ecuaciones con dos incógnitas, pero manteniendo las variables x e y en la fórmula. A simple vista, no parece posible obtener dicha segunda ecuación².

²En realidad los parámetros a y b son los valores iniciales de los números de Fibonacci, con lo cual realmente es posible formar $f(n+1, a, b) = f(n, x, y) = xF_{n-1} + yF_{n-2}$, y ya con dos ecuaciones se pueden hallar los valores $x = a + b$, e $y = a$.

4.3.3. Tercera generalización

Una alternativa posible, donde sí vamos a poder encontrar dos ecuaciones, es:

$$f(n, a, b) = \frac{a(\phi - 1) + b}{\sqrt{5}}\phi^n + \frac{a\phi - b}{\sqrt{5}}(1 - \phi)^n$$

donde $\phi = \frac{1+\sqrt{5}}{2}$, y $f(n, a, b) = T_n = T_{n-1} + T_{n-2}$, con $T_0 = a$, y $T_1 = b$. Es decir, sigue la recurrencia donde un número es igual a la suma de los dos anteriores, y además se especifican las condiciones iniciales. La expresión es simplemente el resultado de resolver la ecuación de recurrencia con las condiciones iniciales (obteniendo una expresión no recursiva).

En este caso, como se han añadido dos parámetros (a, b) a la función general, necesitaríamos considerar dos ecuaciones que contengan a x e y . Afortunadamente, para la generalización propuesta se puede escoger, por ejemplo, $f(n, a, b) = f(n - 1, x, y)$, y $f(n + 1, a, b) = f(n, x, y)$. La clave está en que x e y representan condiciones iniciales, las cuales podemos fijar. Posteriormente se resuelve el sistema formado por dichas ecuaciones lineales para hallar x e y , que en este caso resultan ser $x = b$, e $y = a + b$.

4.3.4. Cuarta generalización

En este caso, la función a considerar tiene una descripción totalmente diferente, aunque posiblemente más sencilla. No va a ser una única fórmula, sino un conjunto de expresiones. Considérese que $f(n, a, b)$ es una secuencia $G_n = G_{n-1} + G_{n-2}$, donde $G_0 = a$ y $G_1 = b$ son los valores iniciales. En ese caso tenemos:

$$\begin{array}{ccc} f(n, a, b) & \rightarrow & G_n, \quad G_0 = a, \quad G_1 = b \\ \parallel & & \parallel \\ f(n - 1, x, y) & \rightarrow & H_{n-1}, \quad H_0 = x, \quad H_1 = y \end{array}$$

donde H sigue la misma relación de recurrencia, con diferentes condiciones iniciales. Como $G_n = H_{n-1}$, entonces $G_1 = b = x = H_0$, y $G_2 = a + b = y = H_1$, completando el caso recursivo.

4.4. Metodología aplicada cuando se desconoce una fórmula para la función: paridad de un número

En este caso, el objetivo es diseñar una función recursiva por cola Booleana que determine si un número natural es impar. Se trata de una versión recursiva de la función *odd*, donde no será posible utilizar la función *mod* (resto de la división entera).

Se trata de un ejemplo similar al presentado en la Sec 4.3.4, donde no se ha especificado una función mediante una fórmula concreta. Al no poder utilizar ni la función *odd* ni *mod*, simplemente definimos una descripción de la función que queremos implementar:

$$g(x) = x \text{ es impar}$$

El siguiente paso consistiría en crear una generalización añadiendo un parámetro. En este caso podemos elegir:

$$f(x, y) = y \oplus (x \text{ es impar})$$

Donde \oplus denota la función XOR (O exclusiva). La simplificación hacia el caso base ($0 = FALSE$) consiste en decrementar x . Por tanto podemos considerar:

$$\begin{array}{ccc} f(x, y) & \longrightarrow & y \oplus (x \text{ es impar}) \\ \parallel & & \parallel \\ f(x-1, z) & \longrightarrow & z \oplus ((x-1) \text{ es impar}) \end{array} \quad \Rightarrow z = \bar{y}$$

Donde \bar{y} denota y -negado. El resultado $z = \bar{y}$ se puede ver analizando la siguiente tabla de verdad:

y	b	$y \oplus b$	$z \oplus \bar{b}$	\bar{b}	z
0	0	0	0	1	1
0	1	1	1	0	1
1	0	1	1	1	0
1	1	0	0	0	0

donde $b = (x \text{ es impar})$, por lo que $\bar{b} = ((x-1) \text{ es impar})$.

Por último, como $f(0, y) = y \oplus FALSE = y$, la función queda definida de la siguiente manera:

$$f(x, y) = \begin{cases} y & \text{si } x = 0 \\ f(x-1, \bar{y}) & \text{si } x > 0 \end{cases}$$

donde $g(x) = f(x, FALSE)$.