8. RECURSIVIDAD

El juicio de los hombres entendidos
descubre por las cosas claras las oscuras,
por las pequeñas las grandes,
por las próximas las remotas
y por la apariencia la realidad.
Séneca

Recursividad o recursión, como también se le llama, es un concepto que se aplica de manera especial en las matemáticas y en la programación de computadores; y de forma similar en numerosas situaciones de la vida cotidiana. En matemáticas se habla de *definición inductiva* para referirse a los numerosos métodos que hacen uso de la recursividad; mientras que en programación, este tema, se aplica a los algoritmos en el momento de la especificación, y a las funciones en la implementación (Joyanes, 1999).

La utilización del concepto de recursividad en los lenguajes de programación se la debe al profesor John McCarthy, del *Massachu-setts Institute of Tenology* (MIT), quien recomendó su inclusión en el diseño de Algol60 y desarrolló el lenguaje Lisp, que introdujo estructuras de datos recursivas junto con procedimientos y funciones recursivas (Baase, 2002).

La recursividad es una alternativa a la utilización de estructuras iterativas. Un algoritmo recursivo ejecuta cálculos repetidas veces mediante llamados consecutivos a sí mismo. Esto no significa que los algoritmos recursivos sean más eficientes que los iterativos, incluso, en algunos casos, los programas pueden requerir más tiempo y memoria para ejecutarse, pero este costo puede ser compensado por una solución más intuitiva y sustentada matemáticamente en una prueba por inducción.

La mayoría de los lenguajes de programación actuales permiten la implementación de algoritmos recursivos, mediante procedimientos o funciones. Para el caso de lenguajes que no permiten su implementación directa, ésta puede simularse utilizando estructuras de datos de tipo pila.

La recursividad es un concepto que se aplica indistintamente a algoritmos, procedimientos y programas. No obstante, en este libro se aborda desde la perspectiva del diseño de algoritmos.

8.1 LA RECURSIVIDAD Y EL DISEÑO DE ALGORITMOS

En el ámbito del diseño de algoritmo se define como recursividad la técnica de plantear la solución de un problema invocando consecutivamente el mismo algoritmo con un problema cada vez menos complejo, hasta llegar a una versión con una solución conocida. Por ejemplo, para calcular el factorial de un número.

Ejemplo 73. Función recursiva para calcular el factorial de un número

Se conoce como factorial de un número al producto de los enteros comprendidos entre 1 y *n* inclusive, y se representa mediante: *n!*

De manera que:

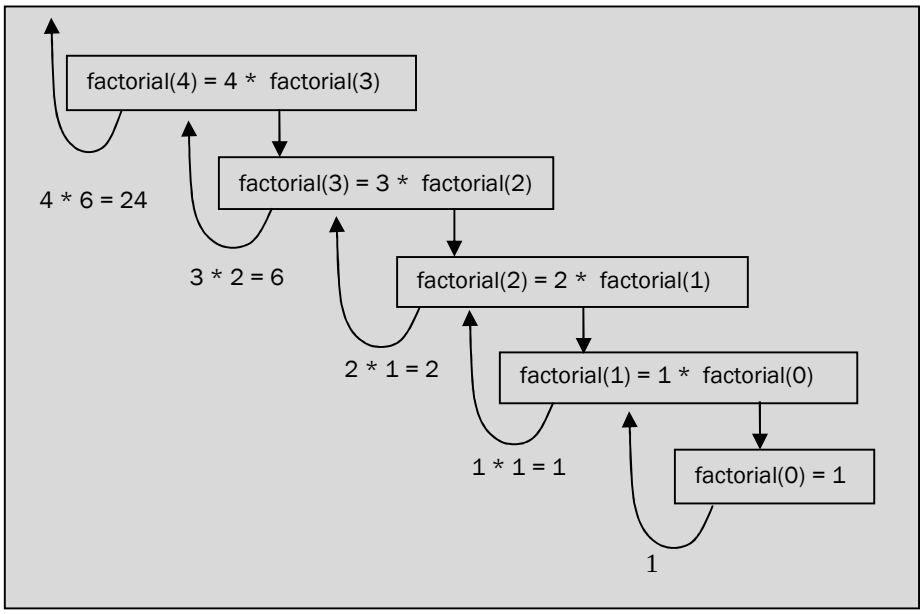
$n! = 1 * 2 * 3 * 4 * \ldots * (n-2) * (n - 1) * n$

También se sabe que el factorial está definido para el caso de que $n = 0$, cuyo resultado es 1. Es decir, $0! = 1$.

De esto se desprende que para calcular el factorial de 4 (4!) se calculará el producto de 4 por el factorial de 3 (3!); para calcular el factorial de 3 (3!), el producto de 3 por el factorial de 2 (2!); para calcular el factorial de 2 (2!), 2 por factorial de 1 (1!); para calcular factorial de 1 (1!), 1 por el factorial de 0. Ahora, como se conoce que el factorial de 0 es 1 (0! = 1), entonces el problema está resuelto. Más adelante se desarrolla completamente este ejercicio.

Como se aprecia en la figura 126, la recursividad consiste en escribir una función que entre sus líneas o sentencias incluyen un llamado a sí misma con valores diferentes para sus parámetros, de manera que progresivamente llevan a la solución definitiva.

Figura 126. Esquema de una función recursiva



La recursividad es otra forma de hacer que una parte del código se ejecute repetidas veces, pero sin implementar estructuras iterativas como: *mientras, para* o *hacer mientras.*

8.2 ESTRUCTURA DE UNA FUNCIÓN RECURSIVA

Se ha mencionado que consiste en implementar funciones que se invocan a sí mismas. Esto implica que la función debe tener una estructura tal que le permita invocarse a sí misma y ejecutarse tantas veces como se requiera para solucionar el problema, pero a la vez, sin hacer más invocaciones de las estrictamente necesarias, evitando que se genere una secuencia infinita de llamadas a sí misma. Cuando una función recursiva especifica en su definición cuándo autoinvocarse y cuándo dejar de hacerlo se dice que aquella está correctamente definida (Lipschutz, 1993)

Una función recursiva bien definida ha de cumplir las dos condiciones siguientes:

1. Debe existir un *criterio base* cuya solución se conoce y que no implica una llamada recursiva. En el caso de utilizar una función recursiva para calcular el factorial, el criterio base indica que factorial de 0 es 1 (0! = 1). Es decir, si la función recibe como parámetro el 0 ya no necesita invocarse nuevamente, simplemente retorna el valor correspondiente, en este caso el 1.

   factorial(n) = ? (requiere llamada a la función recursiva)
   factorial(0) = 1 (no requiere llamado a la función)

2. Cada vez que la función se invoque a sí misma, directa o indirectamente, debe hacerlo con un valor más cercano al *criterio base*. Dado que el criterio base es un caso particular del problema en el que se conoce la solución esto indica que con cada llamado a la función recursiva se estará acercando a la solución conocida. Continuando con el ejemplo de la función factorial, se ha determinado que el *criterio base* es *0! = 1*, entonces, cada vez que se invoque la función deberá utilizarse como parámetro un valor más cercano a 0.

El factorial de un número es el producto del número por el factorial del número menor, hasta llegar al factorial de 0 que es constante. Con base en esto se puede afirmar que está definido para cualquier entero positivo, así:

0! = 1	
1! = 1 * 1 = 1	1! = 1 * 0!
2! = 2 * 1 = 2	2! = 2 * 1!
3! = 3 * 2 * 1 = 6	3! = 3 * 2!
4! = 4 * 3 * 2 * 1 = 24	4! = 4 * 3!
5! = 5 * 4 * 3 * 2 * 1 = 120	5! = 5 * 4!

Por lo tanto, la solución del factorial se puede expresar como:

a. Si n = 0 → n! = 1
*b. Si n > 0 → n! = n * (n – 1)!*

Y esta es la definición correcta de una función recursiva, donde *a* es el criterio base, es decir, la solución conocida, y *b* es la invocación recursiva con un argumento más cercano a la solución. Expresado en forma de función matemática se tiene:

Siendo *f(n)* la función para calcular el factorial de *n*, entonces:

$$f(n) = \begin{cases} 1 & \text{Si } n = 0 \\ n * f(n-1) & \text{Si } n > 0 \end{cases}$$

El pseudocódigo para esta función se presenta en el cuadro 135.

Cuadro 135. Función recursiva para calcular el factorial de un número

```
1.  Entero factorial(entero n)
2.      Si n = 0 entonces
3.          Retornar  1
4.      Si no
5.          Retornar (n * factorial(n – 1))
6.      Fin si
7.  Fin factorial
```

En la definición se puede apreciar claramente el cumplimiento de las dos condiciones de que trata esta sección. En las líneas 2 y 3 se examina el cumplimiento del *criterio base*, para el cual se conoce la solución (0! = 1).

```
2.      Si n = 0 entonces
3.          Retornar 1
```

Si aun no se ha llegado a dicho criterio, es necesario invocar nuevamente la función recursiva, pero con un valor menor, como se aprecia en las líneas 4 y 5.

```
4.  Si no
5.      Retornar (n * factorial(n – 1))
```

8.3 EJECUCIÓN DE FUNCIONES RECURSIVAS

La ejecución de una función recursiva requiere que dinámicamente se le asigne la memoria suficiente para almacenar sus datos. Por cada ejecución se reserva espacio para los parámetros, las variables locales, las variables temporales y el valor devuelto por la función. El código se ejecuta desde el principio con los nuevos datos. Cabe aclarar que no se hace copia del código recursivo en cada marco de activación. (Schildt, 1993).

El espacio en que se ejecuta cada invocación de una función recursiva se denomina *Marco de activación* (Baase, 2002). Este marco, además de proporcionar espacio para guardar las variables con que opera la función, también, proporciona espacio para otras necesidades contables, como la dirección de retorno, que indica la instrucción que se ha de ejecutar una vez salga de la función recursiva. Así, se crea un marco de referencia en el que la función se ejecuta únicamente durante una invocación.

Ahora bien, como la función será invocada un número indeterminado de veces (depende del valor de los parámetros) y por cada invocación se creará un marco de activación, el compilador deberá asignar una región de la memoria para la creación de la *pila de marcos.* A este espacio se hace referencia mediante un registro llamado *apuntador de marco*, de modo

que mientras se ejecuta una invocación de la función, se conoce dónde están almacenadas las variables locales, los parámetros de entrada y el valor devuelto.

Una ejecución a mano que muestra los estados de los marcos de activación se denomina *rastreo de activación* (Baase, 2002) y permite analizar el tiempo de ejecución de la función y comprender como funciona realmente la recursividad en el computador.

Cada invocación activa tiene un marco de activación único. Una invocación de función está activa desde el momento en que se entra en ella hasta que se sale, después de haber resuelto el problema que se le paso como parámetro. Todas las activaciones de la función recursiva que están activas simultáneamente tienen marcos de activación distintos. Cuando se sale de una invocación de función su marco de activación se libera automáticamente para que otras invocaciones puedan hacer uso de ese espacio y la ejecución se reanuda en el punto donde se hizo la invocación de la función.

Para ilustrar mejor estas ideas, en la figura 127 se presenta el rastreo de activación para la función factorial.

Figura 127. Rastreo de activación para la función factorial

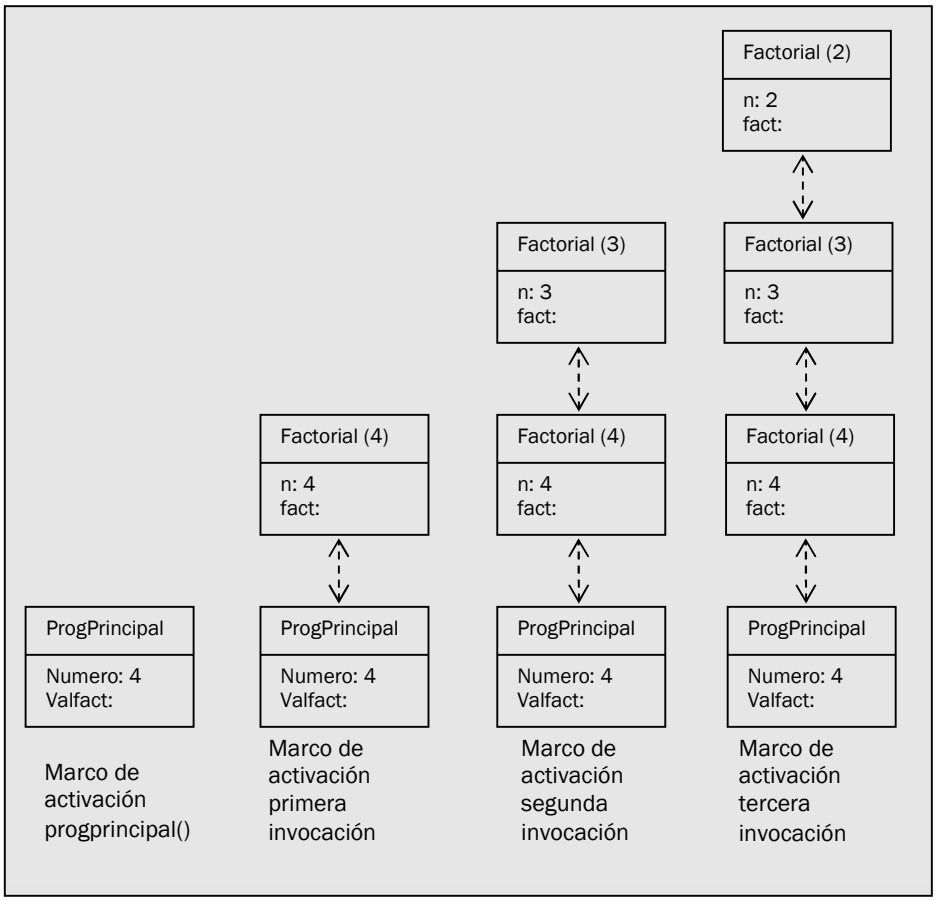
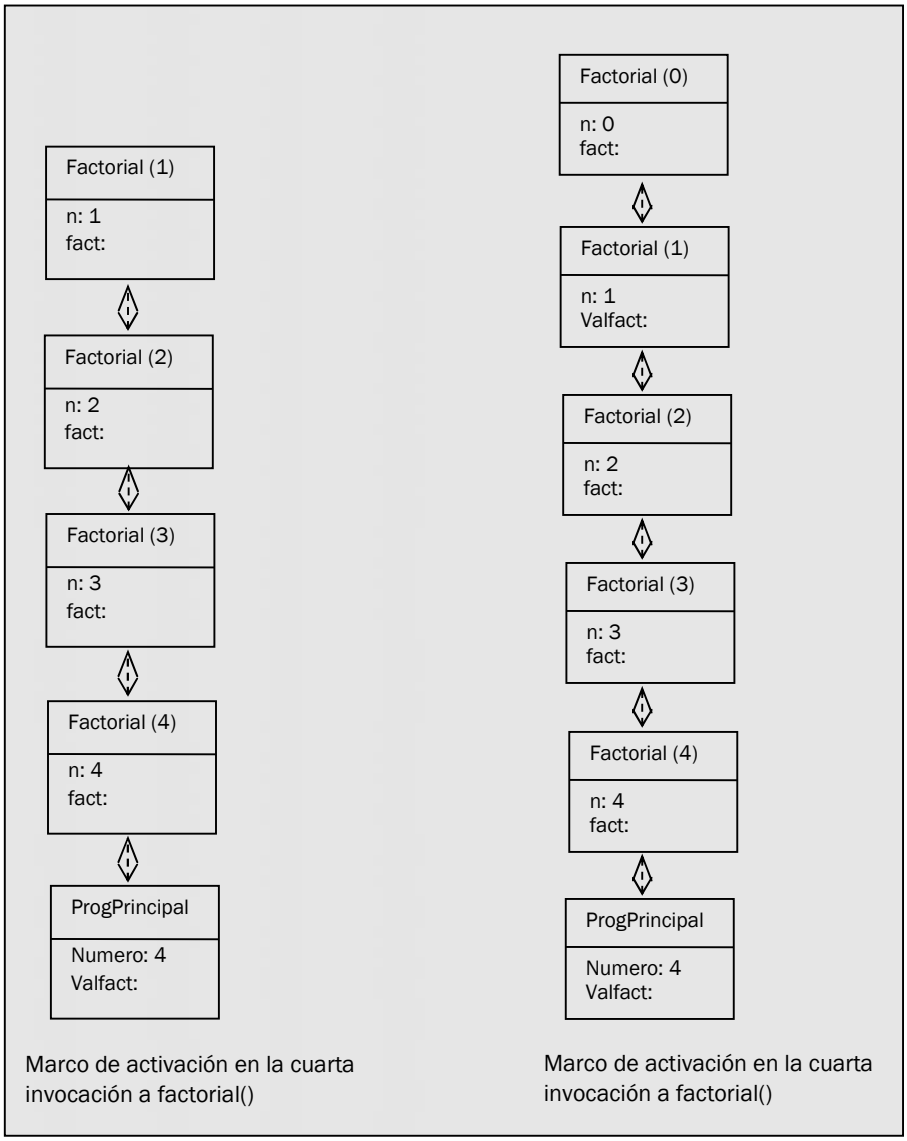


Figura 127. (Continuación)



Una vez que la función ha encontrado el criterio base, es decir, un caso con solución no recursiva, éste marco devuelve a quien lo invocó el valor determinado para dicho criterio y libera el espacio ocupado; en este caso, el último marco retorna 1 para el marco anterior y desaparece. Y continúa un proceso regresivo en que cada marco de activación desarrolla sus operaciones utilizando el valor devuelto por la función recursiva invocada y retorna el valor obtenido al marco que lo invocó.

8.4 ENVOLTURAS PARA FUNCIONES RECURSIVAS

Es común que las funciones recursivas se concentren en la búsqueda de un valor o en el desarrollo de una tarea concreta, como calcular el factorial, invertir una cadena, encontrar un valor en una estructura de datos u ordenar una lista y no se ocupen de otras tareas que no requieren ser recursivas como leer el dato a buscar, comprobar la validez de un argumento o imprimir los resultados.

Es decir, cuando se programa utilizando recursividad se encontrará que hay algunas tareas que no requieren un tratamiento recursivo y que pueden estar antes o después de la ejecución de la función recursiva. Para manejar estas tareas se definen programas, procedimientos o funciones *envoltura*.

Se denomina *envoltura* a los procedimientos no recursivos que se ejecutan antes o después de la invocación de una función recursiva y que se encargan de preparar los parámetros de la función y de procesar los resultados (Baase, 2002).

En el ejemplo del factorial se utiliza como envoltura un algoritmo que se encarga de leer el número que se utiliza como argumento de la función, invocar la función y mostrar el resultado. El seudocódigo se presenta en el cuadro 136.

Cuadro 136. Pseudocódigo del programa principal para calcular el factorial

```
1.  Inicio
2.      Entero: num
3.      Leer  num
4.      Imprimir "factorial de", num, " = ", factorial(num)
5.  Fin algoritmo
```

8.5 TIPOS DE RECURSIVIDAD

La recursividad puede presentarse de diferentes maneras y dependiendo de ellas se han establecido al menos cuatro tipos diferentes de implementaciones recursivas.

Recursividad simple: se presenta cuando una función incluye un llamado a sí misma con un argumento diferente. Ejemplo de este tipo de recursividad es la función factorial().

Este tipo de recursividad se caracteriza porque puede pasarse fácilmente a una solución iterativa.

Recursividad múltiple: el cuerpo de una función incluye más de una llamado a la misma función, por ejemplo, la función para calcular un valor de la serie Fibonacci (esta función se explica en la sección 8.7).

Recursividad anidada: se dice que una función recursiva es anidada cuando entre los parámetros que se pasan a la función se incluye una invocación a la misma. Un ejemplo de recursividad anidada es la solución al problema de Ackerman[§].

Recursividad cruzada o indirecta: en este tipo de recursividad, el cuerpo de la función no contiene un llamado a sí misma, sino a otra función; pero, la segunda incluye un llamado a la primera. Puede ser que participen más de dos funciones. Este tipo de implementaciones también se conoce como *cadenas recursivas*. Como ejemplo de este tipo de recursividad se tiene la función para validar una expresión matemática.

8.6 EFICIENCIA DE LA RECURSIVIDAD

En general, una versión iterativa se ejecutará con más eficiencia en términos de tiempo y espacio que una versión recursiva. Esto se debe a que, en la versión iterativa, se evita la información general implícita al entrar y salir de una función. En la versión recursiva, con frecuencia, es necesario apilar y desapilar variables anónimas en cada campo de activación. Esto no es necesario en el caso de una implementación iterativa donde los resultados se van reemplazando en los mismos espacios de memoria.

Por otra parte, la recursividad es la forma más natural de solucionar algunos tipos de problemas, como es el caso de: ordenamiento rápido (*QuickSort*), las torres de Hanoy, las ocho reinas, la conversión de prefijo a posfijo o el recorrido de árboles, que aunque pueden implementarse soluciones iterativas, la solución recursiva surge directamente de la definición del problema.

El optar por métodos recursivos o iterativos es, más bien, un conflicto entre la eficiencia de la máquina y la eficiencia del programador (Langsam, 1997). Considerando que el costo de la programación tiende a aumentar y el costo de computación a disminuir, 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, como tampoco vale la pena implementar soluciones recursivas para problemas que pueden solucionarse fácilmente de manera iterativa. Muchos de los ejemplos de soluciones recursivas de este capítulo se presentan con fines didácticos más no por su eficiencia.

Conviene tener presente que la demanda de tiempo y espacio extra, en las soluciones recursivas, proviene principalmente de la creación de los espacios de activación de las funciones y del apilamiento de resultados parciales, de manera que éstas pueden optimizarse reduciendo el uso de variables locales. A la vez que las soluciones iterativas que utilizan pilas pueden requerir tanto tiempo y espacio como las recursivas.

[§] Wilhelm Ackerman (29 de marzo 1896 - 24 de diciembre 1962) matemático alemán, concibió la función doblemente recursiva que lleva su nombre como un ejemplo de la teoría computacional y demostró que no es primitiva recursiva.

8.7 EJEMPLOS DE SOLUCIONES RECURSIVAS

Ejemplo 74. Sumatoria recursiva

Diseñar una función recursiva para calcular la sumatoria de los primeros *n* números enteros positivos, de la forma:

$1 + 2 + 3 + 4 + 5 + 6 + \ldots + (n-1) + n$

Se define la función *f(n)* donde *n* puede ser cualquier número mayor o igual a 1 ($n >= 1$).

Se conoce que al sumar 0 a cualquier número éste se mantiene. De ahí se desprende que la sumatoria de cero es cero. Para cualquier otro entero positivo, la sumatoria se calcula adicionando su propio valor a la sumatoria del número inmediatamente inferior. Así:

$f(0) = 0$
$f(1) = 1 + f(0)$
$f(2) = 2 + f(1)$
$f(3) = 3 + f(2)$
$f(4) = 4 + f(3)$
$f(n) = n + f(n-1)$

De esto se desprende que

a. Si $n = 0 \rightarrow f(n) = 0$
b. Si $n > 0 \rightarrow f(n) = n + f(n-1)$

De esta manera se plantea una solución recursiva donde *a* representa el criterio base y *b* la invocación recursiva de la función.

$$f(n) = \begin{cases} 0 & \text{Si } n = 0 \\ n + f(n-1) & \text{Si } n > 0 \end{cases}$$

El pseudocódigo de la función sumatoria se presenta en el cuadro 137.

Cuadro 137. Función recursiva para calcular la sumatoria de un número

```
1.  Entero sumatoria(entero n)
2.      Si n = 0 entonces
3.          Retornar 0
4.      Si no
5.          Retornar  (n + sumatoria(n – 1))
6.      Fin si
7.  Fin sumatoria
```

Ejemplo 75. Potenciación recursiva

Se define como cálculo de una potencia a la solución de una operación de la forma x^n, donde *x* es un número entero o real que se conoce como base y *n* es un entero no negativo conocido como exponente.

La potencia x^n es el producto de *n* veces *x*, de la forma:

$x^n = x * x * x * \ldots * x$ (*n* veces *x*)

Por las propiedades de la potenciación se tiene que cualquier número elevado a 0 da como resultado 1 y que cualquier número elevado a 1 es el mismo número.

$x^0 = 1$
$x^1 = x$

Para este ejercicio se extiende la primera propiedad, también, para el caso de x = 0. Aunque normalmente se dice que este resultado no está definido, para explicar recursividad esto es irrelevante.

Para proponer una solución recursiva es necesario considerar la potencia x^n en función de una expresión más cercana a la identidad: $x^0 = 1$.

Se propone el siguiente ejemplo:

$2^0 = 1$	
$2^1 = 2$	$2^1 = 2 * 2^0 = 2$
$2^2 = 4$	$2^2 = 2 * 2^1 = 2 * 2 = 4$
$2^3 = 8$	$2^3 = 2 * 2^2 = 2 * 4 = 8$
$2^4 = 16$	$2^4 = 2 * 2^3 = 2 * 8 = 16$
$2^5 = 32$	$2^5 = 2 * 2^4 = 2 * 16 = 32$

Como se aprecia en la columna de la derecha, cada potencia puede plantearse utilizando la solución de la potencia anterior, con exponente menor.

En consecuencia:

a. Si $n = 0 \rightarrow x^n = 1$
b. Si $n > 0 \rightarrow x^n = x * x^{n-1}$

Donde *a* es el criterio base y *b* es la invocación recursiva.

En el cuadro 138 se presenta el pseudocódigo para calcular recursivamente cualquier potencia de cualquier número.

Cuadro 138. Función recursiva para calcular una potencia

```
1.  Entero potencia(entero x, entero n)
2.      Si(n = 0)
3.          Retornar 1
4.      Si no
5.          Retornar (x * potencia(x, n-1))
6.      Fin si
7.  Fin potencia
```

En las líneas 2 y 3 se soluciona el problema cuando se presenta el criterio base, en caso contrario, se debe hacer una invocación recursiva de la función, según las líneas 4 y 5.

Ejemplo 76. Serie Fibonacci

La serie Fibonacci tiene muchas aplicaciones en ciencias de la computación, en matemáticas y en teoría de juegos. Fue publicada por primera vez en 1202 por un matemático italiano del mismo nombre, en su libro titulado *Liberabaci* (Brassard, 1997).

Fibonacci planteó el problema de la siguiente manera: supóngase que una pareja de conejos produce dos descendientes cada mes y cada nuevo ejemplar comienza su reproducción después de dos meses. De manera que al comprar una pareja de conejos, en los meses uno y dos se tendrá una pareja, pero al tercer mes se habrán reproducido y se contará con dos parejas, en el mes cuatro solo se reproduce la primera pareja, así que el número aumentará a tres parejas; en el mes cinco, comienza la reproducción de la segunda pareja, con lo cual se obtendrán cinco parejas, y así sucesivamente.

Si ningún ejemplar muere, el número de conejos que se tendrán cada mes está dado por la sucesión: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, …, que corresponde a la relación que se muestra en el cuadro 139:

Cuadro 139. Términos de la serie Fibonacci

Mes (*n*)	0	1	2	3	4	5	6	7	8	9	...
Pares de conejos *f(n)*	0	1	1	2	3	5	8	13	21	34	...

Cuya representación formal es:

a. f(0) = 0
b. f(1) = 1
c. f(n) = f(n-1) + f(n-2)

Es decir, en el mes 0 aún no se tiene ningún par de conejos, en el mes uno se adquiere la primera pareja, en el mes dos se mantiene la misma pareja comprada, porque aún no comienzan a reproducirse. Pero a partir del mes tres hay que calcular la cantidad de parejas que se tienen considerando la condición de que cada par genera un nuevo par cada mes, pero solo comienza a reproducirse después de dos meses.

De esta manera, el problema está resuelto para el mes cero y para el mes uno, estos se toman como criterio base (*a, b*) y a partir del segundo mes se debe tener en cuenta los meses anteriores. Esto genera una invocación recursiva de la forma:

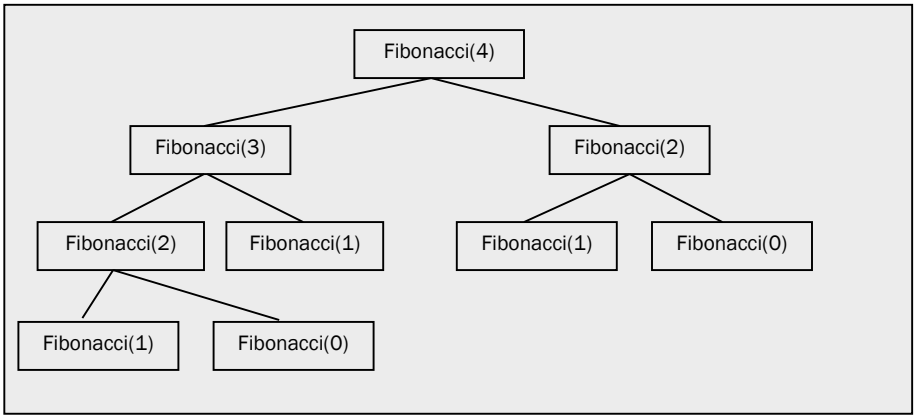
f(n) = f(n-1) + f(n-2) para *n >= 2.*

En el pseudocódigo que se presenta en el cuadro 140 se aprecia que cada activación tiene definidos dos criterios bases (líneas 2) y, si aún no se ha llegado a estos, en cada marco de activación se generan dos nuevas invocaciones recursivas (línea 5). En la figura 128 se presenta el árbol que se forma al calcular el valor de la serie para el número 4.

Cuadro 140. Función recursiva para calcular el n-ésimo termino de la serie Fibonacci

```
1.  Entero Fibonacci(Entero n)
2.      Si (n = 0) o (n = 1) entonces
3.          Retornar n
4.      Si no
5.          Retornar (Fibonacci(n – 1) + Fibonacci(n – 2))
6.      Fin si
7.  Fin Fibonacci
```

Figura 128. Marcos de activación de la serie Fibonacci



Ejemplo 77. Máximo común divisor

El Máximo Común Divisor (MCD) entre dos o más números es el mayor de los divisores comunes. Se conocen dos métodos para encontrar el MCD entre dos números: el primero consiste en descomponer cada número en sus factores primos, escribir los números en forma de potencia y luego tomar los factores comunes con menor exponente y el producto de estos será el MCD; el segundo, es la búsqueda recursiva en la que se verifica si el segundo número es divisor del primero, en cuyo caso será el MCD, si no lo es se divide el segundo número sobre el módulo del primero sobre el segundo y así sucesivamente con un número cada vez más pequeño que tiende a 1 como el divisor común a todos los números. Este último se conoce como algoritmo de Euclides.

Por ejemplo, si se desea encontrar el MCD de 24 y 18, aplicando el algoritmo de Euclides, se divide el primero sobre el segundo (24/18= 1, residuo = 6), si la división es exacta se tiene que el MCD es el segundo número, pero si no lo es, como en este caso, se vuelve a buscar el MCD entre el segundo número (18) y el módulo de la división (6) y de esa manera hasta que se lo encuentre. Si los dos números son primos se llegará hasta 1.

Si *f(a,b)* es la función que devuelve el Máximo Común Divisor de *a* y *b*, entonces se tiene que:

$c = a \bmod b$

a. *Si* $c = 0 \rightarrow f(a,b) = b$
b. *Si* $c > 0 \rightarrow f(a,b) = f(b,c)$

La función recursiva para calcular el MCD de dos números aplicando el algoritmo de Euclides se presenta en el cuadro 141.

Cuadro 141. Función recursiva para calcular el MCD

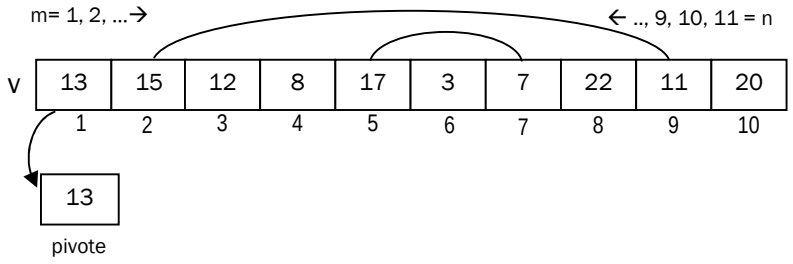
```
1.  Entero mcd(Entero a, Entero b)
2.      Entero c
3.      c = a Mod b
4.      Si c = 0 entonces
5.          Retornar b
6.      Si no
7.          Retornar mcd(b, c)
8.      Fin si
9.  Fin mcd
```

Ejemplo 78. Algoritmo de ordenamiento rápido recursivo

Este algoritmo recursivo de ordenamiento de vectores o listas se basa en el principio *dividir para vencer*, que aplicado al caso se traduce en que es más fácil ordenar dos vectores pequeños que uno grande. Se trata de tomar un elemento del vector como referencia, a éste se le llama *pivote*, y dividir el vector en tres partes: los elementos menores al pivote, el pivote y los elementos mayores al pivote.

Para particionar el vector se declara dos índices y se hacen dos recorridos, uno desde el primer elemento hacia adelante y otro desde el último elemento hacia atrás. El primer índice se mueve hasta encontrar un elemento mayor al pivote, mientas que el segundo índice se mueve hasta encontrar un elemento menor al pivote, cuando los dos índices se han detenido se hace el intercambio y se continúa los recorridos. Cuando los dos índices se cruzan se suspende el particionamiento. Como ejemplo, considérese el vector de la figura 112 y aplíquesele las instrucciones del cuadro 142.

Figura 129. Recorridos para particionar un vector

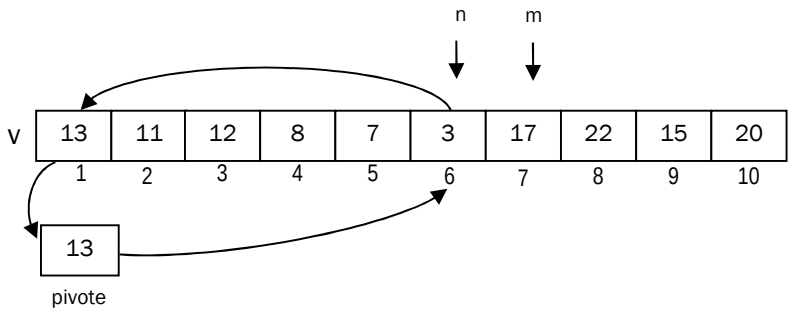


Cuadro 142. Recorridos para particionar un vector

```
1   Entero m = 1, n = 11, pivote = v[1]
2   Mientras m <= n hacer
3       Hacer
4           m = m + 1
5       Mientras v[m] < pivote
6       Hacer
7           n = n – 1
8       Mientras v[n] > pivote
9       Si m < n entonces
10          aux = v[m]
11          v[m] = v[n]
12          v[n] = aux
13      Fin si
14  Fin mientras
```

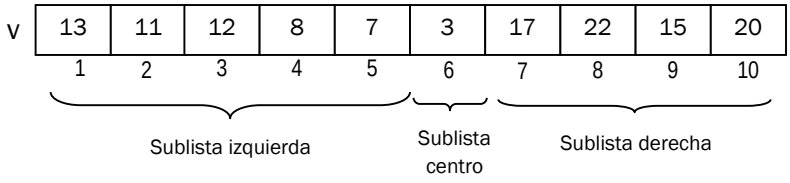
Después de ejecutar las instrucciones del cuadro 142 el vector quedaría como se muestra en la figura 130.

Figura 130. Recorridos para particionar un vector



Después de pasar los datos mayores al pivote al lado derecho y los menores al lado izquierdo se intercambia el pivote con el dato indicado por el índice que recorre el vector de derecha a izquierda (n), como lo muestran las flechas en la figura 130. Después de dicho intercambio, los elementos quedan en el orden que se muestra en la figura 131.

Figura 131. Orden del vector después de la primera partición



Aunque las divisiones son puramente formales, ahora se tienen tres sublistas:

lista-izquierda = {3, 11, 12, 8, 7}
lista-centro = {13}
lista-derecha = {17, 22, 15, 20}

Como se aprecia en las sublistas, los datos se han movido de un lado del pivote al otro, pero no están ordenados. Por ello, el siguiente paso en el algoritmo *Quicksort* consiste en tomar cada una de las sublistas y realizar el mismo proceso, y continuar haciendo particiones de las sublistas que se generan hasta reducir su tamaño a un elemento.

En el cuadro 143 se presenta la función recursiva *Quicksort* con el algoritmo completo para ordenar un vector aplicando éste método.

Cuadro 143. Función recursiva Quicksort

```
1   Entero[] Quicksort(entero v[], entero inf, entero sup)
2       Entero: aux, pivote = v[inf], m = inf, n = sup+1
3       Mientras m <= n hacer
4           Hacer
5               m = m + 1
6           Mientras v[m] < pivote
7           Hacer
8               n = n – 1
9           Mientras v[n] > pivote
10          Si m < n entonces
11              Aux = v[m]
12              v[m] = v[n]
13              v[n] = aux
14          Fin si
15      Fin mientras
16      v[inf] = v[n]
17      v[n] = pivote
18      Si inf < sup entonces
19          v[] = Quicksort(v[], inf, n-1)
```

Cuadro 143. (Continuación)

```
20              v[] = Quicksort(v[], n+1,sup)
21          Fin si
22          Retornar v[]
23      Fin Quicksort
```

La función *Quicksort* recibe un vector de enteros, en este ejemplo, y dos valores: *inf* y *sup*, correspondientes a las posiciones que delimitan el conjunto de elementos a ordenar. La primera invocación a la función se hará con todo el vector, enviando como parámetros el índice del primero y del último elemento (*1* y *n*), pero en las que siguen el número de elementos se reducen significativamente en cada llamada, pues corresponde a las sub-listas que se generan en cada partición.

En esta versión de la función se toma como pivote el primer elemento, pero esto no necesariamente tiene que ser así, se puede diseñar una función para buscar un elemento con un valor intermedio que permita una partición equilibrada de la lista.

8.8 EJERCICIOS PROPUESTOS

Diseñar las soluciones recursivas para las necesidades que se plantean a continuación

1. Mostrar los números de 1 a *n*
2. Generar la tabla de multiplicar de un número
3. Sumar los divisores de un número
4. Determinar si un número es perfecto
5. Calcular el producto de dos números sin utilizar el operador *, teniendo en cuenta que una multiplicación consiste en sumar el multiplicando tantas veces como indica el multiplicador.
6. Calcular el Mínimo Común Múltiplo (MCM) de dos números.
7. Convertir un número decimal a binario
8. Convertir un número binario a decimal
9. Determinar si una cadena es palíndromo
10. Mostrar los primeros *n* términos de la serie Fibonacci

11. Invertir el orden de los dígitos que conforman un número
12. Determinar si un número es primo.