

CAPÍTULO 14

Recursividad

- 14.1.** La naturaleza de la recursividad
- 14.2.** Recursividad directa e indirecta
- 14.3.** Recursión *versus* iteración
- 14.4.** Recursión infinita
- 14.5.** Resolución de problemas complejos con recursividad

CONCEPTOS CLAVE
RESUMEN
EJERCICIOS
PROBLEMAS

INTRODUCCIÓN

La recursividad (recursión) es aquella propiedad que posee una función por la cual dicha función puede llamarse a sí misma. Se puede utilizar la recursividad como una alternativa a la iteración. Una solución recursiva es normalmente menos eficiente en términos de tiempo de computadora que una solución iterativa debido a las operaciones auxiliares que llevan consigo

las llamadas supplementarias a las funciones; sin embargo, en muchas circunstancias el uso de la recursión permite a los programadores especificar soluciones naturales, sencillas, que serían, en caso contrario, difíciles de resolver. Por esta causa, la recursión es una herramienta poderosa e importante en la resolución de problemas y en la programación.

14.1. LA NATURALEZA DE LA RECURSIVIDAD¹

Los programas examinados hasta ahora, generalmente estructurados, se componen de una serie de funciones que llaman unas a otras de modo disciplinado. En algunos problemas es útil disponer de funciones que se llamen a sí misma. Un *subprograma recursivo* es un subprograma que se llama a sí mismo ya sea directa o indirectamente. La recursividad es un tópico importante examinado frecuentemente en cursos de programación y de introducción a las ciencias de la computación.

En este libro se dará una importancia especial a las ideas conceptuales que soportan la recursividad. En matemáticas existen numerosas funciones que tienen carácter recursivo; de igual modo numerosas circunstancias y situaciones de la vida ordinaria tienen carácter recursivo.

Hasta el momento casi siempre se han visto subprogramas que llaman a otros subprogramas distintos. Así, si se dispone de dos procedimientos proc1 y proc2, la organización de un programa tal y como se suele haber visto hasta este momento podría adoptar una forma similar a esta:

```
procedimiento proc1(...)  

inicio  

...  

fin_procedimiento  
  

procedimiento proc2(...)  

inicio  

...  

    proc1(...)           // llamada a proc1  

...  

fin_procedimiento
```

Cuando diseñan programas recursivos se tendría esta situación:

```
procedimiento proc1(...)  

inicio  

...  

    proc1(...);  

...  

fin_procedimiento
```

o bien esta otra:

```
procedimiento proc1(...)  

inicio  

...  

    proc2(...)           //llamada a proc2  

...  

fin_procedimiento  
  

procedimiento proc2(...)  

inicio  

...  

    proc1(...)           //llamada a proc1  

...  

fin_procedimiento
```

¹ Las palabras inglesas «recursive» y «recursion» no han sido aceptadas todavía por el Diccionario de la Real Academia de la Lengua Española (www.rae.es). La última edición (22.^a, Madrid, 2001) editada conjuntamente por todas las Academias de la Lengua de España, Latinoamérica y Estados Unidos, recoge sólo los términos sinónimos siguientes en sus acepciones Mat (Matemáticas): *recurrencia*, «propiedad de aquellas secuencias en las que cualquier término se puede calcular conociendo los precedentes», y *recurrente*, «Dicho de un proceso: que se repita».

EJEMPLO 14.1

El factorial de un entero no negativo n , escrito $n!$ (y pronunciado n factorial), es el producto

$$n! = n \cdot (n - 1) \cdot (n - 2) \cdot \dots \cdot 1$$

en el cual

$$\begin{aligned} 0! &= 1 \\ 1! &= 1 \\ 2! &= 2 \cdot 1 = 2 \cdot 1! \\ 3! &= 3 \cdot 2 \cdot 1 = 3 \cdot 2! \\ 4! &= 4 \cdot 3 \cdot 2 \cdot 1 = 4 \cdot 3! \\ \dots \end{aligned}$$

así:

$$5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 5 \cdot 4! = 120$$

de modo que una definición recursiva de la función factorial n es:

$$n! = n \cdot (n - 1)! \quad \text{para } n > 1$$

El factorial de un entero n , mayor o igual a 0, se puede calcular de modo *iterativo* (no recursivo), teniendo presente la definición de $n!$ del modo siguiente:

$$\begin{array}{ll} n! = 1 & \text{si } n = 0 \\ n! = n \cdot (n - 1)! & \text{si } n > 0 \end{array}$$

El algoritmo que resuelve el factorial de forma iterativa de un entero n , mayor o igual que 0, se puede calcular utilizando un bucle `for`:

```
var
entero: contador
    real: factorial
inicio
    ...
factorial ← 1;
desde contador ← n hasta 1 decremento 1
    factorial ← factorial * contador
fin_desde
fin
```

En el caso de implementar una función se requerirá una sentencia de retorno que devuelva el valor del factorial, tal como

```
devolver(factorial)
```

El algoritmo que resuelve la función de modo *recursivo* ha de tener presente una condición de salida. Así, en el caso del cálculo de $6!$, la definición es $6! = 6 \times 5!$ y $5!$ de acuerdo a la definición es $5 \times 4!$. Este proceso continúa hasta que $1! = 1 \times 0!$ por definición. El método de definición de una función en términos de sí misma se llama en matemáticas una definición **inductiva** y conduce naturalmente a una implementación recursiva. El caso base de $0! = 1$ es esencial dado que se detiene, potencialmente, una cadena de llamadas recursivas. Este caso base o condición de salida deben fijarse en cada caso de una solución recursiva. El algoritmo que resuelve $n!$ de modo recursivo se apoya en la definición siguiente:

$n! = 1$	si $n = 0$
$n! = n * (n - 1) * (n - 2) * \dots * 1$	si $n > 0$

en consecuencia, el algoritmo mencionado que calcula el factorial será:

```

si ( $n = 0$ ) entonces
    fac  $\leftarrow 1$ 
si_no
    contador  $= n - 1$ 
    fac  $\leftarrow n * \text{fac}(\text{contador})$ 
fin_si

```

Otro pseudocódigo que resuelve la función factorial es:

```

si  $n = 1$  entonces
    fac  $\leftarrow n$ 
si_no
    fac  $\leftarrow n * \text{fac}(n - 1)$ 
fin_si

```

Así una función recursiva de factorial es:

```

entero función factorial(E entero: n)
    inicio
        si ( $n = 1$ ) entonces
            devolver (1)
        si_no
            devolver ( $n * \text{factorial}(n - 1)$ )
        fin_si
    fin_función

```

Nota

Dado que el valor de un factorial de un número entero aumenta considerablemente a medida que aumenta el valor de n , es conveniente en el diseño del algoritmo definir el tipo de dato a devolver por la función como un valor real, al objeto de no tener problema de desbordamiento cuando traduzca a un código fuente en un lenguaje de programación.

EJEMPLO 14.2

Deducir la definición recursiva del producto de números naturales.

El producto $a * b$, donde a y b son enteros positivos, tiene dos soluciones.

Solución iterativa

$$a * b = \underbrace{a + a + a + \dots + a}_{b \text{ veces}}$$

Solución recursiva

$$\begin{array}{ll} a * b = a & \text{si } b = 1 \\ a * b = a * (b - 1) + a & \text{si } b > 1 \end{array}$$

Así, por ejemplo, 7×3 será:

$$7 * 3 = 7 * 2 + 7 = 7 * 1 + 7 + 7 = 7 + 7 + 7 = 21$$

En pseudocódigo se tiene:

```

entero funcion producto (E entero, a, b)
inicio
    si b = 1 entonces
        devolver (a)
    si no
        devolver (a*producto (a, b-1))
    fin_si
fin

```

EJEMPLO 14.3

Definir la naturaleza de la serie de Fibonacci: 0, 1, 1, 2, 3, 5, 8, 13, 21, ...

Se observa en esta serie que comienza con 0 y 1, y tiene la propiedad de que cada elemento es la suma de los dos elementos anteriores, por ejemplo:

$$\begin{aligned}
 0 + 1 &= 1 \\
 1 + 1 &= 2 \\
 2 + 1 &= 3 \\
 3 + 2 &= 5 \\
 5 + 3 &= 8 \\
 \dots
 \end{aligned}$$

Entonces se puede decir que:

```

fibonacci(0) = 0
fibonacci(1) = 1
...
fibonacci(n) = fibonacci(n - 1) + fibonacci(n - 2)

```

y la definición recursiva será:

```

fibonacci(n) = n      si n = 0      o   n = 1
fibonacci(n) = fibonacci(n - 1) + fibonacci(n - 2)  si n > = 2

```

Obsérvese que la definición recursiva de los números de fibonacci es diferente de las definiciones recursivas del factorial de un número y del producto de dos números. Así, por ejemplo, simplificando el nombre de la función por fib

```
fib(6) = fib(5) + fib(4)
```

o lo que es igual, fib(6) ha de aplicarse en modo recursivo dos veces, y así sucesivamente. Las funciones iterativa y recursiva implementadas en Java son

```

public class Fibonacci
{
    //Fibonacci iterativo
    public static long fibonacci(int n)
    {
        long f = 0, fsig = 1;
        for (int i = 0; i < n; i++)
        {
            long aux = fsig;

```

```

        fsig += f;
        f = aux;
    }
    return(f);
}

//Fibonacci recursivo
public static long fibonaccir(int n)
{
    // si n es menor que 0 devuelve -1 como señal de error
    if (n < 0)
        return -1;
    // especificar else no es necesario, ya que
    // cuando se ejecuta return se retorna a la sentencia llamadora
    // y la siguiente instrucción ya no se ejecuta
    if (n == 0)
        return(0);
    else
        if (n == 1)
            return(1);
        else
            return(fibonaccir(n-1)+fibonaccir(n-2));
}

public static void main(String[] args)
{
    System.out.println("Fibonacci_iterativo("+8+")=="+fibonaccii(8));
    System.out.println("Fibonacci_recursivo("+8+")=="+fibonaccii(8));
}
}

```

14.2. RECURSIVIDAD DIRECTA E INDIRECTA

En **recursión directa** el código del subprograma recursivo F contiene una sentencia que invoca a F, mientras que en **recursión indirecta** el subprograma F invoca al subprograma G que invoca a su vez al subprograma P, y así sucesivamente hasta que se invoca de nuevo al subprograma F.

Si una función, procedimiento o método se invoca a sí misma, el proceso se denomina **recursión directa**; si una función, procedimiento o método puede invocar a una segunda función, procedimiento o método que a su vez invoca a la primera, este proceso se conoce como *recursión indirecta o mutua*.

Un requisito para que un algoritmo recursivo sea correcto es que no genere una secuencia infinita de llamadas sobre sí mismo. Cualquier algoritmo que genere una secuencia de este tipo no puede terminar nunca. En consecuencia, la definición recursiva debe incluir un **componente base** (*condición de salida*) en el que $f(n)$ se defina directamente (es decir, no recursivamente) para uno o más valores de n .

Debe existir una “forma de salir” de la secuencia de llamadas recursivas. Así en la función $f(n) = n!$ para n entero

$$f(n) \begin{cases} 1 & n \leq 1 \\ n*f(n - 1) & n > 1 \end{cases}$$

la condición de salida o base es $f(n) = 1$ para $n \leq 1$.

En el caso de la serie de Fibonacci

$$F_0 = 0, \quad F_1 = 1, \quad F_n = F_{n-1} + F_{n-2} \text{ para } n > 1.$$

$F_0 = 0$ y $F_1 = 1$ constituyen el componente base o condiciones de salida y $F_n = F_{n-1} + F_{n-2}$ es el componente recursivo.

C++ permite escribir funciones recursivas. Una función recursiva correcta debe incluir un componente base o condición de salida.

PROBLEMA 14.1

Escribir una función recursiva en C++ que calcule el factorial de un número n y un programa que maneje dicha función.

Recordemos que

$$\begin{array}{ll} n! = 1 & \text{si } n = 0 \\ n! = n * (n - 1)! & \text{si } n \geq 1 \end{array}$$

La función recursiva que calcula $n!$

```
int Factorial (int n)
{
    // cálculo de n!
    if (n <= 1)
        return 1;
    return n * Factorial(n - 1);
}
```

En el algoritmo anterior se ha considerado que el valor resultante es de tipo entero; sin embargo, observe la secuencia de valores de la función factorial.

n	$n!$
0	1
1	1
2	2
3	6
4	24
5	120
6	720
7	5040
8	40320
9	362880
10	3628800

Como se puede ver, los valores crecen muy rápidamente, y para $n = 8$ ya sobrepasa el valor normal del mayor entero manejado en computadoras de 16 bits (32767). Por consiguiente, será preciso cambiar el tipo de dato devuelto que ha de ser float, double, unsigned int, long, etc. En consecuencia, el programa fac.cpp que calcula el factorial de un número puede ser:

```
//Programa fac.cpp
#include <iostream>
```

```
#using namespace std;

// en C++ las funciones han de ser declaradas
// o definidas antes de su uso

double Factorial (int n);

int main()
{
    // declaración
    int num;
    // escribir ('Por favor introduzca un número: ')
    cout << "Por favor introduzca un número: ";
    // leer(num)
    cin >> num;
    // escribir (num, ' != ', Factorial(num); endl)
    // endl significa salto de línea
    cout << num << " != " << Factorial(num) << endl;
    // devolver éxito, es decir ejecución válida
    return 0;
}

// definición de la función Factorial
// el paso de parámetros de tipo simple por defecto es por valor

double Factorial (int n)
{
    if (n <=1)
        return (1);
    else
        return (n * Factorial(n - 1));
    // en C++ los paréntesis en la sentencia return son opcionales
}
```

Una variante de este programa podría ser el cálculo del factorial correspondiente a los números naturales 0 a 10. Para ello bastaría sustituir la función `main` anterior por una función tal como ésta e incluyendo una llamada al archivo `#include <iomanip>`

```
// Programa principal

int main()
{
    int i;
    for (i = 0; i<=10; i++)
        cout << setw(2) << i << " != " << Factorial(i) << endl;
        // setw da formato a la salida y establece la anchura del
        // campo a 2
    return 0;
}
```

PROBLEMA 14.2

Escribir una función de Fibonacci de modo recursivo y un programa que manipule dicha función, de modo que calcule el valor del elemento de acuerdo a la posición ocupada en la serie.

Nota

El código fuente de este programa se ha escrito en lenguaje C++.

```
// Función de Fibonacci: fibo.cpp
#include <iostream>
using namespace std;

long fibonacci (long n);

int main()
{
    long resultado, num;

    cout << "Introduzca un entero : ";
    cin >> num;
    resultado = fibonacci (num);
    cout << "El valor de Fibonacci(" << num << ") = " << resultado << endl;
    return 0;
}

// definición recursiva de la función de fibonacci
long fibonacci(long n)
{
    if ((n == 0) || (n == 1))
        return n;
    // no es necesaria la especificación de else, pero se puede poner
    return fibonacci(n - 1) + fibonacci (n - 2);
}
```

La salida resultante de la ejecución del programa anterior:

```
Introduzca un entero : 2
El valor de Fibonacci (2) = 1

Introduzca un entero : 20
El valor de Fibonacci (20) = 832040
```

14.2.1. Recursividad indirecta

La recursividad indirecta se produce cuando un subprograma llama a otro, que eventualmente terminará llamando de nuevo al primero. El programa ALFABETO.CPP visualiza el alfabeto utilizando recursión mutua o indirecta.

Nota

El código fuente de este programa también se ha escrito en lenguaje C++.

```
// Listado ALFABETO.CPP
#include <iostream>
#include <stdio.h>
```

```

using namespace std;
// A y B equivalen a procedimientos
void A(int c);
void B(int c);

int main()
{
    A('Z');
    cout << endl;
    return 0;
}

void A(int c)
{
    if (c > 'A')
        B(c);
    putchar(c);
}

void B(int c)
{
    A(--c);
}

```

El programa principal llama a la función recursiva `A()` con el argumento '`Z`' (la última letra del alfabeto). La función `A` examina su parámetro `c`. Si `c` está en orden alfabético después que '`A`', la función llama a `B()`, que inmediatamente llama a `A()`, pasándole un parámetro predecesor de `c`. Esta acción hace que `A()` vuelva a examinar `c`, y nuevamente una llamada a `B()`, hasta que `c` sea igual a '`A`'. En este momento, la recursión termina ejecutando `putchar()` veintiséis veces y visualizando el alfabeto, carácter a carácter.

14.2.2. Condición de terminación de la recursión

Cuando se implementa un subprograma recursivo será preciso considerar una condición de terminación, ya que en caso contrario el subprograma continuaría indefinidamente llamándose a sí mismo y llegaría un momento en que la memoria se podría agotar. En consecuencia, sería necesario establecer en cualquier subprograma recursivo la condición de parada que termine las llamadas recursivas y evitar indefinidamente las llamadas. Así, por ejemplo, en el caso de la función `factorial`, definida anteriormente, la condición de salida puede ser cuando el número sea 1 o 0, ya que en ambos casos el factorial es 1.

```

real función factorial(E entero: n)
inicio
    si(n = 1) o (n = 0) entonces
        devolver (1)
    si_no
        devolver (n * factorial (n - 1))
    fin_si
fin_función

```

14.3. RECUSIÓN VERSUS ITERACIÓN

En las secciones anteriores se han estudiado varias funciones que se pueden implementar fácilmente o bien de modo recursivo o bien de modo iterativo. En esta sección compararemos los dos enfoques y examinaremos las razones por las que el programador puede elegir un enfoque u otro según la situación específica.

Tanto la iteración como la recursión se basan en una estructura de control: *la iteración utiliza una estructura repetitiva y la recursión utiliza una estructura de selección*. La iteración y la recursión implican ambas repetición: la iteración utiliza explícitamente una estructura repetitiva mientras que la recursión consigue la repetición mediante llamadas repetidas. La iteración y recursión implican cada una un test de terminación (*condición de salida*). La iteración termina cuando la condición del bucle no se cumple mientras que la recursión termina cuando se reconoce un caso base o la condición de salida se alcanza.

La recursión tiene muchas desventajas. Se invoca repetidamente al mecanismo de recursividad y en consecuencia se necesita tiempo suplementario para realizar las mencionadas llamadas.

Esta característica puede resultar cara en tiempo de procesador y espacio de memoria. Cada llamada de una función recursiva produce que otra copia de la función (realmente sólo las variables de función) sea creada; esto puede consumir memoria considerable. Por el contrario, la iteración se produce dentro de una función, de modo que las operaciones suplementarias de las llamadas a la función y asignación de memoria adicional son omitidas.

En consecuencia, ¿cuáles son las razones para elegir la recursión? La razón fundamental es que existen numerosos problemas complejos que poseen naturaleza recursiva y, en consecuencia, son más fáciles de implementar con algoritmos de este tipo. Sin embargo, en condiciones críticas de tiempo y de memoria, es decir, cuando el consumo de tiempo y memoria sean decisivos o concluyentes para la resolución del problema, la solución a elegir debe ser, normalmente, la iterativa.

Cualquier problema que se puede resolver recursivamente se puede resolver también iterativamente (no recursivamente). Un enfoque recursivo se elige normalmente con preferencia a un enfoque iterativo cuando el enfoque recursivo es más natural para la resolución del problema y produce un programa más fácil de comprender y depurar. Otra razón para elegir una solución recursiva es que una solución iterativa puede no ser clara ni evidente.

Consejo de programación

Se ha de evitar utilizar recursividad en situaciones de rendimiento crítico o exigencia de altas prestaciones en tiempo y memoria, ya que las llamadas recursivas emplean tiempo y consumen memoria adicional.

Consejo de carácter general

Si una solución de un problema se puede expresar iterativa o recursivamente con igual facilidad, es preferible la solución iterativa, ya que se ejecuta más rápidamente (no existen llamadas adicionales a funciones que consumen tiempo de proceso) y utiliza menos memoria (la pila necesaria para almacenar las sucesivas llamadas necesarias en la recursión). Hay veces, sin embargo, que, pese a todo, es preferible la solución recursiva.

EJEMPLO 14.4

La función factorial de un número ya expuesta anteriormente ofrece un ejemplo claro de comparación entre funciones definidas de modo iterativo o modo recursivo y, a continuación, se muestra su implementación en C#.

El factorial $n!$, de un número n era

```
0! = 1  
n! = n * (n - 1)! para n > 0
```

Solución recursiva

```
// código en C#  
public class Prueba1
```

```
{
    // factorial recursivo
    // Precondición      n está definido y n >= 0
    // Postcondición     ninguna
    // Devuelve          n!
    public static long factorial(int n)
    {
        if (n < 0)
            return -1;
        if (n == 0)
            return 1;
        else
            return n * factorial(n - 1);
    }
    public static void main()
    {
        // escribir(factorial(4))
        System.Console.WriteLine(factorial(4));
    }
}
```

Solución iterativa

```
// código en C#
public class Prueba2
{
    // factorial iterativo
    // Precondición      n está definido y n >= 0
    // Postcondición     ninguna
    // Devuelve          n!
    public static long factorial(int n)
    {
        if (n < 0)
            return -1;
        long fact = 1;
        while (n > 0)
        {
            fact = fact * n;
            n = n - 1;
        }
        return fact;
    }
    public static void main()
    {
        // escribir(factorial(4))
        System.Console.WriteLine(factorial(4));
    }
}
```

Directrices en la toma de decisión iteración/recursión

1. Considérese una solución recursiva sólo cuando una solución iterativa *sencilla* no sea posible.
2. Utilícese una solución recursiva sólo cuando la ejecución y eficiencia de la memoria de la solución esté dentro de límites aceptables considerando las limitaciones del sistema.

3. Si son posibles las dos soluciones, iterativa y recursiva, la solución recursiva siempre requerirá más tiempo y espacio debido a las llamadas adicionales que se realizan.
4. En ciertos problemas, la recursión conduce naturalmente a soluciones que son mucho más fáciles de leer y comprender que su correspondiente iterativa. En estos casos los beneficios obtenidos con la claridad de la solución suelen compensar el coste extra (en tiempo y memoria) de la ejecución de un programa recursivo.

14.4. RECUSIÓN INFINITA

La iteración y la recursión pueden producirse infinitamente. Un bucle infinito ocurre si la prueba o test de continuación de bucle nunca se vuelve falsa; una recursión infinita ocurre si la etapa de recursión no reduce el problema en cada ocasión de modo que converja sobre el caso base o condición de salida.

En realidad la **recursión infinita** significa que cada llamada recursiva produce otra llamada recursiva y ésta a su vez otra llamada recursiva y así para siempre. En la práctica dicho código se ejecutará hasta que la computadora agota la memoria disponible y se produzca una terminación anormal del programa.

El flujo de control de un algoritmo recursivo requiere tres condiciones para una terminación normal:

- Un test para detener (o continuar) la recursión (*condición de salida o caso base*).
- Una llamada recursiva (para continuar la recursión).
- Un caso final para terminar la recursión.

EJEMPLO 14.5

Se desea calcular la suma de los primeros N enteros positivos.

La función no recursiva que realiza la tarea solicitada es:

```
entero función CalculoSuma (E entero: N)
var
    entero: suma, i
inicio
    suma ← 0
    desde i ← 1 hasta N hacer
        suma ← suma + i
    fin_desde
    devolver (suma)
fin_función
```

La función `CalculoSuma` implementada recursivamente requiere la definición previa de la suma de los primeros N enteros matemáticamente en forma recursiva, tal como se muestra a continuación:

$$\text{suma}(N) = \begin{cases} 1 & \text{si } N = 1 \\ N + \text{suma}(N-1) & \text{en caso contrario} \end{cases}$$

La definición anterior significa que si N es 1, entonces la función `suma(N)` toma el valor 1. En caso contrario, significa que la función `suma(N)` toma el valor resultante de la suma de N y el resultado de `suma(N-1)`. Por ejemplo, la función `suma(5)` se evalúa tal como se muestra en la Figura 14.1 de la página siguiente.

El pseudocódigo fuente de la función recursiva `suma` es:

```
entero función suma (E entero: n)
inicio
    // test para parar o continuar (condición de salida)
    si (n = 1) entonces
        devolver (1)
```

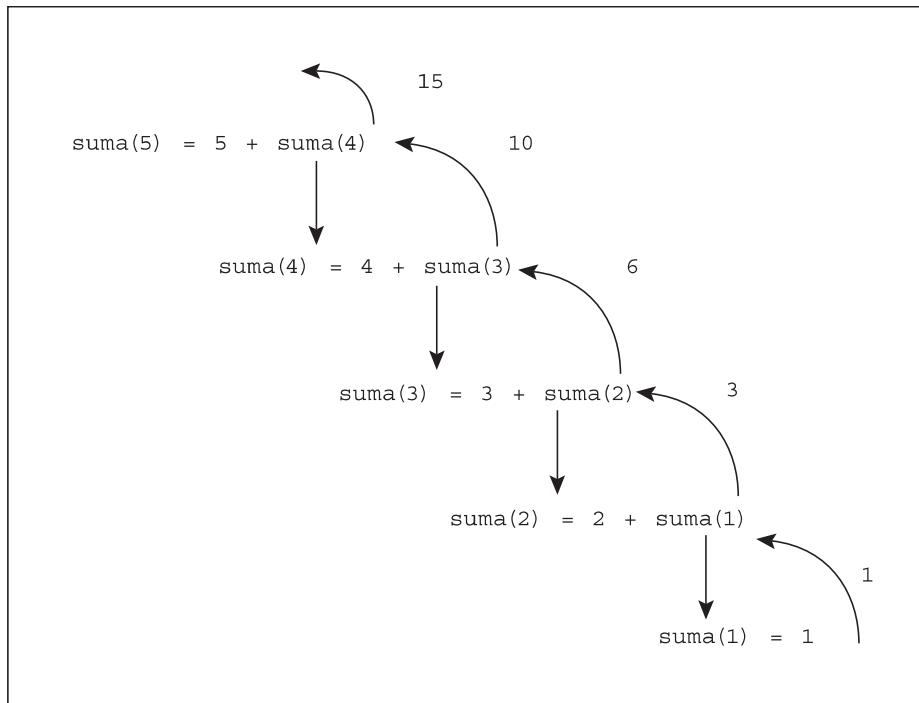


Figura 14.1. Secuencia de llamadas recursivas que evalúan la función Suma (N) (en el ejemplo Suma (4)).

```

//caso final - se detiene la recursión
si_no
  devolver(n + suma (n - 1))
//caso recursivo
//la recursión continúa con una llamada recursiva
fin_si
fin_función
  
```

y el código fuente en Turbo Pascal es:

```

program Sumas;
{solución interactiva}
function CalculoSuma (N: integer): integer;
var
  suma, i: integer;
begin
  suma := 0;
  for i:= 1 to N do
    suma := suma + i;
  CalculoSuma := suma
end;

{solución recursiva}
function suma(n: integer): integer;
begin
  { test para parar o continuar (condición de salida)}
  if n = 1 then
    suma := 1
  else
    suma := n + suma(n - 1)
end;
  
```

```

{ caso final - se detiene la recursión}
    la recursión continúa con una llamada recursiva }
else
    suma := n + suma (n - 1);
end;
begin
writeln('Suma recursiva ', suma(4))
writeln('Suma iterativa ', CalculoSuma(4))
end

```

Cuando se realizan llamadas recursivas se han de pasar argumentos diferentes de los parámetros de entrada; así, en el ejemplo de la función `suma`, el argumento que se pasa en la función recursiva es $n - 1$ y el parámetro es n . La Figura 14.2 muestra el flujo de control de la función `suma` de modo recursivo.

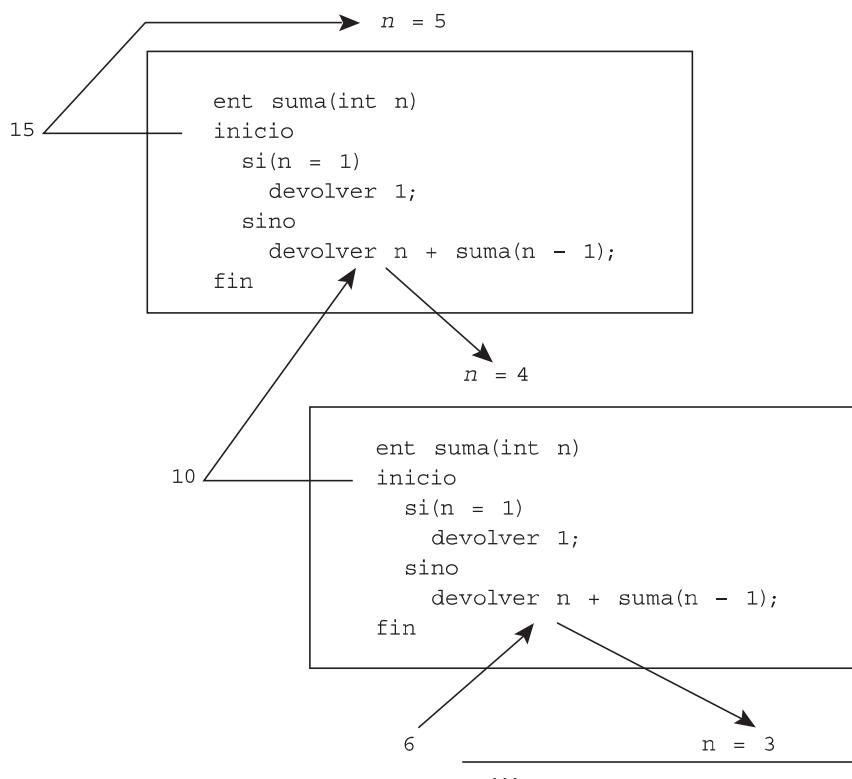


Figura 14.2. Flujo de control de la función `suma` recursiva.

PROBLEMA 14.2

Deducir cuál es la condición de salida de la función `mcd()` que calcula el mayor divisor común de dos números enteros $b1$ y $b2$ (el **mcd**, máximo común divisor, es el entero mayor que divide a ambos números) y un programa que la manipule.

El **mcd** de los enteros $b1$ y $b2$ se define como el entero mayor que divide a ambos números. El **mcd** no está definido si $b1$ y $b2$ son cero. Los valores negativos de $b1$ y $b2$ se sustituyen por su valores absolutos. Supongamos dos números 6 y 124; el procedimiento clásico de obtención del **mcd** es la realización de divisiones sucesivas, se comienza dividiendo ambos números (124 entre 6) si el resto no es 0, se divide el número menor por el resto y así sucesivamente hasta que el resto sea 0.

$$\begin{array}{r}
 124 \quad | \quad 6 \\
 04 \quad | \quad 20 \\
 \hline
 6 \quad | \quad 4 \\
 2 \quad | \quad 1
 \end{array}
 \qquad
 \begin{array}{r}
 4 \quad | \quad 2 \\
 0 \quad | \quad 2
 \end{array}
 \qquad
 \begin{array}{c}
 \searrow \\
 (mcd = 2)
 \end{array}
 \qquad
 \begin{array}{r}
 20 \quad | \quad 1 \quad | \quad 2 \\
 \hline
 124 \quad | \quad 6 \quad | \quad 4 \quad | \quad 2 \\
 \hline
 4 \quad | \quad 2 \quad | \quad 0 \quad | \quad
 \end{array}
 \qquad
 mcd = 2$$

En el caso de 124 y 6, el **mcd** es 2. Suponga ahora que los números son $b1=18$ y $b2=45$

$$\begin{array}{r}
 18 \quad | \quad 45 \\
 18 \quad | \quad 0 \\
 \hline
 45 \quad | \quad 18 \\
 09 \quad | \quad 2
 \end{array}
 \qquad
 \begin{array}{r}
 18 \quad | \quad 9 \\
 0 \quad | \quad 2
 \end{array}
 \qquad
 \begin{array}{c}
 (mcd = 9)
 \end{array}$$

El **mcd** de 18 y 45 es 9. En consecuencia, la condición de salida es que el resto sea cero. Por tanto:

1. Si $b2$ es cero, la solución es $b1$.
2. Si $b2$ no es cero, la solución es $mcd(b2, b1 \bmod b2)$.

El código fuente de la función es:

```

entero función mcd(ENTERO: b1, b2)
inicio
    si (b2 <> 0) entonces // condición de salida
        devolver (mcd (b2, b1 mod b2))
    si_no
        devolver (b1)
    fin_si
fin_función
  
```

Un programa en C++ que gestiona la función `mcd` es `mcd.cpp`.

```

#include <iostream>
using namespace std;

// Programa mcd.cpp, escrito en lenguaje C++
int mcd(int n, int m);

void main()
{
    // datos locales
    int m, n;

    cout << "Introduzca dos enteros positivos : ";
    cin >> m >> n;
    cout << endl;
    cout << "El máximo común divisor es : " << mcd(m, n) << endl;
}

// Función recursiva mcd
int mcd(int n,int m)
// devuelve el máximo común divisor de m y n
  
```

```

{
  if (m != 0) // condición de salida
    return mcd(m, n % m);
  else
    return n;
} // final de mcd

```

Al ejecutarse el programa se produce la siguiente salida:

```

Introduzca dos enteros positivos : 6   40
El máximo común divisor es : 2

```

El código de la función recursiva en Turbo Pascal es

```

function mcd (n, m: integer): integer;
begin
  if m <> 0 then
    mcd := mcd(m, n mod m)
  else
    mcd := n
end;

```

14.5. RESOLUCIÓN DE PROBLEMAS COMPLEJOS CON RECURSIVIDAD

Muchos problemas de computadora tienen una formulación simple y elegante que se traduce directamente a código recursivo. En esta sección se describen una serie de ejemplos que incluyen problemas clásicos resueltos mediante recursividad. Entre ellos se destacan problemas matemáticos, las Torres de Hanoi, método de búsqueda binaria, ordenación rápida, árboles de expresión, etc. Explicamos con detalle algunos de ellos.

14.5.1. Torres de Hanoi

Este juego (un algoritmo clásico) tiene sus orígenes en la cultura oriental y en una leyenda sobre el Templo de Brahma. El problema en cuestión supone la existencia de 3 varillas o postes en los que se alojaban discos, cada disco es ligeramente inferior en diámetro al que está justo debajo de él, y pretende determinar los movimientos necesarios para trasladar los discos de una varilla a otra cumpliendo las siguientes reglas:

- En cada movimiento sólo puede intervenir un disco.
- Nunca puede quedar un disco sobre otro de menor tamaño.

La Figura 14.3 ilustra el problema. Los cuatro discos situados en la varilla I se desean trasladar a la varilla F conservando la condición de que cada disco sea ligeramente inferior en diámetro al que tiene situado debajo de él.

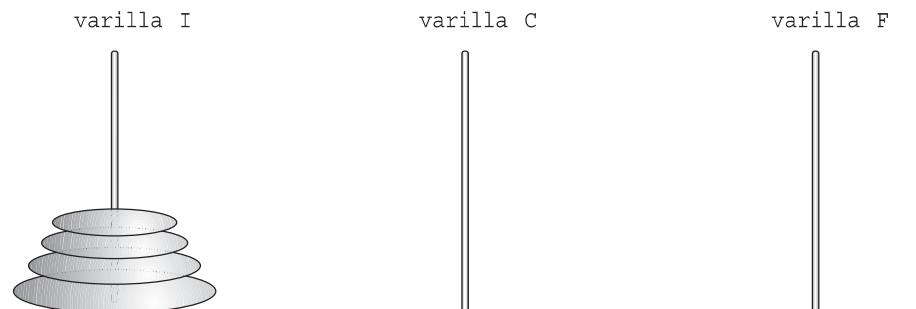


Figura 14.3.

Este problema es claramente recursivo, pues mover cuatro discos de la varilla I a la F consiste en trasladar los tres discos superiores de la varilla origen a otra considerada como auxiliar (C). Figuras 14.3 y 14.4,

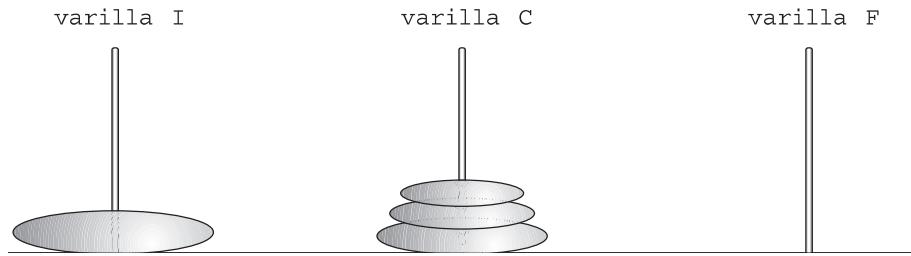


Figura 14.4.

trasladar el disco más grande de la varilla origen al destino (de I a F). Figuras 14.4 (antes) y 14.5 (después).

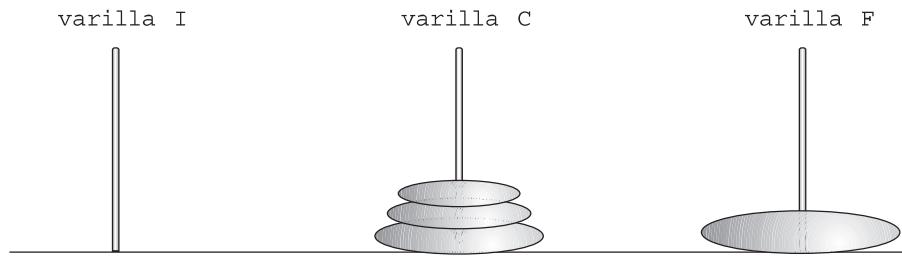


Figura 14.5.

y pasar los tres de la varilla auxiliar al destino. Figura 14.5 (antes) y 14.6 (después).

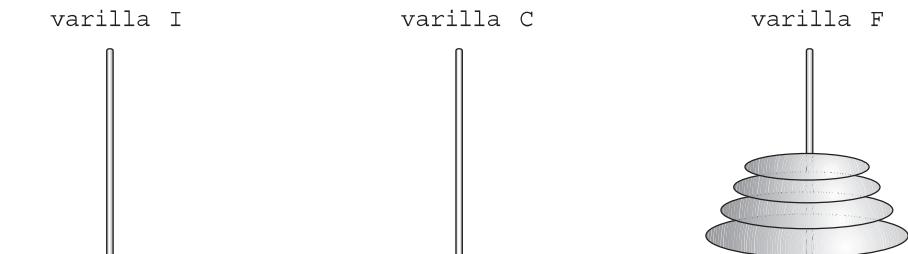
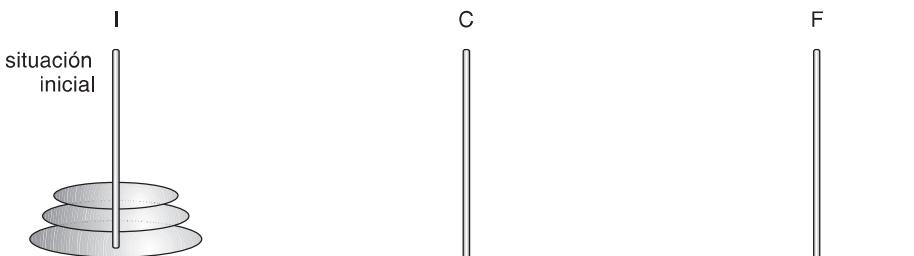
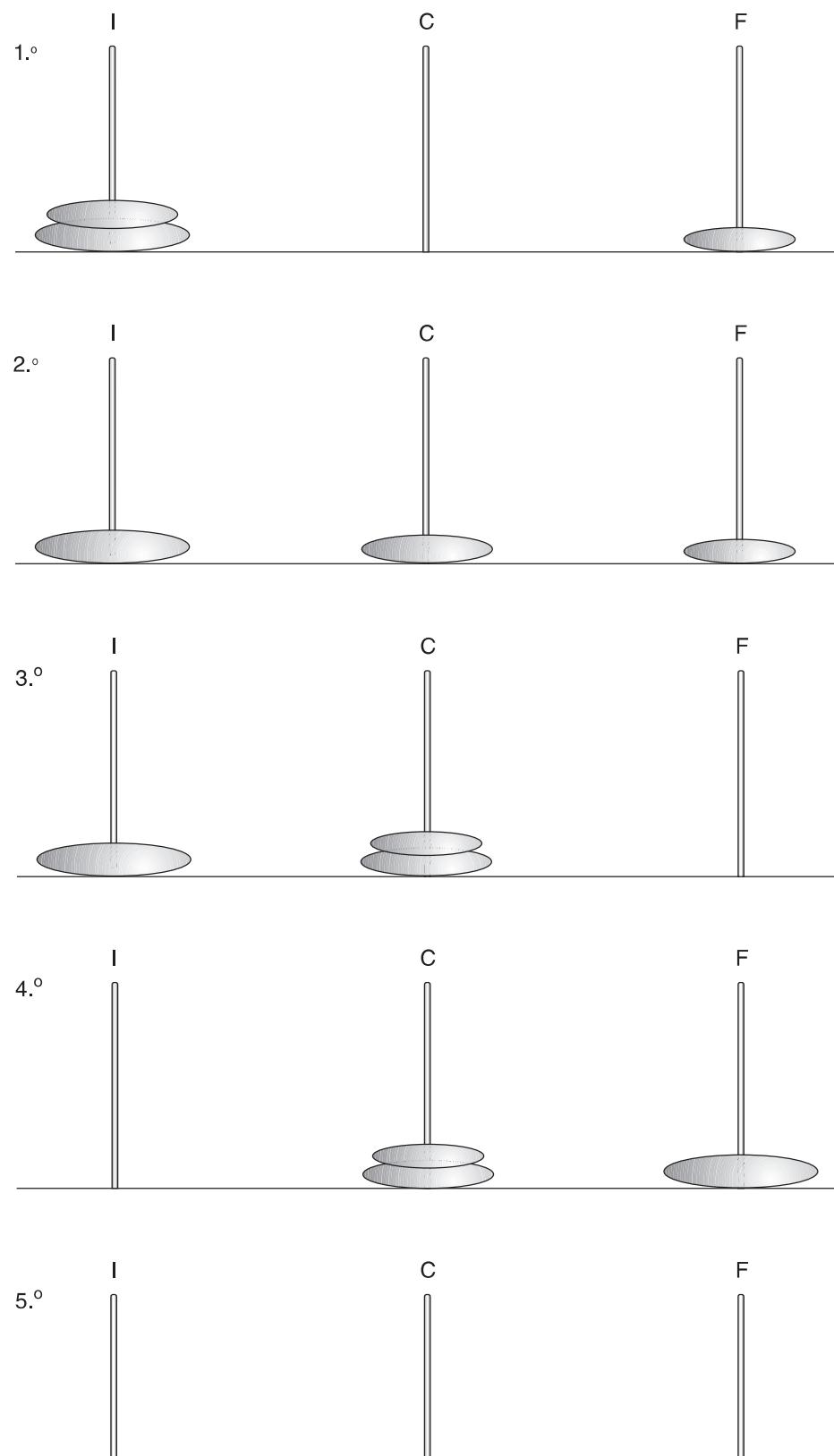


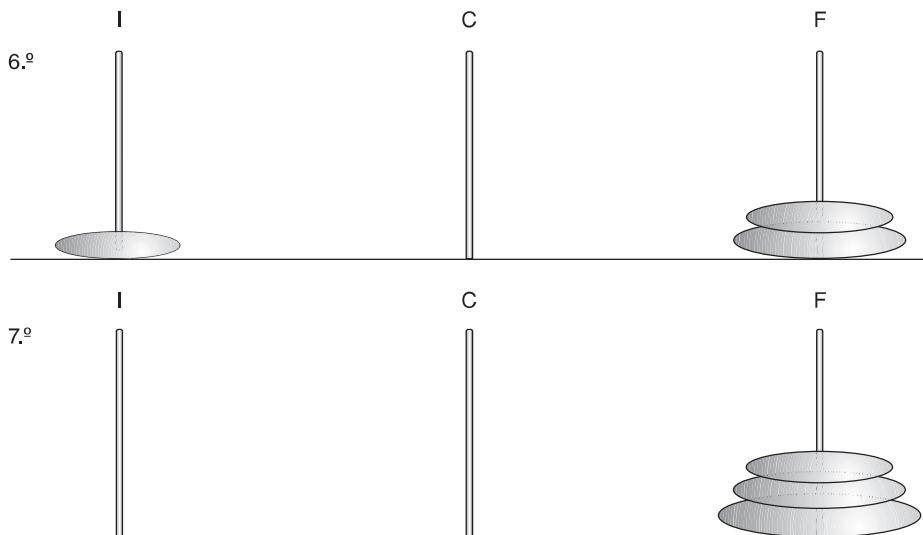
Figura 14.6.

Nuevamente se observa que mover los tres discos superiores de un origen a un destino requiere mover dos de origen a auxiliar, uno de origen a destino y dos de auxiliar a destino. Por último, trasladar dos discos de origen a destino implica trasladar uno de origen a auxiliar, otro de origen a destino y completar la operación pasando el de la varilla auxiliar a destino.

Los movimientos que se realizarían detallados gráficamente para el caso de $N = 3$, son







Diseño del algoritmo

El algoritmo se escribe generalizando para n discos y tres varillas. La función de Hanoi declara las varillas o postes como objetos cadena. En la lista de parámetros, el orden de las variables o varillas es:

varinicial varcentral varfinal

lo que implica que se están moviendo discos desde la varilla inicial a la final utilizando la varilla central como auxiliar para almacenar los discos. Si $n = 1$ se tiene la condición de parada, ya que se puede manejar moviendo el único disco desde la varilla inicial a la varilla final. El algoritmo sería el siguiente:

1. **Si n es 1**
 - 1.1 Mover el disco 1 de varinicial a varfinal
2. **Si_no**
 - 1.2 Mover $n - 1$ discos desde varinicial hasta la varilla auxiliar utilizando varfinal
 - 1.3 Mover el disco n desde varinicial a varfinal
 - 1.4 Mover $n - 1$ discos desde la varilla auxiliar o central a varfinal utilizando la varilla inicial.

Es decir, si n es 1, se alcanza la condición de salida o terminación del algoritmo. Si n es mayor que 1, las etapas recursivas 1.2, 1.3 y 1.4 son tres subproblemas más pequeños, que aproximan a la condición de salida.

Las Figuras 14.7, 14.8 y 14.9 muestran el algoritmo anterior:

Etapa 1: Mover $n - 1$ discos desde varilla inicial (I).



Figura 14.7.

Etapa 2: Mover un disco desde I a F.



Figura 14.8.

Etapa 3: Mover $n - 1$ discos desde varilla central (C).

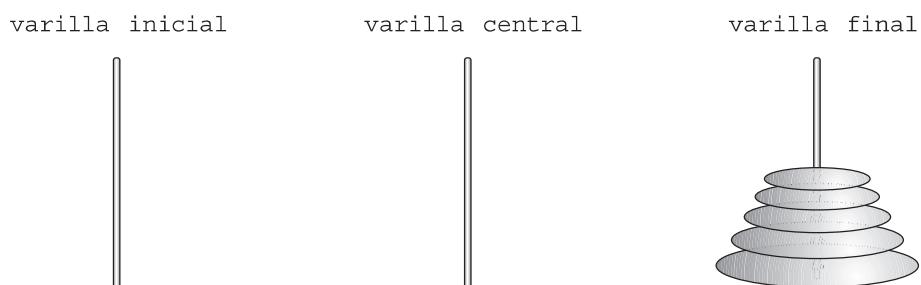


Figura 14.9.

La primera etapa en el algoritmo mueve $n - 1$ discos desde la varilla inicial a la varilla central utilizando la varilla final. Por consiguiente, el orden de parámetros en la llamada a la función recursiva es *varinicial*, *varfinal* y *varcentral*.

```
// utilizar varfinal como almacenamiento auxiliar
Hanoi(n - 1, varinicial, varfinal, varcentral);
```

La segunda etapa mueve simplemente el disco mayor desde la varilla inicial a la varilla final:

```
escribir "mover", varinicial, "a", varfinal, fin de línea;
```

La tercera etapa del algoritmo mueve $n - 1$ discos desde la varilla central a la varilla final utilizando *varinicial* para almacenamiento temporal. Por consiguiente, el orden de parámetros en la llamada a la función recursiva es: *varcentral*, *varinicial* y *varfinal*.

```
// utilizar varinicial como almacenamiento auxiliar
Hanoi(n - 1, varcentral, varinicial, varfinal);
```

Implementación de las Torres de Hanoi en C++

La implementación del algoritmo se apoya en los nombres de las tres varillas o alambres "inicial", "central" y "final" que se pasan como parámetros a la función. El programa comienza solicitando al usuario que introduzca el número de discos N . Se llama a la función recursiva *Hanoi* para obtener un listado de los movimientos que transferirán los N discos desde la varilla "inicial" a la varilla "final". El algoritmo requiere $2^N - 1$ movimientos. Para el caso de 10 discos, el juego requerirá 1.023 movimientos. En el caso de prueba para $N = 3$, el número de movimientos es $2^3 - 1 = 7$.

```
// archivo Torres.cpp
// función recursiva Torres de Hanoi

void Hanoi(char varinicial, char varfinal, char varcentral, int n)
{
    if (n == 1)
        cout << " Mover disco 1 de varilla " << varinicial << " a varilla "
            << varfinal << endl;
    else
    {
        Hanoi(varinicial, varcentral, varfinal, (n - 1));
        cout << " Mover disco " << n << " desde varilla " << varinicial <<
            " a varilla " << varfinal << endl;
        Hanoi(varcentral, varfinal, varinicial, n - 1);
    }
}
```

Una ejecución de la función para el caso de mover tres discos desde las varillas A a C tomando la varilla B como varilla central o auxiliar, se puede conseguir con la siguiente sentencia:

```
Hanoi ('A', 'C', 'B', 3);
```

que resuelve el problema de tres discos desde A a C. La salida generada sería:

```
Mover disco 1 de varilla A a varilla C
Mover disco 2 de varilla A a varilla B
Mover disco 1 de varilla C a varilla B
Mover disco 3 de varilla A a varilla C
Mover disco 1 de varilla B a varilla A
Mover disco 2 de varilla B a varilla C
Mover disco 1 de varilla A a varilla C
```

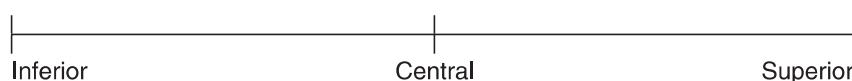
Consideraciones de eficiencia en las Torres de Hanoi

Es de destacar que la función `Hanoi` resolverá el problema de las Torres de Hanoi para cualquier número de discos. El problema de tres discos se resuelve en un total de $7 (2^3 - 1)$ llamadas a la función `Hanoi` mediante 7 movimientos de disco. El problema de cinco discos se resuelve con $31 (2^5 - 1)$ llamadas y 31 movimientos. En general, como ya se ha expresado anteriormente, el número de movimientos requeridos para resolver el problema de n discos es $2^n - 1$. Cada llamada a la función requiere la asignación e inicialización de un área local de datos en la memoria, por lo que el tiempo de computadora se incrementa exponencialmente con el tamaño del problema. Por estas razones, la ejecución del programa con un valor de n mayor que 10 requiere gran cantidad de prudencia para evitar desbordamientos de memoria y ralentización de tiempo.

14.5.2. Búsqueda binaria recursiva

Recordemos que la búsqueda binaria era aquel método de búsqueda de una clave especificada dentro de una lista o array ordenado de n elementos que realizaba una exploración de la lista hasta que se encontraba o no la coincidencia con la clave especificada. El algoritmo de búsqueda binaria se puede describir recursivamente.

Supóngase que se tiene una lista ordenada A con un límite inferior y un límite superior. Dada una clave (valor buscado) se comienza la búsqueda en la posición central de la lista (índice central).

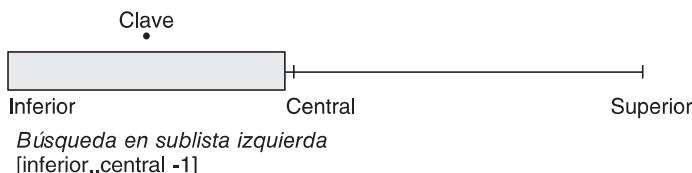


```
Central = (inferior + superior) div 2
```

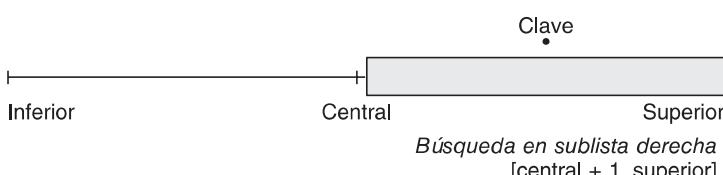
```
Comparar A[central] y clave
```

Si se produce coincidencia (se encuentra la clave), se tiene la condición de terminación que permite detener la búsqueda y devolver el índice central. Si no se produce la coincidencia (no se encuentra la clave), dado que la lista está ordenada, se centra la búsqueda en la “sublista inferior” (a la izquierda de la posición central) o en la “sublista derecha” (a la derecha de la posición central).

1. Si $\text{clave} < A[\text{central}]$, el valor buscado sólo puede estar en la mitad izquierda de la lista con elementos en el rango inferior a $\text{central} - 1$.



2. Si $\text{clave} > A[\text{central}]$, el valor buscado sólo puede entrar en la mitad derecha de la lista con elementos en el rango de índices, $\text{Central} + 1$ a Superior.



3. El proceso recursivo continúa la búsqueda en sublistas más y más pequeñas. La búsqueda termina o con éxito (*aparece la clave buscada*) o sin éxito (*no aparece la clave buscada*), situación que ocurrirá cuando el límite superior de la lista sea más pequeño que el límite inferior. La condición $\text{Inferior} > \text{Superior}$ será la condición de salida o terminación y el algoritmo devuelve el índice -1 .

En notación matemática y algorítmica se podría representar la búsqueda binaria de la siguiente forma:

```
BusquedaBR(inferior, superior, clave)      // BR, binaria recursiva
= { devolver no encontrada
    si inferior > superior
    devolver central
    si elemento[central] = clave
    devolver BusquedaBR(central + 1, superior, clave)
    si elemento[central] < clave
    devolver BusquedaBR(inferior, central - 1, clave)
    si elemento[central] > clave
```

en donde *central* es el punto central entre *inferior* y *superior*. Su codificación en Java podría ser:

```
public class Bbin
{
    private int busquedaBinaria(int[] a, int iz, int de, int c)
    {
        int central;

        if (de < iz)
            return(-1);
        else
        {
            central = (iz + de)/2;
            if (c < a[central])
                return(busquedaBinaria(a, iz, central - 1, c));
```

```

        else
            if (a[central] < c)
                return(búsquedaBinaria(a, central + 1, de, c));
            else
                return(central);
        }
    }

public int búsquedaB(int[] a, int c)
{
    // los arrays en Java comienzan con el subíndice 0
    // a.length se encuentra predefinido y devuelve
    // la longitud del array
    return(búsquedaBinaria(a, 0, a.length - 1, c));
}
}

```

14.5.3. Ordenación rápida (*QuickSort*)

El algoritmo conocido como *quicksort* (ordenación rápida) recibe su nombre de su autor, Tony Hoare. La idea del algoritmo es simple, se basa en la división en particiones de la lista a ordenar. El método es, posiblemente, el más pequeño de código, más rápido, más elegante y más interesante y eficiente de los algoritmos conocidos de ordenación.

El método se basa en dividir los n elementos de la lista a ordenar en tres partes o particiones: una partición *izquierda*, una partición *central* que sólo contiene un elemento denominado *pivote* o elemento de partición y una partición *derecha*. La partición o división se hace de tal forma que todos los elementos de la primera sublistas (partición izquierda) son menores que todos los elementos de la segunda sublistas (partición derecha). Las dos sublistas se ordenan entonces independientemente.

La lista se divide en particiones (sublistas) eligiendo uno de los elementos de la lista y se utiliza como *pivote* o *elemento de partición*. Si se elige una lista cualquiera con los elementos en orden aleatorio, se puede elegir cualquier elemento de la lista como pivote; por ejemplo, el primer elemento de la lista. Si la lista tiene algún orden parcial, que se conoce, se puede tomar otra decisión para el pivote. Idealmente, el pivote se debe elegir de modo que se divida la lista exactamente por la mitad, de acuerdo al tamaño relativo de las claves. Por ejemplo, si se tiene una lista de enteros de 1 a 10, 5 o 6 serían pivotes ideales, mientras que 1 o 10 serían elecciones “pobres” de pivotes.

Una vez que el pivote ha sido elegido, se utiliza para ordenar el resto de la lista en dos sublistas: una tiene todas las claves menores que el pivote y la otra en la que todos los elementos (claves) son mayores que o iguales que el pivote (o al revés). Estas dos listas parciales se ordenan recursivamente utilizando el mismo algoritmo; es decir, se llama sucesivamente al propio algoritmo *quicksort*. La lista final ordenada se consigue concatenando la primera sublista, el pivote y la segunda lista, en ese orden, en una única lista. La primera etapa de *quicksort* es la división o “particionado” recursivo de la lista hasta que todas las sublistas constan de sólo un elemento.

EJEMPLO 14.6

1. lista inicial 2 96 18 38 12 45 10 55 81 43 39
pivot elegido 39

2	10	18	38	12	39	96	55	81	43	45
<= pivote					↑	>= pivote				

2. lista inicial 13 81 92 43 65 31 57 26 75 0

0	13	92	43	65	31	57	26	75	81	
<= pivote					↑	>= pivote				

EJEMPLO 14.7 (Pivote: primer elemento de la lista)

1. Lista original

5	2	1	8	3	7	9
---	---	---	---	---	---	---

pivot elegido

5

sublista izquierdal, Izqda1 (elementos menores que 5)

2	1	3
---	---	---

sublista derechal, Dcha1 (elementos mayores o iguales a 5)

8	7	9
---	---	---

2. Sublista Izda1

2	1	3
---	---	---



sublista Izda2 1
sublista Dcha2 3

Sublista Izda1

Izda pivot2 Dcha

1	2	3
---	---	---

3. Sublista Dcha1

8	7	9
---	---	---



sublista Izda2 7
sublista Dcha2 9

Sublista Dcha1

Izda pivot3 Dcha

7	8	9
---	---	---

4. Lista ordenada final

Sublista izquierda

1 2 3

Pivot

5

Sublista derecha

7 8 9

El algoritmo *quicksort* requiere una estrategia de partición y la selección idónea del pivote. Las etapas fundamentales del algoritmo dependen del pivote elegido, aunque la estrategia de partición suele ser similar. La primera etapa en el algoritmo de partición es obtener el elemento pivote; una vez que se ha seleccionado se ha de buscar el sistema para situar en la sublista izquierda todos los elementos menores o iguales que el pivote y en la sublista derecha todos los elementos mayores que el pivote y dejar el pivote como separador de ambas sublistas.

EJEMPLO 14.8

Lista:

8 1 4 9 6 3 5 2 7 0

Etapa 1:

En esta etapa se efectúa la selección del pivote. Lo primero que se hace es calcular la posición central y si el primer elemento es mayor que el central se intercambian,

Lista:

6 1 4 9 8 3 5 2 7 0

si el primer elemento es mayor que el último, se intercambian

Lista:

0 1 4 9 8 3 5 2 7 6

si el central es mayor que el último, se intercambian.

Lista: 0 1 4 9 6 3 5 2 7 8

Se toma ahora el central como pivote y se intercambia con el elemento extremo.

Pivote 6
Lista: 0 1 4 9 8 3 5 2 7 6

Etapa 2:

La etapa 2 requiere mover todos los elementos menores al pivote, entre el primero y el penúltimo, a la parte izquierda del array y los elementos mayores a la parte derecha.

0 1 4 9 8 3 5 2 7 6

Para ello se recorre la lista de izquierda a derecha utilizando un contador i que se inicializa en la posición más baja (`Inferior`) buscando un elemento mayor al pivote. También se recorre la lista de derecha a izquierda buscando un elemento menor. Para hacer esto se utilizará un contador j inicializado en la posición más alta, `Superior-1`.

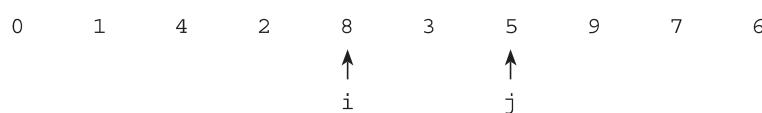
El contador i se detiene en el elemento 9 (mayor que el pivote) y el contador j se detiene en el elemento 2 (menor que el pivote).



Ahora se intercambian 9 y 2 para que estos dos elementos se sitúen correctamente en cada sublistas.

0 1 4 2 8 3 5 9 7 6

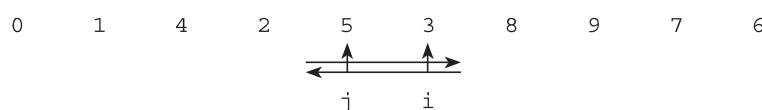
A medida que el algoritmo continúa, i se detiene en el elemento mayor, 8, y j se detiene en el menor, 5.



Se intercambian los elementos mientras que i y j no se cruzan, por tanto se intercambian 8 y 5.

0 1 4 2 8 3 5 9 7 6

Continúa la exploración.



En esta posición los contadores i y j se encuentran sobre el mismo elemento del array y en este caso se detiene la búsqueda y no se realiza ningún intercambio, ya que el elemento al que accede el contador j está ya correctamente situado. Las dos sublistas ya han sido creadas (la lista original se ha dividido en dos particiones).



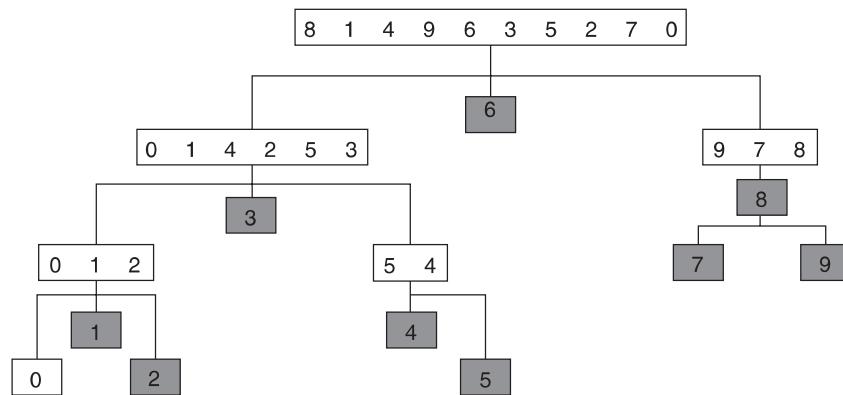
Ahora ya lo único que se necesita es intercambiar el elemento que está en la posición i con el elemento pivote.

Etapa 3:

Intercambiar el elemento de la posición i con el pivote, de modo que se tendrá la secuencia prevista inicialmente:



Resumiendo el proceso general sería:

**14.5.3.1. Algoritmo quicksort**

El primer problema a resolver en el diseño del algoritmo de *quicksort* es seleccionar el pivote. Aunque la posición del pivote, en principio puede ser cualquiera, una de las decisiones más ponderadas es aquella que considera el pivote como el elemento central o próximo al central de la lista. La Figura 14.10 muestra las operaciones del algoritmo para ordenar la lista de elementos enteros L.

```
// algoritmo quicksort
// ordenar a[0:n-1]

Seleccionar un elemento de a[0:n-1] como elemento central
    (este elemento es el pivote)
Dividir los elementos restantes en particiones izquierda y derecha,
    de modo que ningún elemento de la izquierda tenga una clave (valor) mayor que
    el pivote y que ningún elemento a la derecha tenga una clave más pequeña que la
    del pivote.
Ordenar la partición izquierda utilizando quicksort recursivamente.
Ordenar la partición derecha utilizando quicksort recursivamente.
```

14.5.4. Ordenación MERGESORT

La idea básica de este método de ordenación es la mezcla (*merge*) de listas ya ordenadas. El algoritmo puede considerarse que aplica la técnica “divide y vence”, el proceso es simple: si se ordena la primera mitad de la lista, se ordena la segunda mitad de la lista y una vez ordenadas se mezclan, la mezcla da lugar a una lista de elementos ordenada. A su vez, la ordenación de la sublista mitad sigue los mismos pasos, ordenar la primera mitad, ordenar la segunda mitad y mezclar. La sucesiva división de la lista actual en dos hace que el problema (número de elementos) cada vez sea más pequeño; así hasta que la lista actual tenga un elemento y, por tanto, se considera ordenada, es el caso base y a partir de dos sublistas de un número mínimo de elementos se mezclan, dando cada vez lugar a listas ordenadas de cada vez más elementos hasta alcanzar la lista total.

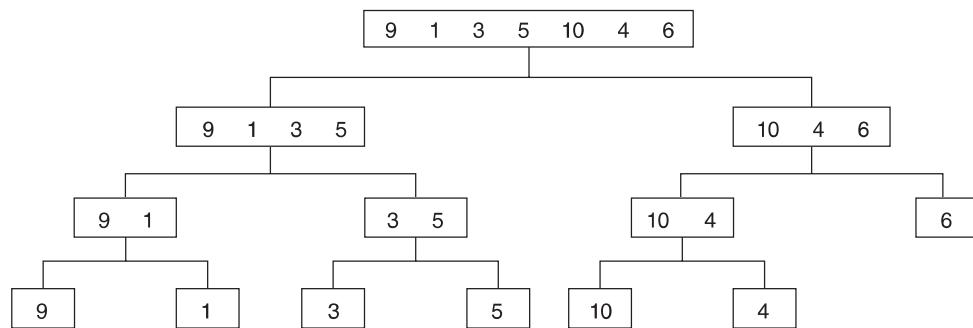
Es decir, que el método consiste, pues, en dividir el vector por su posición central en dos partes y tratar análogamente cada una de ellas hasta que consten de un único elemento. Hay que tener en cuenta que un vector con un único elemento siempre se encuentra ordenado. A la salida de los procesos recursivos las partes ordenadas (subvectores) se mezclan de forma que resultan otras, de mayor longitud, también ordenadas.

EJEMPLO 14.9

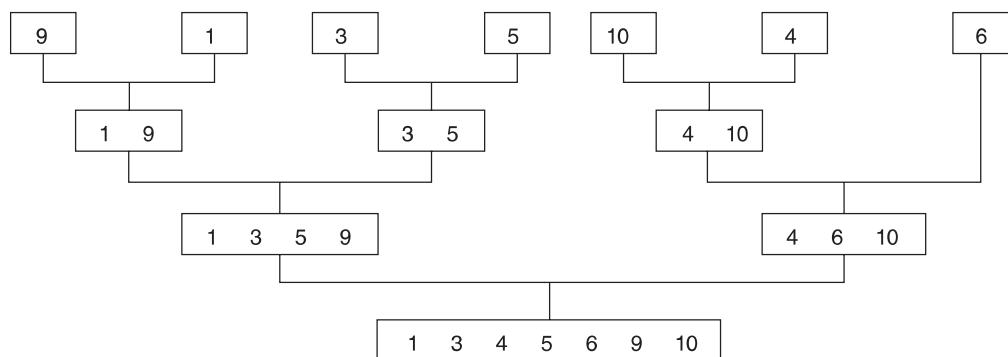
Seguir la estrategia del algoritmo «mergesort» para ordenar la lista:

9 1 3 5 10 4 6

Se representa el proceso con las siguientes figuras en las que aparecen las divisiones.



La mezcla comienza con las sublistas de un solo elemento, que dan lugar a otra sublista del doble de elementos ordenados. El proceso continúa hasta que se construye un única lista ordenada. A continuación se muestra la creación de las sublistas ordenadas:



14.5.4.1. Algoritmo mergesort en JAVA

Este algoritmo de ordenación se diseña fácilmente con ayuda de las llamadas recursivas para dividir las listas en dos mitades; posteriormente se invoca al método de mezcla de dos listas ordenadas. La delimitación de las dos listas se puede hacer con tres índices: `primero`, `central` y `último`, que apuntan a los elementos del array significados por los identificadores. Así, si se tiene una lista de 10 elementos los valores de los índices:

```
primero = 0; ultimo = 9; central = (primero+ultimo)/2 = 4
```

La primera sublista comprende los elementos $a_0 \dots a_4$ y la segunda los elementos siguientes $a_{4+1} \dots a_9$. Los pasos del algoritmo *mergesort* para el *array* (arreglo) *a*:

```

procedimiento mergesort(E/S arr: a, E entero: primero, ultimo)
  inicio
    Si primero < ultimo entonces
      central ← (primero+ultimo) div 2
      mergesort(a, primero, central)
      // ordena primera mitad de la lista
      mergesort(a, central+1, ultimo)
      // ordena segunda mitad de la lista
      mezcla(a, primero, central, ultimo)
      {fusiona las dos sublistas ordenadas, delimitadas por
       los extremos}
    fin_si
  fin_procedimiento

```

La codificación en **Java** consta del método mergesort() y del método auxiliar mezcla().

```

public class PruebaMS
{
  public void mergesort(double[] a, int primero, int ultimo)
  {
    int central;
    if (primero < ultimo)
    {
      central = (primero+ultimo)/2;
      // división entera puesto que los operandos son enteros
      mergesort(a, primero, central);
      mergesort(a, central+1, ultimo);
      mezcla(a, primero, central, ultimo);
    }
  }

  private void mezcla(double[] a, int izda, int medio, int drcha)
  {
    double [] tmp = new double[a.length];
    int x, y, z;

    x = z = izda;
    y = medio+1;
    // bucle para la mezcla, utiliza tmp[] como array auxiliar
    while (x<=medio && y<=drcha)
    {
      if (a[x] <= a[y])
        tmp[z++] = a[x++];
      else
        tmp[z++] = a[y++];
    }
    // bucle para mover elementos que quedan de sublistas
    while (x <= medio)
      tmp[z++] = a[x++];
    while (y <= drcha)
      tmp[z++] = a[y++];

    // Copia de elementos de tmp[] al array a[]
    System.arraycopy(tmp, izda, a, izda, drcha-izda+1);
  }
}

```

```

public static void main(String[] args)
{
    PruebaMS up=new PruebaMS();
    double[] a = {9,1,3,5,10,4,6};
    up.mergesort(a, 0, a.length-1);
    for (int i = 0; i < a.length; i++)
        System.out.println(a[i]);
}
}

```

CONCEPTOS CLAVE

- Clases de recursividad: directa e indirecta
- Concepto de recursividad.
- Complejidad de los métodos de ordenación.
- Eficiencia en cuanto al tiempo de ejecución
- Iteración *versus* recursión.
- *Notación O*.
- Requisitos de un algoritmo recursivo.

RESUMEN

Un subprograma se dice que es recursivo si tiene una o más sentencias que son llamadas a sí mismo. La recursividad puede ser directa e indirecta, la recursividad indirecta ocurre cuando el subprograma —método, procedimiento o función— $f()$ llama a $p()$ y éste a su vez llama a $f()$. La recursividad es una alternativa a la iteración en la resolución de algunos problemas matemáticos. Los aspectos más importantes a tener en cuenta en el diseño y construcción de métodos recursivos son los siguientes:

- Un algoritmo recursivo correspondiente con un método normalmente contiene dos tipos de casos: uno o más casos que incluyen al menos una llamada recursiva y uno o más casos de terminación o parada del problema en los que éste se soluciona sin ninguna llamada recursiva sino con una sentencia simple. De otro modo, un método recursivo debe tener dos partes: una parte de terminación en la que se deja de hacer llamadas, es el caso base, y una llamada recursiva con sus propios parámetros.
- Muchos problemas tienen naturaleza recursiva y la solución más fácil es mediante un método recursivo. De igual modo, aquellos problemas que no entrañen una solución recursiva se deberán seguir resolviendo mediante algoritmos iterativos.
- Todo algoritmo recursivo puede ser transformado en otro de tipo iterativo, pero para ello a veces se nece-

sita utilizar pilas donde almacenar los cálculos parciales.

- Los métodos con llamadas recursivas utilizan memoria extra en las llamadas; existe un límite en las llamadas, que depende de la memoria de la computadora. En caso de superar este límite ocurre un error de overflow.
- Cuando se codifica un método recursivo se debe comprobar siempre que tiene una condición de terminación; es decir, que no se producirá una recursión infinita. Durante el aprendizaje de la recursividad es usual que se produzca ese error.
- Para asegurarse de que el diseño de un método recursivo es correcto se deben cumplir las siguientes tres condiciones:
 1. No existe recursión infinita. Una llamada recursiva puede conducir a otra llamada recursiva y ésta conducir a otra, y así sucesivamente; pero cada llamada debe de aproximarse más a la condición de terminación.
 2. Para la condición de terminación, el método devuelve el valor correcto para ese caso.
 3. En los casos que implican llamadas recursivas: si cada uno de los métodos devuelve un valor correcto, entonces el valor final devuelto por el método es el valor correcto.

EJERCICIOS

- 14.1.** La suma de una serie de números consecutivos de 1 se puede definir recursivamente como:

```
suma(1) = 1
suma(n) = n + suma(n-1)
```

Escribir la función recursiva que acepte n como un argumento y calcule la suma de los números de 1 a n .

- 14.2.** El valor de x^n se puede definir recursivamente como:

```
 $x^0 = 1$ 
 $x^n = x * x^{n-1}$ 
```

Escribir una función recursiva que calcule y devuelva el valor de x^n .

- 14.3.** Reescribir la función escrita en el Ejercicio 14.2 de modo que se utilice un algoritmo repetitivo para calcular el valor de x^n .

- 14.4.** Convierta la siguiente función iterativa en una recursiva. La función calcula un valor aproximado de e , la base de los logaritmos naturales, sumando las series

$$1 + 1/1! + 1/2! + \dots + 1/n!$$

hasta que los términos adicionales no afecten a la aproximación

```
real función loge()
    var
        // Datos locales
        real: enl, delta, fact
        entero: n
    inicio
        enl ← 1.0
        fact ← 1.0
        delta ← 1.0
    hacer
        enl ← delta
        n ← n + 1
```

```
fact ← fact * n
delta ← 1.0 / fact
mientras (enl <> enl + delta)
    devolver (enl)
fin_función
```

- 14.5.** Explique por qué la siguiente función puede producir un valor incorrecto cuando se ejecute:

```
real función factorial (E real: n)
inicio
    si (n = 0 o n = 1) entonces
        devolver(1)
    si_no
        devolver (n * factorial (n-1))
    fin_si
fin_función
```

- 14.6.** Proporcionar funciones recursivas que representen los siguientes conceptos:

- El producto de dos números naturales.
- El conjunto de permutaciones de una lista de números.

- 14.7.** El elemento mayor de un array entero de n -elementos se puede calcular recursivamente. Definir la función:

```
entero función max(E entero: x, y)
```

que devuelve el mayor de dos enteros x e y . Definir la función

```
entero función maxarray(E arr: a,
E entero: n)
```

que utiliza recursión para devolver el elemento mayor de a

Condición de parada: $n == 1$

Incremento recursivo: $maxarray = max(max(a[0] \dots$
 $a[n-2]), a[n-1])$

PROBLEMAS

- 14.1.** La expresión matemática $C(m, n)$ en el mundo de la teoría combinatoria de los números representa el número de combinaciones de m elementos tomados de n en n elementos

$$C(m, n) = \frac{m!}{n!(m-n)!}$$

Escribir y probar una función que calcule $C(m, n)$ donde $n!$ es el factorial de n .

- 14.2.** Un palíndromo es una palabra que se escribe exactamente igual leído en un sentido o en otro. Palabras tales como level, deed, ala, etc., son ejemplos de palíndromos. Escribir una función recursiva que devuelva un valor de 1 (verdadero), si una palabra pasada como argumento es un palíndromo y devuelva 0 (falso) en caso contrario.

- 14.3.** La suma de los primeros n números enteros responde a la fórmula:

$$1 + 2 + 3 + \dots + n = n(n + 1)/2$$

Inicializar el array `A` que contiene los primeros 50 enteros. La media de estos elementos del array es entonces $51/2 = 25.5$. Comprobar la solución aplicando la función recursiva `media (float a[], int n)`.

- 14.4.** Leer un número entero positivo $n < 10$. Calcular el desarrollo del polinomio $(x + 1)^n$. Imprimir cada potencia x^2 en la forma x^{**i} .

Sugerencia:

$$(x + 1)^n = C_{n,n}x^n + C_{n,n-1}x^{n-1} + C_{n,n-2}x^{n-2} + \dots + C_{n,2}x^2 + C_{n,1}x^1 + C_{n,0}x^0$$

donde $C_{n,n}$ y $C_{n,0}$ son 1 para cualquier valor de n .

La relación de recurrencia de los coeficientes binomiales es:

$$C(n, 0) = 1$$

$$C(n, n) = 1$$

$$C(n, k) = C(n - 1, k - 1) + C(n - 1, k)$$

Estos coeficientes constituyen el famoso Triángulo de Pascal y será preciso definir la función que genera el triángulo

$$\begin{array}{ccccccc} 1 & & & & & & \\ 1 & 1 & & & & & \\ 1 & 2 & 1 & & & & \\ 1 & 3 & 3 & 1 & & & \\ 1 & 4 & 6 & 4 & 1 & & \\ \dots & & & & & & \end{array}$$

- 14.5.** Escribir un programa en el que el usuario introduzca 10 enteros positivos y calcule e imprima su factorial.