

**Tema V**

**Memoria dinámica**



## Capítulo 16

# Punteros

---

16.1	Introducción al uso de punteros . . . . .	336
16.2	Aplicaciones no recursivas de los punteros . . . . .	344
16.3	Ejercicios . . . . .	348

---

Las estructuras de datos estudiadas hasta ahora se almacenan estáticamente en la memoria física del computador. Cuando se ejecuta un subprograma, se destina memoria para cada variable global del programa y tal espacio de memoria permanecerá reservado durante toda su ejecución, se usen o no tales variables; por ello, en este caso hablamos de asignación *estática* de memoria.

Esta rigidez presenta un primer inconveniente obvio: las estructuras de datos estáticas no pueden crecer o menguar durante la ejecución de un programa. Obsérvese sin embargo que ello no implica que la cantidad de memoria usada por un programa durante su funcionamiento sea constante, ya que depende, por ejemplo, de los subprogramas llamados. Y, más aún, en el caso de subprogramas recursivos se podría llegar fácilmente a desbordar la memoria del computador.

Por otra parte, la representación de ciertas construcciones (como las listas) usando las estructuras conocidas (concretamente los arrays) tiene que hacerse situando elementos consecutivos en componentes contiguas, de manera que las operaciones de inserción de un elemento nuevo o desaparición de uno ya existente requieren el desplazamiento de todos los posteriores para cubrir el vacío producido, o para abrir espacio para el nuevo.

Estos dos aspectos, tamaño y disposición rígidos, se superan con las llamadas estructuras de datos *dinámicas*. La definición y manipulación de estos objetos

se efectúa en Pascal mediante un mecanismo nuevo (el puntero), que permite al programador referirse directamente a la memoria.

Además estas estructuras de datos son más flexibles en cuanto a su forma: árboles de tamaños no acotados y con ramificaciones desiguales, redes (como las ferroviarias por ejemplo), etc.

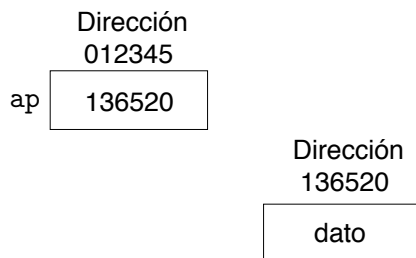
No obstante, esta nueva herramienta no está exenta de peligros en manos de un programador novel, que deberá tratar con mayor cuidado las estructuras creadas, atender especialmente al agotamiento del espacio disponible, etc.

En este capítulo se estudia el concepto de puntero o apuntador (en inglés *pointer*) así como sus operaciones asociadas. Éste es pues un capítulo técnico, que muy bien podría dar como primera impresión que las aportaciones de este nuevo mecanismo son sobre todo dificultades y peligros. Naturalmente, no es así: una vez identificados los peligros y superados los detalles técnicos (capítulo presente), el siguiente capítulo logrará sin duda convencer de que las múltiples ventajas superan con creces los inconvenientes mencionados.

## 16.1 Introducción al uso de punteros

Un *puntero* es una variable que sirve para señalar la posición de la memoria en que se encuentra otro dato almacenando como valor la dirección de ese dato.

Para evitar confusión entre la variable puntero y la variable a la que apunta (o variable *referida*) conviene imaginar gráficamente este mecanismo. En la siguiente figura se muestra la variable puntero `ap`, almacenada en la dirección 012345, y la celda de memoria que contiene la variable a la que apunta.



Puesto que *no* es el contenido real de la variable puntero lo que nos interesa, sino el de la celda cuya dirección contiene, es más común usar el siguiente diagrama

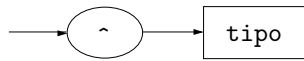
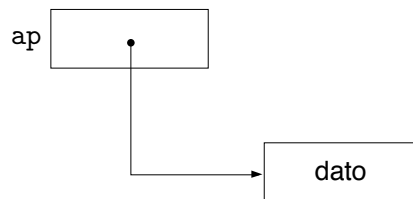


Figura 16.1. Definición de un tipo puntero.



que explica por sí mismo el porqué de llamar puntero a la variable `ap`.

Este último diagrama muestra que un puntero tiene dos componentes: la dirección de memoria a la que apunta (contenido del puntero) y el elemento referido (contenido de la celda de memoria cuya dirección está almacenada en el puntero).

- ☞ Al usar punteros conviene tener muy claro que la variable puntero y la variable a la que apunta son dos variables distintas y, por lo tanto, sus valores son también distintos. A lo largo del capítulo se tendrá ocasión de diferenciar claramente entre ambas.

### 16.1.1 Definición y declaración de punteros

Antes de poder usar punteros en un programa es necesario declararlos: en primer lugar habrá que definir el tipo del dato apuntado y, posteriormente, declarar la(s) variable(s) de tipo puntero que se usará(n).

Por consiguiente, una variable puntero sólo puede señalar a objetos de un mismo tipo, establecido en la declaración. Así por ejemplo, un puntero podrá señalar a caracteres, otro a enteros y otro a vectores pero, una vez que se declara un puntero, sólo podremos usarlo para señalar variables del tipo para el cual ha sido definido. Esta exigencia permite al compilador mantener la consistencia del sistema de tipos, así como conocer la cantidad de memoria que debe reservar o liberar para el dato apuntado.

El tipo puntero es un tipo de datos simple. El diagrama sintáctico de su definición aparece en la figura 16.1, en la que hay que resaltar que el circunflejo (^) indica que se está declarando un puntero a variables del tipo `tipo`.

Naturalmente, para usar punteros en un programa, no basta con definir el tipo puntero: es necesario declarar alguna variable con este tipo. En este fragmento de programa se define el tipo `tApuntChar` como un puntero a variables de tipo `char` y, posteriormente, se declara la variable `apCar` de tipo puntero a caracteres.

```
type
  tApuntChar = ^char;
var
  apCar: tApuntChar
```

Una variable de tipo puntero ocupa una cantidad de memoria fija, independientemente del tipo del dato señalado, ya que su valor es la dirección en que reside éste. Por otra parte, si bien el tipo de la variable `apCar` es `tApuntChar`, el dato señalado por `apCar` se denota mediante `apCar^`, cuyo tipo es por lo tanto `char`.

Como ocurre con las otras variables, el valor de un puntero estará en principio indefinido. Pero, además, los objetos señalados no tienen existencia inicial, esto es, ni siquiera existe un espacio en la memoria destinado a su almacenamiento. Por ello, no es correcto efectuar instrucciones como `WriteLn(apCar1^)` hasta que se haya creado el dato apuntado. Estas operaciones se estudian en el siguiente apartado.

### 16.1.2 Generación y destrucción de variables dinámicas

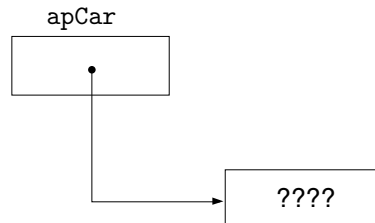
En la introducción del tema se destacó la utilidad de la memoria dinámica, de variables que se generan cuando se necesitan y se destruyen cuando han cumplido su cometido. La creación y destrucción de variables dinámicas se realiza por medio de los procedimientos predefinidos `New` y `Dispose`, respectivamente. Así pues, la instrucción

```
New(apCar)
```

tiene un efecto doble:

1. Reserva la memoria para un dato del tipo apropiado (en este caso del tipo `char`).
2. Coloca la dirección de esta nueva variable en el puntero.

que, gráficamente, se puede expresar así:



- Obsérvese que la operación de generación `New` ha generado el dato apuntado por `apCar`, pero éste contiene por el momento una información desconocida.

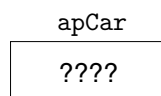
Para destruir una variable dinámica se usa el procedimiento estándar `Dispose`. La instrucción

```
Dispose(apCar)
```

realiza las dos siguientes acciones

1. Libera la memoria asociada a la variable referida `apCar` (dejándola disponible para otros fines).
2. Deja indefinido el valor del puntero.

Gráficamente, esos efectos llevan a la siguiente situación:



En resumen, una variable dinámica sólo se creará cuando sea necesario (lo que ocasiona la correspondiente ocupación de memoria) y, previsiblemente, se destruirá una vez haya cumplido con su cometido (con la consiguiente liberación de la misma).

### 16.1.3 Operaciones básicas con datos apuntados

Recuérdese que el dato referido por el puntero `apCar` se denota `apCar`, que es de tipo `char`. Por consiguiente, son válidas las instrucciones de asignación, lectura y escritura y demás operaciones legales para los caracteres:

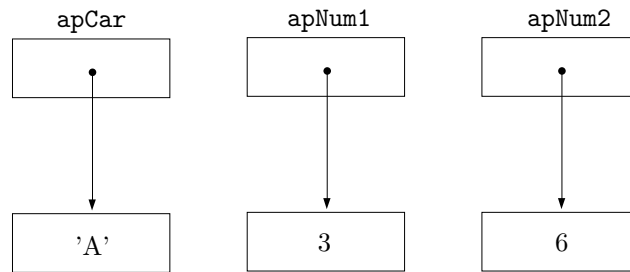


Figura 16.2.

```

type
  tApCharacter = ^char;
var
  apCar: tApCharacter;
...
New(apCar);
ReadLn(apCar^); {supongamos que se da la letra 'B'}
apCar^:= Pred(apCar^);
WriteLn(apCar^);

```

escribiéndose en el output la letra 'A'.

Igualmente, suponiendo que se ha declarado

```

type
  tApNumero = ^integer;
var
  apNum1, apNum2: tApNumero;

```

el siguiente fragmento de programa es válido:

```

New(apNum1);
New(apNum2);
apNum1^:= 2;
apNum2^:= 4;
apNum2^:= apNum1^ + apNum2^;
apNum1^:= apNum2^ div 2;

```

y ambos fragmentos de instrucciones llevan a la situación de la figura 16.2, que será referida varias veces en este apartado.

En general, las operaciones válidas con un dato apuntado dependen de su tipo. Por ejemplo, el fragmento de programa siguiente

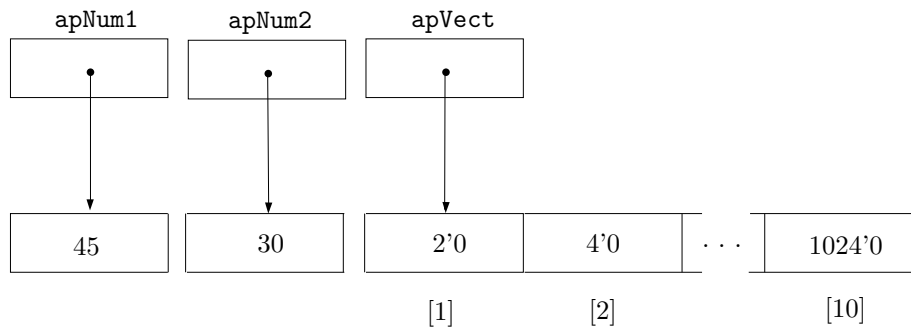


Figura 16.3.

```

type
  tVector10 = array[1..10] of real;
  tApNumero = ^integer;
  tApVector10 = ^tVector10;
var
  apNum1, apNum2: ApNumero;
  apVect: tApVector10;
  i: integer;
  ...
  New(apNum1);
  New(apNum2);
  New(apVect);
  apNum1^ := 45;
  apNum2^ := 30;
  apVect^[1] := 2;
  for i:= 2 to 10 do
    apVect^[i] := apVect^[i-1] * 2;

```

deja el estado de la memoria como se indica en la figura 16.3.

Por su parte, las operaciones siguientes, por ejemplo, no serían legales:

```

ReadLn(apVect^); {Lectura de un array de un solo golpe}
apNum1^ := apNum2^ + apVect^[1]; {Tipos incompatibles}

```

#### 16.1.4 Operaciones básicas con punteros

Sólo las operaciones de comparación (con la igualdad) y asignación están permitidas entre punteros. En la situación de la figura 16.2, la comparación `apNum1 = apNum2` resulta ser falsa. Más aún, tras ejecutar las instrucciones

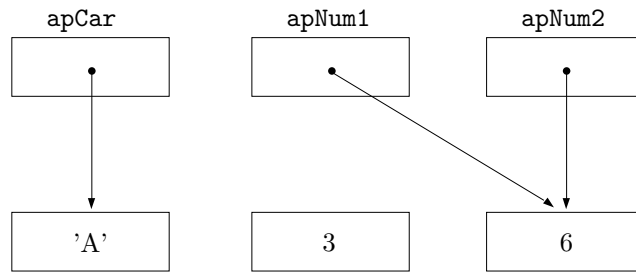


Figura 16.4.

```
apNum1^ := 6;
apNum2^ := 6;
```

la comparación:

```
apNum1 = apNum2
```

seguiría siendo `False`, ya que las direcciones apuntadas no coinciden, a pesar de ser iguales los datos contenidos en dichas direcciones.

También es posible la asignación

```
apNum1 := apNum2
```

cuyo resultado puede verse en la figura 16.4 en la que se observa que ambos punteros señalan a la misma dirección, resultando ahora iguales al compararlos:

```
apNum1 = apNum2
```

produce un resultado `True` y, como consecuencia, `apNum1^` y `apNum2^` tienen el mismo valor, `6`. Además, esta coincidencia en la memoria hace que los cambios efectuados sobre `apNum1^` o sobre `apNum2^` sean indistintos:

```
ApNum1^ := 666;
WriteLn(ApNum2^); {666}
```

- ☉☉ Obsérvese que el espacio de memoria reservado inicialmente por el puntero `apNum1` sigue situado en la memoria. Lo adecuado en este caso habría sido liberar ese espacio con `Dispose` antes de efectuar esa asignación.

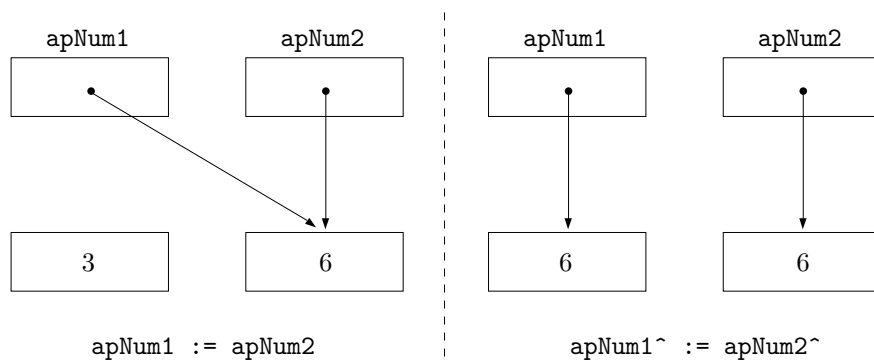
En ambos casos, asignación y comparación, es necesario conservar la consistencia de los tipos utilizados. Por ejemplo, en la situación representada en la figura 16.2, la siguiente asignación es errónea

```
apCar := apNum1
```

porque, aunque ambas variables sean punteros, en su declaración queda claro que no apuntan a variables del mismo tipo.

- ☉☉ Obsérvese que las instrucciones `apNum1 := apNum2` y `apNum1^ := apNum2^` tienen distinto efecto. A continuación se muestra gráficamente el resultado de cada una de las instrucciones anteriores, partiendo de la situación de la figura 16.2,

Supuesto que `apNum1^=3` y `apNum2^=6` sean True



donde se observa que, al asignar los punteros entre sí, se *comparte* la variable referida; en nuestro caso la celda de memoria que contenía a `apNum1^` se pierde, sin posibilidad de recuperación (a menos que otro puntero señalara a esta celda). Por otra parte, al asignar las variables referidas se tiene el mismo valor *en dos celdas distintas* y los punteros permanecen sin variar.

### 16.1.5 El valor *nil*

Un modo alternativo para dar valor a un puntero es, simplemente, diciendo que no apunta a ningún dato. Esto se puede conseguir utilizando la constante predefinida `nil`.

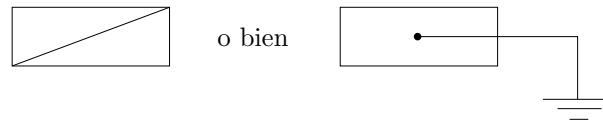


Figura 16.5. puntero iniciado en **nil**.

Por ejemplo, la siguiente asignación define el puntero **apCar**:

```
apCar := nil
```

Gráficamente, el hecho de que una variable puntero no apunte a nada se representa cruzando su celda de memoria con una diagonal, o bien mediante el símbolo (prestado de la Electricidad) de conexión a tierra, como se indica en la figura 16.5.

En cuanto al tipo del valor **nil** es de resaltar que esta constante es común a cualquier tipo de puntero (sirve para todos), pues sólo indica que el puntero está anulado.

En el capítulo siguiente se pondrá de manifiesto la utilidad de esta operación. Por el momento, digamos solamente que es posible saber si un puntero está anulado, mediante la comparación con **nil**, por ejemplo mediante **apNum1 = nil**.

☞ Obsérvese que no es necesario iniciar los punteros apuntando a **nil**. En el ejemplo anterior se inician usando **New**, esto es, creando la variable referida.

## 16.2 Aplicaciones no recursivas de los punteros

Se ha comentado en la introducción que las aplicaciones más importantes de los punteros, están relacionadas con las estructuras de datos recursivas, que estudiaremos en el capítulo siguiente. El apartado que pone cierre a éste presenta algunas situaciones con estructuras no recursivas en las que los punteros resultan útiles.

El hecho de que los punteros sean objetos de tipo simple es interesante y permite, entre otras cosas:

1. La asignación de objetos no simples en un solo paso.
2. La definición de funciones cuyo resultado no es simple.

### 16.2.1 Asignación de objetos no simples

Esta aplicación adquiere mayor relevancia cuando se consideran registros de gran tamaño en los que el problema es el elevado coste al ser necesaria la copia de todas sus componentes. Por ejemplo, en la situación

```

type
  tFicha = record
    nombre: ...
    direccion: ...
    ...
  end; {tFicha}
var
  pers1, pers2: tFicha;

```

la asignación `pers1 := pers2` es altamente costosa si el tipo `tFicha` tiene grandes dimensiones.

Una forma de economizar ese gasto consiste en usar sólo su posición, haciendo uso de punteros:

```

var
  p1, p2: ^tFicha;

```

en vez de las variables `pers1` y `pers2`. Entonces, la instrucción

```
p1 := p2
```

es casi instantánea, por consistir tan sólo en la copia de una dirección.

El artificio anterior resulta útil, por ejemplo, cuando hace falta intercambiar variables de gran tamaño. Si se define el tipo

```

type
  tApFicha = ^tFicha;

```

el procedimiento siguiente efectúa el intercambio rápidamente, con independencia del tamaño del tipo de datos `fichas`:

```

procedure Intercambiar(var p1, p2: tApFicha);
var
  aux: tApFicha;
begin
  aux := p1;
  p1 := p2;
  p2 := aux
end; {Intercambiar}

```

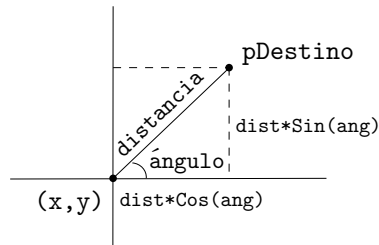


Figura 16.6.

La ordenación de vectores de elementos grandes es un problema en el que la aplicación mostrada demuestra su utilidad. Por ejemplo, la definición

```
type
  tListaAlumnos = array [1..100] of tFicha;
```

se puede sustituir por un array de punteros,

```
type
  tListaAlumnos = array [1..100] of tApFicha;
```

de este modo la ordenación se realizará de una manera mucho más rápida, debido esencialmente al tiempo que se ahorra al no tener que copiar literalmente todas las fichas que se cambian de posición.

### 16.2.2 Funciones de resultado no simple

Esta aplicación tiene la misma base que la anterior, cambiar el objeto por el puntero al mismo. A continuación vemos un sencillo ejemplo en el que se aprovecha esta característica.

Supongamos, por ejemplo, que hemos de definir un programa que, dado un punto del plano, un ángulo y una distancia, calcule un nuevo punto, alcanzado al recorrer la distancia según el rumbo dado, según la figura 16.6.

Una primera aproximación al diseño del programa podría ser la siguiente:

*Lectura de datos: punto base, distancia y ángulo*  
*Cálculo del punto final*  
*Escritura de resultados*

Naturalmente, en primer lugar se deben definir los tipos, constantes y variables que se usarán en el programa: se definirá un tipo `tPunto` como un registro

de dos componentes de tipo real y un tipo dinámico `tApPunto` que señalará al tipo `punto`, además se necesitarán dos variables `distancia` y `angulo` para leer los valores de la distancia y el ángulo del salto, una de tipo `punto` para el origen del salto y otra de tipo `tApPunto` para el punto de destino.

```

type
  tPunto = record
    x, y: real
  end; {tPunto}
  tApPunto = ^tPunto;
var
  angulo, distancia: real;
  origen: tPunto;
  pDestino: tApPunto;

```

El cálculo del punto final se puede realizar mediante la definición (y posterior aplicación) de una función. Ésta tendría que devolver un punto, formado por dos coordenadas; dado que esto no es posible, se devolverá un puntero al tipo `tPunto` (de ahí la necesidad del tipo `tApPunto`). Dentro de la función utilizaremos el puntero `pPun` para generar la variable apuntada y realizar los cálculos, asignándolos finalmente a `Destino` para devolver el resultado de la función. La definición de la función es absolutamente directa:

```

function Destino(orig: tPunto; ang, dist: real): tApPunto;
  var
    pPun: tApPunto;
  begin
    New(pPun);
    pPun^.x := orig.x + dist * Cos(ang);
    pPun^.y := orig.y + dist * Sin(ang);
    Destino := pPun
  end; {Destino}

```

Finalmente, si se añaden los procedimientos de lectura de datos y escritura de resultados tenemos el siguiente programa:

```

Program Salto (input, output);
  type
    tPunto = record
      x, y: real
    end {tPunto};
    tApPunto = ^tPunto;

```

```

var
  angulo, distancia: real;
  origen: tPunto;
  pDestino: tApPunto;

function Destino(orig: tPunto; ang, dist: real): tApPunto;
var
  pPun: tApPunto;
begin
  New(pPun);
  pPun^.x:= orig.x + dist * Cos(ang);
  pPun^.y:= orig.y + dist * Sin(ang)
  Destino:= pPun
end; {Destino}

begin
  Write('Introduzca la x y la y del punto origen: ');
  ReadLn(origen.x, origen.y);
  Write('Introduzca el rumbo y la distancia: ');
  ReadLn(angulo, distancia);
  pDestino:= Destino(origen, angulo, distancia);
  WriteLn('El punto de destino es:');
  WriteLn('X = ', pDestino^.x:20:10, ' Y = ', pDestino^.y:20:10)
end. {Salto}

```

La idea más importante de este programa estriba en que la función devuelve un valor de tipo puntero, que es un tipo simple, y por lo tanto correcto, como resultado de una función. Sin embargo, la variable referenciada por el puntero es estructurada, y almacena las dos coordenadas del punto de destino. Mediante esta estratagema, conseguimos obtener un resultado estructurado de una función.

### 16.3 Ejercicios

1. Implementar algún algoritmo de ordenación de registros (por ejemplo, fichas de alumnos), mediante un array de punteros, tal como se propone en el apartado 16.2.1.
2. Complete los tipos de datos apropiados para que sea correcta la siguiente instrucción:

```

for i:= 1 to n do begin
  New(p);
  p^:= i
end {for}

```

- (a) ¿Cuál es el efecto de ejecutarla?
  - (b) ¿Cuál es el efecto de añadir las siguientes instrucciones tras la anterior?  

```
WriteLn(p);  
WriteLn(p^);  
Dispose(p)
```
  - (c) ¿De qué forma se pueden recuperar los valores  $1 \dots N - 1$  generados en el bucle **for**?
3. Escriba una función que reciba una matriz cuadrada de  $3 \times 3$  y calcule (si es posible) su inversa, devolviendo un puntero a dicha matriz.



## Capítulo 17

# Estructuras de datos recursivas

---

17.1 Estructuras recursivas lineales: las listas enlazadas .	351
17.2 Pilas . . . . .	362
17.3 Colas . . . . .	370
17.4 Árboles binarios . . . . .	376
17.5 Otras estructuras dinámicas de datos . . . . .	387
17.6 Ejercicios . . . . .	389
17.7 Referencias bibliográficas . . . . .	391

---

En este capítulo se introducen, a un nivel elemental, las estructuras de datos recursivas como la aplicación más importante de la memoria dinámica.

Se presentan las *listas*, el tipo de datos recursivo más sencillo y que tiene múltiples aplicaciones, y los casos particulares de las *pilas* y las *colas*, como listas con un “acceso controlado”. También se introducen los árboles binarios como ejemplo de estructura de datos no lineal, y se termina apuntando hacia otras estructuras cuyo estudio queda fuera de nuestras pretensiones.

### 17.1 Estructuras recursivas lineales: las listas enlazadas

Las representaciones para listas que introducimos en este capítulo nos permitirán insertar y borrar elementos más fácil y eficientemente que en una implementación estática (para listas acotadas) usando el tipo de datos **array**.

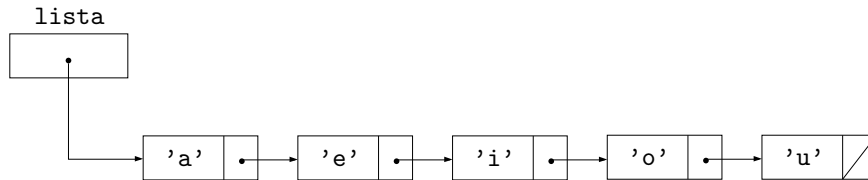


Figura 17.1. Representación de una lista enlazada dinámica.

A medida que desarrollemos las operaciones de inserción y borrado, centraremos nuestra atención en las acciones necesarias para mantener diferentes clases particulares de listas: las pilas y las colas. Por lo común, son las distintas operaciones requeridas las que determinarán la representación más apropiada para una lista.

### 17.1.1 Una definición del tipo lista

Una *lista* es una colección lineal de elementos que se llaman *nodos*. El término colección lineal debe entenderse de la siguiente manera: tenemos un primer y un último nodo, de tal manera que a cada nodo, salvo el último, le corresponde un único sucesor, y a cada nodo, salvo el primero, le corresponde un único predecesor. Se trata, pues, de una estructura de datos cuyos elementos están situados secuencialmente.

Una definición del tipo listas se puede realizar usando punteros y registros. Para ello consideramos que cada nodo de la lista es un registro con dos componentes: la primera almacena el **contenido** del nodo de la lista y, la segunda, un puntero que señala al **siguiente** elemento de la lista, si éste existe, o con el valor **nil** en caso de ser el último. Esta construcción de las listas recibe el nombre de *lista enlazada dinámica*. En la figura 17.1 se muestra gráficamente esta idea.

Esencialmente, una lista será representada como un puntero que señala al principio (o cabeza) de la lista. La definición del tipo `tLista` de elementos de tipo `tElem` se presenta a continuación, junto con la declaración de una variable del tipo lista:

```

type
  tElem = char; {o lo que corresponda}
  tLista = ^tNodo;
  tNodo = record
    contenido: tElem;
    siguiente: tLista
  end; {tNodo}

```

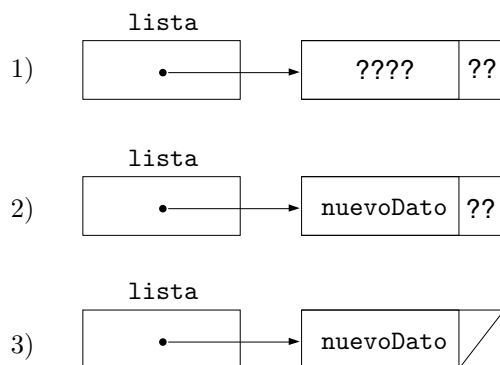


Figura 17.2. Inserción de un elemento en una lista vacía.

```
var
  lista : tLista;
```

Sobre el código anterior debe señalarse que se define `tLista` como un puntero a un tipo no definido todavía. Esto está permitido en Pascal (supuesto que tal tipo es definido posteriormente, pero en el mismo grupo de definiciones) precisamente para poder construir estructuras recursivas.

### 17.1.2 Inserción de elementos

Supóngase que nuestra `lista` está iniciada con el valor `nil`. Para introducir un elemento `nuevoDato` en ella, habrá que completar la siguiente secuencia de pasos, que se muestra gráficamente en la figura 17.2:

1. En primer lugar, habrá que generar una variable del tipo `tNodo`, que ha de contener el nuevo eslabón: esto se hace mediante la sentencia `New(lista)`.
2. Posteriormente, se asigna `nuevoDato` al campo `contenido` del nodo recién generado. La forma de esta asignación dependerá del tipo de datos de la variable `nuevoDato`.
3. Y por último, hay que anular (con el valor `nil`) el campo `siguiente` del nodo para indicar que es el último de la `lista`.

Para insertar un `nuevoDato` al principio de una lista no vacía, `lista`, se ha de proceder como se indica (véase la figura 17.3):

1. Una variable auxiliar `listaAux` se usa para apuntar al principio de la `lista` con el fin de no perderla. Esto es:

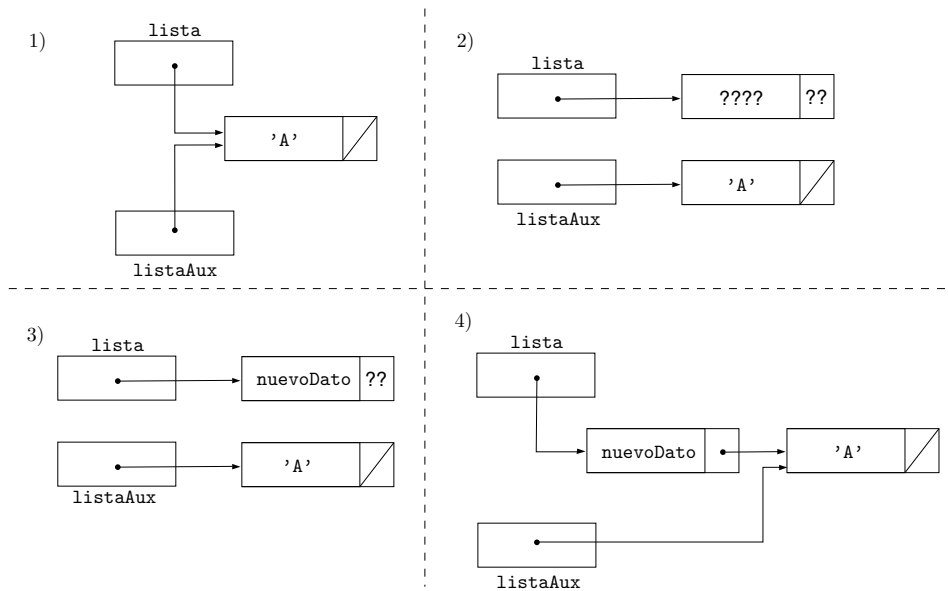


Figura 17.3. Adición de un elemento al principio de una lista.

```
listaAux:= lista
```

2. Después se asigna memoria a una variable del tipo `tNodo` (que ha de contener el nuevo elemento): esto se hace mediante la sentencia `New(lista)`.
3. Posteriormente, se asigna `nuevoDato` al campo `contenido` del nuevo nodo:

```
lista^.contenido:= nuevoDato
```

4. Y, por último, se asigna la `listaAux` al campo `siguiente` del nuevo nodo para indicar los demás elementos de la lista, es decir:

```
lista^.siguiente:= listaAux
```

A continuación se van a definir algunas de las operaciones relativas a listas. Para empezar, se desarrolla un procedimiento para añadir un elemento al principio de una lista, atendiendo al diseño descrito por los pasos anteriores:

```

procedure AnnadirPrimero(var lista: tLista; nuevoDato: tElem);
  {Efecto: se añade nuevoDato al comienzo de lista}
  var
    listaAux: tLista;
begin
  listaAux:= lista;
  New(lista);
  lista^.contenido:= nuevoDato;
  lista^.siguiente:= listaAux
end; {AnnadirPrimero}

```

Obsérvese que este procedimiento sirve tanto para cuando `lista` es vacía como para cuando no lo es.

### 17.1.3 Eliminación de elementos

A continuación se presenta el procedimiento `Eliminar` que sirve para eliminar el primer elemento de una lista no vacía. La idea es bastante simple: sólo hay que actualizar el puntero para que señale al siguiente elemento de la lista (si existe).

```

procedure EliminarPrimero(var lista: tLista);
  {PreC.: la lista no está vacía}
  {Efecto: el primer nodo ha sido eliminado}
  var
    listaAux: tLista;
begin
  listaAux:= lista;
  lista:= lista^.siguiente;
  Dispose(listaAux)
end; {EliminarPrimero}

```

### 17.1.4 Algunas funciones recursivas

#### La longitud de una lista

La naturaleza recursiva de la definición dada del tipo lista permite definir fácilmente funciones que trabajen usando un proceso recursivo. Por ejemplo, la función `Longitud` de una lista se puede definir recursivamente (es evidente que el caso base es `Longitud(nil) = 0`) siguiendo la línea del siguiente ejemplo:

$$\begin{aligned}
 \text{Longitud}([a,b,c]) &\rightsquigarrow 1 + \text{Longitud}([b,c]) \\
 &\rightsquigarrow 1 + (1 + \text{Longitud}([c])) \\
 &\rightsquigarrow 1 + (1 + (1 + \text{Longitud}(\text{nil}))) \\
 &\rightsquigarrow 1 + (1 + (1 + 0)) \rightsquigarrow 3
 \end{aligned}$$

De aquí se deduce que, si `lista` no está vacía, y llamamos `Resto(lista)` a la `lista` sin su primer elemento, tenemos:

$$\text{Longitud}(\text{lista}) = 1 + \text{Longitud}(\text{Resto}(\text{lista}))$$

Por lo tanto, en resumen,

$$\text{Longitud}(\text{lista}) = \begin{cases} 0 & \text{si } \text{lista} = \text{nil} \\ 1 + \text{Longitud}(\text{Resto}(\text{lista})) & \text{en otro caso} \end{cases}$$

Finalmente, `Resto(lista)` se representa en Pascal como `lista^.siguiente`. Por lo tanto, la definición recursiva en Pascal de la función `Longitud` es la siguiente:

```

type
  natural = 0..MaxInt;
  ...
function Longitud(lista: tLista): natural;
  {Dev. el número de nodos de lista}
begin
  if lista = nil then
    Longitud := 0
  else
    Longitud := 1 + Longitud(lista^.siguiente)
end; {Longitud}

```

El uso de la recursividad subyacente en la definición del tipo de datos `lista` hace que la definición de esta función sea muy simple y clara, aunque también es posible y sencilla una versión iterativa, que se deja como ejercicio.

### Evaluación de polinomios: La regla de Horner

Supongamos que se plantea escribir un programa para evaluar en un punto dado  $x$  un polinomio con coeficientes reales

$$P(x) = a_0x^n + a_1x^{n-1} + a_2x^{n-2} + \dots + a_{n-1}x + a_n,$$

Conocido  $x$ , un posible modo de evaluar el polinomio es el siguiente:

*Conocido x*  
*Dar valor inicial 0 a valor*  
*para cada i entre 0 y N*  
*Añadir  $a_{N-i} * x^i$  a valor*  
*Devolver valor*

La *regla de Horner* consiste en disponer el polinomio de forma que su evaluación se pueda hacer simplemente mediante la repetición de operaciones aritméticas elementales (y no tener que ir calculando los valores de las potencias sucesivas de  $x$ ).

No es difícil observar que

$$P(x) = \left( \cdots \left( ((a_0x + a_1)x + a_2)x + a_3 \right) x + \cdots a_{n-1} \right) x + a_n$$

y que, con esta expresión, el valor del polinomio se puede calcular de acuerdo con el siguiente diseño:

```
Dar valor inicial a0 a valor
para cada i entre 1 y N hacer
    valor := ai + x * valor
Devolver valor
```

En este apartado volvemos a considerar la representación de polinomios en un computador una vez que se han introducido algunas herramientas de programación dinámica.

Hay dos formas de representar un polinomio conocida la lista de sus coeficientes: de más significativo a menos, o viceversa. A saber:

$$\begin{aligned} P_1(x) &= a_0x^n + a_1x^{n-1} + a_2x^{n-2} + \cdots + a_{n-1}x + a_n \\ &= \left( \cdots \left( ((a_0x + a_1)x + a_2)x + a_3 \right) x + \cdots a_{n-1} \right) x + a_n \\ P_2(x) &= b_0 + b_1x + b_2x^2 + \cdots + b_{n-1}x^{n-1} + b_nx^n \\ &= b_0 + x \left( b_1 + x \left( b_2 + x \left( b_3 + \cdots + x \left( b_{n-1} + xb_n \right) \cdots \right) \right) \right) \end{aligned}$$

En el supuesto de que tengamos los coeficientes ordenados de grado menor a mayor (si lo están del otro modo la modificación del programa es mínima) según su grado y almacenados en una lista (`lCoef`), es posible dar una versión recursiva de la función anterior.

La recursividad de la regla de Horner se aprecia mejor si consideramos la siguiente tabla, en la que se renombra el resultado de cada uno de los paréntesis de la expresión de Horner:

$$\begin{aligned} c_0 &= a_0 \\ c_1 &= a_1 + xc_0 \end{aligned}$$

$$\begin{aligned}
 c_2 &= a_2 + xc_1 \\
 c_3 &= a_3 + xc_2 \\
 \vdots &= \quad \vdots \\
 c_{n-1} &= a_{n-1} + xc_{n-2} \\
 c_n &= a_n + xc_{n-1}
 \end{aligned}$$

Claramente se observa que  $b_n = P(x)$  y, en consecuencia, ya podemos escribir su expresión recursiva.

```

Horner(x,lCoef) =
    lCoef^.contenido + x * Horner(x,lCoef^.siguiente)

```

El caso base se da cuando la lista de coeficientes se queda vacía (**nil**). Un sencillo ejemplo nos muestra que, en este caso, la función debe valer cero.

Supongamos que queremos evaluar un polinomio constante  $P(x) = K$ , entonces su lista de coeficientes es  $[K]$ , usando la expresión recursiva anterior tendríamos que

```

Horner(x,nil) = 0
Horner(x,[K]) = K + x * Horner(x,nil)

```

Una vez conocidos el caso base y la expresión recursiva definiremos una función **Horner2** con dos variables: el punto donde evaluar y una lista de reales que representan los coeficientes del polinomio ordenados de mayor a menor grado.

```

function Horner2(x: real; lCoef: tLista): real;
    {Dev. P(x), siendo P el polinomio de coeficientes lCoef}
begin
    if lCoef = nil then
        Horner2:= 0
    else
        Horner2:= lCoef^.contenido + x * Horner2(x,lCoef^.siguiente)
    end; {Horner2}

```

### 17.1.5 Otras operaciones sobre listas

A continuación se enumeran algunas de las operaciones que más frecuentemente se utilizan al trabajar con listas:

1. Determinar el número de elementos de una lista.

2. Leer o modificar el  $k$ -ésimo elemento de la lista.
3. Insertar o eliminar un elemento en la  $k$ -ésima posición.
4. Insertar o eliminar un elemento en una lista ordenada.
5. Combinar dos o más listas en una única lista.
6. Dividir una lista en dos o más listas.
7. Ordenar los elementos de una lista de acuerdo con un criterio dado.
8. Insertar o eliminar un elemento en la  $k$ -ésima posición de una lista de acuerdo con un criterio dado.
9. Buscar si aparece un valor dado en algún lugar de una lista.

Como ejemplo de implementación de algunas de estas operaciones, a continuación se presentan procedimientos que eliminan o insertan un nuevo elemento en una lista.

### El procedimiento `EliminarK`

En primer lugar, se presenta un procedimiento que elimina el  $k$ -ésimo elemento de una lista (que se supone de longitud mayor que  $k$ ). En el caso en que  $k = 1$ , podemos utilizar el procedimiento `EliminarPrimero` desarrollado anteriormente, por lo que en este apartado nos restringimos al caso  $k > 1$ .

El primer esbozo de este procedimiento es muy sencillo:

*Localizar el nodo  $(k-1)$ -ésimo de lista*  
*Asociar el puntero siguiente del nodo  $(k-1)$ -ésimo al nodo  $(k+1)$ -ésimo*

El proceso se describe gráficamente en la figura 17.4.

Emplearemos un método iterativo para alcanzar el  $(k - 1)$ -ésimo nodo de la lista, de modo que se irá avanzando nodo a nodo desde la cabeza (el primer nodo de la lista) hasta alcanzar el  $(k - 1)$ -ésimo usando un puntero auxiliar `apAux`.

Para alcanzar el nodo  $(k - 1)$ -ésimo, se empezará en el primer nodo, mediante la instrucción

```
listaAux:= lista;
```

y luego se avanzará iterativamente al segundo, tercero,  $\dots$ ,  $(k - 1)$ -ésimo ejecutando

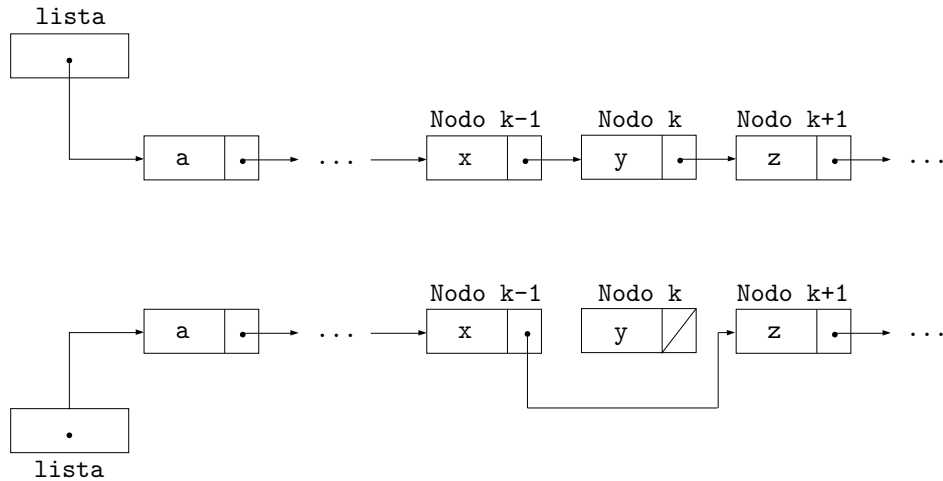


Figura 17.4. Eliminación del  $k$ -ésimo nodo de una lista.

```
for i:= 2 to k - 1 do
  apAux:= apAux^.siguiente
```

Una vez hecho esto, sólo hace falta saltarse el nodo  $k$ -ésimo, liberando después la memoria que ocupa. Con esto, el código en Pascal de este procedimiento podría ser:

```
procedure EliminarK(k: integer; var lista: tLista);
  {PreC.:  $1 < k \leq$  longitud de la lista}
  {Efecto: se elimina el elemento  $k$ -ésimo de lista}
  var
    apAux, nodoSupr: tLista;
    i: integer;
begin
  apAux:= lista;
  {El bucle avanza apAux hasta el nodo  $k-1$ }
  for i:= 2 to k-1 do
    apAux:= apAux^.siguiente;
    {Actualizar punteros y liberar memoria}
    nodoSupr:= apAux^.siguiente;
    apAux^.siguiente:= nodoSupr^.siguiente;
    Dispose(nodoSupr)
  end; {EliminarK}
```

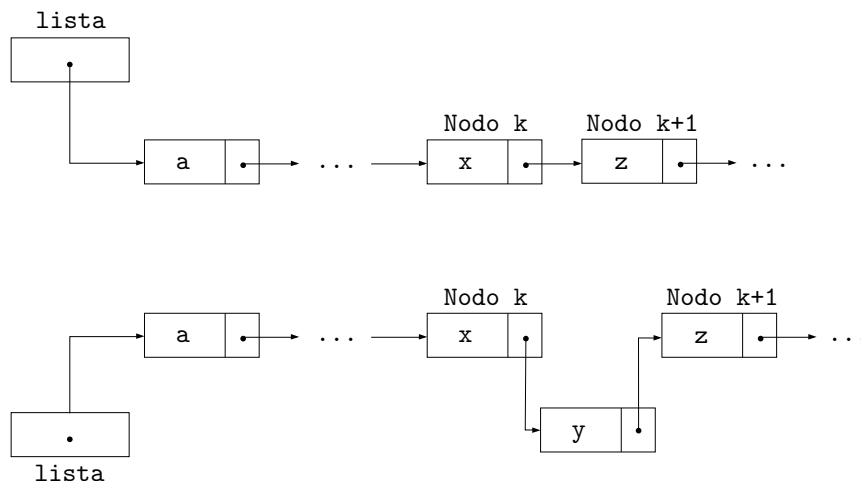


Figura 17.5. Inserción de un nodo tras el  $k$ -ésimo nodo de una lista.

Obsérvese que el procedimiento anterior funciona también cuando el nodo eliminado es el último nodo de la lista.

### El procedimiento InsertarK

Otro procedimiento importante es el de inserción de un elemento tras cierta posición de la lista. En particular, a continuación se implementará la inserción de un nuevo nodo justo después del  $k$ -ésimo nodo de una lista (de longitud mayor que  $k$ ). Puesto que la inserción de un nuevo nodo al comienzo de una lista ya se ha presentado, nos restringiremos al caso  $k \geq 1$ .

De nuevo el primer nivel de diseño es fácil:

*Localizar el nodo  $k$ -ésimo de lista*

*Crear un nuevoNodo y asignarle el contenido nuevoDato*

*Asociar el puntero siguiente del nodo  $k$ -ésimo a nuevoNodo*

*Asociar el puntero siguiente de nuevoNodo al nodo  $(k+1)$ -ésimo*

En la figura 17.5 se pueden observar las reasignaciones de punteros que hay que realizar. También en este caso es necesario usar un puntero auxiliar para localizar el nodo tras el cual se ha de insertar el nuevo dato y se volverá a hacer uso del bucle **for**.

La creación del nuevo nodo y la reasignación de punteros es bastante directa, una vez que **apAux** señala al nodo  $k$ -ésimo. El código correspondiente en Pascal podría ser:

```

New(nuevoNodo);
nuevoNodo^.contenido:= nuevoDato;
nuevoNodo^.siguiente:= apAux^.siguiente;
apAux^.siguiente:= nuevoNodo

```

Uniendo el bucle, para alcanzar el  $k$ -ésimo nodo, con las asignaciones anteriores se obtiene la implementación completa del procedimiento `InsertarK`:

```

procedure InsertarK(k: integer; nuevoDato: tElem; var lista: tLista);
  {PreC.:  $k \geq 1$  y lista tiene al menos k nodos}
  {Efecto: nuevoDato se inserta tras el k-ésimo nodo de lista}
  var
    nuevoNodo, apAux: tLista;
    i: integer;
begin
  apAux:= lista;
  {El bucle avanza apAux hasta el nodo k}
  for i:= 1 to k-1 do
    apAux:= apAux^.siguiente;
    {Actualización de punteros}
  New(nuevoNodo);
  nuevoNodo^.contenido:= nuevoDato;
  nuevoNodo^.siguiente:= apAux^.siguiente;
  apAux^.siguiente:= nuevoNodo
end; {InsertarK}

```

Son muy frecuentes las situaciones en las que se accede a una lista sólo a través de su primer o último elemento. Por esta razón tienen particular interés las implementaciones de listas en las que el acceso está restringido de esta forma, siendo las más importantes las pilas y las colas. A continuación se definen estos tipos de listas junto con determinadas operaciones asociadas: se dará su representación y las operaciones de modificación y acceso básicas, y se presentarán ejemplos donde se vea el uso de las mismas.

## 17.2 Pilas

Una *pila* es un tipo de lista en el que todas las inserciones y eliminaciones de elementos se realizan por el mismo extremo de la lista.

El nombre de pila procede de la similitud en el manejo de esta estructura de datos y la de una “pila de objetos”. Estas estructuras también son llamadas listas LIFO,<sup>1</sup> acrónimo que refleja la característica más importante de las pilas.

---

<sup>1</sup>Del inglés *Last-In-First-Out*, es decir, el último en entrar es el primero en salir.

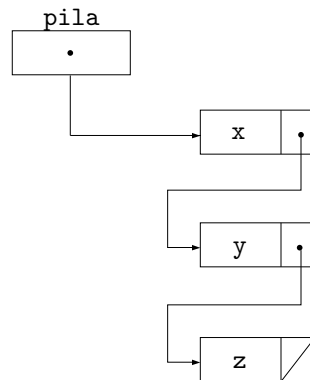


Figura 17.6. Representación gráfica del concepto de pila.

Es fácil encontrar ejemplos de pilas en la vida real: hay una pila en el aparato dispensador de platos de un autoservicio, o en el cargador de una pistola automática, o bien en el tráfico en una calle sin salida. En todos estos casos, el último ítem que se añade a la pila es el primero en salir. Gráficamente, podemos observar esto en la figura 17.6.

Dentro del contexto informático, una aplicación importante de las pilas se encuentra en la implementación de la recursión, que estudiaremos con cierto detalle más adelante.

### 17.2.1 Definición de una pila como lista enlazada

Puesto que una pila es una lista en la cual sólo se insertan o eliminan elementos por uno de sus extremos, podríamos usar la declaración dada de las listas y usarlas restringiendo su acceso del modo descrito anteriormente.

```

type
  tElem = char; {o lo que corresponda}
  tPila = ^tNodoPila;
  tNodoPila = record
    contenido: tElem;
    siguiente: tPila
  end; {tNodoPila}

```

### 17.2.2 Operaciones básicas sobre las pilas

Entre las operaciones básicas del tipo pila se encuentran la creación de una pila, la consulta del contenido del primer elemento de la pila, la inserción de

un nuevo elemento sobre la pila (que se suele llamar `push`), y la eliminación del elemento superior de la pila (también llamada `pop`).<sup>2</sup>

### Creación de una pila vacía

El primer procedimiento que consideramos es el de crear una pila vacía. La idea es bien simple: sólo hay que asignar el valor `nil` a su puntero.

```
procedure CrearPila(var pila: tPila);
  {PostC.: pila es una pila vacía}
begin
  pila:= nil
end; {CrearPila}
```

### Averiguar si una pila está vacía

Se trata sencillamente de la siguiente función

```
function EsPilaVacía(pila: tPila): boolean;
begin
  EsPilaVacía:= pila = nil
end; {EsPilaVacía}
```

### Consulta de la cima de una pila

Una función especialmente importante es la que permite consultar el contenido del primer nodo de la pila. En esta implementación sólo hay que leer el campo `contenido` del primer nodo de la pila. La función tiene como argumento una variable de tipo `tPila` y como rango el tipo de sus elementos `tElem`. Está implementada a continuación bajo el nombre de `Cima`.

```
function Cima(pila: tPila): tElem;
  {PreC.: pila no está vacía}
  {Dev. el contenido del primer nodo de pila}
begin
  Cima:= pila^.contenido
end; {Cima}
```

---

<sup>2</sup>Los nombres de estas funciones proceden de los verbos usados en inglés para colocar o coger objetos apilados.

### Añadir un nuevo elemento en la cima de la pila

Este procedimiento toma como parámetros una pila y la variable que se va a apilar; la acción que realiza es la de insertar el objeto como un nuevo nodo al principio de la pila. La definición de este procedimiento es directa a partir del método de inserción de un nodo al principio de una lista visto en el apartado anterior.

```

procedure Apilar(nuevoDato: tElem; var pila: tPila);
  {Efecto: nuevoDato se añade sobre pila}
  var
    pilaAux: tPila;
  begin
    pilaAux:= pila;
    New(pila);
    pila^.contenido:= nuevoDato;
    pila^.siguiente:= pilaAux
  end; {Apilar}

```

### Eliminar la cima de la pila

Para quitar un elemento de la pila sólo debemos actualizar el puntero. La definición del procedimiento no necesita mayor explicación, ya que es idéntica a la del procedimiento Eliminar presentado en el apartado 17.1.3.

```

procedure SuprimirDePila(var pila: tPila);
  {PreC.: pila no está vacía}
  {Efecto: se suprime el dato de la cima de la pila}
  var
    pilaAux: tPila;
  begin
    pilaAux:= pila;
    pila:= pila^.siguiente;
    Dispose(pilaAux)
  end; {SuprimirDePila}

```

## 17.2.3 Aplicaciones

### Pilas y recursión

Una aplicación importante de las pilas surge al tratar la recursión (véase el capítulo 10). En efecto, en cada llamada recursiva se añade una *tabla de activación* en una pila (denominada *pila recursiva*). Dicha tabla incorpora los argumentos y objetos locales con su valor en el momento de producirse la llamada.

Dicho de otro modo, la recursión se puede transformar en un par de bucles que se obtienen apilando las llamadas recursivas (primer bucle) para después ir evaluándolas una a una (con el segundo bucle).

Un ejemplo servirá para aclarar estas ideas; supongamos que tenemos una función definida recursivamente, como, por ejemplo, la función factorial (véase el apartado 10.1):

```
function Fac(num: integer): integer;
  {PreC.: num ≥ 0}
  {Dev. num!}
begin
  if num = 0 then
    Fac:= 1
  else
    Fac:= num * Fac(num - 1)
end; {Fac}
```

Consideremos cómo se ejecuta la función `Fac` aplicada a un entero, por ejemplo el 3:

$$\begin{aligned} \text{Fac}(3) &= 3 * \text{Fac}(2) = 3 * 2 * \text{Fac}(1) \\ &= 3 * 2 * 1 * \text{Fac}(0) = 3 * 2 * 1 * 1 = 3! \end{aligned}$$

El primer bucle de los comentados más arriba consistiría en ir apilando los argumentos sucesivos de `Fac` hasta llegar al caso base, en este ejemplo tenemos 3, 2, 1; el segundo bucle es el encargado de completar las llamadas recursivas usando la parte recursiva de `Fac`, esto es, se parte del caso base  $\text{Fac}(0) = 1$  y se van multiplicando los distintos valores apilados. Atendiendo a esta descripción, un primer nivel de diseño para la versión iterativa del factorial podría ser el siguiente:

```
para cada n entre num y 1
  Apilar n en pilaRec
  Dar valor inicial 1 a fac
  mientras que pilaRec no esté vacía hacer
    fac:= Cima(pilaRec) * fac
    SuprimirDePila(pilaRec)
  Devolver fac
```

La implementación iterativa en Pascal de la función factorial se ofrece a continuación:

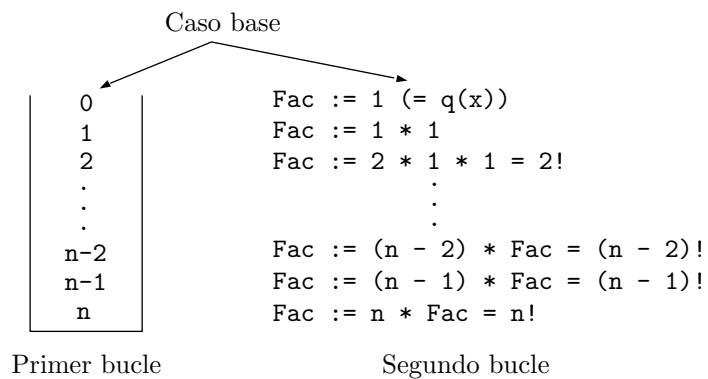


Figura 17.7. Expresión en dos bucles de la función factorial.

```

function FacIter (num: integer): integer;
  {PreC.: num ≥ 0}
  {Dev. num!}
  var
    pilaRec: tPila; {de enteros}
    n, fac: integer;
begin
  n := num;
  CrearPila(pilaRec);
  {Primer bucle: acumulación de las llamadas}
  for n := num downto 1 do
    Apilar (n, pilaRec);
    {Segundo bucle: resolución de las llamadas}
  fac := 1; {Caso base}
  while pilaRec <> nil do begin
    fac := Cima(pilaRec) * fac;
    SuprimirDePila(pilaRec)
  end; {while}
  FacIter := fac
end; {FacIter}

```

En la figura 17.7 puede verse gráficamente el significado de los dos bucles que aparecen en el programa anterior.

La descomposición anterior de la función factorial no es más que un ejemplo del caso general de transformación de recursión en iteración que se expone seguidamente.

Supongamos que se tiene una función definida recursivamente, de la siguiente manera:

```

function F(x: tdato): tResultado;
begin
  if P(x) then
    F:= Q(x)
  else
    F:= E(x, F(T(x)))
end; {F}

```

donde  $P(x)$  es una expresión booleana para determinar el caso base (en el factorial es  $x = 0$ ,  $Q(x)$  es el valor de la función en el caso base,  $T$  es una función que transforma el argumento  $x$  en el de la siguiente llamada (en el factorial es  $T(x) = x - 1$ ), y, finalmente,  $E(x, y)$  es la expresión que combina el argumento  $x$  con el resultado  $y$  devuelto por la llamada recursiva subsiguiente (en el factorial se tiene  $E(x, y) = x * y$ ).

En resumen, cualquier función de la forma de la función  $F$  puede ser descrita mediante un par de bucles de forma que:

1. El primero almacena los parámetros de las sucesivas llamadas recursivas al aplicar la función  $T$  hasta llegar al caso base.
2. El segundo deshace la recursión aplicando la expresión  $E(x, F(T(x)))$  repetidamente desde el caso base hasta el argumento inicial.

La descripción general de estos dos bucles se muestra a continuación:

```

function F(x: tDato): tResultado;
  var
    pilaRec: tPila;
    acum: tDato;
begin
  CrearPila(pilaRec);
  {Primer bucle}
  while not P(x) do begin
    Apilar(x, pilaRec);
    x:= T(x)
  end; {while}
  acum:= Q(x); {Aplicación de F al caso base}
  {Segundo bucle}
  while pilaRec <> nil do begin
    acum:= E(Cima(pilaRec), acum);
    SuprimirDePila(pilaRec)
  end; {while}
  F:= acum
end; {F}

```

El proceso descrito anteriormente para una función recursiva puede llevarse a cabo también para procedimientos recursivos. En el apartado de ejercicios se propone su generalización a un procedimiento recursivo.

### Evaluación postfija de expresiones

¿Quién no ha perdido algún punto por olvidarse un paréntesis realizando un examen? Este hecho, sin duda, le habrá llevado a la siguiente pregunta: ¿No será posible eliminar estos molestos signos del ámbito de las expresiones aritméticas? La respuesta es afirmativa: existe una notación para escribir expresiones aritméticas sin usar paréntesis, esta notación recibe el nombre de *notación postfija* o *notación polaca inversa*.

El adjetivo postfija o inversa se debe a que, en contraposición a la notación habitual (o infija) los operadores se ponen detrás de (y no entre) los argumentos. Por ejemplo, en lugar de escribir  $a + b$  se ha de escribir  $a b +$ , y en vez de  $a * (b + c)$  se escribirá  $a b c + *$ .

Para entender una expresión postfija hay que leerla de derecha a izquierda; la expresión anterior, por ejemplo, es un producto, si seguimos leyendo encontramos un  $+$ , lo cual indica que uno de los factores del producto es una suma. Siguiendo con la lectura vemos que tras el signo  $+$  aparecen  $c$  y  $b$ , luego el primer factor es  $b + c$ ; finalmente hallamos  $a$ , de donde la expresión infija del miembro de la izquierda es  $a * (b + c)$ .

Aunque es probable que el lector no se encuentre a gusto con esta notación y comience a añorar el uso de paréntesis, la ventaja de la notación postfija reside en que es fácilmente implementable en un computador,<sup>3</sup> al no hacer uso de paréntesis ni de reglas de precedencia (esto es, convenios tales como que  $a * b + c$  representa  $(a * b) + c$  y no  $a * (b + c)$ ).

La figura 17.8 muestra gráficamente el proceso de evaluación de la expresión  $2 4 * 3 +$ . Más detalladamente, haciendo uso de una pila, un computador interpretará esa expresión de la siguiente forma:

1. Al leer los dos primeros símbolos, 2 y 4, el computador aún no sabe qué ha de hacer con ellos, así que los pone en una pila.
2. Al leer el siguiente símbolo,  $*$ , saca dos elementos de la pila, los multiplica, y pone en ella su valor, 8.
3. El siguiente símbolo, 3, se coloca en la pila, pues no tenemos ningún operador que aplicar.

---

<sup>3</sup>De hecho, los compiladores de lenguajes de alto nivel traducen las expresiones aritméticas a notación postfija para realizar las operaciones. Hay lenguajes que usan siempre notación postfija, como, por ejemplo, el lenguaje de descripción de página PostScript.

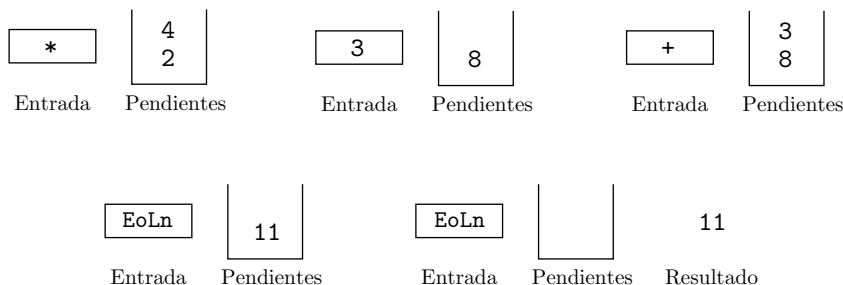


Figura 17.8. Evaluación postfija de una expresión.

- Después se lee el símbolo  $+$ , con lo que se toman dos elementos de la pila, 3 y 8, se calcula su suma, 11, y se pone en la pila.
- Para terminar, ya no hay ningún símbolo que leer y sólo queda el resultado de la operación.

Una vez visto el ejemplo anterior, la evaluación postfija de expresiones aritméticas es muy sencilla de codificar: sólo se necesita una pila para almacenar los operandos y las operaciones básicas de las pilas. La codificación en Pascal se deja como ejercicio.

### 17.3 Colas

Una *cola* es una lista en la que todas las inserciones se realizan por un extremo y todas las eliminaciones se realizan por el otro extremo de la lista.

El ejemplo más importante de esta estructura de datos, del cual recibe el nombre, es el de una cola de personas ante una ventanilla. En esta situación, el primero en llegar es el primero en ser servido; por esto, las colas también se llaman listas FIFO.<sup>4</sup>

Un ejemplo más interesante, dentro de un contexto informático, resulta al considerar la gestión de trabajos de una impresora conectada en red. Todos los archivos por imprimir se van guardando en una cola y se irán imprimiendo según el orden de llegada (sería ciertamente una ruindad implementar esta lista como una pila).

<sup>4</sup>Del inglés *First-In-First-Out*, es decir, el primero en entrar es el primero en salir.

### 17.3.1 Definición del tipo cola

Las colas, como listas que son, podrían definirse de la manera habitual, esto es:

```

type
  tElem = char; {o lo que corresponda}
  tCola = ^tNodoCola;
  tNodoCola = record
    contenido: tElem;
    siguiente: tCola
  end; {tNodoCola}

```

Sin embargo, algunas operaciones, como poner un elemento en una cola, no resultan eficientes, debido a que debe recorrerse la lista en su totalidad para llegar desde su primer elemento hasta el último. Por ello, suele usarse otra definición, considerando una cola como un par de punteros:

```

type
  tElem = char; {o lo que corresponda}
  tApNodo = ^tNodoCola;
  tNodoCola = record
    contenido: tElem;
    siguiente: tApNodo
  end; {tNodoCola}
  tCola = record
    principio: tApNodo;
    final: tApNodo
  end; {tCola}

```

Con esta definición de colas, cualquier operación que altere la posición de los nodos extremos de la lista deberá actualizar los punteros `principio` y `final`; sin embargo, esto resulta ventajoso en comparación con la obligatoriedad de recorrer toda la cola para añadir un nuevo elemento.

### 17.3.2 Operaciones básicas

Veamos cómo implementar las operaciones básicas con colas siguiendo la definición anterior.

#### Creación de una cola vacía

Para crear una cola se necesita iniciar a **nil** los campos `principio` y `final` del registro. Nada más fácil:

```

procedure CrearCola(var cola: tCola);
begin
    cola.principio:= nil;
    cola.final:= nil
end; {CrearCola}

```

Por eficiencia en la implementación de los siguientes procedimientos resultará conveniente considerar una lista vacía a aquella cuyo puntero **final** tiene el valor **nil**; esto permitirá que para comprobar si una cola es o no vacía sólo se necesite evaluar la expresión lógica `cola.final = nil`. Es trivial escribir el subprograma que efectúa esta comprobación.

### Consulta del primer elemento

Esta operación es idéntica a **Cima** de una pila, puesto que nos interesa el contenido del primer nodo de la cola. Para verificar el nivel de comprensión sobre el tema, es un buen ejercicio intentar escribirla (sin mirar el análogo para pilas, como es natural).

### Añadir un elemento

En las colas, los elementos nuevos se sitúan al final y por esta razón es necesario actualizar el puntero **final** (y también **principio** si la cola está vacía). Un simple gráfico servirá para convencerse cómo. El diseño de este procedimiento es el siguiente:

```

si cola no está vacía entonces
    Crear y dar valores al nuevoNodo
    Actualizar los punteros y el final de cola
en otro caso
    Crear y dar valores al nuevoNodo
    Actualizar principio y final de cola

```

La codificación en Pascal de este diseño podría ser la siguiente:

```

procedure PonerEnCola(dato: tElem; var cola: tCola);
    {Efecto: dato se añade al final de cola}
    var
        nuevoNodo: tApNodo;
begin
    New(nuevoNodo);
    nuevoNodo^.contenido:= dato;
    nuevoNodo^.siguiente:= nil;

```

```

if cola.final <> nil then begin
    {Si la cola no está vacía se actualizan los punteros}
    cola.final^.siguiente:= nuevoNodo;
    cola.final:= nuevoNodo
end {then}
else begin
    {Actualización de punteros;}
    cola.principio:= nuevoNodo;
    cola.final:= nuevoNodo
end {else}
end; {PonerEnCola}

```

### Suprimir el primer elemento

Para definir SacarDeCola tenemos que considerar dos casos distintos:

1. Que la cola sea unitaria, pues al sacar su único elemento se queda vacía, con lo que hay que actualizar tanto su **principio** como su **final**.
2. Que la cola tenga longitud mayor o igual a dos, en cuyo caso sólo hay que actualizar el campo **principio**.

A continuación se muestra la implementación del procedimiento:

```

procedure SacarDeCola(var cola: tCola);
    {PreC.: cola es no vacía}
    {Efecto.: se extrae el primer elemento de cola}
    var
        apuntaAux : tApNodo;
begin
    if cola.principio = cola.final then begin
        {La cola es unitaria}
        apuntaAux:= cola.principio;
        Dispose(apuntaAux);
        cola.principio:= nil;
        cola.final:= nil
    end {then}
    else begin
        {Si Longitud(cola) >= 2;}
        apuntaAux:= cola.principio;
        cola.principio:= apuntaAux^.siguiente;
        Dispose(apuntaAux)
    end {else}
end; {SacarDeCola}

```

### 17.3.3 Aplicación: gestión de la caja de un supermercado

Con la ayuda del tipo cola se presenta a continuación un programa de simulación del flujo de clientes en una caja de supermercado.

Una cantidad prefijada de clientes se va incorporando a la cola en instantes discretos con una cierta probabilidad también fijada de antemano;<sup>5</sup> cada cliente lleva un máximo de artículos, de forma que cada artículo se contabiliza en una unidad de tiempo. Cuando a un cliente le han sido contabilizados todos los artículos, es eliminado de la cola. En cada instante se muestra el estado de la cola.

Un primer esbozo en pseudocódigo de este programa sería el siguiente:

```

Leer valores iniciales
Iniciar contadores
repetir
  Mostrar la cola
  Procesar cliente en caja
  Procesar cliente en cola
hasta que se acaben los clientes y los artículos

```

La lectura de los valores iniciales consiste en pedir al usuario del programa el número de clientes, `numClientes`, la probabilidad de que aparezca un cliente por unidad de tiempo, `probLlegada`, y el número máximo de artículos por cliente, `maxArti`. Tras definir los valores iniciales de los contadores de tiempo `t`, y de clientes puestos en cola, `contClientes`, sólo hay que refinar las acciones del bucle del diseño anterior. Así, en un nivel de diseño más detallado *Procesar cliente en caja* se descompondría en

```

si el cliente ya no tiene artículos entonces
  Retirarlo de la cola
en otro caso
  Reducir el número de artículos del cliente

```

Mientras que el refinamiento de *Procesar cliente en cola* sería

```

si quedan clientes y random < probabilidad prefijada entonces
  Añadir un cliente e incrementar el contador de clientes

```

A partir del refinamiento anterior ya podemos pasar a la implementación en Pascal de este programa.

---

<sup>5</sup>Para incluir factores de aleatoriedad se hace uso de `Randomize` y `Random` de Turbo Pascal (véase el apartado A.2.1).

El único tipo que se necesita definir para este programa es el tipo `tCola`. Las variables que almacenarán los datos del programa son el número de personas que llegarán a la caja, `numClientes`, el número máximo de artículos que cada persona puede llevar, `maxArti`, y la probabilidad de que una persona se una a la cola, `probLlegada`. Además se usará la variable `t` para medir los distintos instantes de tiempo, la variable `contClientes` para contar el número de personas que se han puesto en la cola y la variable `caja`, que es de tipo `tCola`.

El encabezamiento del programa es el siguiente:

```

Program SimuladorDeColas (input, output);
type
  tElem = integer;
  tApNodo = ^tNodo;
  tNodo = record
    contenido: tElem;
    siguiente: tApNodo
  end; {tNodo}
  tCola = record
    principio, final: tApNodo
  end; {tCola}
var
  t, contClientes, numClientes, maxArti: integer;
  probLlegada: real;
  caja: tCola;

```

Se usarán los siguientes procedimientos estándar del tipo `tCola`: `CrearCola`, `PonerEnCola`, `SacarDeCola`, y además `MostrarCola` que, como su nombre indica, muestra todos los elementos de la cola. Su implementación se presenta a continuación:

```

procedure MostrarCola(coola: tCola);
  {Efecto: muestra en pantalla todos los elementos de cola}
  var
    apuntaAux: tApNodo;
  begin
    if cola.final= nil then
      WriteLn('La caja está desocupada')
    else begin
      apuntaAux:= cola.principio;
      repeat
        WriteLn(apuntaAux^.contenido);
        apuntaAux:= apuntaAux^.siguiente
      until apuntaAux= nil
    end {else}
  end; {MostrarCola}

```

En el cuerpo del programa, tal como se especifica en el diseño, tras la lectura de datos, se realiza la siguiente simulación en cada instante  $t$ : se procesa un artículo en la caja (se anota su precio), cuando se acaban todos los artículos de un determinado cliente éste se elimina de la cola (paga y se va); por otra parte se comprueba si ha llegado alguien (si su número aleatorio es menor o igual que `probLlegada`), si ha llegado se le pone en la cola y se actualiza el contador de personas.

Finalmente, el cuerpo del programa es el siguiente:

```

begin
  Randomize;
  CrearCola(caja);
  t:= 0;
  contClientes:= 0;
  WriteLn('Simulador de Colas');
  WriteLn('Introduzca número de personas');
  ReadLn(numClientes);
  WriteLn('Introduzca la probabilidad de llegada');
  ReadLn(probLlegada);
  WriteLn('Introduzca máximo de artículos');
  ReadLn(maxArti);
  repeat
    Writeln('Tiempo t =',t);
    MostrarCola(caja);
    t:= t + 1;
    if caja.principio^.contenido= 0 then
      SacarDeCola(caja)
    else with caja.principio^ do
      contenido:= contenido - 1;
      if (Random <= probLlegada) and (contClientes < numClientes)
      then begin
        PonerEnCola (Random(maxArti) + 1, caja);
        contClientes:= contClientes + 1
      end; {if}
    until (contClientes = numClientes) and (caja.principio= nil)
  end. {SimuladorDeColas}

```

## 17.4 Árboles binarios

Es posible representar estructuras de datos más complejas que las listas haciendo uso de los punteros. Las listas se han definido como registros que contienen datos y un puntero al siguiente nodo de la lista; una generalización del concepto de lista es el árbol, donde se permite que cada registro del tipo de dato dinámico tenga más de un enlace. La naturaleza lineal de las listas hace

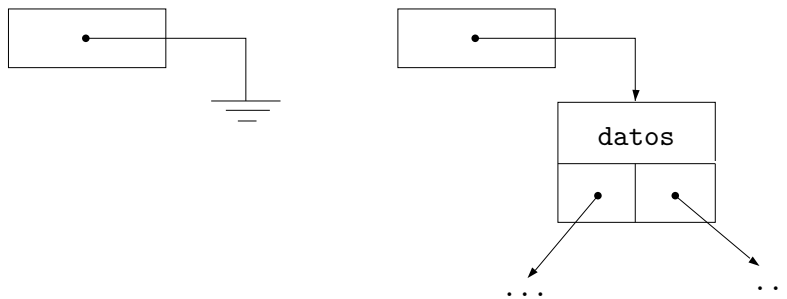


Figura 17.9.

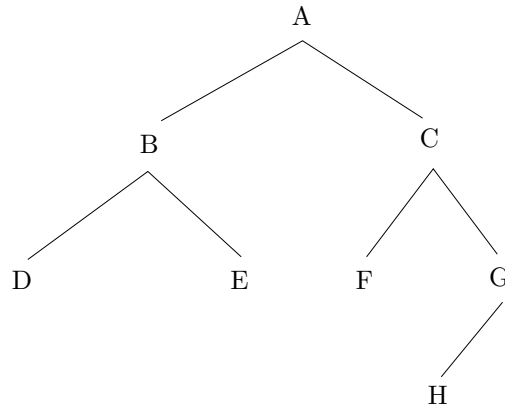


Figura 17.10.

posible, y fácil, definir algunas operaciones de modo iterativo; esto no ocurre con los árboles, que son manejados de forma natural haciendo uso de la recursión.

Los árboles son estructuras de datos recursivas más generales que una lista y son apropiados para aplicaciones que involucren algún tipo de jerarquía (tales como los miembros de una familia o los trabajadores de una organización), o de ramificación (como los árboles de juegos), o de clasificación y/o búsqueda. La definición recursiva de árbol es muy sencilla: Un *árbol* o es vacío o consiste en un nodo que contiene datos y punteros hacia otros árboles. Es decir, la representación gráfica de un árbol es una de las dos que aparecen en la figura 17.9.

En este apartado sólo trataremos con *árboles binarios*, que son árboles en los que cada nodo tiene a lo sumo dos descendientes. En la figura 17.10 vemos la representación gráfica de un árbol binario.

La terminología usada cuando se trabaja con árboles es, cuando menos, curiosa. En ella se mezclan conceptos botánicos como raíz y hoja y conceptos genealógicos tales como hijo, ascendientes, descendientes, hermanos, padres, etc. En el árbol de la figura 17.10, el nodo A es la *raíz* del árbol, mientras que los nodos D, E, F y H son las *hojas* del árbol; por otra parte, los *hijos* de la raíz son los nodos B y C, el *padre* de E es B, ...

La definición en Pascal del tipo árbol binario es sencilla: cada nodo del árbol va a tener dos punteros en lugar de uno.

```

type
  tElem = char;    {o el tipo que corresponda}
  tArbol = ^tNodoArbol;
  tNodoArbol = record
    hIzdo, hDcho: tArbol;
    contenido: tElem
  end; {tNodoArbol}

```

La creación de un árbol vacío, a estas alturas, no representa problema alguno. Sin embargo no ocurre lo mismo para realizar una consulta, ya que debemos saber cómo movernos dentro del árbol.

### 17.4.1 Recorrido de un árbol binario

Definir un algoritmo de recorrido de un árbol binario no es una tarea directa ya que, al no ser una estructura lineal, existen distintas formas de recorrerlo. En particular, al llegar a un nodo podemos realizar una de las tres operaciones siguientes:

- (i) Leer el valor del nodo.
- (ii) Seguir por el hijo izquierdo.
- (iii) Seguir por el hijo derecho.

El orden en el que se efectúen las tres operaciones anteriores determinará el orden en el que los valores de los nodos del árbol son leídos. Si se postula que siempre se leerá antes el hijo izquierdo que el derecho, entonces existen tres formas distintas de recorrer un árbol:

**Preorden:** Primero se lee el valor del nodo y después se recorren los subárboles. Esta forma de recorrer el árbol también recibe el nombre de *recorrido primero en profundidad*.

El árbol de la figura 17.10 recorrido en preorden se leería así: ABDECFGH.

**Inorden:** En este tipo de recorrido, primero se recorre el subárbol izquierdo, luego se lee el valor del nodo y, finalmente, se recorre el subárbol derecho.

El árbol de la figura 17.10 recorrido en inorden se leería así: DBEAFCHG.

**Postorden:** En este caso, se visitan primero los subárboles izquierdo y derecho y después se lee el valor del nodo.

El árbol de la figura 17.10 recorrido en postorden se leería así: DEBFHGCA.

Ahora, escribir un procedimiento recursivo para recorrer el árbol (en cualquiera de los tres órdenes recién definidos) es tarea fácil, por ejemplo:

```

procedure RecorrerEnPreorden(arbol: tArbol);
begin
  if arbol <> nil then begin
    Visitarnodo(arbol); {por ejemplo, Write(arbol^.contenido)}
    RecorrerEnPreorden(arbol^.hIzdo);
    RecorrerEnPreorden(arbol^.hDcho);
  end {if}
end; {RecorrerEnPreorden}

```

para utilizar otro orden en el recorrido sólo hay que cambiar el orden de las acciones *Visitarnodo* y las llamadas recursivas.

### 17.4.2 Árboles de búsqueda

Como un caso particular de árbol binario se encuentran los *árboles binarios de búsqueda* (o *árboles de búsqueda binaria*), que son aquellos árboles en los que el valor de cualquier nodo es mayor que el valor de su hijo izquierdo y menor que el de su hijo derecho. Según la definición dada, no puede haber dos nodos con el mismo valor en este tipo de árbol.

- ☞ Obsérvese que los nodos de un árbol binario de búsqueda se pueden enumerar en orden creciente siguiendo un recorrido en inorden.

La utilidad de los árboles binarios de búsqueda reside en que si buscamos cierta componente, podemos decir en qué mitad del árbol se encuentra comparando solamente con el nodo raíz. Nótese la similitud con el método de búsqueda binaria.

- ☉☉ Una mejora de los árboles de búsqueda consiste en añadir un campo *clave* en cada nodo y realizar las búsquedas comparando los valores de dichas claves en lugar de los valores del campo *contenido*. De esta forma, pueden existir en el árbol dos nodos con el mismo valor en el campo contenido pero con clave distinta. En este texto se implementan los árboles de búsqueda sin campo clave para simplificar la presentación; la modificación de la implementación para incluir un campo clave es un ejercicio trivial.

## Operaciones básicas

Las operaciones básicas para el manejo de árboles de búsqueda son la consulta, la inserción y la eliminación de nodos. Las dos primeras son de fácil implementación, haciendo uso de la natural recursividad de los árboles. La operación de eliminación de nodos es, sin embargo, algo más compleja, como se detalla a continuación.

## Búsqueda de un nodo

Debido al orden intrínseco de un árbol de búsqueda binaria, es fácil implementar una función que busque un determinado valor entre los nodos del árbol y, en caso de encontrarlo, proporcione un puntero a ese nodo. La versión recursiva de la función es particularmente sencilla, todo consiste en partir de la raíz y rastrear el árbol en busca del nodo en cuestión, según el siguiente diseño:

```

si arbol es vacío entonces
  Devolver fallo
en otro caso si arbol^.contenido = dato entonces
  Devolver el puntero a la raíz de arbol
en otro caso si arbol^.contenido > dato entonces
  Buscar en el hijo izquierdo de arbol
en otro caso si arbol^.contenido < dato entonces
  Buscar en el hijo derecho de arbol

```

La codificación en Pascal es directa:

```

function Encontrar(dato: tElem; arbol: tArbol): tArbol;
  {Dev. un puntero al nodo con dato, si dato está en arbol, o
  nil en otro caso}
begin
  if arbol = nil then
    Encontrar := nil
  else with arbol do
    if dato < contenido then

```

```

    Encontrar:= Encontrar (dato, hIzdo)
  else if datos > contenido then
    Encontrar:= Encontrar (dato, hDcho)
  else Encontrar:= arbol
end; {Encontrar}

```

### Inserción de un nuevo nodo

El siguiente procedimiento inserta un nuevo nodo en un árbol binario de búsqueda *árbol*; la inserción del nodo es tarea fácil, todo consiste en encontrar el lugar adecuado donde insertar el nuevo nodo, esto se hace en función de su valor de manera similar a lo visto en el ejemplo anterior. El pseudocódigo para este procedimiento es:

```

si arbol es vacío entonces
  crear nuevo nodo
en otro caso si arbol^.contenido > datoNuevo entonces
  Insertar ordenadamente en el hijo izquierdo de arbol
en otro caso si arbol^.contenido < datoNuevo entonces
  Insertar ordenadamente en el hijo derecho de arbol

```

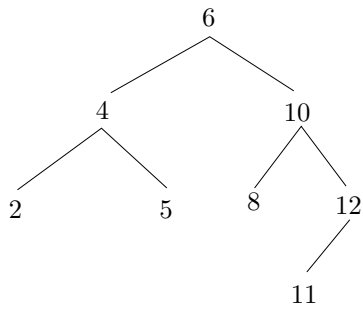
Y la implementación en Pascal es:

```

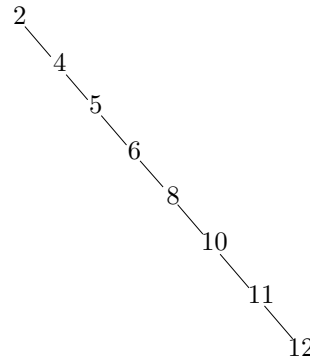
procedure Insertar(datoNuevo: tElem; var arbol: tArbol);
  {Efecto: se añade ordenadamente a arbol un nodo de contenido
  datoNuevo}
begin
  if arbol = nil then begin
    New(arbol);
    with arbol^ do begin
      hIzdo:= nil;
      hDcho:= nil;
      contenido:= datoNuevo
    end {with}
  end {then}
  else with arbol^ do
    if datoNuevo < contenido then
      Insertar(datoNuevo,hIzdo)
    else if datoNuevo > contenido then
      Insertar(datoNuevo,hDcho)
    else {No se hace nada: entrada duplicada}
  end; {Insertar}

```

La forma resultante de un árbol binario de búsqueda depende bastante del orden en el que se vayan insertando los datos: si éstos ya están ordenados el árbol degenera en una lista (véase la figura 17.11).



Datos: 6, 4, 2, 10, 5, 12, 8, 11



Datos: 2, 4, 5, 6, 8, 10, 11, 12

Figura 17.11.

### Supresión de un nodo

Eliminar un nodo en un árbol de búsqueda binaria es la única operación que no es fácil de implementar: hay que considerar los distintos casos que se pueden dar, según el nodo por eliminar sea

1. Una hoja del árbol,
2. Un nodo con un sólo hijo, o
3. Un nodo con dos hijos.

Los dos primeros casos no presentan mayor problema; sin embargo, el tercero requiere un análisis detallado para suprimir el nodo de modo que el árbol resultante siga siendo un árbol binario de búsqueda. En primer lugar hay que situarse en el nodo padre del nodo por eliminar; después procederemos por pasos, analizando qué se necesita hacer en cada caso:

1. Si el nodo por eliminar es una hoja, entonces basta con destruir su variable asociada (usando `Dispose`) y, posteriormente, asignar `nil` a ese puntero.
2. Si el nodo por eliminar sólo tiene un subárbol, se usa la misma idea que al eliminar un nodo interior de una lista: hay que “saltarlo” conectando directamente el nodo anterior con el nodo posterior y desechando el nodo por eliminar.
3. Por último, si el nodo por eliminar tiene dos hijos no se puede aplicar la técnica anterior, simplemente porque entonces habría dos nodos que

conectar y no obtendríamos un árbol binario. La tarea consiste en eliminar el nodo deseado y recomponer las conexiones de modo que se siga teniendo un árbol de búsqueda.

En primer lugar, hay que considerar que el nodo que se coloque en el lugar del nodo eliminado tiene que ser mayor que todos los elementos de su subárbol izquierdo, luego la primera tarea consistirá en buscar tal nodo; de éste se dice que es el *predecesor* del nodo por eliminar (¿en qué posición se encuentra el nodo predecesor?).

Una vez hallado el predecesor el resto es bien fácil, sólo hay que copiar su valor en el nodo por eliminar y desechar el nodo predecesor.

A continuación se presenta un esbozo en pseudocódigo del algoritmo en cuestión; la implementación en Pascal del procedimiento se deja como ejercicio indicado.

*Determinar el número de hijos del nodo N a eliminar  
si N no tiene hijos entonces  
eliminarlo  
en otro caso si N sólo tiene un hijo H entonces  
Conectar H con el padre de N  
en otro caso si N tiene dos hijos entonces  
Buscar el predecesor de N  
Copiar su valor en el nodo a eliminar  
Desechar el nodo predecesor*

### 17.4.3 Aplicaciones

#### Recorrido en anchura de un árbol binario

El uso de colas resulta útil para describir el recorrido en anchura de un árbol. Hemos visto tres formas distintas de recorrer un árbol binario: recorrido en preorden, en inorden y en postorden. El *recorrido primero en anchura* del árbol de la figura 17.10 nos da la siguiente ordenación de nodos: ABCDEFGH. La idea consiste en leer los nodos nivel a nivel, primero la raíz, luego (de izquierda a derecha) los nodos de profundidad 1, los de profundidad 2, etc.

Analicemos cómo trabaja el algoritmo que hay que codificar, para ello seguiremos el recorrido en anchura sobre el árbol de la figura 17.10:

1. En primer lugar se lee la raíz de árbol: A.
2. Luego hay que leer los hijos de la raíz, B y C.
3. Después, se leen D y E (hijos de B) y F y G (hijos de C).
4. Finalmente se lee G, el único nodo de profundidad 3.

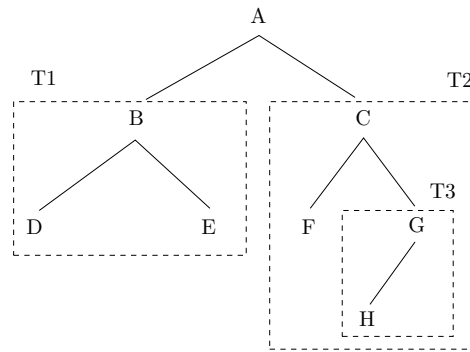


Figura 17.12.

Teniendo en cuenta que los hijos de un nodo son las raíces de sus subárboles hijos se observa que existe una tarea repetitiva, la visita del nodo raíz de un árbol. Lo único que resta es ordenar adecuadamente los subárboles por visitar.

En este punto es donde las colas juegan su papel, pues se va a considerar una cola de subárboles pendientes de visitar, este proceso se representa en la figura 17.12:

1. Tras leer el nodo raíz A se colocan sus dos subárboles hijos en la cola de espera.
2. Se toma el primer árbol de la cola y se visita su raíz, en este caso B, tras lo que se añaden sus subárboles hijos a la cola.
3. ...

La tabla 17.1 muestra, paso a paso, cómo va avanzando el recorrido en anchura y cómo va cambiando el estado de la cola de espera (usando la notación de la figura 17.12 para nombrar a los distintos subárboles con los que se trabaja).

Para la codificación de este recorrido será necesario definir el tipo cola de árboles (el tipo `tCola` por razones de eficiencia) y declarar una variable `enEspera` para almacenar los árboles en espera de ser visitados.

Una primera aproximación en pseudocódigo es la siguiente:

*Crear una cola con el `arbolDato` en ella*  
*Procesar los árboles de esa cola*

La primera acción se refina directamente así:

Recorrido	Cola de espera
[ ]	[T]
[A]	[T1, T2]
[A,B]	[T2,D,E]
[A,B,C]	[D,E,F,T3]
[A,B,C,D]	[E,F,T3]
[A,B,C,D,E]	[F,T3]
[A,B,C,D,E,F]	[T3]
[A,B,C,D,E,F,G]	[H]
[A,B,C,D,E,F,G,H]	[ ]

Tabla 17.1.

```

var
  arbolesEnEspera: cola de árboles
  ...
  CrearCola(arbolesEnEspera);
  PonerEnCola(arbolDato, arbolesEnEspera);

```

donde una cola de árboles se concreta mediante el tipo `tApNodo` (véase el apartado 17.3), siendo sus elementos `tElem` del tipo `tArbol`.

La segunda acción consiste en la serie de pasos de la figura 17.13.

Naturalmente, todo el peso del procedimiento recae en el procesado de los árboles en espera. Para ello hay que leer la raíz del primer árbol de la cola (mientras ésta no esté vacía), sacarlo de ella y añadir al final de la cola sus subárboles hijos. Por tanto, la acción *Procesar los árboles en espera* se puede refinar de la siguiente forma:

```

mientras arbolesEnEspera <> nil hacer
  Sacar el primer árbol de la cola
  Procesar su nodo raíz
  Poner en cola los subárboles no vacíos

```

El refinamiento de cada una de las tareas anteriores es directa haciendo uso de las operaciones descritas para el tipo cola.

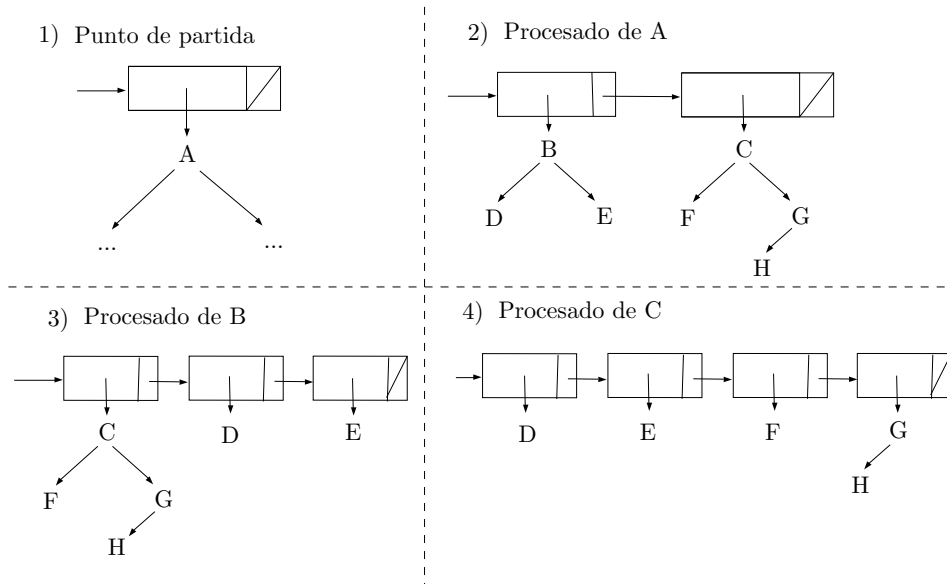


Figura 17.13.

### Árboles de expresiones aritméticas

Mediante un árbol binario pueden expresarse las expresiones aritméticas habituales: suma, resta, producto y cociente. En tal representación cada nodo interior del árbol está etiquetado con el símbolo de una operación aritmética (cuyos argumentos son los valores de las expresiones aritméticas que representan sus subárboles hijos), y sólo las hojas están etiquetadas con valores numéricos.

Es interesante observar la relación entre los distintos tipos de recorrido de un árbol binario con las distintas notaciones para las expresiones aritméticas. Por ejemplo, la siguiente expresión dada en notación habitual (esto es, en notación infija):

$$\left( (3 * (4 + 5)) - (7 : 2) \right) : 6$$

tiene el árbol sintáctico de la figura 17.14.

Si se recorre el árbol en *inorden* entonces se obtiene la notación *infija* (habitual) de la expresión. Si se realiza un recorrido en *postorden* entonces la ordenación de los nodos es la siguiente:

$$3 \ 4 \ 5 \ + \ * \ 7 \ 2 \ : \ - \ 6 \ :$$

que coincide con la expresión en notación *postfija*.

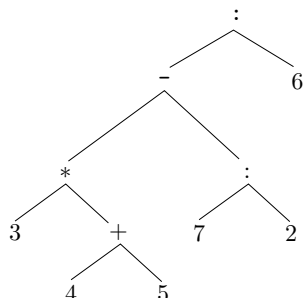


Figura 17.14. Árbol sintáctico de una expresión aritmética.

- ☉☉ El lector atento habrá observado que esta notación postfija no coincide con la definida en el apartado 17.2.3; los argumentos de las operaciones aparecen cambiados de orden, esto no es problema con operaciones conmutativas (suma y producto) pero sí en las que no lo son.

Este problema se subsana definiendo un recorrido en postorden en el que se visita antes el hijo derecho que el izquierdo.

Finalmente, si se recorre el árbol en *preorden* se obtiene una expresión *prefija* de la expresión, en este caso:

$$: - * 3 + 4 5 : 7 2 6$$

## 17.5 Otras estructuras dinámicas de datos

En este último apartado damos una idea de otras aplicaciones de la memoria dinámica sin pasar al estudio de la implementación de las mismas.

### Listas doblemente enlazadas

La implementación de listas dinámicas mostrada en este capítulo adolece de la imposibilidad de acceder directamente al predecesor de un nodo. Este problema causó la inclusión de varios casos especiales en la implementación de algunas operaciones sobre listas, como la inserción o eliminación de nodos intermedios (véase el apartado 17.1.5).

Una lista *doblemente enlazada* o *de doble enlace* se define de modo similar al de las listas de enlace simple, sólo que cada nodo dispone de dos punteros que apuntan al nodo anterior y al nodo siguiente:

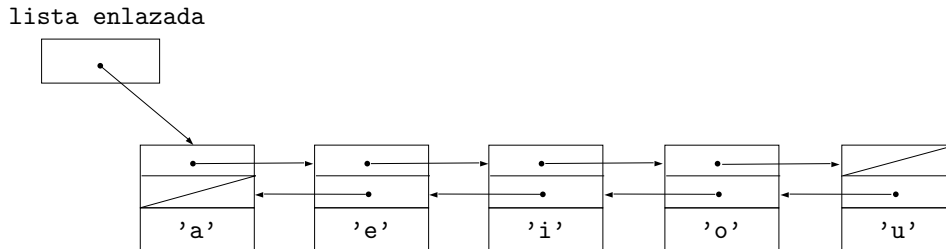


Figura 17.15. Una lista doblemente enlazada.

```

type
  tElem = char; {o lo que corresponda}
  tListaDobleEnlace = ^tNodo;
  tNodo = record
    contenido : tElem;
    anterior, siguiente: tListaDobleEnlace
  end; {tNodo}

```

La figura 17.15 representa una lista doblemente enlazada. Con los nodos primero y último de la lista se pueden tomar dos posturas:

1. Considerar que el anterior del primer nodo es **nil**, y que el siguiente al último también es **nil**.

En este caso resulta útil considerar la lista como un registro con dos componentes: principio y final. Del mismo modo que en la implementación de `tCola`.

2. Considerar que el anterior del primero es el último y que el siguiente al último es el primero.

En este caso, estrictamente hablando, no se obtiene una lista, sino un *anillo* o *lista circular* que carece de principio y de final.

Un análisis más profundo de esta estructura rebasa los límites de este libro; no obstante, en el apartado de comentarios bibliográficos se citan algunas referencias con las que profundizar en el estudio de este tipo de datos.

## Árboles generales

En el apartado 17.4 se han estudiado los árboles binarios; esta estructura se puede generalizar a estructuras en las que los nodos pueden tener más de dos subárboles hijos. Esta generalización puede hacerse de dos formas distintas:

1. Pasando a *árboles  $n$ -arios*, en los que cada nodo tiene a lo sumo  $n$  subárboles hijos.

Un árbol *árbol  $n$ -ario* se puede representar como un árbol de registros de  $n$  componentes, o bien como un vector de subárboles.

2. Considerando *árboles generales*, en los que no existe limitación en el número de subárboles hijo que puede tener.

Los árboles generales se implementan mediante un árbol de listas (puesto que no se sabe el número máximo de hijos de cada nodo).

La elección de un tipo de árbol o de otro depende del problema particular que se esté tratando: el uso de registros acelera el acceso a un hijo arbitrario; sin embargo, un número elevado de registros puede consumir buena parte de la memoria disponible. El otro enfoque, el de un árbol general, resulta apropiado para evitar el derroche de memoria si no se usa la mayoría de los campos de los registros con la contrapartida de un mayor tiempo de acceso a los nodos.

Entre las aplicaciones típicas de los árboles generales se encuentran los árboles de juegos o los árboles de decisión. Veamos brevemente qué se entiende por un árbol de juego (de mesa, como, por ejemplo, las damas o el ajedrez):

La raíz de un árbol de juegos es una posición de las fichas en el tablero; el conjunto de sus hijos lo forman las distintas posiciones accesibles en un movimiento desde la posición anterior. Si no es posible realizar ningún movimiento desde una posición, entonces ese nodo no tiene descendientes, es decir, es una hoja; como ejemplo de árbol de juegos, en la figura 17.16 se muestra un árbol para el juego del tres en raya, representando una estrategia que permite ganar siempre al primero en jugar.

Conviene saber que no es corriente razonar sobre juegos desarrollando explícitamente todo el árbol de posibilidades, sino que suele “recorrerse” implícitamente (hasta cierto punto) al efectuarse llamadas de subprogramas recursivos.

## 17.6 Ejercicios

1. Escriba una versión iterativa de la función `longitud` de una lista.
2. Se denomina *palíndromo* una palabra o frase que se lee igual de izquierda a derecha que de derecha a izquierda. Por ejemplo,

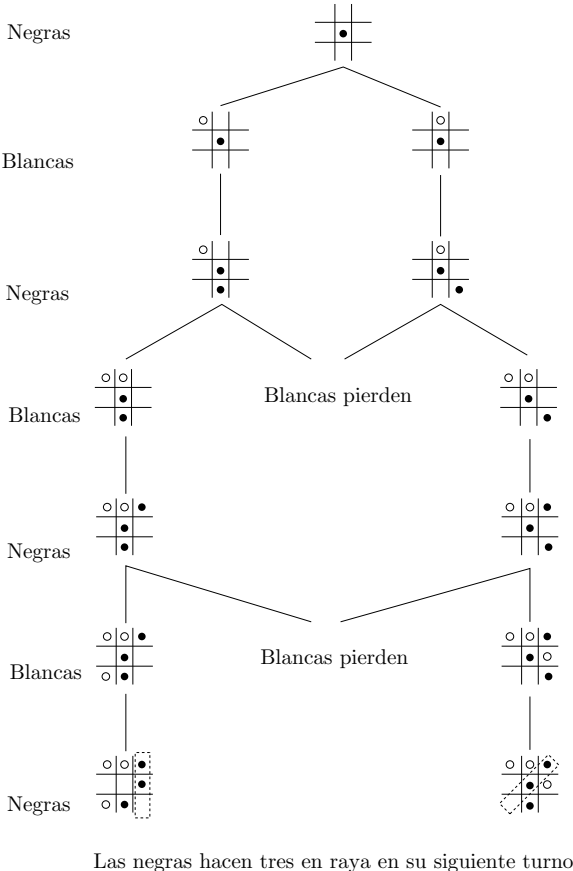


Figura 17.16. Estrategia ganadora para el juego del tres en raya.

## DABALE ARROZ A LA ZORRA EL ABAD

Escriba un programa que lea una cadena de caracteres (terminada por el carácter de fin de línea) y determine si es o no un palíndromo. (Indicación: un método sencillo consiste en crear una lista de letras y su inversa y compararlas para ver si son iguales).

3. Completar el programa de simulación de colas, calculando el tiempo que permanece la caja desocupada y el tiempo medio de espera de los clientes.
4. La función de búsqueda de un dato en un árbol binario de búsqueda se presentó de forma recursiva. Escribir una versión iterativa de dicha función.
5. Escriba un procedimiento que halle el nodo predecesor de un nodo de un árbol binario de búsqueda. Úsese para escribir una codificación completa de la eliminación de un nodo en un árbol de búsqueda binaria.
6. Completar la codificación en Pascal del recorrido en anchura de un árbol. Para el procesado de los nodos visitados límitese a imprimir el contenido de los nodos.
7. Implemente una versión iterativa del procedimiento `PonerEnCola`.
8. Escriba la versión iterativa de un procedimiento recursivo genérico.
9. Escriba una función que evalúe expresiones aritméticas escritas en notación post-fija.

## 17.7 Referencias bibliográficas

En [DL89] se hace un estudio detallado de de las principales estructuras dinámicas de datos con numerosos ejemplos y aplicaciones, siguiendo el mismo estilo que en la primera parte de este texto [DW89].

Una referencia obligada es el clásico libro de N. Wirth [Wir86], que dedica su capítulo 4 al estudio de las estructuras dinámicas de datos. Dentro de ellas hay que destacar el tratamiento de los distintos tipos de árboles: equilibrados, de búsqueda, generales y otros, con interesantes aplicaciones.

Esta parte de nuestro libro debe entenderse como una introducción, ya que las estructuras de datos que pueden construirse con punteros son variadísimas (árboles, grafos, tablas, conjuntos, matrices de gran tamaño y bases de datos, por citar algunos) y pueden llegar a ser muy complejos. Su estudio en detalle corresponde a cursos posteriores. Por citar algunos textos en español sobre el tema destacamos [AM88] y [AHU88].

Finalmente, es obligado mencionar que la idea de los punteros con referencias indirectas de los datos es ampliamente explotada por el lenguaje C y sus extensiones. Precisamente ese uso extensivo es una razón de peso para lograr programas eficientes. Entre las muchas referencias sobre este lenguaje, citamos [KR86].



## **Tema VI**

# **Aspectos avanzados de programación**



## Capítulo 18

# Complejidad algorítmica

---

<b>18.1</b>	<b>Conceptos básicos . . . . .</b>	<b>396</b>
<b>18.2</b>	<b>Medidas del comportamiento asintótico . . . . .</b>	<b>402</b>
<b>18.3</b>	<b>Reglas prácticas para hallar el coste de un programa</b>	<b>408</b>
<b>18.4</b>	<b>Útiles matemáticos . . . . .</b>	<b>418</b>
<b>18.5</b>	<b>Ejercicios . . . . .</b>	<b>422</b>
<b>18.6</b>	<b>Referencias bibliográficas . . . . .</b>	<b>425</b>

---

Con frecuencia, un problema se puede resolver con varios algoritmos, como ocurre, por ejemplo, en los problemas de ordenación y búsqueda de vectores (véase el capítulo 15).

En este capítulo se estudian criterios (presentados escuetamente en el apartado 1.3.3) que permiten al programador decidir cuál de los posibles algoritmos que resuelven un problema es más eficiente. En general, la eficiencia se puede entender como una medida de los recursos empleados por un algoritmo en su ejecución. En particular, usualmente se estudia la eficiencia de un algoritmo en tiempo (de ejecución), espacio (de memoria) o número de procesadores (en algoritmos implementados en arquitecturas paralelas). Como se vio en el apartado 1.3.3, el estudio de la complejidad algorítmica trata de resolver este importantísimo aspecto de la resolución de problemas.

Los criterios utilizados por la complejidad algorítmica no proporcionan medidas absolutas, como podría ser el tiempo total en segundos empleado en la ejecución del programa que implementa el algoritmo, sino medidas relativas al

tamaño del problema. Además, estas medidas son independientes del computador sobre el que se ejecute el algoritmo.

Nuestro objetivo en este tema es proporcionar las herramientas necesarias para el cálculo de la complejidad de los algoritmos. De esta forma, en caso de disponer de más de un algoritmo para solucionar un problema, tendremos elementos de juicio para decidir cuál es mejor desde el punto de vista de la eficiencia.

Con frecuencia, no es posible mejorar simultáneamente la eficiencia en tiempo y en memoria, buscándose entonces un algoritmo que tenga una complejidad razonable en ambos aspectos. Actualmente, y gracias a los avances de la técnica, quizás es más importante el estudio de la complejidad en el tiempo, ya que la memoria de un computador puede ser ampliada fácilmente, mientras que el problema de la lentitud de un algoritmo suele ser más difícil de resolver.

Además, el cálculo de las complejidades en tiempo y en espacio se lleva a cabo de forma muy similar. Por estas razones, se ha decidido dedicar este capítulo esencialmente al estudio de la complejidad en el tiempo.

## 18.1 Conceptos básicos

### El tiempo empleado por un algoritmo se mide en “pasos”

Para medir el tiempo empleado por un algoritmo se necesita una medida adecuada. Claramente, si se aceptan las medidas de tiempo físico, obtendremos unas medidas que dependerán fuertemente del computador utilizado: si se ejecuta un mismo algoritmo con el mismo conjunto de datos de entrada en dos computadores distintos (por ejemplo en un PC-XT y en un PC-Pentium) el tiempo empleado en cada caso difiere notablemente. Esta solución llevaría a desarrollar una teoría para cada computador, lo que resulta poco práctico.

Por otro lado, se podría contar el número de instrucciones ejecutadas por el algoritmo, pero esta medida dependería de aspectos tales como la habilidad del programador o, lo que es más importante, del lenguaje de programación en el que se implemente, y tampoco se debe desarrollar una teoría para cada lenguaje de programación.

Se debe buscar entonces una medida abstracta del tiempo que sea independiente de la máquina con que se trabaje, del lenguaje de programación, del compilador o de cualquier otro elemento de *hardware* o *software* que influya en el análisis de la complejidad en tiempo. Una de las posibles medidas, que es la empleada en este libro y en gran parte de la literatura sobre complejidad, consiste en contar *el número de pasos* (por ejemplo, operaciones aritméticas, comparaciones y asignaciones) que se efectúan al ejecutarse un algoritmo.

### El coste depende de los datos

Considérese el problema de decidir si un número natural es par o impar. Es posible usar la función `Odd` predefinida en Pascal, que permite resolver el problema en tiempo constante.

Una segunda opción es emplear el algoritmo consistente en ir restando 2 repetidamente mientras que el resultado de la sustracción sea mayor que 1, y finalmente comprobar el valor del resto. Es fácil comprobar que se realizan  $n \text{ div } 2$  restas, lo que nos indica que, en este caso, el tiempo empleado depende del dato original  $n$ . Normalmente, el tiempo requerido por un algoritmo es función de los datos, por lo que se expresa como tal: así, escribimos  $T(n)$  para representar la complejidad en tiempo para un dato de tamaño  $n$ .

Este tamaño de entrada  $n$  depende fuertemente del tipo de problema que se va a estudiar. Así, por ejemplo, en el proceso de invertir el orden de los dígitos de un número natural

$$(4351, 0) \rightsquigarrow (435, 1) \rightsquigarrow (43, 15) \rightsquigarrow (4, 153) \rightsquigarrow (0, 1534)$$

no importa el número en cuestión: el dato relevante para ver cuánto tiempo se tarda es la longitud del número que se invierte. Otro ejemplo: para sumar las componentes de un vector de números reales, el tiempo empleado depende del número  $n$  de componentes del vector. En este caso, se puede decir ambas cosas sobre el coste:

- Que el coste de invertir un número, dígito a dígito, es lineal con respecto a su longitud (número de cifras)
- Que el coste es la parte entera de  $\log_{10}(n)+1$ , o sea, una función logarítmica, siendo  $n$  el dato.

Otra situación ejemplar se da en el caso del algoritmo de *suma lenta* (véase el apartado 1.2.1) de dos números naturales  $a$  y  $b$ :

```
while b > 0 do begin
  a:= a + 1;
  b:= b - 1
end {while}
```

Como se puede observar, el coste del algoritmo depende únicamente del segundo parámetro, ya que se ejecutan exactamente  $b$  iteraciones del cuerpo del bucle. Es, por tanto, un algoritmo de complejidad lineal con respecto a  $b$ , es decir,  $T(a, b) = b$ . Conclusión: no siempre todos los datos son importantes de cara a la complejidad.

En definitiva, a la hora de elegir el tamaño de los datos de entrada  $n$  de un algoritmo, conviene que represente la parte o la característica de los datos que influye en el coste del algoritmo.

### El coste esperado, el mejor y el peor

Otro aspecto interesante de la complejidad en tiempo puede ilustrarse analizando el algoritmo de búsqueda secuencial ordenada estudiado en el apartado 15.1.2, en el que se recorre un vector (ordenado crecientemente) desde su primer elemento, hasta encontrar el elemento buscado, o hasta que nos encontremos un elemento en el vector que es mayor que el elemento `elem` buscado. La implementación en Pascal de dicho algoritmo (ya mostrada en el citado apartado) es la siguiente:

```

const
  N = 100; {tamaño del vector}
type
  tIntervalo = 0..N;
  tVector = array[1..N] of integer;

function BusquedaSecOrd(v: tVector; elem: integer): tIntervalo;
  {PreC.: v está ordenado crecientemente, sin repeticiones}
  {Dev. 0 (si elem no está en v) ó i (si v[i] = elem)}
  var
    i: tIntervalo;
begin
  i:= 0;
  repeat
    {Inv.: v[j] ≠ elem ∀j, 1 ≤ j ≤ i}
    i:= i + 1
  until (v[i] >= elem) or (i = N);
  {v[i] = elem o v[j] ≠ elem ∀j, 1 ≤ j ≤ N}
  if v[i] = elem then {se ha encontrado el valor elem}
    BusquedaSecOrd:= i
  else
    BusquedaSecOrd:= 0
end; {BusquedaSecOrd}

```

Intuitivamente se puede ver que, si se tiene la buena fortuna de encontrar el elemento al primer intento, el tiempo es, digamos, de un paso (un intento). En el peor caso (cuando el elemento `elem` buscado es mayor o igual que todos los elementos del vector), se tendrá que recorrer todo el vector `v`, invirtiendo  $n$  pasos. Informalmente, se podría pensar que en un caso “normal”, se recorrería la “mitad” del vector ( $n/2$  pasos).

Como conclusión se puede afirmar que en algunos algoritmos, la complejidad no depende únicamente del tamaño del parámetro, sino que intervienen otros factores que hacen que la complejidad varíe de un caso a otro. Para distinguir esas situaciones se habla de coste *en el mejor caso*, *en el peor caso* y *en el caso medio*:

- $T_{\text{máx}}(\mathbf{n})$ , expresa la complejidad en *el peor caso*, esto es, el tiempo máximo que un algoritmo puede necesitar para una entrada de tamaño  $\mathbf{n}$ .
- $T_{\text{mín}}(\mathbf{n})$ , expresa la complejidad en *el mejor caso*, esto es, el tiempo mínimo que un algoritmo necesita para una entrada de tamaño  $\mathbf{n}$ .
- $T_{\text{med}}(\mathbf{n})$ , expresa la complejidad en *el caso medio*, esto es, el tiempo medio que un algoritmo necesita para una entrada de tamaño  $\mathbf{n}$ . Generalmente, se suele suponer que todas las secuencias de entradas son equiprobables. Por ejemplo, en los algoritmos de búsqueda, se considerará que `elem` puede estar en cualquier posición del vector con idéntica probabilidad, es decir, con probabilidad  $\frac{1}{n}$ .

Generalmente, la complejidad en el mejor caso es poco representativa y la complejidad en el caso medio es difícil de calcular por lo que, en la práctica, se suele trabajar con el tiempo para el peor caso por ser una medida significativa y de cálculo factible en general. No obstante, como ejemplo, se calculan a continuación las tres medidas para el algoritmo de búsqueda secuencial ordenada.

Para fijar ideas, vamos a hallar el coste en términos de los tiempos empleados para las operaciones de sumar ( $s$ ), realizar una comparación ( $c$ ) y realizar una asignación ( $a$ ) por un computador cualquiera. El coste en pasos es más sencillo, ya que basta con dar el valor unidad a cada una de esas operaciones.

Las tres medidas de coste mencionadas se calculan como sigue:

- $T_{\text{mín}}$ : Este tiempo mínimo se alcanzará cuando  $v[1] \geq \text{elem}$ . En tal caso se necesita una asignación para iniciar la variable  $i$ , una suma y una asignación para incrementar el valor de  $i$ , dos comparaciones en el bucle **repeat**, otro test más para  $v[i] = \text{elem}$  y, finalmente, una asignación a la función `BúsquedaSecOrd`. Por lo tanto:

$$T_{\text{mín}}(\mathbf{n}) = 3a + 3t + s$$

que es constante, lo que abreviamos así:

$$T_{\text{mín}}(\mathbf{n}) = k$$

- $T_{\text{máx}}$ : Este tiempo máximo se alcanzará cuando  $v[\mathbf{n}] \leq \text{elem}$ . Por lo tanto:

$$T_{\text{máx}}(\mathbf{n}) = a + n(s + 2t + a) + t + a = k_1n + k_2$$

- $T_{\text{med}}$ : Supóngase que es igualmente probable necesitar 1 vuelta, 2 vueltas, ...,  $n$  vueltas para encontrar el elemento buscado.<sup>1</sup> Además, recordemos que el tiempo empleado por el algoritmo cuando para en la posición  $j$ -ésima del vector es

$$T(j) = k_1j + k_2$$

según lo dicho en el apartado anterior. Entonces, se tiene que:<sup>2</sup>

$$\begin{aligned} T_{\text{med}}(n) &= \sum_{j=1}^n T_j P(\text{parar en la posición } j\text{-ésima}) \\ &= \sum_{j=1}^n (k_1j + k_2) \frac{1}{n} \\ &= \frac{k_1}{n} \frac{n+1}{2} + k_2 \\ &= c_1n + c_2 \end{aligned}$$

de forma que también es lineal con respecto a  $n$ .

Por supuesto,  $T_{\text{med}}(\mathbf{n}) < T_{\text{máx}}(\mathbf{n})$ , como se puede comprobar comparando los valores de  $k_1$  y  $c_1$ , lo que se deja como ejercicio al lector.

### También importa el gasto de memoria

Por otra parte, el estudio de la implementación de la función `Sumatorio` nos sirve para ver cómo el coste en memoria también debe tenerse en cuenta al analizar algoritmos.

Esta función, que calcula la suma de los  $n$  primeros números naturales, siendo  $n$  el argumento de entrada, puede ser implementada de forma natural con un algoritmo recursivo, resultando el siguiente código en Pascal:

```
function Sumatorio(n: integer): integer;
  {PreC.: n ≥ 0}
  {Dev.  ∑i=0n i}
```

---

<sup>1</sup>Se tiene esta situación, por ejemplo, cuando la secuencia ordenada de los  $n$  elementos del vector se ha escogido equiprobablemente entre el dominio `integer`, así como el elemento `elem` que se busca.

<sup>2</sup>Notaremos con  $P$  a la probabilidad.

```

begin
  if n = 0 then
    Sumatorio:= 0
  else
    Sumatorio:= n + Sumatorio(n-1)
end; {Sumatorio}

```

Al calcular el espacio de memoria que ocupa una llamada a esta función ha de tenerse en cuenta que, por su naturaleza recursiva, se generan nuevas llamadas a **Sumatorio** (exactamente **n** llamadas). En cada llamada se genera una tabla de activación (véase el apartado 10.2) del tamaño de un entero (el parámetro **n**), es decir, de tamaño constante. En consecuencia, podemos afirmar que la función **Sumatorio** tiene un coste proporcional a **n**.

Pero la suma de los **n** primeros enteros puede calcularse también de forma intuitiva empleando un algoritmo iterativo. Su sencilla implementación es la siguiente:

```

function SumatorioIter(n: integer): integer;
  {PreC.: n ≥ 0}
  {Dev.  $\sum_{i=0}^n i$ }
  var
    suma, i: integer;
begin
  suma:= 0;
  for i:= 0 to n do
    suma:= suma + i;
  SumatorioIter:= suma
end; {SumatorioIter}

```

En este caso, una llamada a **SumatorioIter**, al no generar otras llamadas sucesivas, consume un espacio de memoria constante: exactamente el necesario para su parámetro y para las dos variables locales, todos de tipo **integer**. Piense el lector en la diferencia de espacio requerida por ambas funciones para **n = 1000**, por ejemplo.

Con estos ejemplos podemos concluir que, a la hora del análisis de algoritmos, es fundamental realizar un estudio de la eficiencia, destacando como aspecto más importante la complejidad en tiempo, y en segundo lugar, la complejidad en espacio.

### Lo importante es el comportamiento asintótico

Es un hecho evidente que datos de un tamaño reducido van a tener asociados, en general, tiempos cortos de ejecución. Por eso, es necesario estudiar el com-

portamiento de éstos con datos de un tamaño considerable, que es cuando los costes de los distintos algoritmos pueden tener una diferenciación significativa.

Para entender mejor la importancia del orden de complejidad, resulta muy ilustrativo observar cómo aumenta el tiempo de ejecución de algoritmos con distintos órdenes. En todos ellos,  $n$  representa el tamaño de los datos y los tiempos están expresados en segundos, considerando un computador que realiza un millón de operaciones por segundo:

$T(n)$ $n$	$\log n$	$n$	$n \log n$	$n^2$	$n^3$	$2^n$	$n!$
10	$3.32 \cdot 10^{-6}$	$10^{-5}$	$3.32 \cdot 10^{-5}$	$10^{-4}$	0.001	0.001024	3.6288
50	$5.64 \cdot 10^{-6}$	$5 \cdot 10^{-5}$	$2.82 \cdot 10^{-4}$	0.0025	0.125	intratable	intratable
100	$6.64 \cdot 10^{-6}$	$10^{-4}$	$6.64 \cdot 10^{-4}$	0.01	1	intratable	intratable
$10^3$	$10^{-5}$	0.001	0.01	1	1000	intratable	intratable
$10^4$	$1.33 \cdot 10^{-5}$	0.01	0.133	100	$10^6$	intratable	intratable
$10^5$	$1.66 \cdot 10^{-5}$	0.1	1.66	$10^4$	intratable	intratable	intratable
$10^6$	$2 \cdot 10^{-5}$	1	19.93	$10^6$	intratable	intratable	intratable

## 18.2 Medidas del comportamiento asintótico

### 18.2.1 Comportamiento asintótico

Como se ha visto en el apartado anterior, la complejidad en tiempo de un algoritmo es una función  $T(n)$  del tamaño de entrada del algoritmo. Pues bien, es el orden de dicha función (constante, logarítmica, lineal, exponencial, etc.) el que expresa el comportamiento dominante para datos de gran tamaño, como se ilustra en el ejemplo que se presenta a continuación.

Supóngase que se dispone de cuatro algoritmos para solucionar un determinado problema, cuyas complejidades son respectivamente, lineal ( $8n$ ), cuadrática ( $2n^2$ ), logarítmica ( $20 \log_2 n$ ) y exponencial ( $e^n$ ). En la figura 18.1 se puede observar cómo sus tiempos relativos de ejecución no son excesivamente diferentes para datos de un tamaño pequeño (entre 1 y 5).

Sin embargo, las gráficas de la figura 18.2 confirman que es realmente el orden de la función de complejidad el que determina el comportamiento para tamaños de entrada grandes, reteniendo únicamente la parte relevante de una función (de coste) para datos de gran tamaño.

A la vista de esto, es evidente que el aspecto importante de la complejidad de algoritmos es su comportamiento asintótico, ignorando otros detalles menores por ser irrelevantes.

Para ello, es preciso formalizar el estudio del orden de complejidad mediante medidas que ayuden a determinar el comportamiento asintótico del coste.

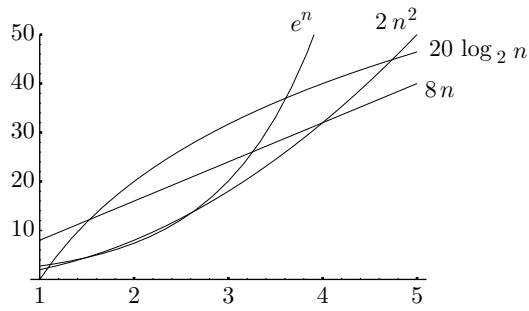


Figura 18.1.

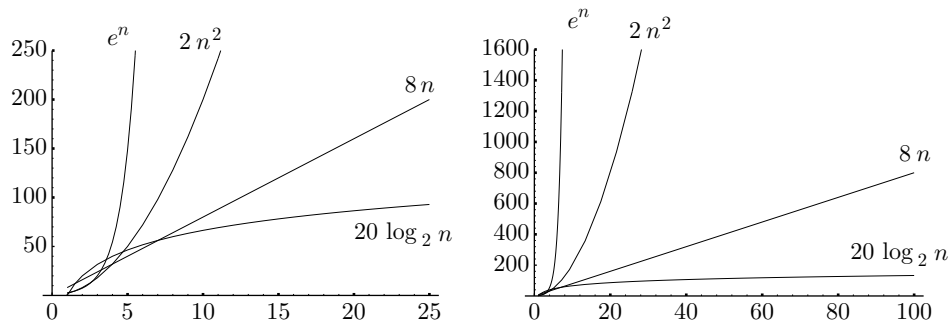


Figura 18.2.

Entre estas medidas destaca la notación  $O$  mayúscula,<sup>3</sup> la notación  $\Omega$  y la notación  $\Theta$ .

### 18.2.2 Notación $O$ mayúscula (una cota superior)

**Definición:** Sean  $f, g : \mathbb{Z}^+ \rightarrow \mathbb{R}^+$ . Se dice que  $f \in O(g)$  o que  $f$  es del orden de  $g$  si existen constantes  $n_0 \in \mathbb{Z}^+$  y  $\lambda \in \mathbb{R}^+$  tales que

$$f(n) \leq \lambda g(n) \quad \text{para todo } n \geq n_0$$

Con la notación<sup>4</sup>  $f \in O(g)$  se expresa que la función  $f$  no crece más deprisa que alguna función proporcional a  $g$ . Esto es, se acota superiormente el comportamiento asintótico de una función salvo constantes de proporcionalidad. Veamos algunos ejemplos:

- Para el algoritmo de búsqueda secuencial ordenada,

$$T_{\text{máx}}(n) = k_1 n + k_2 \in O(n)$$

(lo que se ve tomando cualquier  $\lambda > k_1$  y  $n_0 > \frac{k_2}{\lambda - k_1}$ ).

Para este mismo algoritmo se tiene además que  $T_{\text{med}}(n) = c_1 n + c_2 \in O(n)$ , y para el sumatorio recursivo se cumple que  $S(n) = n + 1 \in O(n)$ .

- Todas las funciones de tiempo constante son  $O(1)$ :

$$f(n) = k \in O(1)$$

lo que se ve tomando  $\lambda = k$  y cualquier  $n_0 \in \mathbb{Z}^+$ . En este caso están  $T_{\text{mín}}(n) = k$  para la búsqueda secuencial ordenada, y  $S(n) = 3$  para el sumatorio iterativo.

- $15n^2 \in O(n^2)$

Como esta notación expresa una cota superior, siempre es posible apuntar alto a la hora de establecerla. Por ejemplo, se puede decir que los algoritmos estudiados hasta ahora son  $O(n!)$ . Naturalmente, esta imprecisión es perfectamente inútil, por lo que se debe procurar que la función  $g$  sea lo más “próxima” posible a  $f$ ; es decir, se debe buscar una cota superior lo menor posible. Así, por ejemplo, aunque  $(5n + 3) \in O(n^2)$ , es más preciso decir que  $(5n + 3) \in O(n)$ .

Para formalizar esta necesidad de precisión, se usan otras medidas del comportamiento asintótico.

<sup>3</sup>Debido a que la expresión inglesa que se utiliza para esta notación es *Big-Oh*, también se conoce como notación  $O$  grande.

<sup>4</sup>En lo sucesivo, emplearemos las dos notaciones  $f \in O(g)$  y  $f(n) \in O(g(n))$  indistintamente, según convenga.

### 18.2.3 Notación $\Omega$ mayúscula (una cota inferior)

**Definición:** Sean  $f, g : \mathbb{Z}^+ \rightarrow \mathbb{R}^+$ . Se dice que  $f \in \Omega(g)$  si existen constantes  $n_0 \in \mathbb{Z}^+$  y  $\lambda \in \mathbb{R}^+$  tales que

$$f(n) \geq \lambda g(n) \quad \text{para todo } n \geq n_0$$

Con la notación  $f \in \Omega(g)$  se expresa que la función  $f$  crece más deprisa que alguna función proporcional a  $g$ . Esto es, se acota inferiormente el comportamiento asintótico de una función, salvo constantes de proporcionalidad. Dicho de otro modo, con la notación  $f \in \Omega(g)$  se indica que la función  $f$  necesita para su ejecución un tiempo mínimo dado por el orden de la función  $g$ .

En los ejemplos anteriores se puede comprobar que:

- Para el algoritmo de búsqueda secuencial ordenada,  $T_{\text{máx}}(n) \in \Omega(n)$ .
- $3n + 1 \in \Omega(n)$ .
- $3n + 1 \in \Omega(1)$ .
- $15n^2 \in \Omega(n^2)$ .
- $15n^2 \in \Omega(n)$ .

Comparando las definiciones anteriores, se tiene que

$$f \in O(g) \Leftrightarrow g \in \Omega(f)$$

### 18.2.4 Notación $\Theta$ mayúscula (orden de una función)

**Definición:** Sean  $f, g : \mathbb{Z}^+ \rightarrow \mathbb{R}^+$ . Se dice que  $f \in \Theta(g)$  si  $f \in O(g)$  y  $g \in O(f)$ ; esto es, si  $f \in O(g) \cap \Omega(g)$ . Al conjunto  $\Theta(g)$  se le conoce como el *orden exacto de  $g$* .

Con la notación  $\Theta$  se expresa que las funciones  $f$  y  $g$  tienen el mismo “grado” de crecimiento, es decir, que  $0 < \lim_{x \rightarrow \infty} \frac{f(x)}{g(x)} < \infty$ . Ejemplos:

- Para el algoritmo de búsqueda secuencial ordenada,  $T_{\text{máx}}(n) \in \Theta(n)$ .
- $3n + 1 \in \Theta(n)$ .
- $15n^2 \in \Theta(n^2)$ .

### 18.2.5 Propiedades de $O$ , $\Omega$ y $\Theta$

Entre las propiedades más importantes de las notaciones  $O$  mayúscula,  $\Omega$  y  $\Theta$  cabe destacar las que se describen a continuación. En ellas se utiliza el símbolo  $\Delta$  si la propiedad es cierta para las tres notaciones, y se asume que

$$f, g, f_1, f_2 : \mathbb{Z}^+ \longrightarrow \mathbb{R}^+$$

**Reflexividad:**  $f \in \Delta(f)$ .

**Escalabilidad:** Si  $f \in \Delta(g)$  entonces  $f \in \Delta(k \cdot g)$  para todo  $k \in \mathbb{R}^+$ .

Una consecuencia de ello es que, si  $a, b > 1$  se tiene que  $O(\log_a n) = O(\log_b n)$ . Por ello, no hace falta indicar la base:  $O(\log n)$ .

**Transitividad:** Si  $f \in \Delta(g)$  y  $g \in \Delta(h)$  se tiene que  $f \in \Delta(h)$ .

**Simetría:** Si  $f \in \Theta(g)$  entonces  $g \in \Theta(f)$

(Obsérvese que las otras notaciones no son simétricas y trátense de dar un contraejemplo.)

**Regla de la suma:** Si  $f_1 \in O(g_1)$  y  $f_2 \in O(g_2)$  entonces  $f_1 + f_2 \in O(\max(g_1, g_2))$  siendo  $\max(g_1, g_2)(n) = \max(g_1(n), g_2(n))$ .

Junto con la escalabilidad, esta regla se generaliza fácilmente así: si  $f_i \in O(g_i)$  para todo  $i = 1, \dots, k$ , entonces  $c_1 f_1 + \dots + c_k f_k \in O(f)$ .

Otra consecuencia útil es que si  $p_k(n)$  es un polinomio de grado  $k$ , entonces  $p_k(n) \in O(n^k)$ .

**Regla del producto:** Si  $f_1 \in \Delta(g_1)$  y  $f_2 \in \Delta(g_2)$  entonces  $f_1 \cdot f_2 \in O(g_1 \cdot g_2)$ .

Una consecuencia de ello es que, si  $p < q$ , entonces  $O(n^p) \subset O(n^q)$ .

**Regla del sumatorio:** Si  $f \in O(g)$  y la función  $g$  es creciente,

$$\sum_{i=1}^n f(i) \in O\left(\int_1^{n+1} g(x) dx\right)$$

Una consecuencia útil es la siguiente:  $\sum_{i=1}^n i^k \in O(n^{k+1})$ .

Estas propiedades no deben interpretarse como meras fórmulas desprovistas de significado. Muy al contrario, expresan la idea de fondo de las medidas asintóticas, que consiste en ver la parte relevante de una función de coste pensando en datos de gran tamaño. Gracias a ello, podemos simplificar considerablemente las funciones de coste sin peligro de pérdida de información (para datos grandes, se entiende). Por ejemplo:

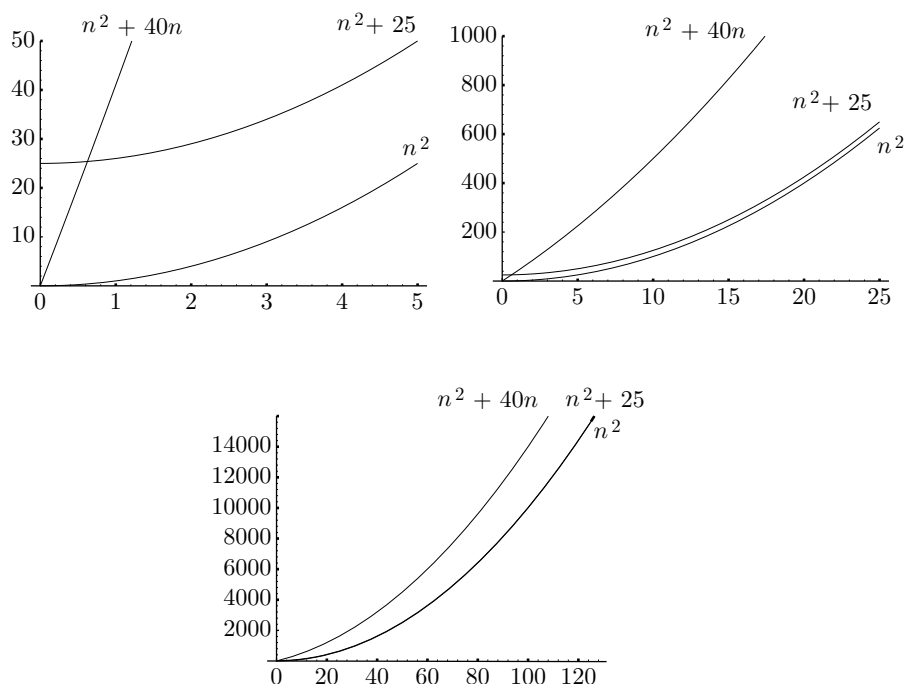


Figura 18.3.

$$c_1n^2 + c_2n + c_3 \sim c_1n^2 \sim n^2$$

Efectivamente, para datos grandes las funciones con el mismo orden de complejidad se comportan esencialmente igual. Además, en la práctica es posible omitir los coeficientes de proporcionalidad: de hecho, no afectan a las medidas estudiadas. Las dos simplificaciones se justifican en las gráficas de la figura 18.3.

### 18.2.6 Jerarquía de órdenes de frecuente aparición

Existen algoritmos de complejidad *lineal* con respecto a los datos de entrada ( $T(n) = c_1n + c_2$ ). También existen algoritmos de complejidad *constante* ( $T(n) = c$ ), independientemente de los datos de entrada. El método de intercambio directo para ordenar arrays tiene un coste *cuadrático*. Otras funciones de coste son *polinómicas* de diversos grados, *exponenciales*, *logarítmicas*, etc.

Estos diferentes comportamientos asintóticos se pueden ordenar de menor a mayor crecimiento. Aunque no pretendemos dar una lista exhaustiva, la siguiente cadena de desigualdades puede orientar sobre algunos de los órdenes de coste más usuales:<sup>5</sup>

$$1 \ll \log(n) \ll n \ll n \log(n) \ll n^2 \ll n^3 \ll \dots \ll 2^n \ll n!$$

A pesar de las apariencias, la relación “ser del orden de” no es una relación de orden total: existen pares de funciones tales que ninguna de las dos es una cota superior de la otra. Por ejemplo, las funciones

$$f(n) = \begin{cases} 1 & \text{si } n \text{ es par} \\ n & \text{si } n \text{ es impar} \end{cases} \quad g(n) = \begin{cases} n & \text{si } n \text{ es par} \\ 1 & \text{si } n \text{ es impar} \end{cases}$$

no verifican  $f \in O(g)$  ni  $g \in O(f)$ .

## 18.3 Reglas prácticas para hallar el coste de un programa

### 18.3.1 Tiempo empleado

En este apartado se dan reglas generales para el cálculo de la complejidad en tiempo en el peor caso de los programas escritos en Pascal. Para dicho cálculo, como es natural, se debe tener en cuenta el coste en tiempo de cada una de las diferentes instrucciones de Pascal, y esto es lo que se detalla a continuación.

#### Instrucciones simples

Se considera que se ejecutan en tiempo constante:

- La evaluación de las expresiones aritméticas (suma, resta, producto y división) siempre que los datos sean de tamaño constante, así como las comparaciones de datos simples.
- Las instrucciones de asignación, lectura y escritura de datos simples.
- Las operaciones de acceso a una componente de un array, a un campo de un registro y a la siguiente posición de un archivo.

Todas esas operaciones e instrucciones son  $\Theta(1)$ .

---

<sup>5</sup>La notación  $\ll$  representa la relación *de orden menor que*.

### Composición de instrucciones

Suponiendo que las instrucciones  $I_1$  e  $I_2$  tienen como complejidades en el peor caso  $T_{I_1}(n)$  y  $T_{I_2}(n)$ , respectivamente, entonces el coste de la composición de instrucciones  $(I_1; I_2)$  en el peor caso es

$$T_{I_1;I_2}(n) = T_{I_1}(n) + T_{I_2}(n)$$

que, aplicando la regla de la suma, es el máximo entre los costes  $T_{I_1}(n)$  y  $T_{I_2}(n)$ .

### Instrucciones de selección

En la instrucción condicional,

**if** *condición* **then**  $I_1$  **else**  $I_2$

siempre se evalúa la condición, por lo que su coste debe agregarse al de la instrucción que se ejecute. Puesto que se está estudiando el coste en el peor caso, se tendrá en cuenta la más costosa. Con todo esto, la complejidad de la instrucción **if-then-else** es:

$$T_{condición}(n) + \max(T_{I_1}(n), T_{I_2}(n))$$

Análogamente, la instrucción de selección por casos

**case** *expresión* **of**  
*caso1*:  $I_1$ ;  
 ...  
*casoL*:  $I_L$   
**end**; {**case**}

requiere evaluar la expresión y una instrucción, en el peor caso la más costosa:

$$T_{expresión}(n) + \max(T_{I_1}(n), \dots, T_{I_L}(n))$$

### Bucles

El caso más sencillo es el de un bucle **for**:

**for**  $j := 1$  **to**  $m$  **do**  $I$

En el supuesto de que en  $I$  no se altera el índice  $j$ , esta instrucción tiene el siguiente coste:

$$m + \sum_{j=1}^m T_{I_j}(n)$$

donde la cantidad  $m$  representa las  $m$  veces que se incrementa  $j$  y la comprobación de si está entre los extremos inferior y superior.

En el caso de que el cuerpo del bucle consuma un tiempo fijo (independientemente del valor de  $j$ ), la complejidad del bucle resulta ser  $m(1 + T_I(n))$ .

En los bucles **while** y **repeat** no hay una regla general, ya que no siempre se conoce el número de vueltas que da, y el coste de cada iteración no siempre es uniforme. Sin embargo, con frecuencia se puede acotar superiormente, acotando precisamente el número de vueltas y el coste de las mismas.

### Subprogramas

El coste de ejecutar un subprograma no recursivo se deduce con las reglas descritas. Por el contrario, en caso de haber recursión, hay que detenerse a distinguir entre los casos básicos (los parámetros que no provocan nuevas llamadas recursivas) y los recurrentes (los que sí las producen). Considérese como ejemplo la versión recursiva de la función factorial (tomada del apartado 10.1):

```
function Fac (n: integer): integer;
  {PreC.: n ≥ 0}
  {Dev. n!}
begin
  if n = 0 then
    Fac := 1
  else
    Fac := n * Fac(n-1)
end; {Fac}
```

Para calcular la complejidad  $T_{\text{Fac}}(n)$  del algoritmo se debe tener en cuenta que, para el caso básico ( $n = 0$ ), el coste es constante,  $\Omega(1)$ ,

$$T_{\text{Fac}}(0) = 1$$

y para los recurrentes ( $n > 0$ ), el coste es de una cantidad constante,  $\Omega(1)$  más el de la llamada subsidiaria provocada:

$$T_{\text{Fac}}(n) = 1 + T_{\text{Fac}}(n - 1)$$

En resumen,

$$T_{\text{Fac}}(n) = \begin{cases} 1 & \text{si } n = 0 \\ 1 + T_{\text{Fac}}(n - 1) & \text{si } n > 0 \end{cases}$$

Así, para  $n > 1$ ,

$$\begin{aligned}
 T_{\text{Fac}}(n) &= 1 + T_{\text{Fac}}(n - 1) \\
 &= 1 + 1 + T_{\text{Fac}}(n - 2) \\
 &= \dots \\
 &= n + T_{\text{Fac}}(0) \\
 &= n + 1
 \end{aligned}$$

y, por lo tanto, el coste es lineal, o sea,  $T_{\text{Fac}}(n) \in \Omega(n)$ .

### 18.3.2 Ejemplos

Una vez descrito cómo calcular la complejidad en tiempo de los programas en Pascal, se presentan algunos algoritmos a modo de ejemplo.

#### Producto de dos números enteros

Supóngase que se pretende calcular el producto de dos números naturales  $n$  y  $m$  sin utilizar la operación de multiplicación. Un primer nivel de diseño de un algoritmo para este problema es:

```

prod:= 0
repetir n veces
  repetir m veces
    prod:= prod + 1

```

Este diseño se implementa directamente en Pascal mediante la siguiente función:

```

function Producto(n, m: integer): integer;
  {PreC.: n,m ≥ 0}
  {Dev. n×m}
  var
    i,j,prod: integer;
begin
  prod:= 0;
  for i:= 1 to n do
    for j:= 1 to m do
      prod:= prod + 1;
  Producto:= prod
end; {Producto}

```

Como el primer bucle **for** se repite  $n$  veces y el segundo  $m$  veces, y puesto que el resto de las instrucciones son asignaciones (con tiempo de ejecución constante), se deduce que el algoritmo tiene una complejidad en tiempo  $T(n, m) \in O(nm)$ .

Para mejorar el algoritmo, se podría utilizar un único bucle, atendiendo al siguiente diseño:

```
prod:= 0
repetir n veces
  prod:= prod + m
```

Este diseño también se implementa fácilmente en Pascal mediante una función:

```
function Producto2(n, m: integer): integer;
  {PreC.: n,m ≥ 0}
  {Dev. n×m}
  var
    i,prod: integer;
begin
  prod:= 0;
  for i:= 1 to n do
    prod:= prod + m;
  Producto2:= prod
end; {Producto2}
```

Se obtiene así un código cuya complejidad es, claramente,  $O(n)$ .

Este algoritmo se puede mejorar ligeramente si se controla que el bucle se repita  $n$  veces si  $n \leq m$ , o  $m$  veces si  $m \leq n$ . La implementación de esta mejora se deja como ejercicio al lector.

Es posible conseguir una complejidad aún menor utilizando el algoritmo conocido con el nombre de *multiplicación a la rusa*.<sup>6</sup> El método consiste en multiplicar uno de los términos por dos mientras que el otro se divide por dos (división entera) hasta llegar a obtener un uno. El resultado se obtendrá sumando aquellos valores que se han multiplicado por dos tales que su correspondiente término dividido por dos sea un número impar. Por ejemplo, al realizar la *multiplicación a la rusa* de  $25 \times 11$ , se obtiene la siguiente tabla de ejecución:

25	11	*
50	5	*
100	2	
200	1	*

---

<sup>6</sup>Curiosamente, muchos autores dicen que este algoritmo es el que utilizaban los romanos para multiplicar.

obteniendo como resultado final  $25 + 50 + 200 = 275$  debido a que sus términos divididos correspondientes (11, 5 y 1) son impares.

- ☉ La justificación de por qué sumar sólo estos valores es la siguiente: si se tiene un producto de la forma  $k \times (2l)$ , un paso del algoritmo lo transforma en otro producto igual  $(2k) \times l$ ; sin embargo, si el producto es de la forma  $k \times (2l + 1)$ , un paso del algoritmo lo transforma en  $(2k) \times l$  (debido a que  $(2l + 1) \text{ div } 2 = l$ ), mientras que el verdadero resultado del producto es  $k \times (2l + 1) = (2k) \times l + k$ . Por lo tanto, se tendrá que sumar la cantidad perdida  $k$  y esto es lo que se hace siempre que el número que se divide es impar.

Dada la sencillez del algoritmo, se presenta directamente su implementación en Pascal:

```
function ProductoRuso(n,m: integer):integer;
  {PreC.: n,m ≥ 0}
  {Dev. n×m}
  var
    prod: integer;
begin
  prod:= 0;
  while m > 0 do begin
    if Odd(m) then
      prod:= prod + n;
    n:= n + n; {n:= n * 2, sin usar *}
    m:= m div 2
  end; {while}
  ProductoRuso:= prod
end; {ProductoRuso}
```

Como la variable  $m$  se divide por dos en cada vuelta del bucle, hasta llegar a 0, el algoritmo es de complejidad  $O(\log m)$ .

## Ordenación de vectores

En el capítulo 15 se presentaron varios algoritmos para la ordenación de vectores, viendo cómo, intuitivamente, unos mejoraban a otros en su eficiencia en tiempo. En este apartado se estudia con detalle la complejidad de algunos de estos algoritmos<sup>7</sup> utilizando las técnicas expuestas en los apartados anteriores.

<sup>7</sup>Dado que no es necesario tener en cuenta todos los detalles de un algoritmo para su análisis, se ha optado en este apartado por estudiar la complejidad sobre el pseudocódigo, llegando hasta el nivel menos refinado que permita su análisis. Los autores consideran que descender a niveles inferiores no es necesario, ya que los fragmentos que tardan un tiempo constante (o acotado por una constante) no importa qué detalles contengan.

Para empezar, se presenta el algoritmo de ordenación por intercambio directo (véase el apartado 15.2.3), que es fácil pero ineficiente, como se demostrará al estudiar su complejidad y compararla con la de los restantes. Como se explicó en dicho apartado, este algoritmo consiste en recorrer el array con dos bucles anidados dependientes. El primero recorre todos los elementos del vector, mientras que el segundo bucle va intercambiando los valores que están en orden decreciente. El boceto del algoritmo es el siguiente:

```

para i entre 1 y n-1 hacer
  Desplazar el menor valor desde vn hasta vi, intercambiando
  pares vecinos, si es necesario
Devolver v ya ordenado

```

Como siempre, para determinar la complejidad es necesario contar el número de veces que se ejecuta el cuerpo de los bucles, ya que las operaciones que intervienen en el algoritmo (asignaciones, comparaciones y acceso a elementos de un vector) se ejecutan en tiempo constante.

El cuerpo del bucle *Desplazar el menor valor... si es necesario* requiere  $n-i$  pasos en la vuelta  $i$ -ésima (uno para cada posible intercambio). Por lo tanto, el coste del algoritmo completo es<sup>8</sup>

$$\sum_{i=1}^n (n-i) = \frac{n(n-1)}{2}$$

y, en consecuencia, su complejidad es cuadrática ( $O(n^2)$ ).

Analicemos ahora el algoritmo *Quick Sort* (véase el apartado 15.2.4) en el peor caso. A grandes trazos, el algoritmo es el siguiente:

```

si v es de tamaño 1 entonces
  v ya está ordenado
si no
  Dividir v en dos bloques A y B
  con todos los elementos de A menores que los de B
fin {si}
Ordenar A y B usando Quick Sort
Devolver v ya ordenado como concatenación
de las ordenaciones de A y de B

```

donde *Dividir v en dos bloques A y B* consiste en

---

<sup>8</sup>La suma que hay que calcular se corresponde con la suma de los términos de una progresión aritmética (véase el apartado 18.4).

*Elegir un elemento p (pivote) de v*  
*para cada elemento del vector hacer*  
     *si el elemento < p entonces*  
         *Colocar el elemento en A, el subvector con los elementos de v*  
         *menores que p*  
     *en otro caso*  
         *Colocar el elemento en B, el subvector con los elementos de v*  
         *mayores que p*

Como se dijo en 15.2.4, se ha optado por elegir como pivote el primer elemento del vector.

En el peor caso (cuando el vector se encuentra ordenado decrecientemente), el algoritmo *Quick Sort* va a tener complejidad  $O(n^2)$ . La razón es que, en tal caso, el cuerpo del bucle *para cada elemento...* se ejecutará, en total,  $(n - 1) + (n - 2) + \dots + 1$  veces, donde cada sumando proviene de cada una de las sucesivas ordenaciones recursivas del subvector A. Esto es así porque en cada llamada se ordena un solo elemento (el pivote), y por tanto dicho subvector tendrá sucesivamente longitud  $(n - 1), (n - 2), \dots, 1$ . Dicha suma, como se vio anteriormente es

$$\frac{n(n - 1)}{2}$$

y por tanto el algoritmo tiene complejidad cuadrática. En resumen, la complejidad en el peor caso es la misma en el algoritmo anterior.

Ciertamente, en el capítulo 15 se presentó este último método como mejora en el tiempo de ejecución. Lo que ocurre es que esa mejora es la que se logra en el caso medio. Sin embargo, el correspondiente cálculo rebasa las pretensiones de este libro.

Para completar este apartado se presenta el análisis de la complejidad en tiempo de un tercer algoritmo de ordenación de vectores, concretamente el de ordenación por mezcla o *Merge Sort* (véase el apartado 15.2.5). El algoritmo es el que sigue:

*si v es de tamaño 1 entonces*  
     *v ya está ordenado*  
*si no*  
     *Dividir v en dos subvectores A y B*  
*fin {si}*  
     *Ordenar A y B usando Merge Sort*  
     *Mezclar las ordenaciones de A y B para generar el vector ordenado.*

En este caso, el paso *Dividir v en dos subvectores A y B* consiste en:

Asignar a A el subvector  $[v_1, \dots, v_{n \text{ div } 2}]$   
 Asignar a B el subvector  $[v_{n \text{ div } 2 + 1}, \dots, v_n]$

mientras que *Mezclar las ordenaciones de A y B* consiste en ir entremezclando adecuadamente las componentes ya ordenadas de A y de B para obtener el resultado buscado.

El análisis de la complejidad de este algoritmo no es complicado. Partiendo de que la operación de *Mezclar las ordenaciones de A y B* se ejecuta en un tiempo proporcional a  $n$  (la longitud del vector por ordenar), el coste en tiempo del algoritmo completo viene dado por la siguiente relación de recurrencia, donde  $k_i$  son cantidades constantes:

$$T(n) = \begin{cases} k_1 & \text{si } n = 1 \\ 2T(\frac{n}{2}) + k_2n + k_3 & \text{si } n > 1 \end{cases}$$

En esta fórmula  $k_1$  representa el coste del caso trivial ( $v$  de tamaño 1);  $T(n/2)$  es el coste de cada llamada recursiva, y  $k_2n + k_3$  es el coste de mezclar los subvectores A y B, ya ordenados.

Esta ecuación se resuelve mediante sustituciones sucesivas cuando  $n$  es una potencia de 2 (es decir, existe  $j$ , tal que  $n = 2^j$ ), de la siguiente forma:

$$\begin{aligned} T(n) &= 2(2T(\frac{n}{4}) + k_2\frac{n}{2} + k_3) + k_2n + k_3 \\ &= 4T(\frac{n}{4}) + 2k_2n + k_4 \\ &= 4(2T(\frac{n}{8}) + k_2\frac{n}{4} + k_3) + 2k_2n + k_4 \\ &= \dots \\ &= 2^jT(1) + jk_2n + k_j \\ &= nk_1 + k_2n \log_2 n + k_j \end{aligned}$$

Y en el caso en que  $n$  no sea una potencia de 2, siempre se verifica que existe  $k > 0$  tal que  $2^k < n < 2^{k+1}$ , y, por tanto, se tiene que  $T(n) \leq T(2^{k+1})$ . En consecuencia, se puede afirmar que, en todo caso,  $T(n) \in O(n \log_2 n)$ .

Esta conclusión indica que *Merge Sort* es un algoritmo de ordenación con una complejidad en tiempo óptima<sup>9</sup> en el peor caso, aunque no es tan bueno en cuanto a la complejidad en espacio, ya que es necesario mantener dos copias del vector. Existen versiones mejoradas de este algoritmo que tienen menor coste en espacio, pero su estudio excede a las pretensiones de este texto.

<sup>9</sup>Hay que recordar que esta complejidad es óptima bajo la notación O-grande, esto es, salvo constantes de proporcionalidad.

### 18.3.3 Espacio de memoria empleado

Aunque el cálculo de la complejidad en espacio es similar al de la complejidad en tiempo, se rige por leyes distintas, como se comenta en este apartado.

En primer lugar, se debe tener en cuenta que la traducción del código fuente a código máquina depende del compilador y del computador, y que esto tiene una fuerte repercusión en la memoria. En consecuencia, y al igual que se razonó para la complejidad en tiempo, no es recomendable utilizar medidas absolutas sino relativas, como celdas de memoria (el espacio para almacenar, por ejemplo, un dato simple: un número o un carácter).

Si llamamos  $S(\mathbf{n})$  al espacio relativo de memoria que el algoritmo ha utilizado al procesar una entrada de tamaño  $\mathbf{n}$ , se definen los conceptos de  $S_{\text{máx}}(\mathbf{n})$ ,  $S_{\text{mín}}(\mathbf{n})$  y  $S_{\text{med}}(\mathbf{n})$  para la complejidad en espacio del *peor caso*, *el mejor caso* y *caso medio*, de forma análoga a los conceptos respectivos de tiempo.

Con estos conceptos, y considerando, por ejemplo, que cada entero necesita una celda de memoria, el espacio necesario para la búsqueda secuencial ordenada es:  $\mathbf{n}$  celdas para el vector, una celda para el elemento buscado y una celda para la variable  $i$ , es decir,

$$S(\mathbf{n}) = \mathbf{n} + 2$$

tanto en el peor caso como en mejor caso y en el caso medio.

De forma análoga se ve que el algoritmo de búsqueda binaria tiene como complejidad en espacio  $S(\mathbf{n}) = \mathbf{n} + 4$ .

Utilizando esta notación, podemos afirmar que la función `Sumatorio` del apartado 18.1 tiene, en su versión iterativa, una complejidad en espacio  $S(\mathbf{n}) = 3$ , debida al espacio ocupado por el parámetro  $\mathbf{n}$  y las variables locales  $i$  y `suma`. La complejidad de la versión recursiva es  $S(\mathbf{n}) = \mathbf{n} + 1$ , puesto que cada una de las tablas de activación ocupa una celda para su parámetro local  $\mathbf{n}$ .

Para el cálculo de la complejidad en espacio es preciso tener en cuenta algunos aspectos relacionados con el manejo de subprogramas:

- La llamada a un subprograma tiene asociado un coste en espacio, dado que se tiene que generar la tabla de activación (véase el apartado 10.2). Más concretamente:
  - Los parámetros por valor necesitan un espacio igual a su tamaño, al igual que los objetos (constantes y variables) locales.
  - Los parámetros por variable sólo necesitan una cantidad de espacio unitaria independientemente de su tamaño.
- Los algoritmos recursivos necesitan una cantidad de espacio dependiente de la “profundidad” de la recursión que determina el tamaño de la pila de tablas de activación (véase el apartado 17.2.3).

Por consiguiente, cuando un subprograma recursivo origine varias llamadas, sólo importará la llamada que provoque una mayor profundidad, pudiéndose despreciar las demás.

Es un error frecuente comparar la complejidad en espacio de los algoritmos recursivos con el número total de llamadas.

### Ejemplo: sucesión de Fibonacci

La sucesión de los números de Fibonacci (véase el apartado 10.3.1) se puede hallar mediante la siguiente función recursiva:

```

function Fib(num: integer): integer;
  {PreC.: num ≥ 0}
  {Dev. fibnum}
begin
  if (num = 0) or (num = 1) then
    Fib:= 1
  else
    Fib:= Fib(num-1) + Fib(num-2)
  end; {Fib}

```

El coste en espacio,  $S(n)$ , del algoritmo descrito es proporcional a la profundidad del árbol de llamadas; es decir,  $S(n) = 1$  en los casos triviales  $n = 0$  y  $n = 1$ ; en los no triviales ( $n \geq 2$ ), **Fib**(**n**) origina dos llamadas subsidiarias, **Fib**(**n-1**) y **Fib**(**n-2**), la primera de las cuales es más profunda. Por lo tanto, en estos casos,

$$S(n) = 1 + \text{máx}(S(n-1), S(n-2)) = 1 + S(n-1)$$

En resumidas cuentas,  $S(n) = n$ , lo que indica que esta función tiene un requerimiento de espacio lineal con respecto a su argumento  $n$ .

## 18.4 Útiles matemáticos

Ya se ha visto en los ejemplos anteriores que, cuando se trabaja con funciones o procedimientos recursivos, la complejidad en el tiempo  $T(n)$  va a venir dada en función del valor de  $T$  en puntos menores que  $n$ . Por ello es útil saber cómo calcular términos generales de sucesiones en las que los términos se definen en función de los valores anteriores (sucesiones recurrentes). En este apéndice se tratan los casos más comunes que pueden surgir a la hora del cálculo de la complejidad en el tiempo de funciones o procedimientos recursivos.

### 18.4.1 Fórmulas con sumatorios

- Si  $x_n$  es una sucesión aritmética, esto es,  $x_n = x_{n-1} + r$ , entonces

$$x_1 + x_2 + \dots + x_n = \frac{(x_1 + x_n)n}{2}$$

- $1 + x + x^2 + \dots + x^{n-1} = \frac{1 - x^n}{1 - x}$ .
- $\sum_{i=0}^{\infty} x^i = 1 + x + x^2 + \dots = \frac{1}{1 - x}$ , siempre que  $|x| < 1$ .
- $\sum_{i=0}^{\infty} \frac{x^i}{i!} = e^x$ .
- $\sum_{i=0}^{\infty} (-1)^i \frac{x^i}{i} = \log(x)$ .
- Si cada  $a_i \in \Delta(n^k)$ , se tiene que  $\sum_{i=1}^n a_i \in \Delta(n^{k+1})$ .

### 18.4.2 Sucesiones de recurrencia lineales de primer orden

Son aquellas sucesiones en las que su término general viene dado en función del término anterior, es decir,  $x_n = f(x_{n-1})$ . En estas sucesiones es necesario conocer el valor de  $x_0$ .

Dependiendo de la forma  $f$  se consideran los siguientes casos:

- Si  $x_n$  es de la forma  $x_n = cx_{n-1}$ , se tiene que  $x_n = c^n x_0$ , ya que  $x_n = cx_{n-1} = c^2 x_{n-2} = \dots = c^n x_0$ .
- Si  $x_n$  es de la forma  $x_n = b_n x_{n-1}$  para  $n \geq 1$ , se tiene que  $x_n = b_1 b_2 \dots b_n x_0$ .
- Si  $x_n$  es de la forma  $x_n = b_n x_{n-1} + c_n$ , realizando el cambio de variable  $x_n = b_1 b_2 \dots b_n y_n$  en la recurrencia de  $x_{n+1}$ , se obtiene:

$$b_1 b_2 \dots b_{n+1} y_{n+1} = b_{n+1} (b_1 b_2 \dots b_n y_n) + c_{n+1}$$

lo que, operando, conduce a

$$x_n = (b_1 b_2 \dots b_n) \times \left( x_0 + \sum_{i=1}^n d_i \right)$$

siendo  $d_n = \frac{c_n}{(b_1 b_2 \dots b_n)}$ .

Como ejemplo, se expone el caso particular que se obtiene cuando  $b_n = b$  y  $c_n = c$ , es decir,  $x_{n+1} = bx_n + c$ . En este caso se realiza el cambio  $x_n = b^n y_n$  y se tiene:

$$b^{n+1} y_{n+1} = b^{n+1} y_n + c \Rightarrow y_{n+1} = y_n + \frac{c}{b^{n+1}}$$

$$\Rightarrow y_n = y_0 + \sum_{i=1}^n \left( \frac{c}{b^i} \right) = y_0 + c \frac{1 - b^n}{1 - b}$$

lo que conduce a  $x_n = x_0 + c \frac{1 - b^n}{(1 - b)b^n}$ .

Estas recurrencias son de gran importancia por sí mismas, pero además, las recurrencias generadas por sustracción y por división se reducen a ellas.

### Recurrencias generadas por sustracción

Existen algoritmos que dan lugar a recurrencias de la forma

$$x_n = Expr(n, x_{n-c})$$

conocido  $x_0$ . Mediante el cambio de variable  $n = kc$  tenemos:

$$\begin{aligned} x_n &= Expr(n, x_{n-c}) \\ &= Expr(kc, x_{kc-c}) \\ x_{kc} &= Expr(kc, x_{(k-1)c}) \end{aligned}$$

Si ahora llamamos a la sucesión  $x_{kc} = y_k$ , tenemos:

$$\begin{aligned} y_0 &= x_0 \\ y_k &= x_{kc} \\ &= Expr(kc, y_{k-1}) \end{aligned}$$

que es lineal de primer orden. Una vez resuelta, se tiene que

$$x_n = y_{n/c}$$

### Recurrencias generadas por división

Otros algoritmos dan lugar a recurrencias de la forma siguiente:

$$x_n = \dots n \dots x_{n/c}$$

conocido  $x_0$ . Mediante el cambio de variable  $n = c^k$  tenemos:

$$\begin{aligned}x_n &= Expr(n, x_{n/c}) \\ &= Expr(c^k, x_{c^k/c}) \\ x_{c^k} &= Expr(c^k, x_{c^{k-1}})\end{aligned}$$

Si ahora llamamos a la sucesión  $x_{c^k} = y_k$ , tenemos:

$$\begin{aligned}y_0 &= x_0 \\ y_k &= x_{c^k} \\ &= Expr(c^k, y_{k-1})\end{aligned}$$

que es lineal de primer orden. Una vez resuelta, se tiene que

$$x_n = y_{\log_c n}$$

Como ejemplo de este tipo de recurrencias, considérese el coste del algoritmo de ordenación por mezcla:

$$T(n) = \begin{cases} k_1 & \text{si } n = 1 \\ 2T(\frac{n}{2}) + k_2n + k_3 & \text{si } n > 1 \end{cases}$$

Como su resolución sigue al pie de la letra el procedimiento descrito, se deja como ejercicio al lector. La solución puede compararse con la ofrecida en 18.3.2.

### 18.4.3 Sucesiones de recurrencia de orden superior

Son las sucesiones generadas por subprogramas recursivos con más de una llamada recursiva.<sup>10</sup>

$$x_n = f(x_{n_1}, x_{n_2}, \dots, x_{n_k}, n)$$

La resolución exacta de este tipo de recurrencias sobrepasa las pretensiones de esta introducción a la complejidad algorítmica. Sin embargo, con frecuencia es posible y suficiente acotar dicho coste. En efecto, es frecuente que la sucesión  $x_i$  sea creciente y que entre los tamaños de las llamadas subsidiarias se puedan identificar el mínimo y el máximo y en general. Si llamamos  $x_{\text{mín}}$  y  $x_{\text{máx}}$  respectivamente a estos valores en la sucesión anterior, se tiene que

$$\left( \sum_{i=1}^k a_i \right) x_{\text{mín}} + f(n) \leq x_n \leq \left( \sum_{i=1}^k a_i \right) x_{\text{máx}} + f(n)$$

Esta desigualdad nos da siempre las acotaciones  $\Theta$  y  $O$  y, cuando coincidan ambas, tendremos el orden  $\Theta$ .

<sup>10</sup>Las distintas llamadas son  $x_{n_i}$ .

**Ejemplo: sucesión de Fibonacci**

En su definición recursiva usual, esta función tiene un coste dado por la siguiente relación de recurrencia:

$$\begin{aligned} t_n &= k_1 && \text{si } k = 0 \text{ o } k = 1 \\ t_n &= x_{n-1} + x_{n-2} + k_2 && \text{si } k \geq 0 \end{aligned}$$

Como  $t$  es creciente, podemos acotarla entre  $f$  y  $g$ , así:

$$\begin{aligned} f_n &= k_1 && \text{si } k = 0 \text{ o } k = 1 \\ f_n &= 2f_{n-2} + k_2 && \text{si } k \geq 0 \\ g_n &= k_1 && \text{si } k = 0 \text{ o } k = 1 \\ g_n &= 2g_{n-1} + k_2 && \text{si } k \geq 0 \end{aligned}$$

Estas dos relaciones de recurrencia son lineales, y se resuelven fácilmente:

$$f_n \in \Theta(2^{n/2}) \quad g_n \in \Theta(2^n)$$

por lo que podemos concluir que la función analizada tiene un coste en tiempo exponencial  $t_n = k^n$ , para una constante  $k$  sin determinar, entre  $\sqrt{2}$  y 2.

**18.5 Ejercicios**

1. Considere las siguientes funciones (dependientes de  $n$ ) de cara a estudiar su comportamiento asintótico:

$$\begin{array}{c} n^2 + 10^3 n + 10^6 \\ (5n^2 + 3)(3n + 2)(n + 1) \\ 2^n \\ 3^n \\ \frac{(n+1)(n^2-n+5)}{n(3+n^2)} \\ \sum_{i=1}^n i \end{array} \quad \left\| \begin{array}{c} n\sqrt{n} \\ \log \sqrt{n} \\ \left(\frac{1}{2}\right)^n \\ 2^n n^2 \\ \frac{1}{n} \\ \sum_{i=1}^n n \end{array} \right\| \quad \left\| \begin{array}{c} \log_{10} n \\ \sqrt{n} + \log n \\ 2^{1/n} \\ \log_e n \\ 1000 \\ \sum_{i=1}^n \sum_{j=1}^i n \end{array} \right\|$$

Se pide lo siguiente:

- (a) Para cada una de ellas, busque una función sencilla que acote superiormente su comportamiento asintótico, (usando para ello la notación  $O$  mayúscula) procurando ajustarse lo más posible.
  - (b) Clasifique las funciones anteriores según sean  $O(2^n)$ ,  $O(n^4)$ ,  $O(\sqrt{n})$ ,  $O(\log n)$  ó  $O(1)$ .
  - (c) Agrupe las funciones del ejercicio anterior que sean del mismo orden  $\Theta$ .
2. Compare las funciones siguientes por su orden de complejidad:

$$\log \log n \quad \left\| \quad \sqrt{\log n} \quad \left\| \quad (\log n)^2 \quad \left\| \quad \sqrt[4]{n}\right.\right.\right.$$

3. Calcule la complejidad en tiempo del siguiente algoritmo de sumar:

```

function Suma (m, n: integer): integer;
  {PreC.: n >= 0}
  {Dev. m + n }
  var
    i: integer;
begin
  for i:= 1 to n do
    m:= Succ (m)
end; {Suma}

```

4. Calcule ahora el coste del siguiente algoritmo de multiplicar,

```

function Producto (a, b: integer): integer;
  {PreC.: b >= 0}
  {Dev. a * b}
  var
    i, acumProd: integer;
begin
  acumProd:= 0;
  for i:= 1 to b do
    acumProd:= acumProd + a
end; {Producto}

```

en los tres casos siguientes:

- (a) Tal como se ha descrito.
  - (b) Cambiando, en el cuerpo de la instrucción **for**, la expresión **acumProd + a** por la llamada **Suma(acumProd, a)**.
  - (c) Cambiando la expresión **acumProd + a** de antes por la llamada **Suma(a, acumProd)**.
5. Considerando los problemas siguientes, esboce algoritmos iterativos para resolverlos e indique su complejidad:
- (a) Resolución de una ecuación de segundo grado.
  - (b) Cálculo del cociente y el resto de la división entera mediante restas sucesivas.
  - (c) Cálculo de un cero aproximado de una función mediante el método de bipartición.
  - (d) Suma de las cifras de un número entero positivo.
  - (e) Determinación de si un número es primo o no mediante el tanteo de divisores.
  - (f) Cálculo de  $\sum_{i=1}^n \frac{i+1}{i!}$ , diferenciando dos versiones:
    - i. Una, en la que cada vez se halla un término del sumatorio.
    - ii. Otra, donde cada factorial del denominador se halla actualizando el del término anterior.

- (g) Ordenación de un array de  $n$  componentes por el método de intercambio directo:

```

for i:= 1 to n - 1 do
  for j:= n downto i + 1 do
    Comparar las componentes j-1 y j
    e intercambiar si es necesario

```

- (h) Producto de dos matrices, una de  $m \times k$  y otra de  $k \times n$ .
6. Para el problema de las Torres de Hanoi de tamaño  $n$ , calcule su complejidad exacta en tiempo y en espacio, así como su orden de complejidad. Calcule el tiempo necesario para transferir los 64 discos del problema tal como se planteó en su origen, a razón de un segundo por movimiento. De esta forma, podrá saber la fecha aproximada del fin del mundo según la leyenda.
7. Considerando los problemas siguientes, esboce algoritmos recursivos para resolverlos e indique su complejidad:
- (a) Cálculo del cociente de dos enteros positivos, donde la relación de recurrencia (en su caso) es la siguiente:

$$\text{Coc}(\text{dividendo}, \text{divisor}) = 1 + \text{Coc}(\text{dividendo} - \text{divisor}, \text{divisor})$$

- (b) Cálculo del máximo elemento de un array; así, si es unitario, el resultado es su único elemento; si no, se halla (recursivamente) el máximo de su mitad izquierda, luego el de su mitad derecha del mismo modo y luego se elige el mayor de estos dos números.
- (c) Cálculo de  $\sum_{i=1}^n \frac{i+1}{i!}$ , usando la relación de recurrencia siguiente, en su caso:

$$\sum_{i=1}^n a_n = a_n + \sum_{i=1}^{n-1} a_n$$

- (d) Cálculo de los coeficientes binomiales  $\binom{n}{k}$ , distinguiendo las siguientes versiones:
- Iterativa, mediante el cálculo de  $\frac{m!}{n!(m-n)!}$ .
  - Iterativa, mediante el cálculo de  $\frac{m(m-1)\dots(n+1)}{(m-n)!}$ .
  - Recursiva, mediante la relación de recurrencia conocida.
- (e) Evaluación de un polinomio, conocido el valor de la variable  $x$ , en los siguientes casos:
- La lista de los coeficientes viene dada en un array, y se aplica la fórmula  $\sum_i \text{coef}_i x^i$ .
  - Los coeficientes están en una lista enlazada, y usamos la regla de Horner (véase el apartado 17.1.4).
- (f) Ordenación de un array de  $n$  componentes por el método de mezcla:

```
procedure MergeSort (var v:  $V_n(\text{elem})$ : iz, der);
  {Efecto: se ordena ascendentemente v[iz..der]}
begin
  if iz < der + 1 then begin
    MergeSort (la mitad izquierda de v);
    MergeSort (la mitad derecha de v);
    Mezclar las dos mitades de v
  end {if}
end; {MergeSort}
```

suponiendo que las mitades del vector son siempre de igual tamaño, para simplificar, y que el proceso de mezclarlas tiene un coste proporcional a su tamaño.

## 18.6 Referencias bibliográficas

La complejidad de algoritmos se estudia más o menos a fondo en casi todos los cursos de estructuras avanzadas de datos y de metodología de la programación. De las muchas referencias mencionables, omitiremos aquí las que tienen un contenido similar, y citaremos en cambio sólo algunas de las que pueden servir para ampliar lo expuesto aquí.

Para empezar, citamos [MSPF95], una suave introducción a los conceptos básicos de la complejidad. En el clásico libro de Brassard y Bratley [BB97] (especialmente en el capítulo 2) se pueden ampliar las técnicas estudiadas para resolver recurrencias. Otro libro interesante es [Wil89], que está completamente dedicado a la complejidad de algoritmos (incluyendo el estudio de algunos algoritmos conocidos de investigación operativa y de teoría de números útiles en criptografía), así como a la complejidad de problemas. En [BKR91], se estudia el tema y su aplicación en el desarrollo de programas y estructuras de datos con el objetivo de minimizar el coste. En este libro se introduce además el coste en el caso medio de algoritmos, así como el análisis de algoritmos paralelos.



## Capítulo 19

# Tipos abstractos de datos

---

<b>19.1</b>	<b>Introducción . . . . .</b>	<b>428</b>
<b>19.2</b>	<b>Un ejemplo completo . . . . .</b>	<b>429</b>
<b>19.3</b>	<b>Metodología de la programación de TADs . . . . .</b>	<b>440</b>
<b>19.4</b>	<b>Resumen . . . . .</b>	<b>446</b>
<b>19.5</b>	<b>Ejercicios . . . . .</b>	<b>447</b>
<b>19.6</b>	<b>Referencias bibliográficas . . . . .</b>	<b>448</b>

---

A medida que se realizan programas más complejos, va aumentando simultáneamente la complejidad de los datos necesarios. Como se explicó en el capítulo 11, este aumento de complejidad puede afrontarse, en primera instancia, con las estructuras de datos proporcionadas por Pascal como son, por ejemplo, los arrays y los registros.

Cuando en un programa, o en una familia de ellos, el programador descubre una estructura de datos que se utiliza repetidamente, o que puede ser de utilidad para otros programas, es una buena norma de programación definir esa estructura como un nuevo tipo de datos e incluir variables de este tipo cada vez que sea necesario, siguiendo así un proceso similar al de la abstracción de procedimientos (véanse los apartados 8.1 y 9.3.1), mediante la cual se definen subprogramas que pueden ser reutilizados. En este proceso, denominado *abstracción de datos*, el programador debe despreocuparse de los detalles menores, concentrándose en las operaciones globales del tipo de datos. Esto es, en términos generales, lo que se persigue con los *tipos abstractos de datos*.

## 19.1 Introducción

El objetivo que se persigue en este apartado es la definición de un conjunto de objetos con una serie de operaciones para su manipulación. Para resolverlo, se utiliza la técnica de la *abstracción de datos*, que permite tratar estas definiciones de tipos de una forma ordenada, mantenible, reutilizable y coherente.

La abstracción de datos pone a disposición del programador-usuario<sup>1</sup> nuevos tipos de datos con sus correspondientes operaciones de una forma totalmente independiente de la representación de los objetos del tipo y de la implementación de las operaciones; de ahí el nombre *tipos abstractos de datos*.

Esta técnica es llamada *abstracción*, porque aplica un proceso consistente en ignorar ciertas características de los tipos de datos por ser irrelevantes para el problema que se intenta resolver. Esas características ignoradas son aquéllas relativas a *cómo* se implementan los datos, centrándose toda la atención en *qué* se puede hacer con ellos. Esto es, las propiedades de un tipo abstracto de datos vienen dadas implícitamente por su definición y no por una representación o implementación particular.

Obsérvese que los tipos de datos básicos de Pascal (véase el capítulo 3) son *abstractos* en el siguiente sentido: el programador puede disponer de, por ejemplo, los enteros y sus operaciones (representados por el tipo `integer`), ignorando la representación concreta (complemento restringido o auténtico, o cualquiera de las explicadas en el apartado 2.2 del tomo I) escogida para éstos.

En cambio, con los tipos definidos por el programador (por ejemplo, las colas presentadas en el apartado 17.3), los detalles de la implementación están a la vista, con los siguientes inconvenientes:

- El programador tiene que trabajar con la representación de un objeto en lugar de tratar con el objeto directamente.
- El programador podría usar el objeto de modo inconsistente si manipula inadecuadamente la representación de éste.

En resumen, desde el punto de vista del programador-usuario se puede afirmar que la introducción de los tipos abstractos de datos suponen un aumento de nivel en la programación, pues bastará con que éste conozca el *qué*, despreocupándose de las características irrelevantes (el *cómo*) para la resolución del problema. Por otra parte, la tarea del programador que implementa el tipo consistirá en escoger la representación concreta que considere más adecuada y ocultar los detalles de ésta en mayor o menor nivel, dependiendo del lenguaje utilizado.

---

<sup>1</sup>Designaremos con este término al programador que utiliza porciones de código puestas a su disposición por otros programadores (o por él mismo, pero de forma independiente).

## 19.2 Un ejemplo completo

Es conveniente concretar todas estas ideas mediante un ejemplo: como se vio en el apartado 11.3, la representación de conjuntos en Pascal tiene una fuerte limitación en cuanto al valor máximo del cardinal de los conjuntos representados (por ejemplo, en Turbo Pascal el cardinal máximo de un conjunto es 256). En el siguiente ejemplo, se pretende representar conjuntos sin esta restricción. Posteriormente, se utilizarán estos conjuntos ilimitados en un programa que escriba los números primos menores que uno dado por el usuario, usando el conocido método de la *criba de Eratóstenes* (véase el apartado 11.3.3). La idea de la representación es disponer de un conjunto inicial con todos los enteros positivos menores que el valor introducido por el usuario e ir eliminando del conjunto aquellos números que se vaya sabiendo que no son primos. De acuerdo con esta descripción, una primera etapa de diseño del programa podría ser:

```
Leer cota ∈ IN
Generar el conjunto inicial, {2, ..., cota}
Eliminar los números no primos del conjunto
Escribir los números del conjunto
```

Detallando un poco más cada una de esas acciones, se tiene: *Generar el conjunto inicial* se puede desarrollar así:

```
Crear un conjunto vacío primos
Añadir a primos los naturales de 2 a cota
```

Para *Eliminar los números no primos del conjunto*, basta con lo siguiente:

```
para cada elemento e ∈ conjunto, entre 2 y √cota
  Eliminar del conjunto todos los múltiplos de e
```

En un nivel de refinamiento inferior, se puede conseguir *Eliminar del conjunto todos los múltiplos de e* de la siguiente forma:

```
coeficiente:= 2;
repetir
  Eliminar e * coeficiente del conjunto
  coeficiente:= coeficiente + 1
hasta que e * coeficiente sea mayor que cota
```

Finalmente, *Escribir los números del conjunto* no presenta problemas y se puede hacer con un simple recorrido de los elementos del `conjunto`.

### 19.2.1 Desarrollo de programas con tipos concretos de datos

Una vez detallado este nivel de refinamiento, se tiene que tomar una decisión sobre el modo de representar los conjuntos. Las posibilidades son múltiples: con el tipo **set** en Pascal, con listas enlazadas, etc.

En este apartado se presenta una implementación del diseño anterior, representando un conjunto de enteros en una lista enlazada con cabecera, con los elementos ordenados ascendentemente y sin repeticiones. Desde el nivel de refinamiento alcanzado en el apartado anterior se puede pasar directamente a la implementación:

```

Program CribaEratostenes (input, output);
  {PreC.: input = [un entero, >=2]}
  type
    tConjunto = ^tNodoEnt;
    tNodoEnt = record
      elem: integer;
      sig: tConjunto
    end; {tNodoEnt}
  var
    cota, e, coef: integer;
    conjunto, aux, puntPrimo, auxElim: tConjunto;
begin
  {Leer cota;}
  Write('Cota: ');
  ReadLn(cota);
  {Generar el conjunto inicial, [2, ..., cota]:}
  New(conjunto);
  aux:= conjunto;
  for e:= 2 to cota do begin
    New(aux^.sig);
    aux:= aux^.sig;
    aux^.elem:= e
  end; {for i}
  aux^.sig:= nil;
  {Eliminar los números no primos del conjunto;}
  puntPrimo:= conjunto^.sig;
  repeat
    e:= puntPrimo^.elem;
    coef:= 2; aux:= puntPrimo;
    while (e * coef <= cota) and (aux^.sig <> nil) do begin
      if aux^.sig^.elem < coef * e then
        aux:= aux^.sig
      else if aux^.sig^.elem = coef * e then begin
        auxElim:= aux^.sig^.sig;
        Dispose(aux^.sig);

```

```

        aux^.sig:= auxElim;
        coef:= coef + 1
    end {else if}
    else if aux^.sig^.elem > coef * e then
        coef:= coef + 1
    end; {while}
    puntPrimo:= puntPrimo^.sig
until (e >= Sqrt(cota)) or (puntPrimo = nil);
{Escribir los números del conjunto:}
aux:= conjunto^.sig;
while aux <> nil do begin
    Write(aux^.elem:4);
    aux:= aux^.sig
end; {while}
WriteLn
end. {CribaEratostenes}

```

Queda claro que a partir del momento en que se ha adoptado esta representación, quedan mezclados los detalles relativos al algoritmo (Criba de Eratóstenes) con los relativos a la representación (lista enlazada ...) y manipulación de los conjuntos de enteros. Este enfoque acarrea una serie de inconvenientes, como son:

- El código obtenido es complejo, y, en consecuencia, se dificulta su corrección y verificación.
- El mantenimiento del programa es innecesariamente costoso: si, por ejemplo, se decidiese cambiar la estructura dinámica lineal por una de manejo más eficiente, como podrían ser los árboles binarios de búsqueda, se debería rehacer la totalidad del programa. Dicho de otro modo, es muy difícil aislar cambios o correcciones.
- En el caso en que se necesitasen conjuntos sin restricciones de cardinal en otros programas, sería necesario volver a implementar en éstos todas las tareas necesarias para su manipulación. En otras palabras, no hay posibilidad de reutilizar código.

Estos inconvenientes son los que se pretende superar con los tipos abstractos de datos.

### 19.2.2 Desarrollo de programas con tipos abstractos de datos

Una forma de solucionar los problemas enunciados anteriormente es el empleo de la abstracción de datos. Así, se definirá un tipo abstracto de datos, entendido

informalmente como una colección de objetos con un conjunto de operaciones definidas sobre estos objetos. Se tendrá siempre en cuenta la filosofía de la abstracción de datos: todas las características del tipo abstracto vienen dadas por su definición y no por su implementación (de la que es totalmente independiente).

En el ejemplo anterior, es evidente que la aplicación de la abstracción de datos conducirá a un tipo abstracto:<sup>2</sup>

```
type
  tConj = Abstracto
```

cuyos objetos son precisamente conjuntos con un número arbitrario de elementos enteros que se podrán manipular con las siguientes operaciones:<sup>3</sup>

```
procedure CrearConj (var conj: tConj);
  {Efecto: conj :=  $\emptyset$ }

procedure AnnadirElemConj (elem: integer; var conj: tConj);
  {Efecto: conj := conj  $\cup$  [elem]}

procedure QuitarElemConj (elem: integer; var conj: tConj);
  {Efecto: conj := conj  $\setminus$  [elem]}

function Pertenece (elem: integer; conj: tConj): boolean;
  {Dev. True (si elem  $\in$  conj) o False (en otro caso)}

procedure EscribirConj (conj: tConj);
  {Efecto: escribe en el output los elementos de conj}

function EstaVacioConj(conj: tConj): boolean;
  {Dev. True (si conj =  $\emptyset$ ) o False (en otro caso)}
```

Como se dijo anteriormente, una característica esencial de los tipos abstractos de datos es su independencia de la implementación. En este momento, y sin saber nada en absoluto acerca de la forma en que está implementado (o en que se va a implementar) el tipo `tConj`, se puede utilizar de una forma abstracta, siendo más que suficiente la información proporcionada por la especificación de las operaciones del tipo. Además, se podrá comprobar inmediatamente cómo el código obtenido es más claro, fácil de mantener y verificar. La nueva versión del programa `CribaEratostenes` es la siguiente:

---

<sup>2</sup>Acéptese esta notación provisional, que se detalla en el apartado 19.2.3.

<sup>3</sup>En los casos en los que se ha considerado conveniente, se ha sustituido la postcondición por una descripción algo menos formal del efecto del subprograma.

```

Program CribaEratostenes (input, output);
  {PreC.: input = [un entero, >=2]}
  type
    tConj = Abstracto;
  var
    cota, e, coef: integer;
    conjunto : tConj;
  begin
    Write('Cota: '); ReadLn(cota);
    CrearConj(conjunto);
    for e:= 2 to cota do
      AnnadirElemConj(e, conjunto);
    for e:= 2 to Trunc(SqRt(cota)) do
      if Pertenece (e, conjunto) then begin
        coef:= 2;
        repeat
          QuitarElemConj(e * coef, conjunto);
          coef:= coef + 1
        until e * coef > cota
        end; {if}
      EscribirConj(conjunto)
    end. {CribaEratostenes}

```

En el ejemplo se puede observar la esencia de la abstracción de datos: el programador-usuario del tipo abstracto de datos se puede olvidar completamente de *cómo* está implementado o de la representación del tipo, y únicamente estará interesado en qué se puede hacer con el tipo de datos `tConj` que utiliza de una forma completamente abstracta.

Además, el lector puede observar cómo se llega a una nueva distribución (mucho más clara) del código del programa: las operaciones sobre el tipo `tConj` dejan de formar parte del programa y pasan a incorporarse al código propio del tipo abstracto.

Las operaciones del tipo `tConj` escogidas son típicas de muchos tipos abstractos de datos, y se suelen agrupar en las siguientes categorías:

**Operaciones de creación:** Son aquéllas que permiten obtener nuevos objetos del tipo, como sucede en `CrearConj`. Estas últimas son conocidas también como operaciones constructoras primitivas. Entre éstas suele incluirse una operación de lectura de elementos del tipo abstracto de datos.

**Operaciones de consulta:** Realizan funciones que, tomando como argumento un objeto del tipo abstracto, devuelven un valor de otro tipo, como hacen `Pertenece` o `EstaVacioConj`, por poner un caso. Usualmente implementan tareas que se ejecutan con relativa frecuencia. Entre éstas se suele incluir

una operación que escriba objetos del tipo abstracto de datos, que en el ejemplo sería un procedimiento `EscribirConj`.

**Operaciones de modificación:** Permiten, como su propio nombre indica, modificar un objeto del tipo abstracto de datos, como, por ejemplo, las operaciones `Annadir` y `Eliminar`.

**Operaciones propias del tipo:** Son operaciones características de los objetos abstraídos en el tipo de datos, como serían en el ejemplo operaciones para calcular la unión, intersección o diferencia de conjuntos.

Es conveniente destacar que esta clasificación de las operaciones no es excluyente: en algún tipo abstracto de datos puede ser necesaria una operación que pertenezca a dos clases, como, por ejemplo, una operación que efectúe una consulta y una modificación al mismo tiempo. No obstante, tal operación no es adecuada desde el punto de vista de la cohesión, un criterio de calidad de *software* aplicable a la programación con subprogramas, que nos recomienda dividir tal operación en dos, una que haga la consulta y otra la modificación.

A modo de resumen, se puede decir que la abstracción de datos, considerada como método de programación, consiste en el desarrollo de las siguientes etapas:

1. Reconocer los objetos candidatos a elementos del nuevo tipo de datos.
2. Identificar las operaciones del tipo de datos.
3. Especificar las operaciones de forma precisa.
4. Seleccionar una buena implementación.

Se han mostrado las tres primeras etapas tomando como guía el ejemplo de los conjuntos de enteros. A continuación se detalla cómo abordar la cuarta y última.

### 19.2.3 Desarrollo de tipos abstractos de datos

En la última versión del ejemplo de la criba de Eratóstenes se ha utilizado el tipo abstracto `tConj` dejándolo sin desarrollar, únicamente incluyendo la palabra *abstracto*. Pero, obviamente, por mucho que las características de un tipo abstracto de datos sean independientes de la implementación, no se puede olvidar ésta.

Lo más adecuado para implementar un tipo abstracto de datos es recurrir a mecanismos que permitan *encapsular* el tipo de datos y sus operaciones, y *ocultar* la información al programador-usuario. Estas dos características son esenciales

para llevar a cabo efectivamente la abstracción de datos. A tal fin, Turbo Pascal dispone de las *unidades* (véase el apartado B.11).

De acuerdo con esto, se da a continuación una posible implementación<sup>4</sup> del tipo abstracto de datos `tConj`, a base de listas de enteros, ordenadas ascendentemente, enlazadas con punteros:<sup>5</sup>

```

unit conjEnt;
  {Implementación mediante listas enlazadas, con cabecera,
   ordenadas ascendentemente y sin repeticiones}

interface
  type
    tElem = integer;
    {Requisitos: definidas las relaciones '=' (equiv.) y
     '>' (orden total)}
    tConj = ^tNodoLista;
    tNodoLista = record
      info: tElem;
      sig: tConj
    end; {tNodoLista}

  procedure CrearConj(var conj: tConj);
    {Efecto: conj:= ∅}

  procedure DestruirConj(var conj: tConj);
    {Cuidado: se perderá toda su información}
    {Efecto: conj:= ? (ni siquiera queda
     vacío: comportamiento impredecible)}

  procedure AnnadirElemConj (elem: tElem; var conj: tConj);
    {Efecto: conj:= conj ∪ [elem]}

  procedure QuitarElemConj (elem: tElem; var conj: tConj);
    {Efecto: conj:= conj \ [elem]}

  function Pertenece (elem: tElem; conj: tConj): boolean;
    {Dev. True (si elem ∈ conj) o False (en otro caso)}

```

---

<sup>4</sup>Se ha optado por no detallar todas las etapas del diseño descendente de las operaciones de los tipos abstractos, por ser éstas sencillas y para no extender demasiado el texto.

Por otra parte, es conveniente que los identificadores de unidades coincidan con el nombre del archivo donde se almacenan una vez compiladas (de ahí la elección de identificadores de, a lo sumo, ocho caracteres).

<sup>5</sup>Obsérvese que se ha incluido una operación destructora, necesaria en la práctica, debido a que el manejo de listas enlazadas ocasiona un gasto de memoria y ésta debe liberarse cuando un conjunto deje de ser necesario.

```

procedure EscribirConj (conj: tConj);
  {Efecto: escribe en el output los elementos de conj}

function EstaVacioConj(conj: tConj): boolean;
  {Dev. True (si conj =  $\emptyset$ ) o False (en otro caso)}

```

### implementation

{Representación mediante listas enlazadas, con cabecera,  
ordenadas ascendentemente y sin repeticiones}

```

procedure CrearConj (var conj: tConj);
begin
  New(conj);
  conj^.sig:= nil
end; {CrearConj}

procedure DestruirConj (var conj: tConj);
  var
    listaAux: tConj;
begin
  while conj <> nil do begin
    listaAux:= conj;
    conj:= conj^.sig;
    Dispose(listaAux)
  end {while}
end; {DestruirConj}

procedure AnnadirElemConj (elem: tElem; var conj: tConj);
  var
    parar, {indica el fin de la búsqueda, en la lista}
    insertar: boolean; {por si elem ya está en la lista}
    auxBuscar, auxInsertar: tConj;
begin
  auxBuscar:= conj;
  parar:= False;
  repeat
    if auxBuscar^.sig = nil then begin
      parar:= True;
      insertar:= True
    end {then}
    else if auxBuscar^.sig^.info >= elem then begin
      parar:= True;
      insertar:= auxBuscar^.sig^.info > elem
    end {then}
  until parar;
  if insertar then
    New(auxInsertar);
    auxInsertar^.sig:= auxBuscar;
    auxInsertar^.info:= elem;
    auxBuscar^.sig:= auxInsertar;
  end

```

```

    auxBuscar:= auxBuscar^.sig
until parar;
if insertar then begin
    auxInsertar:= auxBuscar^.sig;
    New(auxBuscar^.sig);
    auxBuscar^.sig^.info:= elem;
    auxBuscar^.sig^.sig:= auxInsertar
end {if}
end; {AnnadirElemConj}

procedure QuitarElemConj (elem: tElem; var conj: tConj);
var
    parar, {indica el fin de la búsqueda, en la lista}
    quitar: boolean; {por si el elem no está en la lista}
    auxBuscar, auxQuitar: tConj;
begin
    auxBuscar:= conj;
    parar:= False;
    repeat
        if auxBuscar^.sig = nil then begin
            parar:= True;
            quitar:= False
        end {then}
        else if auxBuscar^.sig^.info >= elem then begin
            parar:= True;
            quitar:= auxBuscar^.sig^.info = elem
        end
        else
            auxBuscar:= auxBuscar^.sig
    until parar;
    if quitar then begin
        auxQuitar:= auxBuscar^.sig^.sig;
        Dispose(auxBuscar^.sig);
        auxBuscar^.sig:= auxQuitar
    end {if}
end; {QuitarElemConj}

function Pertenece (elem: tElem; conj: tConj): boolean;
var
    parar, {indica el fin de la búsqueda, en la lista}
    esta: boolean; {indica si elem ya está en la lista}
    auxBuscar: tConj;
begin
    auxBuscar:= conj^.sig;
    parar:= False;

    repeat

```

```

    if auxBuscar = nil then begin
        parar:= True;
        esta:= False
    end
    else if auxBuscar^.info = elem then begin
        parar:= True;
        esta:= True
    end
    else if auxBuscar^.info > elem then begin
        parar:= True;
        esta:= False
    end
    else
        auxBuscar:= auxBuscar^.sig
    until parar;
    pertenece:= esta
end; {Pertenece}

procedure EscribirConj (conj: tConj);
var
    puntAux: tConj;
begin
    if EstaVacioConj(conj) then
        WriteLn('[]')
    else begin
        puntAux:= conj^.sig;
        Write('[', puntAux^.info);
        while not EstaVacioConj(puntAux) do begin
            puntAux:= puntAux^.sig;
            Write(', ', puntAux^.info)
        end; {while}
        WriteLn(']')
    end {else}
end; {EscribirConj}

function EstaVacioConj (conj: tConj): boolean;
begin
    EstaVacioConj:= conj^.sig = nil
end; {EstaVacioConj}
end. {conjEnt}

```

Para finalizar, se recuerda que en el programa *CribaEratostenes* se había dejado incompleta la declaración de tipos, así, únicamente

```

type
    tConj = Abstracto;

```

Con los elementos de que se dispone ahora, esta declaración se sustituirá por la siguiente:

```
uses conjEnt;
```

que hace que el tipo `tConj` declarado en la unidad esté disponible para su empleo en el programa.

- ☉☉ Una implementación en Pascal estándar obligaría a prescindir de las unidades, obligando a incluir todas las declaraciones del tipo y de los subprogramas correspondientes a las operaciones del tipo abstracto `tConj` en todo programa en que se utilice. Con ello se perderían gran parte de las ventajas aportadas por la abstracción de datos, como, por ejemplo, la *encapsulación*, la *ocultación* de información y el aislamiento de los cambios.

La representación escogida para la implementación no es la más eficiente. En efecto, las operaciones de inserción, eliminación y consulta requieren, en el peor caso, el recorrido del conjunto completo.

Una posibilidad más interesante consiste en representar los conjuntos mediante árboles de búsqueda (véase el apartado 17.4.2) en vez de usar listas. No es necesario reparar ahora en los pormenores de esta estructura de datos; en este momento, nos basta con saber que las operaciones de modificación y consulta son ahora mucho más eficientes.

Pues bien, si ahora decidiésemos mejorar la eficiencia de nuestra implementación, cambiando las listas por árboles, no tenemos que modificar las operaciones de la interfaz, sino tan sólo su desarrollo posterior en la sección **implementation**. De este modo, se puede apreciar una de las ventajas de la abstracción de datos:

- El programador-usuario no tiene que pensar en ningún momento en la representación del tipo abstracto de datos, sino únicamente en su especificación.
- Los cambios, correcciones o mejoras introducidas en la implementación del tipo abstracto de datos repercuten en el menor ámbito posible, cual es la unidad en que se incluye su representación. En nuestro ejemplo, el cambio de la implementación basada en listas por la basada en árboles no afecta ni a una sola línea del programa `Criba`: el programador-usuario sólo se sorprenderá con una mayor eficiencia de su programa tras dicho cambio. Obsérvese el efecto que hubiera tenido el cambio de representación en la primera versión de `Criba`, y compare.

Una vez presentado este ejemplo y con él las ideas generales de la abstracción de datos, se dan en el siguiente apartado nociones metodológicas generales acerca de esta técnica.

### 19.3 Metodología de la programación de tipos abstractos de datos

En los apartados anteriores se han introducido los tipos abstractos de datos de un modo paulatino, práctico e informal. Es necesario, y es lo que se hace en este apartado, precisar las ideas generales introducidas y presentar los aspectos necesarios para la correcta utilización de los tipos abstractos de datos como método de programación.

Siguiendo las ideas de J. Martin [Mar86], se puede definir un tipo abstracto de datos como un sistema con tres componentes:

1. Un conjunto de objetos.
2. Un conjunto de descripciones sintácticas de operaciones.
3. Una descripción semántica, esto es, un conjunto suficientemente completo de relaciones que especifiquen el funcionamiento de las operaciones.

Se observa que, mientras en el apartado anterior se describían los tipos abstractos de datos como un conjunto de objetos y una colección de operaciones sobre ellos, ahora se subraya la descripción de la sintaxis y la semántica de esas operaciones.

Esto es necesario dado que la esencia de los tipos abstractos de datos es que sus propiedades vienen descritas por su especificación, y, por tanto, es necesario tratar ésta adecuadamente.

Además, este tratamiento riguroso de las especificaciones de los tipos abstractos de datos permite ahondar en la filosofía expuesta al comienzo del tema: separar *qué* hace el tipo abstracto de datos (lo cual viene dado por la especificación) de *cómo* lo lleva a cabo (lo que se da en su implementación). Estos dos aspectos se repasan brevemente en los dos siguientes apartados.

#### 19.3.1 Especificación de tipos abstractos de datos

A la hora de afrontar la especificación de un tipo abstracto de datos se dispone de diversos lenguajes variando en su nivel de formalidad. En un extremo, se tiene el lenguaje natural, en nuestro caso el español. Las especificaciones así expresadas presentan importantes inconvenientes, entre los que destacan su ambigüedad (que puede llegar a inutilizar la especificación, por prestarse a interpretaciones incorrectas y/o no deseadas), y la dificultad para comprobar su corrección y “completitud”.

Ante estos problemas, es necesario recurrir a lenguajes más precisos, como puede ser el lenguaje matemático, que posee las ventajas de su precisión, concisión y universalidad (aunque en su contra se podría argumentar su dificultad de uso, lo cierto es que en los niveles tratados en este libro no es preciso un fuerte aparato matemático). Por ejemplo, algunas de las operaciones del tipo abstracto `tConj` se han especificado como sigue:

```
procedure CrearConj(var conj: tConj);  
  {Efecto: conj:= ∅}  
  
procedure AnnadirElemConj(elem: integer; var conj: tConj);  
  {Efecto: conj:= conj ∪ {elem}}  
  
function Pertenece (elem: tElem; conj: tConj): boolean;  
  {Dev. True (si elem ∈ conj) o False (en otro caso)}  
  
function EstaVacioConj(conj: tConj): boolean;  
  {Dev. True (si conj = ∅) o False (en otro caso)}
```

Obsérvese que la sintaxis de su uso viene dada por los encabezamientos de las operaciones y que su semántica se da en el comentario que les sigue. Las especificaciones de esta forma proporcionan un modelo más o menos formal que describe el comportamiento de las operaciones sin ningún tipo de ambigüedad. Además, se satisfacen las dos propiedades que deben cumplir las especificaciones: precisión y brevedad.

Por último, se debe resaltar la importancia de la especificación como medio de comunicación entre el programador-diseñador del tipo abstracto de datos, el implementador y el programador-usuario: como se ha repetido anteriormente, la información pública del tipo abstracto de datos debe ser únicamente su especificación. Así, una especificación incorrecta o incompleta impedirá a los implementadores programar adecuadamente el tipo abstracto de datos, mientras que los programadores-usuarios serán incapaces de predecir correctamente el comportamiento de las operaciones del tipo, lo que producirá errores al integrar el tipo abstracto de datos en sus programas.

### 19.3.2 Implementación de tipos abstractos de datos

Como se indicó anteriormente, la tercera etapa de la abstracción de datos es la implementación del tipo de acuerdo con la especificación elaborada en la etapa anterior.

La idea de partida que se ha de tomar en esta tarea es bien clara: escoger una representación que permita implementar las operaciones del tipo abstracto de datos simple y eficientemente, respetando, por supuesto, las eventuales restricciones existentes.

Ahora bien, en el momento de llevar a la práctica esta idea, se ha de tener presente que el verdadero sentido de la abstracción de datos viene dado por la separación del *qué* y del *cómo*, de la especificación y de la implementación, y de la ocultación al programador-usuario de la información referente a esta última.

De acuerdo con esto, y como ya se adelantó en el apartado 19.2.3, resulta imprescindible disponer en el lenguaje de programación empleado de herramientas que permitan la implementación separada de los programas y los datos. Estas herramientas se denominan *unidades*, *módulos* o *paquetes* en algunos de los lenguajes de programación imperativa más extendidos. Pero estas herramientas no están disponibles en Pascal estándar, por lo que es imposible realizar la abstracción de datos de una forma completa<sup>6</sup> en este lenguaje, como se advirtió en el apartado anterior. Sin embargo, en Turbo Pascal sí que se dispone de la posibilidad de compilación separada mediante el empleo de *unidades* (véase el apartado B.11), y esto es suficiente para llevar a la práctica la abstracción de datos.

En otros términos, las unidades de Turbo Pascal posibilitan una total “encapsulación” de los datos, al incluir la definición y la implementación del tipo abstracto en la misma unidad. Ésta es una característica positiva, al aumentar la modularidad de los programas y facilitar el aislamiento de los cambios.

No obstante, las unidades permiten ocultar sólo parcialmente las características del tipo abstracto de datos dadas por la representación escogida. Es preciso recalcar la parcialidad de la ocultación de información, ya que es necesario incluir la representación concreta en la declaración del tipo abstracto (en el ejemplo `tConj = ^tNodoLista;` o `tConj = árbol de búsqueda`) contenida en la sección de interfaz de la unidad, y, por tanto, el programador-usuario conocerá parte de los detalles de la representación.<sup>7</sup>

Para finalizar, se puede decir que la implementación típica de un tipo abstracto de datos en Turbo Pascal, siguiendo las ideas de Collins y McMillan [CM], tiene la siguiente estructura:

```
unit TipoAbstractoDeDatos;
```

---

<sup>6</sup>En Pascal estándar habría que incluir la implementación del tipo de datos y de sus operaciones en cada programa que lo necesite, con lo cual no existe ninguna separación entre la especificación y la implementación, que estará totalmente visible al programador-usuario.

<sup>7</sup>En otros lenguajes, como, por ejemplo, Modula2, se puede conseguir la ocultación total de información mediante el empleo de los llamados *tipos opacos*.

**interface**

**uses** *Otras unidades necesarias;*

*Declaraciones de constantes, tipos, y variables necesarios para definir el tipo abstracto*

*Encabezamientos de las operaciones del tipo abstracto de datos*

**implementation**

**uses** *Otras unidades necesarias;*

*Información privada, incluyendo las implementaciones de las operaciones del tipo abstracto de datos y de los tipos participantes, así como las constantes, tipos y variables necesarios para definir éstos, y las operaciones privadas*

**begin**

*Código de iniciación, si es preciso*

**end.** {TipoAbstractoDeDatos}

**19.3.3 Corrección de tipos abstractos de datos**

La verificación de un tipo abstracto de datos debe hacerse a dos niveles:

En primer lugar, se debe estudiar si la implementación de las diferentes operaciones satisface la especificación, bien mediante una verificación *a posteriori*, o bien a través de una derivación correcta de programas (en los términos presentados en el apartado 5.4) partiendo de la especificación y desarrollando las operaciones de acuerdo con ésta. Siguiendo esta segunda opción, por ejemplo, para la operación `AnnadirElemConj`, se partiría de la especificación

{Efecto:  $\text{conj} := \text{conj} \cup [\text{elem}]$ }

y se llegaría al código

```
procedure AnnadirElemConj (elem: tElem; var conj: tConj);
  var
    parar, {indica el fin de la búsqueda, en la lista}
    insertar: boolean; {por si elem ya está en la lista}
    auxBuscar, auxInsertar: tConj;
begin
  auxBuscar := conj;
  parar := False;
  repeat
    if auxBuscar^.sig = nil then begin
```

```

    parar:= True;
    insertar:= True
end
else if auxBuscar^.sig^.info >= elem then begin
    parar:= True;
    insertar:= auxBuscar^.sig^.info > elem
end
else
    auxBuscar:= auxBuscar^.sig
until parar;
if insertar then begin
    auxInsertar:= auxBuscar^.sig;
    New(auxBuscar^.sig);
    auxBuscar^.sig^.info:= elem;
    auxBuscar^.sig^.sig:= auxInsertar
end {if}
end; {AnnadirElemConj}

```

que verifica la especificación, como es fácil comprobar examinando el código. En efecto, la implementación propuesta para `AnnadirElemConj`, hace que se agregue `elem` a la representación de `conj`. Más detalladamente, en el caso del que `elem` no pertenezca al conjunto, se asigna el valor `True` a la variable `insertar`, provocando la ejecución de la rama **then** de la instrucción `if insertar...`, que añade un nuevo nodo (cuya información es `elem`) en la lista que representa a `conj`. Esto demuestra informalmente que `AnnadirElemConj` consigue el efecto descrito en la especificación.

Por otra parte, en los tipos abstractos de datos, se debe estudiar la corrección en una segunda dirección, ya que es necesario establecer propiedades de los objetos del tipo, y mediante ellas comprobar que los objetos que se manejan pertenecen realmente al tipo. Estas propiedades se formalizan en los llamados *invariantes de la representación* de la forma que se explica seguidamente. El sentido de esta segunda parte de la verificación es el de una comprobación de tipos exhaustiva: al escoger una representación, por ejemplo, las listas en el caso del tipo abstracto `tConj`, se activa la comprobación de tipos implícita de Pascal, es decir, el compilador se asegura de que todo objeto del tipo `tConj` es una lista del tipo `^tNodoLista`.

Pero esta comprobación de tipos implícita no es suficiente, por el hecho de que no toda lista de enteros es una representación legal de un conjunto, concretamente porque puede tener elementos repetidos. Por tanto, es necesaria una verificación más profunda que debe ser realizada por el programador-diseñador. Esta verificación se llevará a cabo formalizando las propiedades pertinentes de los objetos en los citados invariantes de representación.

Por ejemplo, si se decide representar los conjuntos de enteros mediante listas sin repetición, este invariante se podría formalizar como sigue:

$$\{\text{Inv. } \forall i, j \in I, \text{ si } i \neq j, \text{ entonces } l_i \neq l_j\}$$

donde  $I$  es el conjunto de “índices” correspondientes a los elementos de la lista, y  $l_i$  es el elemento que ocupa la posición  $i$ -ésima en la lista. De esta forma se expresa que no deben existir dos elementos iguales en la lista.

No hay normas generales para establecer el invariante de representación, ya que depende del tipo abstracto de datos y de la representación escogida. Incluso, en alguna ocasión, el invariante puede ser trivial, como ocurre en la representación de los conjuntos de enteros mediante árboles binarios de búsqueda: en este caso, como en la implementación de estos árboles no se repiten las entradas (véase el apartado 17.4.2), se tiene asegurado que no habrá elementos duplicados, con lo cual el invariante de representación se formaliza simplemente como:

$$\{\text{Inv. } \textit{cierto}\}$$

Una vez conocida la forma del invariante de representación, es necesario asegurarse de que lo verifica la representación particular de todo objeto del tipo abstracto de datos. Esto es sencillo, partiendo del hecho de que todo objeto del tipo es obtenido a partir de las operaciones constructoras o mediante la aplicación de las operaciones de selección o de las propias del tipo. Por ello, basta con comprobar (de forma inductiva) que todo objeto generado por estas operaciones verifica el invariante, suponiendo que los eventuales argumentos del tipo también lo cumplen. Así, volviendo a la representación mediante listas, la operación `CrearConj` lo cumple trivialmente, ya que genera el conjunto vacío representado, valga la redundancia, por la lista vacía, y se verifica que

$$\forall i, j \in I, \text{ si } i \neq j, \text{ entonces } l_i \neq l_j$$

puesto que, en este caso,  $I = \emptyset$ .

En cuanto a la operación `AnnadirElemConj`, dado que el argumento recibido `conj` verifica el invariante, también lo verificará al completarse la ejecución, ya que en ella se comprueba explícitamente que `elem` no pertenece a `conj`.

El lector puede comprobar como ejercicio que el resto de las operaciones implementadas en el ejemplo también preservan el invariante.

El problema de la corrección en tipos abstractos de datos puede tratarse más formal y profundamente, pero ello escapa a las pretensiones de este libro. En las referencias bibliográficas se citan textos que profundizan en este tema.

Finalmente, se debe destacar que la modularidad introducida por los tipos abstractos de datos supone una gran ayuda en el estudio de la corrección de

programas grandes. Para éstos, dicho estudio resulta tan costoso que se ve muy reducido o incluso abandonado en la mayoría de los casos. Dada esta situación, es muy importante disponer de una biblioteca de tipos abstractos de datos correctos cuya verificación se ha realizado independientemente.

## 19.4 Resumen

En este apartado se enumeran algunas ideas que ayudan a fijar los conceptos expuestos en este capítulo, así como otros aspectos relacionados:

- Los tipos abstractos de datos permiten al programador concentrarse en las características o propiedades deseadas para dichos tipos (o en qué servicio deben prestar al programador), olvidándose, temporalmente, de cómo están implementados (o cómo se van a implementar). Por consiguiente, los tipos abstractos de datos pueden considerarse como cajas negras: el programador-usuario sólo ve su comportamiento y no sabe (ni le interesa saber) qué contienen.
- La definición del tipo abstracto de datos debe expresarse como una especificación no ambigua, que servirá como punto de partida para la derivación correcta de la implementación o para enfrentarla *a posteriori* a la implementación en un proceso de verificación formal.
- Para la aplicación de la abstracción de datos, es imprescindible que el lenguaje de programación escogido posibilite la implementación separada para poder llevar a cabo la ocultación de información y la “encapsulación”, esenciales en esta técnica.
- La abstracción de datos optimiza los niveles de independencia (en la línea de lo comentado en los capítulos 8 y 9) del código de la unidad del tipo abstracto de datos y de los programas de aplicación. Como consecuencia de ello, aumenta la facilidad de mantenimiento de los programas, al aislar en la unidad de implementación del tipo abstracto los cambios, correcciones y modificaciones relativos a la representación del tipo abstracto, evitando que se propaguen (innecesariamente) a los programas de aplicación.
- La abstracción de datos se puede considerar como precursora de la técnica de orientación a objetos (véase el apartado 5.1.3 del tomo I), en la que también se encapsulan datos y operaciones (con la diferencia de que se añaden los mecanismos de herencia y polimorfismo, entre otros).<sup>8</sup>

---

<sup>8</sup>A partir de la versión 5.5 de Turbo Pascal se permite una aplicación limitada de esta técnica, pero su explicación excede los propósitos de este libro.

## 19.5 Ejercicios

1. Construya una unidad de Turbo Pascal para los siguientes tipos abstractos de datos. El tipo abstracto debe incluir operaciones de construcción, modificación y consulta, así como operaciones propias del tipo si se considera conveniente:
  - (a) Listas (de enteros). Consúltese el apartado 17.1.
  - (b) Listas ordenadas (de enteros). Consúltese el apartado 17.1.
  - (c) Pilas (de enteros). Consúltese el apartado 17.2.
  - (d) Colas (de enteros). Consúltese el apartado 17.3.
  - (e) Árboles binarios (de enteros). Consúltese el apartado 17.4.
  - (f) Árboles binarios de búsqueda, conteniendo en sus nodos cadenas de caracteres. Consúltese el apartado 17.4.2.
2. Desarrolle una implementación alternativa para el tipo abstracto `tConj`, basando la representación en:
  - (a) Un vector de booleanos. Obviamente, en este caso se deben limitar los elementos posibles (supóngase por ejemplo, que los elementos están comprendidos entre 1 y 1000, ambos inclusive).
  - (b) Árboles binarios de búsqueda (véanse los apartados 19.2.3 y 17.4.2).

Compare el comportamiento que tienen las operaciones en ambos tipos abstractos de datos.
3. Implemente las operaciones `Union`, `Interseccion` y `Diferencia` (que calculan, obviamente, la unión, intersección y diferencia de dos conjuntos) como operaciones propias del tipo abstracto de datos `tConj`, representado mediante árboles binarios de búsqueda.
4. Programe un algoritmo que ordene una secuencia utilizando dos pilas. Para ello debe mover elementos de una pila a otra asegurándose de que los items menores llegan a la cima de una de las pilas y los mayores a la cima de la otra. Utilice el tipo abstracto pila desarrollado en el ejercicio 1c.
5. Escriba una función para sumar dos polinomios utilizando alguno de los tipos abstractos de datos propuestos en el ejercicio 1.
6. Escriba una unidad para la aritmética entera (sin límite en sus valores máximo y mínimo), utilizando alguno de los tipos abstractos de datos propuestos en el ejercicio 1.
7. Una *cola doble* es una estructura de datos consistente en una lista de elementos sobre la cual son posibles las siguientes operaciones:
  - *Meter*( $x,d$ ): inserta el elemento  $x$  en el extremo frontal de la cola doble  $d$ .
  - *Sacar*( $x,d$ ): elimina y devuelve el elemento que está al frente de la cola doble  $d$ .

- *Inyectar*( $x, d$ ): inserta el elemento  $x$  en el extremo posterior de la cola doble  $d$ .
- *Expulsar*( $x, d$ ): elimina y devuelve el elemento que está en el extremo posterior de la cola doble  $d$ .

Programar una unidad de Turbo Pascal para el tipo abstracto de datos *cola doble*.

## 19.6 Referencias bibliográficas

En este capítulo se introducen los tipos abstractos de datos, eludiendo las profundidades formales que hay tras ellos y evitando también dar catálogos exhaustivos de los tipos abstractos de datos más usuales. En [Mar86, Har89] se estudian los aspectos formales de los tipos abstractos de datos basados en especificaciones algebraicas. Además, ambos textos pueden considerarse como catálogos de los tipos abstractos de datos de más amplia utilización, incluyendo la especificación, implementaciones y aplicaciones comunes de cada uno de ellos. En [Pn93] se puede encontrar un estudio actual, riguroso y completo de los tipos abstractos de datos en nuestro idioma, junto con especificaciones de los más usuales. Igualmente, [HS90] es una lectura obligada en cuanto se aborda el estudio de tipos abstractos de datos.

Con un enfoque más aplicado, en [LG86, CMM87] se da un tratamiento completo de los tipos abstractos de datos bien adaptado a Pascal, ofreciendo propuestas de implementación en este lenguaje.

Por supuesto, los tipos abstractos de datos pueden ser implementados en lenguajes de programación diferentes de Pascal. De hecho, Pascal estándar no cuenta con mecanismos apropiados para manejarlos, ya que el concepto de tipo abstracto de datos es posterior a la creación de este lenguaje. El propio N. Wirth incluyó este concepto en su Módulo2 a través de los módulos, y las unidades de Turbo Pascal no son más que un remedo (incompleto, por cierto) de este mecanismo. Otros lenguajes con facilidades para el desarrollo de tipos abstractos de datos son Ada [Bar87], que dispone de paquetes, y C++ [Str84], haciendo uso de clases. Además, algunos lenguajes proporcionan al programador técnicas de compilación separada más adecuadas a los tipos abstractos y sus necesidades que las incompletas unidades de Turbo Pascal.

Finalmente, debemos indicar que el ejercicio 4 está tomado de [Mar86]. Los ejercicios 5, 6 y 7 están tomados de [Wei95].

## Capítulo 20

# Esquemas algorítmicos fundamentales

---

20.1 Algoritmos devoradores . . . . .	450
20.2 Divide y vencerás . . . . .	453
20.3 Programación dinámica . . . . .	455
20.4 Vuelta atrás . . . . .	462
20.5 Anexo: algoritmos probabilistas . . . . .	468
20.6 Ejercicios . . . . .	470
20.7 Referencias bibliográficas . . . . .	473

---

Cuando se estudian los problemas y algoritmos usualmente escogidos para mostrar los mecanismos de un lenguaje algorítmico (ya sea ejecutable o no), puede parecer que el desarrollo de algoritmos es un cajón de sastre en el que se encuentran algunas ideas de uso frecuente y cierta cantidad de soluciones *ad hoc*, basadas en trucos más o menos ingeniosos.

Sin embargo, la realidad no es así: muchos problemas se pueden resolver con algoritmos construidos en base a unos pocos modelos, con variantes de escasa importancia. En este capítulo se estudian algunos de esos esquemas algorítmicos fundamentales, su eficiencia y algunas de las técnicas más empleadas para mejorarla.

## 20.1 Algoritmos devoradores

La estrategia de estos algoritmos es básicamente iterativa, y consiste en una serie de etapas, en cada una de las cuales se consume una parte de los datos y se construye una parte de la solución, parando cuando se hayan consumido totalmente los datos. El nombre de este esquema es muy descriptivo: en cada fase (bocado) se consume una parte de los datos. Se intentará que la parte consumida sea lo mayor posible, bajo ciertas condiciones.

### 20.1.1 Descripción

Por ejemplo, la descomposición de un número  $n$  en primos puede describirse mediante un esquema devorador. Para facilitar la descripción, consideremos la descomposición de 600 expresada así:

$$600 = 2^3 \cdot 3^1 \cdot 5^2$$

En cada fase, se elimina un divisor de  $n$  cuantas veces sea posible: en la primera se elimina el 2, y se considera el correspondiente cociente, en la segunda el 3, etc. y se finaliza cuando el número no tiene divisores (excepto el 1).

El esquema general puede expresarse así:

```
procedure Resolver  $P$  ( $D$ : datos; var  $S$ : solución);
begin
  Generar la parte inicial de la solución  $S$ 
  (y las condiciones iniciales)
  while  $D$  sin procesar del todo do begin
    Extraer de  $D$  el máximo trozo posible  $T$ 
    Procesar  $T$  (reduciéndose  $D$ )
    Incorporar el procesado de  $T$  a la solución  $S$ 
  end {while}
end; {Resolver  $P$ }
```

La descomposición de un número en factores primos se puede implementar sencillamente siguiendo este esquema:

```
procedure Descomponer( $n$ : integer);
  {PreC.:  $n > 1$ }
  {Efecto: muestra en la pantalla la descomposición de  $n$ 
  en factores primos}
  var
     $d$ : integer;
```

```

begin
  d:= 2;
  while n > 1 do begin
    Dividir n por d cuantas veces (k) se pueda
    Escribir 'dk'
    d:= d + 1
  end {while}
end; {Descomponer}

```

### 20.1.2 Adecuación al problema

Otro ejemplo, quizá el más conocido de este esquema, es el de encontrar el “cambio de moneda” de manera óptima (en el sentido de usar el menor número de monedas posible) para una cantidad de dinero dada:<sup>1</sup> en cada fase, se considera una moneda de curso legal, de mayor a menor valor, y se cambia la mayor cantidad de dinero posible en monedas de ese valor.

Se advierte, sin embargo, que no siempre es apropiado un algoritmo devorador para resolver este problema. Por ejemplo, si el sistema de monedas fuera de

1, 7 y 9 pesetas

el cambio de 15 pesetas que ofrece este algoritmo consta de

7 monedas: 1 de 9 y 6 de 1

mientras que el cambio óptimo requiere tan sólo

3 monedas: 2 de 7 y 1 de 1

El algoritmo presentado para el cambio de moneda resultará correcto siempre que se considere un sistema monetario donde los valores de las monedas son cada uno múltiplo del anterior. Si no se da esta condición, es necesario recurrir a otros esquemas algorítmicos (véase el apartado 20.3.3).

La enseñanza que extraemos del ejemplo es la siguiente: generalmente el desarrollo de esta clase de algoritmos no presenta dificultades, pero es complicado asegurarse de que esta técnica es apropiada para el problema planteado. Por otra parte, hay que señalar que este esquema se aplica con frecuencia en la resolución de problemas a sabiendas de que la solución proporcionada no es la óptima, sino sólo relativamente buena. Esta elección se debe a la rapidez de la resolución, circunstancia que muchas veces hace que no merezca la pena buscar una solución mejor.

---

<sup>1</sup>Para simplificar, supondremos que se dispone de cuantas monedas se necesite de cada valor.

### 20.1.3 Otros problemas resueltos vorazmente

Existen gran cantidad de problemas cuya solución algorítmica responde a un esquema voraz. Entre ellos, los dos siguientes son ampliamente conocidos.

#### Problema de la mochila

Se desea llenar una mochila hasta un volumen máximo  $V$ , y para ello se dispone de  $n$  objetos, en cantidades limitadas  $v_1, \dots, v_n$  y cuyos valores por unidad de volumen son  $p_1, \dots, p_n$ , respectivamente. Puede seleccionarse de cada objeto una cantidad cualquiera  $c_i \in \mathbb{R}$  con tal de que  $c_i \leq v_i$ . El problema consiste en determinar las cantidades  $c_1, \dots, c_n$  que llenan la mochila maximizando el valor  $\sum_{i=1}^n v_i p_i$  total.

Este problema puede resolverse fácilmente seleccionando, sucesivamente, el objeto de mayor valor por unidad de volumen que quede y en la máxima cantidad posible hasta agotar el mismo. Este paso se repetirá hasta completar la mochila o agotar todos los objetos. Por lo tanto, se trata claramente de un esquema voraz.

#### Árbol de expansión mínimo (algoritmo de Prim)

El problema del árbol de expansión mínimo<sup>2</sup> se encuentra por ejemplo en la siguiente situación: consideremos un mapa de carreteras, con dos tipos de componentes: las ciudades (*nodos*) y las carreteras que las unen. Cada tramo de carreteras (*arco*) está señalado con su longitud.<sup>3</sup> Se desea implantar un tendido eléctrico siguiendo los trazos de las carreteras de manera que conecte todas las ciudades y que la longitud total sea mínima.

Una forma de lograrlo consiste en

*Empezar con el tramo de menor coste  
repetir*

*Seleccionar un nuevo tramo  
hasta que esté completa una red que conecte todas las ciudades*

donde cada nuevo tramo que se selecciona es el de menor longitud entre los no redundantes (es decir, que da acceso a una ciudad nueva).

Un razonamiento sencillo nos permite deducir que un árbol de expansión cualquiera (el mínimo en particular) para un mapa de  $n$  ciudades tiene  $n - 1$  tramos. Por lo tanto, es posible simplificar la condición de terminación que controla el algoritmo anterior.

<sup>2</sup>También llamado árbol de recubrimiento (del inglés, *spanning tree*.)

<sup>3</sup>Este modelo de datos se llama *grafo ponderado*.

## 20.2 Divide y vencerás

La idea básica de este esquema consiste en lo siguiente:

1. Dado un problema  $P$ , con datos  $D$ , si los datos permiten una solución directa, se ofrece ésta;
2. En caso contrario, se siguen las siguientes fases:
  - (a) Se dividen los datos  $D$  en varios conjuntos de datos más pequeños,  $D_i$ .
  - (b) Se resuelven los problemas  $P(D_i)$  parciales, sobre los conjuntos de datos  $D_i$ , recursivamente.
  - (c) Se combinan las soluciones parciales, resultando así la solución final.

El esquema general de este algoritmo puede expresarse así:

```

procedure Resolver  $P$  ( $D$ : datos; var  $S$ : solución);
begin
  if los datos  $D$  admiten un tratamiento directo then
    Resolver  $P(D)$  directamente
  else begin
    Repartir  $D$  en varios conjuntos de datos,  $D_1, \dots, D_k$ 
      (más cercanos al tratamiento directo).
    Resolver  $P(D_1), \dots, P(D_k)$ 
      (llamemos  $S_1, \dots, S_k$  a las soluciones obtenidas).
    Combinar  $S_1, \dots, S_k$ , generando la solución,  $S$ , de  $P(D)$ .
  end
end; {Resolver  $P$ }

```

Como ejemplo, veamos que el problema de ordenar un vector admite dos soluciones siguiendo este esquema: los algoritmos *Merge Sort* y *Quick Sort* (véanse los apartados 15.2.5 y 15.2.4, respectivamente).

Tal como se presentó en el apartado antes citado, el primer nivel de diseño de *Merge Sort* puede expresarse así:

```

si  $v$  es de tamaño 1 entonces
   $v$  ya está ordenado
si no
  Dividir  $v$  en dos subvectores  $A$  y  $B$ 
fin {si}
  Ordenar  $A$  y  $B$  usando Merge Sort
  Mezclar las ordenaciones de  $A$  y  $B$  para generar el vector ordenado.

```

Esta organización permite distinguir claramente las acciones componentes de los esquemas divide y vencerás comparándolo con el esquema general. El siguiente algoritmo, *Quick Sort*, resuelve el mismo problema siguiendo también una estrategia divide y vencerás:

```

si v es de tamaño 1 entonces
  v ya está ordenado
si no
  Dividir v en dos bloques A y B
  con todos los elementos de A menores que los de B
fin {si}
  Ordenar A y B usando Quick Sort
  Devolver v ya ordenado como concatenación
  de las ordenaciones de A y de B

```

Aunque ambos algoritmos siguen el mismo esquema, en el primero la mayor parte del trabajo se efectúa al combinar las subsoluciones, mientras que en el segundo la tarea principal es el reparto de los datos en subconjuntos. De hecho, lo normal es operar reestructurando el propio vector, de modo que no es preciso combinar las subsoluciones concatenando los vectores. Además, como se analizó en el apartado 18.3.2, el algoritmo *Merge Sort* resulta ser más eficiente en el peor caso, ya que su complejidad es del orden de  $n \log n$  frente a la complejidad cuadrática de *Quick Sort*, también en el peor caso.

### 20.2.1 Equilibrado de los subproblemas

Para que el esquema algorítmico divide y vencerás sea eficiente es necesario que el tamaño de los subproblemas obtenidos sea similar. Por ejemplo, en el caso del algoritmo *Quick Sort*, y en relación con estos tamaños, se podrían distinguir dos versiones:

- La presentada anteriormente, cuya complejidad en el caso medio es del orden de  $n \log n$ .
- La degenerada, en la que uno de los subproblemas es la lista unitaria es de tamaño 1, y en la que se tiene una complejidad cuadrática. De hecho, esta versión de *Quick Sort* es equivalente al algoritmo de ordenación por inserción (véase el apartado 15.2.2).

No obstante, hay problemas en los que el esquema divide y vencerás no ahorra coste, ni siquiera equilibrando los subproblemas. Por ejemplo, el cálculo de  $\prod_{i=a}^b i$ . En efecto, su versión recursiva con los subproblemas equilibrados

$$\prod_{i=a}^b i = \begin{cases} 1 & \text{si } b < a \\ \prod_{i=a}^m i * \prod_{i=m+1}^b i, & \text{para } m = (a + b) \text{ div } 2, \text{ e. o. c.} \end{cases}$$

tiene un coste proporcional a  $b - a$ , al igual que su versión degenerada, donde uno de los subproblemas es el trivial:

$$\prod_{i=a}^b i = \begin{cases} 1 & \text{si } b < a \\ a * \prod_{i=a+1}^b i & \text{e. o. c.} \end{cases}$$

siendo por tanto preferible su versión iterativa conocida.

## 20.3 Programación dinámica

### 20.3.1 Problemas de programación dinámica

Consideremos un problema en el que se desea obtener el valor óptimo de una función, y para ello se ha de completar una secuencia de etapas  $e_1, \dots, e_n$ . Supongamos que en la etapa  $e_i$  se puede escoger entre las opciones  $o_1, \dots, o_k$ , cada una de las cuales divide el problema en dos subproblemas más sencillos e independientes:

$$e_1, \dots, e_{i-1} \quad \text{y} \quad e_{i+1}, \dots, e_n$$

Entonces, bastaría con resolver los  $k$  pares de subproblemas (uno por cada opción para  $e_i$ ) y escoger el par que ofrece el mejor resultado (para la función por optimizar).

Supongamos que la solución óptima se obtiene si  $e_i = o$ :

$$e_1, \dots, e_{i-1}, e_i = o, e_{i+1}, \dots, e_n$$

Entonces, sus mitades anterior y posterior

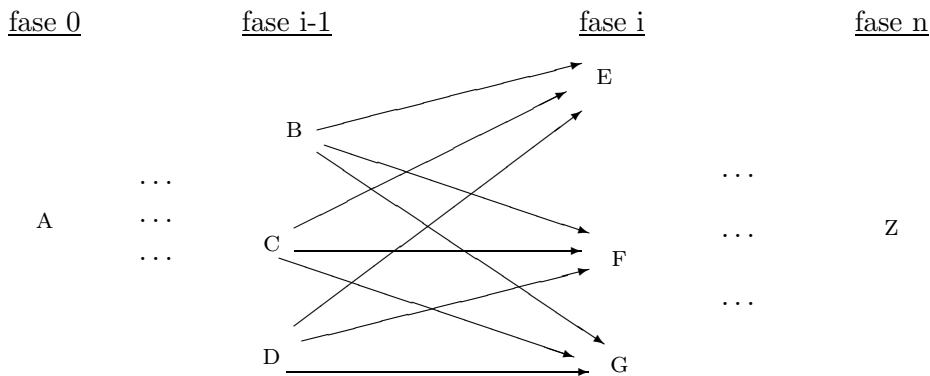
$$\begin{aligned} &e_1, \dots, e_{i-1}, o \\ &o, e_{i+1}, \dots, e_n \end{aligned}$$

deben ser las soluciones óptimas a los subproblemas parciales planteados al fijar la etapa  $i$ -ésima. Esta condición se conoce como el *principio de optimalidad de Bellman*. Por ejemplo, si para ir desde el punto  $A$  hasta el punto  $C$  por el camino más corto se pasa por el punto  $B$ , el camino mínimo desde  $A$  hasta  $C$  consiste en la concatenación del camino mínimo desde  $A$  hasta  $B$  y el camino mínimo desde  $B$  hasta  $C$ .

Por tanto, la resolución de estos problemas consiste en definir una secuencia de etapas. Por otra parte, al fijar una etapa cualquiera se divide el problema en dos problemas más sencillos (como en los algoritmos divide y vencerás), que deben ser independientes entre sí. El principal inconveniente es que la elección de una etapa requiere en principio tantear varios pares de subproblemas, con lo que se dispara el coste de la resolución de estos problemas.

**Ejemplo**

Consideremos un grafo como el de la figura, organizado por fases,<sup>4</sup> y se plantea el problema de averiguar el recorrido más corto entre A y Z, conociendo las longitudes de los arcos:



El problema propuesto consiste en formar una secuencia de etapas, en cada una de las cuales se opta por un arco que conduce a uno de los nodos de la siguiente, accesible desde nuestro nodo actual, determinado por las etapas anteriores.

Si en la fase  $i$ -ésima se puede escoger entre los nodos  $E$ ,  $F$  y  $G$ , el camino más corto entre  $A$  y  $Z$  es el más corto entre los tres siguientes

*ir desde A hasta E, e ir desde E hasta Z*  
*ir desde A hasta F, e ir desde F hasta Z*  
*ir desde A hasta G, e ir desde G hasta Z*

de manera que, una vez fijada una elección (por ejemplo  $E$ ), las subsoluciones *ir desde A hasta E* e *ir desde E hasta Z* componen la solución global. Esto es, se verifica el principio de optimalidad de Bellman.

En resumen, el mejor trayecto entre los puntos  $P$  y  $Q$  (en fases no consecutivas  $f_{inic}$  y  $f_{fin}$ ), se halla así:

*Elegir una etapa intermedia  $i$  (por ejemplo,  $(f_{inic} + f_{fin}) \text{ div } 2$ )*  
*(sean  $\{o_1, \dots, o_k\}$  los puntos en la etapa  $i$ -ésima)*  
*Elegir  $O \in \{o_1, \dots, o_k\}$  tal que el recorrido  $\text{MejorTray}(P, O)$*   
*junto con  $\text{MejorTray}(O, Q)$  sea mínimo*

La recursión termina cuando los nodos a enlazar están en etapas consecutivas, existiendo un único tramo de  $P$  a  $Q$  (en cuyo caso se elige éste) o ninguno (en cuyo

<sup>4</sup>Este tipo de grafos se llaman *polietápicos*.

caso no hay trayecto posible entre los nodos a enlazar, lo que puede consignarse, por ejemplo, indicando una distancia infinita entre esos puntos).

Estas ideas pueden expresarse como sigue, desarrollando a la vez un poco más el algoritmo<sup>5</sup> descrito antes:

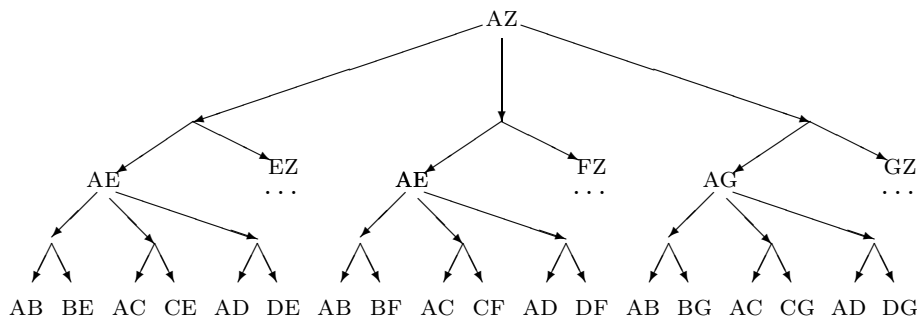
```

procedure MejorTray (P, Q: puntos; fP, fQ: nums.fase; var Tr: trayecto;
                    var Dist: distancia);
begin
  if consecutivos(P, Q) then
    Tr := [P, Q]
    Dist := longArco(P, Q), dato del problema
  else begin
    Sea fmed ← (fP + fQ) div 2,
    y sean {o1, ..., ok} los puntos de paso de la etapa i-ésima
    Se parte de distPQ ← ∞ y trPQ ← []
    para todo O ∈ {o1, ..., ok} hacer begin
      MejorTray(P, O, fP, fmed, TrPO, DistPO)
      MejorTray(O, Q, fmed, fQ, TrOQ, DistOQ)
      if DistPQ > DistPO + DistOQ then begin
        DistPQ := DistPO + DistOQ;
        TrPQ := TrPO concatenado con TrOQ
      end {if}
    end; {para todo}
    Tr := TrPQ
    Dist := DistPQ
  end {else}
end; {MejorTray}

```

### 20.3.2 Mejora de este esquema

El planteamiento anterior se caracteriza por su ineficiencia debida al gran número de llamadas recursivas. En el caso anterior, por ejemplo, el recorrido AZ puede descomponerse de múltiples formas, como se recoge en el siguiente árbol:



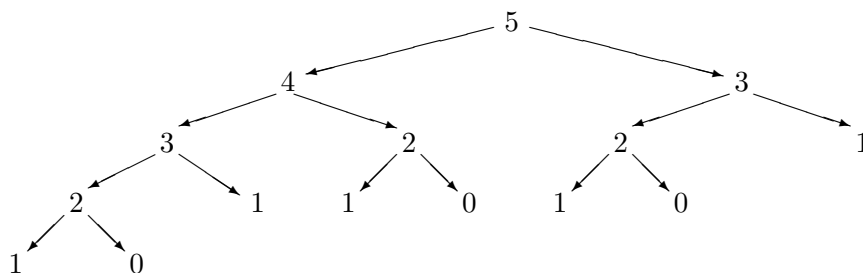
<sup>5</sup>Se advierte que este algoritmo es tremendamente ineficiente y, por tanto, nada recomendable; en el siguiente apartado se verán modos mejores de afrontar esta clase de problemas.

Sin embargo, se observa que un buen número de tramos se calcula repetidamente (AB, AC, AD, ...), por lo que se puede mejorar el planteamiento evitando los cálculos idénticos reiterados, concretamente mediante las técnicas de tabulación.

En este apartado estudiaremos en primer lugar en qué consisten esas técnicas, y después retomaremos el algoritmo anterior para ver cómo puede mejorarse su comportamiento mediante la tabulación.

### Tabulación de subprogramas recursivos

En ocasiones, una función  $f$  con varias llamadas recursivas genera, por diferentes vías, llamadas repetidas. La función de Fibonacci (véase el apartado 10.3.1) es un ejemplo clásico:



Al aplicarse al argumento 5, la llamada  $\text{Fib}(1)$  se dispara cinco veces.

Una solución para evitar la evaluación repetida consiste en dotar a la función de memoria de modo que recuerde los valores para los que se ha calculado junto con los resultados producidos. Así, cada cálculo requerido de la función se consultará en la tabla, extrayendo el resultado correspondiente si ya se hubiera efectuado o registrándolo en ella si fuera nuevo.

El esquema es bien sencillo: basta con establecer una tabla (la memoria de la función) global e incluir en ella las consultas y actualizaciones mencionadas:

```

function  $f'(x: \text{datos}; \text{var } T: \text{tablaGlobal}): \text{resultados};$ 
begin
  if  $x$  no está en la tabla  $T$  then begin
    Hallar las llamadas recursivas  $f'(x_i, T)$  y combinarlas, hallando
    el valor  $f'(x, T)$  requerido
    Incluir en la tabla  $T$  el argumento  $x$  y el resultado  $f'(x, T)$  obtenido
  end; {if}
   $f' := T[x]$ 
end; {f'}

```

Por ejemplo, la función de Fibonacci definida antes puede tabularse como sigue. En primer lugar, definimos la tabla:

```

type
  tDominio = 0..20;
  tTabla = array [tDominio] of record
           definido: boolean;
           resultado: integer
           end; {record}
var
  tablaFib: tTabla;

```

cuyo estado inicial debe incluirse en el programa principal:

```

tablaFib[0].definido:= True;
tablaFib[0].resultado:= 1;
tablaFib[1].definido:= True;
tablaFib[1].resultado:= 1;
for i:= 2 to 20 do
  tablaFib[i].definido:= False

```

Entonces, la función `Fib` resulta

```

function Fib(n: tDominio; var t: tTabla): integer;
begin
  if not t[n].definido then begin
    t[n].definido:= True;
    t[n].resultado:= Fib(n-1,t) + Fib(n-2,t)
  end; {if}
  Fib:= t[n].resultado
end; {Fib}

```

que se llama, por ejemplo, mediante `Fib(14, tablaFib)`.

- ☉☉ Un requisito indispensable para que una función se pueda tabular correctamente es que esté libre de (producir o depender de) efectos laterales, de manera que el valor asociado a cada argumento sea único, independiente del punto del programa en que se requiera o del momento en que se invoque. Esta observación es necesaria, ya que en esta técnica se utilizan variables globales; como se explicó en el apartado 8.5.4, un uso incontrolado de estas variables puede acarrear la pérdida de la corrección de nuestros programas.

### Tabulación de algoritmos de programación dinámica

Veamos ahora cómo las técnicas de tabulación descritas mejoran la eficiencia del algoritmo del grafo polietápico.

En efecto, se puede alterar el procedimiento descrito de modo que cada operación realizada se registre en una tabla y cada operación por realizar se consulte previamente en esa tabla por si ya se hubiera calculado. En el ejemplo anterior, se podría anotar junto a los puntos del propio mapa el mejor trayecto que conduce a ellos cada vez que se calcule. De este modo, no será necesario repetir esos cálculos cuando vuelvan a requerirse. Los cambios descritos son mínimos:

- Se crea una tabla global (donde registraremos los mejores trayectos y las correspondientes distancias), y se rellena inicialmente con los trayectos y distancias de los puntos consecutivos (que son los datos del problema).
- Entonces en vez de comprobar si un camino es directo, se hará lo siguiente:

*si consecutivos( $P, Q$ ) entonces  
el mejor trayecto es la secuencia  $[P, Q]$  y su longitud es  $longArco(P, Q)$*

comprobaremos si está ya tabulado:

*si tabulado( $P, Q$ ) entonces  
extraer de la tabla el trayecto,  $trPQ$ , y su longitud,  $distPQ$   
...*

- Por último, al finalizar un cálculo, se debe añadir a la tabla la acción siguiente:

*Registrar en la tabla el trayecto ( $TrPQ$ ) y la distancia ( $distPQ$ ) hallados*

Esta solución es bastante satisfactoria. Sin embargo, este tipo de problemas permite, en general, establecer un orden entre los subproblemas requeridos y, por consiguiente, entre los cálculos necesarios. En el ejemplo anterior, los cálculos pueden hacerse por fases: una vez hallado (y anotado) el mejor trayecto que lleva a cada uno de los puntos de una fase  $i$ , es sencillo y rápido hallar (y anotar) los mejores caminos hasta los puntos de la fase siguiente,  $i + 1$ . Más aún, como ya no se necesita la información correspondiente a la fase  $i$ -ésima, es posible prescindir de ella, con el consiguiente ahorro de memoria.

### 20.3.3 Formulación de problemas de programación dinámica

El problema del cambio de moneda (véase el apartado 20.1.2) se puede formular también de modo que los valores de las monedas no guardan relación alguna: se dispone de una colección ilimitada de monedas, de valores  $v_1, \dots, v_n$

enteros cualesquiera, y se trata ahora de dar el cambio óptimo para la cantidad  $C$ , entendiendo por óptimo el que requiere el menor número total de monedas.<sup>6</sup>

Una formulación siguiendo el esquema de programación dinámica es la siguiente: si hay alguna moneda de valor igual a la cantidad total, el cambio óptimo es precisamente con *una* moneda; de lo contrario, el primer paso consiste en escoger una moneda entre las de valor menor que la cantidad dada y esa elección debe minimizar el cambio de la cantidad restante:

```

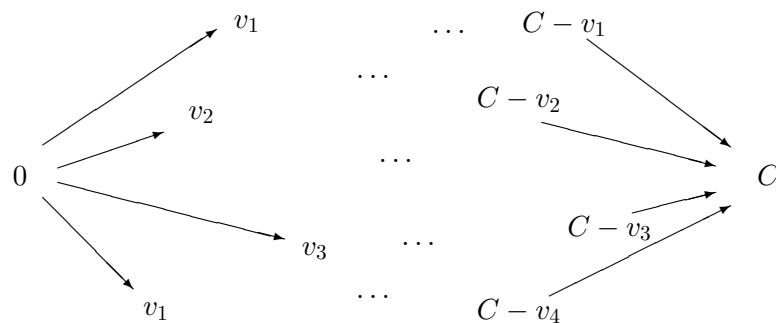
function NumMon( $C$ : integer): integer;
begin
  if alguna de las monedas  $v_1, \dots, v_n$  es igual a  $C$  then
    NumMon := 1
  else
    NumMon := 1 +  $\min_{v_i < C}$  (NumMon( $C - v_i$ ))
end; {NumMon}

```

La tabulación es sencilla: para obtener el cambio óptimo de una cantidad  $C$ , será necesario consultar (posiblemente varias veces) los cambios de cantidades menores. Por lo tanto, es fácil establecer un orden entre los subproblemas, tabulando los cambios correspondientes a las cantidades  $1, 2, \dots, C$ , ascendentemente.

Este ejemplo nos permite extraer dos consecuencias:

- En algunos problemas no se enuncian explícitamente las etapas que deben superarse desde el estado inicial a una solución; más aún, a veces el problema no consta de un número de fases conocido de antemano, sino que éstas deben ser “calculadas” por el algoritmo propuesto, como se muestra gráficamente en la siguiente figura:



<sup>6</sup>Para simplificar, supondremos que una de las monedas tiene el valor unidad, de manera que siempre es posible completar el cambio.

- El ejemplo presentado obedece a un planteamiento matemático con aplicaciones en campos muy diversos. Concretamente, si llamamos  $k_1, \dots, k_n$  al número de monedas de cada valor  $v_1, \dots, v_n$  respectivamente, el esquema anterior resuelve el siguiente problema de optimización:

$$\text{mín} \sum_{i=1}^n k_i \quad \text{sujeto a que} \quad \sum_{i=1}^n k_i v_i = C$$

que es muy similar al planteamiento general numérico de esta clase de problemas: hallar enteros no negativos  $x_1, \dots, x_n$  que minimicen la función  $g(k_1, \dots, k_n)$  definida así:

$$g(k_1, \dots, k_n) = \sum_{i=1}^n f_i(k_i)$$

y de manera que se mantenga  $\sum_{i=1}^n k_i v_i \leq C$ .

### Nota final

Es importante subrayar que un problema planteado no siempre admite una descomposición en subproblemas independientes: en otras palabras, antes de expresar una solución basada en tal descomposición, debe comprobarse que se verifica el principio de optimalidad.

En general, la resolución de esta clase de problemas resulta inviable sin la tabulación. Además, frecuentemente resulta sencillo fijar un orden en los cálculos con lo que, a veces, es posible diseñar un algoritmo iterativo de resolución.

## 20.4 Vuelta atrás

Consideremos el problema de completar un rompecabezas. En un momento dado, se han colocado unas cuantas piezas, y se tantea la colocación de una nueva pieza. Por lo general, será posible continuar de diversos modos, y cada uno de ellos podrá ofrecer a su vez diversas posibilidades, multiplicándose así las posibilidades de tanteo. La búsqueda de soluciones es comparable al recorrido de un árbol, por lo que se le llama *árbol de búsqueda* (véase el apartado 17.5) o también *espacio de búsqueda*.

Por otra parte, el tanteo de soluciones supone muchas veces abandonar una vía muerta cuando se descubre que no conduce a la solución, deshaciendo algunos movimientos y regresando por otras ramas del árbol de búsqueda a una posición anterior. De ahí viene la denominación de esta clase de algoritmos: *vuelta atrás* o búsqueda con *retroceso*.<sup>7</sup>

<sup>7</sup>En inglés, *backtrack* o *backtracking*.

Un ejemplo conocido de problema que se adapta bien a este esquema es el de situar ocho damas en un tablero de ajedrez de manera que ninguna esté amenazada por otra. La siguiente figura muestra una solución:

			*				
							*
				*			
		*					
*							
						*	
	*						
					*		

Como cada dama debe estar en una fila distinta, es posible representar una solución como un vector  $(D_1, \dots, D_8)$ , donde cada componente  $D_i$  es un entero de  $\{1, \dots, 8\}$  que representa la columna en que se encuentra la dama  $i$ -ésima.

La figura 20.1 muestra un fragmento del árbol de búsqueda en una fase intermedia en que se está construyendo la solución mostrada. Se ha usado el símbolo  $\emptyset$  para representar el fallo en un intento de ampliar la solución por una rama, y obedece a la imposibilidad de situar una dama en ciertas casillas por estar a tiro de las situadas con anterioridad.

Las variantes más conocidas a este tipo de problemas son las siguientes:

1. Generar *todas* las soluciones posibles.

El esquema de búsqueda de *todas* las soluciones, desde la fase  $i$ -ésima, se puede resumir como sigue:

```

procedure  $P$  ( $i$ : numero de fase;  $S$ : solucParcial);
  {Efecto: genera todas las soluciones, desde la fase  $i$ -ésima, a partir
  de la solución parcial  $S$ }
begin
  para todo  $p_i \in \{\text{posibles pasos válidos en esta fase}\}$  hacer begin
    Añadir  $p_i$  a  $S$  (Sea  $S'$  la solución  $S$ , dando un paso más  $p_i$ )
    if  $p_i$  completa una solución then
      Registrar la solución  $S'$ 
    else
      Resolver  $P(i+1, S')$ 
    end {para todo}
end; { $P$ }
  
```

En particular, el procedimiento que genera todas las soluciones en el problema de las ocho damas es el siguiente:

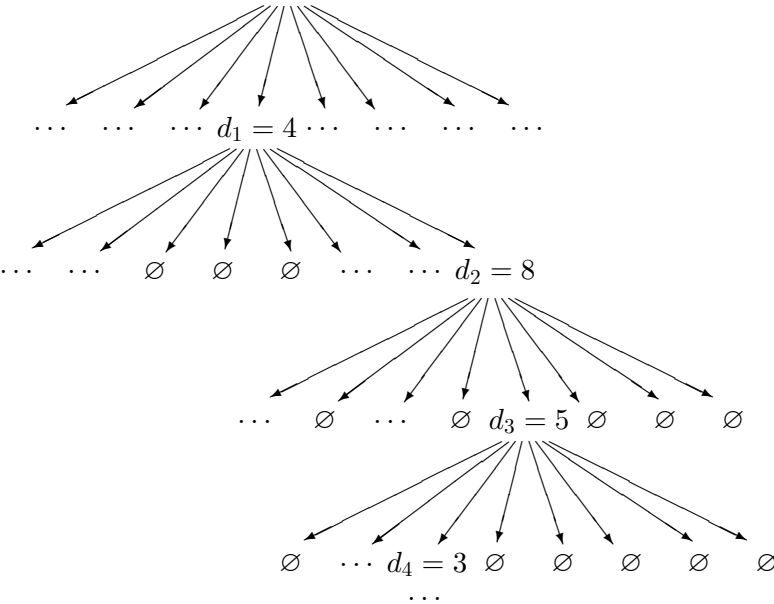


Figura 20.1.

```

type
  tDominio = 1..8;
  ...
procedure OchoDamas (i: numero de dama; S: solucParcial);
  var
    col: tDominio;
begin
  for col:= 1 to 8 do
    if puede situarse la dama i-ésima en la columna col sin
      estar a tiro de las anteriores (1, ..., i-1) then begin
      Situar la dama i-ésima en la columna col
      (con lo que se tiene la situación S')
      if i = 8 then
        Registrar esta solución
      else
        Resolver OchoDamas (i+1, S')
      end {if}
    end; {OchoDamas}

```

2. Tantear *cuántas* soluciones existen.

El esquema es una variante del anterior, estableciendo un contador a cero antes de iniciar la búsqueda de soluciones y cambiando la acción de *registrar una solución* por incrementar el contador.

3. Generar *una* solución, si existe, o indicar lo contrario.

En este caso, la búsqueda termina cuando se halla una solución o se agotan las vías posibles.

Un modo fácil de conseguirlo es modificar el primer esquema añadiendo un parámetro (booleano) para indicar cuándo se ha encontrado una solución y parando la búsqueda en caso de éxito. En el caso del problema de las ocho damas quedaría así:

```

procedure OchoDamas(i: numero de dama; S: solucParcial;
  var halladaSol: boolean);
  var
    col: tDominio;
begin
  halladaSol:= False;
  col:= 0;
  repeat
    col:= col + 1;
    if puede situarse la dama i-ésima en la columna col sin
      estar a tiro de las anteriores (1, ..., i-1) then begin

```

```

    Situar la dama  $i$ -ésima en la columna col
    (con lo que se tiene la situación  $S'$ )
  if i = 8 then begin
    Registrar esta solución;
    halladaSol:= True
  end {then}
  else
    Resolver OchoDamas ( $i+1$ ,  $S'$ , halladaSol)
  end {then}
until halladaSol or (col = 8)
end; {OchoDamas}

```

En resumen, la técnica de vuelta atrás ofrece un método para resolver problemas tratando de completar una o varias soluciones por etapas  $e_1, e_2, \dots, e_n$  donde cada  $e_i$  debe satisfacer determinadas condiciones. En cada paso se trata de extender una solución parcial de todos los modos posibles, y si ninguno resulta satisfactorio se produce la vuelta atrás hasta el último punto en que aún quedaban alternativas sin explorar. El proceso de construcción de una solución es semejante al recorrido de un árbol: cada decisión ramifica el árbol y conduce a posiciones más definidas de la solución o a una situación de fallo.

### 20.4.1 Mejora del esquema de vuelta atrás

El árbol de búsqueda completo asociado a los esquemas de vuelta atrás crece frecuentemente de forma exponencial conforme aumenta su altura. Por ello, cobra gran importancia podar algunas de sus ramas siempre que sea posible. Los métodos más conocidos para lograrlo son los siguientes:

#### Exclusión previa

Frecuentemente, un análisis detallado del problema permite observar que ciertas subsoluciones resultarán infructuosas más o menos pronto. Estos casos permiten organizar el algoritmo para que ignore la búsqueda en tales situaciones.

El problema de las ocho damas es un claro ejemplo de ello: es obvio que cada dama deberá estar en una fila distinta, lo que facilita la organización de las fases y limita la búsqueda de soluciones.

#### Fusión de ramas

Cuando la búsqueda a través de distintas ramas lleve a resultados equivalentes, bastará con limitar la búsqueda a una de esas ramas.

En el problema de las ocho damas, por ejemplo, se puede limitar el recorrido de la primera dama a los cuatro primeros escaques, ya que las soluciones correspondientes a los otros cuatro son equivalentes y deducibles de los primeros por simetría.

### Reordenación de la búsqueda

Cuando lo que se busca no son todas las soluciones, sino sólo una, es interesante reorganizar las ramas, situando en primer lugar las que llevan a un subárbol de menor tamaño o aquéllas que ofrecen mayores expectativas de éxito.

### Ramificación y poda

Con frecuencia, el método de búsqueda con retroceso resulta impracticable debido al elevado número de combinaciones de prueba que aparecen. Cuando lo que se busca es precisamente *una* solución *óptima*, es posible reducir el árbol de búsqueda: suprimiendo aquellas fases que, con certeza, avanzan hacia soluciones no óptimas, por lo que se puede abandonar la búsqueda por esas vías (ésta es la idea que refleja el nombre<sup>8</sup> de esta técnica). Como consecuencia de esta reducción del número de soluciones por inspeccionar, se puede producir una mejora sustancial en la eficiencia de los algoritmos de búsqueda con retroceso convirtiéndolos en viables.

Por ejemplo, consideremos nuevamente el problema de las ocho damas, pero con la siguiente modificación: los escaques del tablero están marcados con enteros positivos, y se trata de hallar la posición de las damas que, sin amenazarse entre sí, cubre casillas cuya suma sea mínima. Ahora, si suponemos que se ha encontrado ya una solución (con un valor suma  $S$ ), en el proceso de búsqueda de las posiciones siguientes se puede “podar” en cuanto una fase intermedia rebasa la cantidad  $S$ , ya que se tiene la seguridad de que la solución obtenida al extender esa solución no mejorará en ningún caso la que ya tenemos.

Este ejemplo nos permite resaltar la conveniencia de reordenar la búsqueda: es obvio que, cuanto menor sea el valor ofrecido por una solución provisional, mayores posibilidades se tendrá de efectuar podas en las primeras fases. En nuestro ejemplo, las posibilidades de reorganizar la búsqueda son dos: establecer un orden entre las fases y dentro de cada fase.

---

<sup>8</sup>En inglés, *branch and bound*.

## 20.5 Anexo: algoritmos probabilistas

Por desgracia, los esquemas anteriores no siempre proporcionan soluciones a cualquier problema planteado; además sucede a veces que, aun existiendo una solución algorítmica a un problema, el tiempo requerido para encontrar una solución es tal que los algoritmos resultan inservibles. Una posibilidad consiste en conformarse con una solución aproximada, o bien con una solución que resulte válida sólo casi siempre. Ambas posibilidades se basan en la aplicación de métodos estadísticos.

En realidad, este planteamiento no constituye un esquema algorítmico, sino más bien una trampa consistente en alterar el enunciado de un problema intratable computacionalmente para ofrecer una respuesta aceptable en un tiempo razonable.

En este anexo incluimos sólo un par de ejemplos de los dos criterios más frecuentemente adoptados en esta clase de algoritmos:

- El primero consiste en buscar una *solución aproximada*, lo que es posible en bastantes problemas numéricos, sabiendo además el intervalo de confianza de la solución encontrada. La precisión de la solución depende frecuentemente del tiempo que se invierta en el proceso de cálculo, por lo que puede lograrse la precisión que se desee a costa de una mayor inversión del tiempo de proceso.
- El segundo criterio consiste en buscar una respuesta que constituya una solución del problema con cierta probabilidad. La justificación es que cuando se trata de tomar una decisión entre, por ejemplo, dos opciones posibles, no cabe pensar en soluciones aproximadas, aunque sí en *soluciones probablemente acertadas*. También aquí es posible invertir una cantidad de tiempo mayor para aumentar, esta vez, la probabilidad de acierto.

Aunque existen clasificaciones sistemáticas más detalladas de esta clase de algoritmos, en este apartado sólo pretendemos dar una idea del principio en que se basan, y ello mediante dos ejemplos ampliamente conocidos.

### 20.5.1 Búsqueda de una solución aproximada

Deseamos hallar el área de un círculo de radio 1 (naturalmente, sin usar la fórmula). Un modo poco convencional de lograrlo consiste en efectuar  $n$  lanzamientos al azar al cuadrado  $[-1, 1] \times [-1, 1]$  y concluir con que la proporción de ellos que caiga dentro del círculo será proporcional a este área. En otras palabras, el procedimiento consiste en generar  $n$  puntos  $(p_x, p_y)$  aleatoriamente, extrayendo ambas coordenadas del intervalo  $[-1, 1]$  uniformemente:

```

function NumAciertos(numPuntos: integer): integer;
  {Dev. el número de lanzamientos que caen dentro del círculo}
  var
    i, total: integer;
begin
  total:= 0;
  for i:= 1 to numPuntos do begin
    GenerarPunto( $p_x$ ,  $p_y$ ); {del intervalo  $[-1,1]^2$ , uniformemente}
    if ( $p_x, p_y$ )  $\in$  círculo de radio 1 then
      total:= total + 1
    end; {for}
  NumAciertos:= total
end; {NumAciertos}

```

Según la ley de los grandes números, si `númPuntos` es muy grande la proporción de aciertos tenderá a la proporción del área del círculo:

$$\frac{\text{numAciertos}}{\text{numEnsayos}} \rightarrow \frac{\pi}{4}$$

De hecho, este algoritmo suele presentarse como un modo de estimar  $\pi$ :

$$\pi \simeq 4 * \frac{\text{numAciertos}}{\text{numEnsayos}}$$

### 20.5.2 Búsqueda de una solución probablemente correcta

Se desea averiguar si un entero  $n$  es o no primo. Supuesto  $n$  impar,<sup>9</sup> el método convencional consiste en tantear los divisores  $3, 5, \dots, \text{Trunc}(\sqrt{n})$ . Los  $\frac{\text{Trunc}(\sqrt{n})}{2} - 1$  tanteos que requiere el peor caso, hacen este método inviable cuando se trata de un  $n$  grande.

Una alternativa es la siguiente: supongamos que se tiene un test<sup>10</sup>

```

function Test(n: integer): boolean;

```

que funciona aleatoriamente como sigue: aplicado a un número primo, resulta ser siempre `True`; por el contrario, aplicado a un número compuesto lo descubre (resultando ser `False`) con probabilidad  $p$  (por ejemplo,  $\frac{1}{2}$ ). Por lo tanto, existe peligro de error sólo cuando su resultado es `True`, y ese resultado es erróneo con una probabilidad  $q = 1 - p$ .

<sup>9</sup>De lo contrario el problema ya está resuelto.

<sup>10</sup>La naturaleza del test no importa en este momento. Bástenos con saber que, efectivamente, existen pruebas eficientes de esta clase.

Entonces, la aplicación repetida de esa función

```

function RepTest(n, k: integer): boolean;
  var
    i: integer;
    pr: boolean;
begin
  pr:= False;
  i:= 0;
  repeat
    i:= i + 1;
    pr:= Test(n)
  until pr or (i = k);
  RepTest:= pr
end; {RepTest}

```

se comporta así: aplicada a un primo, resulta ser siempre **True**, y aplicada a un número compuesto lo descubre con una probabilidad  $1 - q^k$ . La probabilidad de error cuando el resultado es **True**,  $q^k$ , puede disminuirse tanto como se quiera a costa de aumentar el número de aplicaciones de la función **test**.

## 20.6 Ejercicios

1. Desarrollar completamente el programa para descomponer un entero positivo en sus factores primos descrito en el apartado 20.1.1, escribiendo finalmente la solución en la siguiente forma:

$$\begin{array}{r|l}
 600 & 2 \\
 300 & 2 \\
 150 & 2 \\
 75 & 3 \\
 25 & 5 \\
 5 & 5 \\
 1 & 
 \end{array}$$

2. Escribir el algoritmo de cambio de moneda descrito en apartado 20.1.2 ajustándose al esquema devorador general.
3. Desarrolle completamente un procedimiento devorador para el problema de la mochila descrito en el apartado 20.1.3.
4. Desarrolle completamente un procedimiento devorador para el algoritmo de Prim del árbol de recubrimiento mínimo. Téngase en cuenta la observación hecha al final del apartado 20.1.3 que nos permite simplificar la condición de terminación del bucle.
5. Se tienen dos listas  $A$  y  $B$  ordenadas. Escribir un algoritmo devorador que las mezcle produciendo una lista  $C$  ordenada.

6. Supongamos que el vector  $V$  consta de dos trozos  $V_{1,\dots,k}$  y  $V_{k+1,\dots,n}$  ordenados. Escribir un algoritmo devorador que los mezcle, produciendo un vector  $W_{1,\dots,n}$  ordenado.
7. Se desea hallar el máximo valor de un vector  $V$  de  $n$  enteros. Siguiendo una estrategia divide y vencerás,<sup>11</sup> una solución consiste en lo siguiente: si el vector tiene una sola componente, ésa es la solución; de lo contrario, se procede como sigue:
- se divide el vector  $V_{1,\dots,n}$  en dos trozos:  $A = V_{1,\dots,k}$  y  $B = V_{k+1,\dots,n}$
  - se hallan los máximos para  $A$  y  $B$  respectivamente
  - se combinan las soluciones parciales  $M_A$  y  $M_B$  obtenidas, resultando que la solución final es el máximo de entre  $M_A$  y  $M_B$ .

Desarrolle completamente el algoritmo.

8. (a) Desarrolle otro algoritmo que halle el máximo valor de un vector, esta vez siguiendo un esquema recursivo simple en el que se compare el primer elemento del vector con el valor máximo del resto del vector.
- (b) Compare la complejidad de este algoritmo con el del ejercicio anterior.
9. Sea  $A$  una matriz cuadrada de  $n \times n$ . Si  $n$  es par,  $A$  puede descomponerse en cuatro submatrices cuadradas, así:

$$A = \left( \begin{array}{c|c} A' & B' \\ \hline C' & D' \end{array} \right)$$

y su determinante puede hallarse así:

$$|A| = |A'| * |D'| - |B'| * |C'|.$$

Escriba un algoritmo divide y vencerás para hallar el determinante de una matriz de dimensión  $n$ , siendo  $n$  una potencia de 2.

10. (a) Desarrolle un programa que halle  $\text{Fib}(15)$  según la definición recursiva usual de la función de Fibonacci, y modifíquelo para que cuente el número de llamadas  $\text{Fib}(0)$ ,  $\text{Fib}(1)$ ,  $\dots$  que se efectúan.
- (b) Implemente una variante de la función del apartado (a), en la que se le dote de memoria.
11. Defina en Pascal la función *ofuscación* de Dijkstra,

$$\begin{aligned} \text{Ofusc}(0) &= 0 \\ \text{Ofusc}(1) &= 1 \\ \text{Ofusc}(n) &= \left\{ \begin{array}{ll} \text{Ofusc}(\frac{n}{2}), & \text{si } n \text{ es par} \\ \text{Ofusc}(n+1) + \text{Ofusc}(n-1), & \text{si } n \text{ es impar} \end{array} \right\}, \forall n \geq 2 \end{aligned}$$

dotándola de memoria, y averigüe en qué medida ha mejorado su eficiencia, siguiendo los pasos del ejercicio anterior.

<sup>11</sup>Aunque el algoritmo descrito resuelve correctamente el problema planteado, debe advertirse que ésta no es la manera más apropiada para afrontar este problema (véase el apartado 20.2.1).

12. Redefina la función que halla recursivamente los números combinatorios  $\binom{m}{n}$ , para  $m \in \{0, \dots, 10\}$ ,  $n \in \{0, \dots, m\}$ , dotándola de memoria.
13. Desarrolle completamente el algoritmo que halla el camino mínimo en un grafo polietápico incorporando la tabulación descrita en el apartado 20.3.2
14. (a) Desarrolle completamente el problema del cambio de moneda descrito en el apartado 20.3.3 para el juego de monedas de 1, 8 y 9 pesetas. Dar dos versiones del mismo: una sin tabular y otra mediante tabulación.  
(b) Inserte los contadores necesarios para tantear el número de llamadas recursivas requeridas para descomponer 70 ptas. de manera óptima en las dos versiones anteriores.
15. Desarrolle un algoritmo que dé la lista de todas las descomposiciones posibles de  $N$  (número natural que es un dato de entrada) en sumas de doses, treses y cincos.
16. Supongamos que las soluciones parciales al problema de las ocho damas se concretan en dos variables: un indicador del número de damas situadas correctamente (con un valor entre 0 y 8), y un **array** [1..8] of 1..8 cuyas primeras casillas registran las posiciones de las damas ya situadas en una subsolución.
  - (a) Escriba una función que indique si dos damas están a tiro.
  - (b) Escriba una función que, conocida una subsolución y una posición (de 1 a 8) candidata para situar la dama siguiente, indique si es ello posible sin amenazar las anteriores.
  - (c) Desarrolle completamente el algoritmo descrito en el apartado 1, generalizado al problema de las  $n$  damas, con tres variantes:
    - i. La generación de *todas* las posiciones válidas.
    - ii. Averiguar *cuántas* soluciones válidas hay.
    - iii. Buscar *una* posición válida, si existe, o un mensaje de advertencia en caso contrario.
  - (d) Aplique las técnicas de poda expuestas en el apartado 20.4.1 para situar ocho damas, libres de amenazas, en un tablero de ocho por ocho, ocupando casillas con una suma máxima, sabiendo que la casilla de la fila  $i$ , columna  $j$ , vale  $10^{8-i} + j$  puntos.
17. Utilizando las técnicas explicadas en el apartado 20.5.1, hallar aproximadamente el área limitada por la función  $f(x) = \log(x)$ , el eje de abscisas y las rectas  $x = 1$  y  $x = 5$ . Compárese el resultado obtenido con el valor exacto de  $\int_1^5 \log(x) dx$ .



### 18. Experimento de Buffon

Supongamos un entarimado de parquet formado por tablillas paralelas de 10 cm de ancho.

- (a) Hallar, mediante simulación, la probabilidad de que, al lanzar una aguja de 5 cm de largo, caiga sobre una línea entre dos baldosas.
- (b) Sabiendo que esa probabilidad es  $\frac{1}{\pi}$ , estimar  $\pi$  de ese modo.

## 20.7 Referencias bibliográficas

Se introduce sucintamente en este capítulo un tema que requiere, para ser tratado cabalmente, un libro completo, por lo que los interesados en el tema necesitarán ampliar esta introducción. Afortunadamente, existe una enorme cantidad de referencias de calidad dedicada al estudio del diseño de los esquemas algorítmicos fundamentales, por lo que nos ceñimos a las fuentes en castellano a las que más debemos: los apuntes manuscritos de D. de Frutos [Fru84], los clásicos [AHU88], [HS90] y [BB97] y el más reciente [GGSV93].

Los ejemplos escogidos aquí para explicar cada uno de los esquemas algorítmicos no son, ciertamente, originales: al contrario, su utilización es frecuentísima con este fin, y ello se debe a que requieren un número pequeño de ideas para explicar, casi por sí mismos, los importantes conceptos que se han expuesto en este capítulo. En [Bie93] se abunda en esta idea, llegando a proponer el uso de estos ejemplos convertidos en pequeños rompecabezas cuyo manejo de forma natural consiste en cada uno de los esquemas algorítmicos fundamentales.

Los esquemas devorador y de programación dinámica se presentan frecuentemente usando como ejemplo el problema *de la mochila 0-1* (o mochila entera), alternativo al del cambio de moneda. Ambos pueden considerarse formulaciones del mismo argumento con distinta ambientación, admiten interesantes variaciones y han suscitado gran cantidad de observaciones. Además de las referencias citadas con carácter general, en [Wri75] pueden

leerse diferentes soluciones a los mismos; en [CK76] se estudian además las condiciones generales en que estos problemas admiten soluciones voraces.

Desde que apareció el problema de las  $n$  reinas (hace más de cien años), se han estudiado muchas de sus propiedades para tableros de distintos tamaños. En [BR75] puede encontrarse una visión panorámica del mismo y de otro problema clásico (la teselación con *pentominós*), ejemplificando con ambos el uso y mejora de las técnicas de vuelta atrás.

La referencia [Dew85a] es una bonita introducción a los algoritmos probabilistas. [BB90] ofrece una visión más detallada y profunda. El test para descubrir números compuestos descrito en 20.5.2 se encuentra en casi cualquier referencia sobre algoritmos probabilistas. Entre la bibliografía seleccionada, [BB97] y [Wil89] ofrecen claras explicaciones de los mismos.