Algorítmica y programación para ingenieros

Aula Teòrica 18

Algorítmica y programación para ingenieros

M. Isabel Gallego Fernández
Manuel Medina Llinàs

EDICIONS UPC

Primera edición: setiembre de 1993

Diseño de la cubierta: Manuel Andreu

C. Jordi Girona Salgado, 31, 08034 Barcelona

Producción: Servei de Publicacions de la UPC
y CPDA (Centre de Publicacions d'Abast)
Avda. Diagonal, 647, ETSEIB, 08028 Barcelona

Depósito legal: B-25.444-93
ISBN 84-7653-326-8

A:
Toni, Pau, Marc,
Anna y Marcel

Agradecimientos

Patricia Borensztejn, por la iniciativa con la que nos ha impulsado a la realización de esta obra, y su colaboración inestimable en las versiones previas de la misma.

Jordi Buch y **Enric Peig**, por sus colaboración en la verificación de los programas listados en el libro.

Miguel Valero y **Leandro Navarro**, por sus comentarios sobre la adecuación de la obra a los estudios de ingeniería de telecomunicación.

Pere Botella, a quien nunca agradeceremos suficientemente su esfuerzo desinteresado por fomentar una enseñanza de la programación de la máxima calidad en nuestra universidad, y en particular un par de contribuciones con las que indudablemente ha mejorado la calidad de este libro:

- los consejos y textos sobre la didáctica de la algorítmica y la programación, con los que nos regaló al principio de nuestra andadura por los caminos de la enseñanza de la algorítmica para los estudiantes de telecomunicación, antes incluso de la creación de la facultad de informática de Barcelona, de la que actualmente es decano, y
- el prologo con que tan amablemente ha accedido a participar en la confección de este libro.

Prólogo

Los autores de este texto, Manuel Medina e Isabel Gallego, antiguos amigos y compañeros de docencia en la UPC, me piden que les escriba este prólogo. No puedo dejar de sentirme honrado con tal petición, que atiendo encantado, y que les agradezco, y tampoco puedo evitar un cierto sonrojo tras la lectura de los agradecimientos: creo que, al menos en mi caso, no hay para tanto. Pero el sonrojo se mezcla también con un cierto grado de satisfacción, que espero se me permita.

No fuí el único agente encargado de introducir en la UPC determinada forma de hacer las cosas en la enseñanza de la programación, pero no por ello eludo la responsabilidad que me corresponde. La lectura de los textos del profesor Dijkstra (hace ya mas de 20 años!) y, algo más tarde, la aparición del lenguaje Pascal (junto a los libros del profesor Wirth) nos llevaron a algunos a enfocar esa enseñanza hacia el diseño correcto de algoritmos y a la formación de buenos hábitos en su escritura, dejando los aspectos sintácticos de la codificación en su justo lugar: cómo un vehículo (que hay que usar con suma precisión, eso sí) para la expresión y pase por máquina de los algoritmos diseñados. Tras todos estos años podemos sentirnos confortados al ver que ese estilo, esa forma de introducir la programación, se ha convertido en la forma habitual, no solo en la UPC, sino que lo hemos exportado casi sin querer a otras universidades e incluso, aunque menos, a ciertas experiencias en enseñanza secundaria. Dos cosas muestran, en un rápido y simple análisis, que la opción era correcta:

a) los estudiantes que han seguido esa enseñanza tienen una sólida capacidad para diseñar algoritmos y una fácil adaptación a diferentes lenguajes de programación (la industria actual así lo reconoce), y

b) cuesta creer que nos equivoquemos tantos al mismo tiempo.

Sí puede, pues, hablarse de un estilo "UPC" en la forma de enseñar a programar, está claro que los autores de este texto se enmarcan de manera clara en esa orientación. El libro es claro, preciso y conciso, y se adapta muy bien al propósito que sus autores se proponen en el título, ya que puede servir de base para diversas asignaturas de programación de los planes de estudio de la UPC en la formación de diversas ingenierías.

Hay una pregunta que está en el aire y que me gustaría contestar: ¿debe la enseñanza de la programación ser la misma para las ingenierías informáticas que para el resto de ingenierías de la UPC?. Mi opinión al respecto es que, no en cuanto a los contenidos y a los énfasis que se pone en determinados temas, pero sí lo debe ser en cuanto al estilo: la diferencia distingue al "ingeniero en informática" del "ingeniero buen usuario de la informática", y la similitud garantiza un estilo riguroso (de ingeniería) en todos los casos. Y eso es precisamente lo que hace este texto, aportando ni más ni menos que lo que propone su título (que no es poco).

Pere Botella

Degà de la
Facultat d'Informàtica

Juliol 1993

Presentación

Esta obra ha sido concebida en tres partes, cada una de las cuales está orientada a cubrir uno de los siguientes objetivos del libro:

- El primero es dotar al lector de las herramientas básicas para expresar sus ideas sobre la forma de resolver un problema en forma de algoritmo. Para ello, el primer bloque de capítulos del libro introduce, desde el principio, los conceptos necesarios para el desarrollo de programas en genral. A través de los capítulos 1 a 5 se van presentando los distintos tipos de sentencias y datos elementales, y se ofrece al lector la posibilidad de familiarizarse con su uso, a través de variados ejemplos. Su lectura y asimilación permitirá al lector el desarrollo de algoritmos sencillos, y practicar los mecanismos básicos de su traducción a los lenguajes de programación, a través de los cuales se habrá ido introduciendo la sintaxis propia de éstos.

- El segundo objetivo es dotar al lector de las herramientas necesarias para desarrollar algoritmos básicos en el campo de la ingeniería. Estas herramientas comienzan con la descripción de una buena y sencilla metodología de programación (capítulos 6 y 7) basada en el concepto de módulo (programación modular) y su materialización en procedimientos y funciones, al traducir el algoritmo a un lenguaje de programación. Esta metodologia de programación se complementa con la introducción de los tipos de datos estructurados (capítulo 8) y el estudio de su adecuación a los algoritmos que deseamos desarrollar. Todo ello nos permite introducir un conjunto de algoritmos típicos (capítulos 9, 10 y 11), con los que completamos este objetivo de dos formas:
  - dando unos ejemplos de desarrollo de algoritmos, y
  - ofreciendo al lector un conjunto de algoritmos básicos, que él a su vez podrá emplear en el desarrollo de algoritmos más complejos.

- El tercer objetivo, el más ambicioso, introduce las herramientas que permitirán al lector aprovechar al máximo la capacidad de memoria del ordenador que esté programando. Para ello se le describen las sentencias de asignación y liberación de memoria, y los tipos de datos a ellas asociados, tipos de datos avanzados. En los capítulos 12 y 13 se practica en la utilización de estas sentencias y tipos de datos mediante el desarrollo de los procedimientos de manejo de las estructuras de datos

dinámicas más comunmente empleadas en los equipos informáticos de comunicación y control (pilas, colas y listas).

El enfoque dado en los últimos capítulos, para solucionar el problema del manejo de las estructuras dinámicas, está basado en las modernas técnicas de programación orientada a objeto, sin llegar a introducir todos los conceptos asociados a esta técnica de programación, pues no es el objetivo de este libro abordar la enseñanza de la programación en lenguajes orientados a objeto. Lo que sí nos ha parecido una meta alcanzable, es la introducción de la metodología en la que se fundamentan esos lenguajes de programación: la definición y el desarrollo de operaciones asociadas a tipos de datos complejos, a través de las cuales se consigue mostrar al usuario-programador tan sólo la visión imprescindible de la estructura, es decir, la que necesita para utilizarla, ocultándole los aspectos y detalles de la estructura, que no sean relevantes para la invocación de dichas operaciones.

Por lo que respecta a la metodología empleada en la redacción de este libro, debemos resaltar que en cada capítulo se da una importancia máxima al método de concepción y desarrollo de los algoritmos, dejando los programas, que de ellos se derivarán, en un segundo plano. De esta forma se resta importancia a una de las tareas, tradicionalmente de mayor importacia, y que sin embargo es de las más sencillas del conjunto de las necesarias para programar: la codificación de los algoritmos en un lenguaje de programación. De hecho, en los entornos modernos de programación, esa traducción es realizada automáticamente por las herramientas "CASE", por lo que el programador podrá olvidarse de ella en un futuro no muy lejano.

Esta aproximación tiene la ventaja de permitir al lector concentrarse primero en la descripción del algoritmo, en un lenguaje casi natural. Dejando los problemas de la codificación en un lenguaje de programación concreto para una segunda etapa, una vez se considere correcto el algoritmo que ha diseñado.

Para conseguir el objetivo de la desmitificación de la codificación en un lenguaje de programación concreto, en este libro, los algoritmos se traducen a dos lenguajes de programación, los dos más empleados en entornos técnicos:

Pascal, para la realización de programas con requisitos poco exigentes en cuanto al uso de los recursos del ordenador, fácil de leer y por tanto ideal para la construcción rápida de prototipos, y

C, para la realización de programas en los que se deba optimizar la utilización de recursos, o en los que se deban emplear éstos de una forma especial. El C es el lenguaje más empleado para la programación en entornos técnicos profesionales.

Los algoritmos empleados en esta obra para ilustrar los conceptos y herramientas introducidas en cada capítulo, han sido seleccionados pensando en el carácter técnico del

lector al que va destinada esta obra. Por ésta razón, además de los algoritmos típicos de ordenaciones o búsquedas, que aparecen en todos los libros de introducción a la programación y algorítmica, se han incluído algoritmos de aplicación de métodos numéricos. Con ello no se pretende ofrecer al lector un conjunto significativo de éstos, sino mostrarle cómo las herramientas de diseño de algoritmos, que se le han ido introduciendo, le permiten afrontar con soltura el desarrollo de los algoritmos que su trabajo técnico le exige conocer, y no solo los típicos problemas de manejo de caracteres, propios de aplicaciones de gestión informatizada de datos.

Por último, para concluir esta presentación del contenido de este libro, debemos recomendar al lector que ya conoce algún lenguaje de programación, que, a pesar de ello, repase sus conocimientos estudiando los algoritmos desarrollados en los primeros capítulos de esta obra. Ello le permitirá, sin apenas esfuerzo, familiarizarse con la nomenclatura empleada en el libro, en general, y especialmente con el lenguaje de descripción de algoritmos empleado. La progresiva dificultad de los algoritmos propuestos, hace que, pretender evitar los capítulos de introducción, pueda tener como consecuencia una pérdida de tiempo excesiva, para comprender los algoritmos más complejos del final del libro, que de otra forma serían mucho más fáciles de asimilar.

Indice

1. Introducción.

1.1 Definiciones.

1.1.1 Algoritmo.

Método preciso y finito para resolver un problema mediante una serie de pasos.

Todo algoritmo se debe caracterizar por su:

- **precisión** en la indicación del orden en que deben ejecutarse cada uno de los pasos, que debe ser siempre el mismo, para unas mismas condiciones de ejecución.
- **finitud** en la consecución del resultado deseado, es decir, la obtención de éste se debe alcanzar después de ejecutar un número finito de pasos.

1.1.2 Pseudocódigo.

Es la descripción de un algoritmo, realizada en un lenguaje específico de descripción de algoritmos (algorítmico). El lenguaje empleado para la descripción de un algoritmo, debe permitirnos alcanzar la condición de precisión impuesta al algoritmo.

Existen dos tipos de lenguajes algorítmicos:

- **semiformales**: cuyo objetivo fundamental es simplificar tanto la descripción del algoritmo, como la comprensión del mismo por un humano. Las reglas gramaticales y sintácticas que definen este tipo de lenguajes se han simplificado al mínimo para garantizar la precisión requerida en su interpretación.
- **formales**: su objetivo fundamental es permitir a un ordenador interpretar el algoritmo descrito, manteniendo al máximo la legibilidad de la descripción por un humano. En estos tipos de lenguaje, las reglas gramaticales y sintácticas

deben ser mucho más precisas e inequívocas de modo que no requieran una complejidad excesiva para su interpretación por el ordenador.

1.1.3 Programa.

Es la descripción de un algoritmo en un lenguaje de programación, es decir, un lenguaje orientado a optimizar la descripción de los pasos que debe realizar el ordenador para alcanzar el objetivo final del algoritmo.

Según el objetivo del programa y el lenguaje en el que se describe, se distinguen diversos tipos de programas:

- **fuente**: es el programa escrito por el programador a partir del algoritmo. En el caso de no emplear un lenguaje de descripción formal de algoritmos, es el primero que introducimos en el ordenador, origen, por tanto, de todo el proceso para conseguir que el ordenador ejecute nuestro algoritmo, razón por la cual se le llama "fuente". El programa fuente se caracteriza por ser relativamente independiente del ordenador en el que se debe ejecutar. Introducimos el programa fuente en el ordenador mediante un programa "**editor**".

- **objeto**: es el programa obtenido por el ordenador a partir del programa fuente, después de un proceso de "**compilación**", realizado a su vez por un programa **compilador**. En este proceso se traducen las instrucciones (pasos) descritos por el programador en el programa fuente, a operaciones directamente interpretables por el ordenador real en el que se debe ejecutar el programa.

- **ejecutable o binario**: es el programa obtenido por el ordenador a partir del programa objeto, después de un proceso de "**montaje**", realizado a su vez por un programa "**montador**" o "**linker**". En este proceso se complementa el programa objeto obtenido a partir del fuente, escrito por el programador, con otros programas objeto, escritos por otros programadores. En general, todos los programas objeto deben ser montados con programas suministrados por el fabricante del ordenador o del compilador, y agrupados en la denominada "**biblioteca del sistema**" (**System Library**), donde se encuentran recopilados (a modo de biblioteca) todos los programas necesarios para realizar operaciones básicas en el ordenador, como por ejemplo: leer caracteres o números del teclado, escribir en la pantalla, realizar operaciones aritméticas, etc.

- **de utilidad**: son los programas de uso general para cualquier aplicación. Los programas objeto de la librería del sistema se podrían considerar de utilidad, pero en general se reserva este nombre para programas de mayor envergadura y complejidad, como: compiladores, editores, montadores, etc.

- **de aplicación**: son los desarrollados para ser empleados por los usuarios finales de un ordenador, para ejercer una actividad no orientada a la producción de nuevos programas. Las aplicaciones pueden ser de innumerables tipos y, como ejemplo, podemos citar: contabilidad, bases de datos, gestión de pacientes de un hospital o de reservas de un hotel, correo electrónico, etc.

1.1.4 Lenguaje de programación.

Es el lenguaje empleado por el programador para dar al ordenador las instrucciones necesarias para la ejecución de un algoritmo determinado, constituyendo un programa fuente.

Existen diversos tipos de lenguajes de programación, según la proximidad de su gramática y tipos de instrucciones a las específicas del ordenador en el que deba ejecutarse el programa:

- **de alto nivel**: son aquellos en los que las instrucciones son descritas en un lenguaje próximo al empleado por los humanos. Permiten la ejecución del programa en ordenadores muy diversos y requieren su traducción a las características de cada ordenador en concreto, mediante el proceso de compilación. Ejemplos de este tipo de lenguajes son: PASCAL, FORTRAN, BASIC, COBOL, ADA, C, MODULA-2, etc.).

- **de bajo nivel o ensambladores**: son aquellos en los que las instrucciones se corresponden de forma casi unívoca con las ejecutables por el ordenador. Estos programas son difícilmente interpretables por los humanos no iniciados en el lenguaje, y no requieren su traducción a un ordenador concreto, sino que tan sólo hace falta cambiar la sintaxis para obtener el programa objeto, siguiendo un proceso denominado "**ensamblado**", realizado por un programa llamado **ensamblador**, que sustituye a la fase de compilación.

1.1.5 Acción.

Es cada uno de los pasos en los que hemos descrito un algoritmo. Las acciones son operaciones elementales que podemos expresar en un lenguaje algorítmico. Serán más o menos complejas dependiendo del nivel de detalle de descripción del algoritmo. Los algoritmos descritos de forma muy general están desglosados en acciones muy complejas, mientras que los descritos de forma detallada están desglosados en acciones muy sencillas, próximas al lenguaje de programación.

1.1.6 Sentencia.

Es una instrucción que podemos expresar en un lenguaje de programación. La precisión de las sentencias determina la precisión de la ejecución del programa, y para conseguirla deben ajustarse a una sintaxis muy concreta.

Por extensión, se denomina también sentencia a las acciones expresadas en un lenguaje algorítmico.

1.2 Tipos de sentencias.

En la descripción de un algoritmo o programa debemos expresar, no sólo las acciones a realizar, sino también las condiciones en las que éstas deben realizarse. Para conseguir controlar la secuencia (orden) en la que deben ejecutarse cada una de las sentencias de un algoritmo, se ha definido un reducido, pero suficiente, número de tipos de sentencias:

- **secuenciales**: son aquellas que se ejecutan en secuencia, es decir, después de la anterior y antes de la siguiente. Son las acciones elementales que se deben ejecutar para alcanzar el objetivo final del algoritmo. Un caso particular de sentencia secuencial es la de asignación, descrita en la sección siguiente.

- **selectivas**: son aquellas que permiten condicionar la ejecución de una o más sentencias al cumplimiento de una condición. En general, podemos decir que permiten seleccionar una acción a realizar entre varias alternativas.

- **iterativas**: son las que permiten conseguir que una determinada acción se ejecute más de una vez.

2. Sentencias de asignación.

2.1 Definiciones.

2.1.1 Formas de la sentencia de asignación.

Son las sentencias en las que, como su nombre indica, se asigna un valor a una variable. Esta asignación se puede hacer de forma **implícita**, al ejecutar una acción cualquiera, o de forma **explícita**, mediante una expresión de tipo matemático.

Así, por ejemplo, en la descripción del algoritmo de control de un ascensor, podemos tener la acción "Enviar el ascensor al piso P", después de cuya ejecución, una hipotética variable "PisoActual", que indicase el piso en el que se encuentra actualmente el ascensor, valdría P. Esta sentencia realiza implícitamente la asignación: PisoActual = P. De estas asignaciones implícitas nos ocuparemos en el capítulo dedicado a los procedimientos.

2.1.2 Expresión.

Una expresión es la descripción de una o varias operaciones a realizar sobre una o varias variables, es decir, una sentencia que evalúa una fórmula matemática.

Hay dos tipos de expresiones:

- **aritméticas**, en aritméticas (+ , - , * , etc.).
- **lógicas**, en las que las operaciones a realizar con las variables son lógicas (unión, intersección, etc.).

2.1.3 Precedencia.

La dificultad que representa escribir una fórmula matemática en forma de sentencia, es decir, en una sola línea de texto, en los lenguajes de programación y en los algorítmicos,

se subsana mediante unas **reglas de precedencia**, que determinan de forma inequívoca el orden en el que se ejecutará cada una de las operaciones indicadas en la expresión. Las podemos resumir de la forma siguiente:

I. se evalúan los paréntesis, primero los más interiores.

II. dentro de un paréntesis o expresión, las operaciones se ejecutarán en el siguiente orden:

- (signo), not

*, /, div (división entera), mod (módulo), and (y lógica)

+, - (resta), or (o lógica)

Las operaciones del mismo grupo (ej. * y /) se evalúan de izquierda a derecha.

2.1.4 Sentencias de entrada y salida.

Como caso particular de sentencia de asignación, mencionaremos aquí, por su utilidad, las sentencias de entrada y salida de datos:

- **leer** (<expresión **lista** de variables>).

  Permite asignar a las variables contenidas en la **lista** los valores tecleados por el usuario en el teclado del ordenador. Junto con la lista de variables, en esta sentencia podemos especificar un texto que se visualizará en la pantalla del ordenador al solicitar los valores de las variables de la lista, para orientar al usuario sobre las variables a las que serán asignadas los valores solicitados.

- **escribir** (<expresión **lista** de variables>).

  Permite presentar en la pantalla del ordenador los valores de las variables contenidas en la **lista**. Como en el caso anterior, además de las variables podemos especificar un texto explicativo de los valores visualizados.

2.2 Algoritmo ejemplo.

{Calcula el área de un polígono en forma de "L"}

ALGORITMO expresion

VAR a, b, c ENTERO

PRINCIPIO
  Calcular el área del cuadrado superior;
  Calcular el área del cuadrado inferior;
  Sumar las dos áreas calculadas;
  Escribir el resultado;
FIN.

Algoritmo 2.1: calcula el área de un polígono irregular.

2.3 Programa PASCAL.

```
PROGRAM expresion;

VAR a, b, c: INTEGER;

BEGIN

{ Calcular el área del cuadrado superior}
  a := 3*5;

{ Calcular el área del cuadrado inferior}
  b := 7*21;

{ Sumar las dos áreas calculadas}
  c := a + b;

{ Escribir el resultado}
  write ( c )

END
```

Programa PASCAL 2.1: calcula el área de un polígono irregular.

2.4 Programa C.

```
/*******************************/
/* PROGRAMA expresion */
/*******************************/

#include <stdio.h>

main ( )

{
   int a, b, c;

/* Calcular el área del cuadrado superior */
   a = 3 * 5;
/* Calcular el área del cuadrado inferior */
   b = 7 * 21;
/* Sumar las dos áreas calculadas */
   c = a + b;
/* Escribir el resultado */
   printf ( "%d", c );
}
```

Programa C 2.1: calcula el área de un polígono irregular.

2.5 Ejercicios de aplicación.

a).- Calcular el área de un rectángulo de base 3 y altura 7.

b).- Convertir una hora expresada en segundos a horas, minutos y segundos.

3. Sentencias selectivas.

3.1 Sentencia condicional.

3.1.1 Definiciones.

3.1.1.1 Forma de una sentencia.

Una sentencia puede ser:

- **simple**: es decir, descrita en el algoritmo mediante una sola acción.
- **compuesta**: descrita mediante más de una acción.
- **estructurada**: es una sentencia compuesta de dos partes:
  * control de ejecución y/o secuenciamiento, es decir, que determina el número de veces que deben realizarse las acciones descritas en la sentencia.
  * acción/es, es decir, la o las acciones que se realizarán una, más o ninguna vez, dependiendo del resultado de evaluar la parte de control de ejecución de la sentencia. En la descripción de una sentencia estructurada, esta/s acción/es las consideraremos como una única "sentencia compuesta" para simplificar la nomenclatura.

3.1.1.2 Metalenguaje.

Es un lenguaje empleado para describir otro lenguaje. El más conocido es el BNF.

Nosotros lo emplearemos en la descripción de los distintos tipos de sentencias.

Mediante este lenguaje podemos indicar si una parte de la descripción de la sentencia es opcional o si puede aparecer más de una vez en una ejemplarización concreta de la sentencia:

- [<algo>] indica que <algo> es **opcional**, es decir, puede usarse o no al emplear la sentencia descrita.

- [<algo1>|<algo2>] indica que, al emplear la sentencia descrita, puede aparecer **alternativamente** <algo1> o <algo2>.

- $[<algo>]_m^x$ indica que, al emplear la sentencia descrita, <algo> puede aparecer **repetida** un **mínimo** de **m** veces y un **máximo** de **x**.

3.1.1.3 Formato de la sentencia condicional.

SI <expresión condición> ENTONCES < sentencia si >
[SINO <sentencia no>]

- **<expresión condición>**, expresión cuyo resultado sólo puede ser verdadero o falso.

- **<sentencia si>**, sentencia a ejecutar caso de ser cierta **<expresión condición>**.

- **<sentencia no>**, sentencia a ejecutar caso de **no** ser cierta **<expresión condición>**. Esta parte de la sentencia es opcional.

- Tanto **<sentencia si>** como **<sentencia no>** pueden ser sentencias compuestas, es decir, formadas por más de una sentencia simple.

3.1.2 Algoritmo ejemplo.

```
ALGORITMO máximo;
VAR i, j, k, max: ENTERO
PRINCIPIO
   leer ( i, j, k );
   max := i;
   SI j > max  ENTONCES  max := j;
   SI k > max  ENTONCES  max := k;
   escribir  ( 'El mayor de ', i, j, k, ' es',max )
FIN.
```

Algoritmo 3.1: máximo de tres números.

3.1.3 Programa PASCAL.

```
PROGRAM maximo;

VAR i, j, k, max: INTEGER;

BEGIN
{ leer ( i, j, k );}
   readln ( i, j, k );
   max := i;
{ SI j > max ENTONCES max := j }
   IF j > max THEN max := j;
{ SI k > max ENTONCES max := k }
   IF k > max THEN max := k;
{ escribir ( 'El mayor de ', i, j, k,' es ', max ) }
   writeln ( 'El mayor de ', i:5, j:5, k:5,' es ', max:5 )
END.
```

Programa PASCAL 3.1: máximo de tres números.

NOTA: ":5" indica que el dato correspondiente debe escribirse empleando 5 caracteres.

3.1.4 Programa C.

```
/**************************/
/* PROGRAMA maximo */
/**************************/

#include <stdio.h>

main ( )

{
   int i, j, k, max;

/* leer ( i, j, k ) */
   scanf ("%d %d %d", &i, &j, &k);
   printf ( "\n" );
   max = i;
```

```
/*  SI j > max  ENTONCES max := j */
    if ( j > max ) max = j;
/*  SI k > max  ENTONCES  max := k */
    if ( k > max )  max = k;
/*  escribir ( 'El mayor de ', i, j, k, ' es ', max ) */
    printf ( "El mayor de %-5d %-5d %-5d es :  %5d\n", i, j, k, max );
}
```

Programa C 3.1: máximo de tres números.

NOTA: "-5" indica que el dato correspondiente debe escribirse empleando 5 caracteres.

3.1.5 Ejercicios de aplicación.

a).- Determinar el máximo de tres números.

b).- Determinar el máximo y el mínimo de tres números.

c).- Determinar si un número es cuadrado perfecto.

3.2 Sentencia CASO.

3.2.1 Definición.

Esta sentencia tiene la forma siguiente:

CASO <expresión> DE

[< lista de constantes i > : < sentencia i > ;]$_1^n$

[SINO <sentencia sino>]

FINCASO

<expresión> se evalúa al inicio de la ejecución de la sentencia CASO, obteniéndose un determinado "valor". Entonces,

se ejecuta la **<sentencia i>**, tal que "valor" sea igual a una de las constantes contenidas en **<lista de constantes i>**.

si "valor" no coincide con ninguno de los valores de las constantes de las n listas, se ejecuta la **<sentencia sino>**, si existe y, si no existe, no se realiza ninguna acción.

3.2.2 Algoritmo ejemplo.

```
v := valor;
CASO v DE
   1,2:      escribir ( "v vale 1 o 2" );
   3:        escribir ( "v vale 3" );
   SINO      escribir ( "v no vale 1, 2 o 3" )
FINCASO
```

Esta sentencia es equivalente a:

```
v := valor;
SI v = 1 o v = 2 ENTONCES escribir ( "v vale 1 o 2" )
                 SINO  SI v = 3 ENTONCES escribir ( "v vale 3" )
                                 SINO escribir ( "v no vale 1, 2 o 3" )
```

Es decir, que todo algoritmo expresado en función de la sentencia CASO puede ser también expresado en función de sentencias SI/ENTONCES/SINO encadenadas.

4. Sentencias iterativas.

Las sentencias iterativas son las que permiten ejecutar más de una vez una sentencia (compuesta).

En toda sentencia iterativa tenemos asociados los siguentes conceptos:

- **Sentencia del bucle**: sentencia, generalmente compuesta, que se ejecutará en cada iteración (ejecución o pasada) del bucle.

- **Condición de ejecución**: expresión booleana, que puede tomar sólo dos valores (cierta o falsa), del resultado de cuya evaluación se deduce la ejecución o no de la sentencia del bucle una vez (más).

- **Inicialización**: establecimiento del **estado inicial o de entrada** de las variables, cuyo valor condiciona la ejecución de la sentencia del bucle, antes de ejecutarse éste por primera vez. Consideraremos que una variable influye en la ejecución del bucle cuando es referenciada bien en la sentencia del bucle, o bien en la condición de ejecución del bucle.

- **Condición final o de salida**: estado de las variables manejadas en el bucle, que determina la **no ejecución** del bucle ninguna vez más. También la podemos interpretar como el estado de esas variables después de ejecutarse la sentencia del bucle por última vez.

4.1 Sentencia PARA.

4.1.1 Definición.

La sentencia PARA es la más sencilla de utilizar de las sentencias iterativas. La emplearemos cuando el número de iteraciones del bucle se conozca antes de iniciarse la ejecución de éste. Una variable de contaje, **contador**, va contando las pasadas (iteraciones) por el bucle.

La sentencia tiene la siguiente definición:

PARA < contador > := < expresión inicial >
[HASTA | DEC-HASTA] < expresión final >
[SALTO < incremento >]
HACER < sentencia del bucle >

Conceptos:

- **<contador>**, es la variable empleada para contabilizar las iteraciones del bucle realizadas.

- **<expresión inicial>**, el resultado de su evaluación determina el valor inicial, v_i, de <contador>, es decir, el valor con el que se ejecutará por primera vez el bucle.

- **<expresión final>**, el resultado de su evaluación determina el valor final, v_f, de <contador>, es decir, el valor con el que se ejecutará por última vez el bucle.

- **<incremento>**, determina la forma en la que se modificará <contador> al final de cada ejecución del bucle. Su valor se sumará o restará a <contador>, según hayamos usado la palabra reservada HASTA o DEC-HASTA antes de <expresión final>, respectivamente. La indicación de este valor en la sentencia es opcional, caso de no indicarse, el valor que toma por defecto es 1.

- **<sentencia del bucle>**, es la que constituye el denominado "cuerpo del bucle". En general será una sentencia compuesta, es decir, que estará formada por más de una sentencia simple, aunque puede ser una sola acción, o incluso no existir (acción nula).

4.1.2 Algoritmo ejemplo.

```
ALGORITMO sumatoria;

VAR i, j, sum : ENTERO

PRINCIPIO
   Inicializa sumatoria;
   PARA i := 1 HASTA 10 HACER
   PRINCIPIO
      Calcula término i-ésimo;
      Suma a sumatoria el término i-ésimo;
   FIN;
```

```
    escribe ( "La suma es ", sumatoria );
FIN.
```

Algoritmo 4.1: calcular sumatoria (i^3) para i = 1,10.

4.1.3 Programa PASCAL

```
    PROGRAM sumatoria;

    VAR i, j, sum : INTEGER;

    BEGIN
{   Inicializa sumatoria}
    sum := 0;
{   PARA i := 1 HASTA 10 HACER}
    FOR i := 1 TO 10 DO
        BEGIN
{       Calcula término i-ésimo }
        j := i * i * i;
{       Suma a sumatoria el término i-ésimo }
        sum := sum + j;
        END
{   escribe ("La suma es ", sumatoria) }
    write (" La suma es ", sum );
    END .
```

Programa PASCAL 4.1 : calcular sumatoria (i^3) para i = 1,10.

4.1.4 Programa C.

```
/**************************/
/* PROGRAMA  reales */
/**************************/

#include <stdio.h>

main ( )

{
```

```
   int i,sum;
/* Inicializa sumatoria */
   sum= 0;
/* PARA i:=1 HASTA 10 HACER */
   for ( i= 1 ;i <= 10 ; i++ ) {
/*     Calcula término i-ésimo */
       j = i * i * i;
/*     Suma a sumatoria el término i-ésimo */
       sum = sum + j;
   }
/* escribe ("La suma es ", sumatoria) */
   printf ( "La suma es", sum );
}
```

Programa C 4.1: calcular sumatoria (i^3) para i = 1,10.

4.1.5 Ejercicios de aplicación.

a).- Calcular el factorial de un número x.

b).- Hallar la suma de todos los números multiplos de 5 comprendidos entre dos enteros a y b.

c).- Imprimir los 15 primeros números de la serie de Fibonacci ($a_i = a_{i-1} + a_{i-2}$, $a_0 = a_1 = 1$).

d).- Calcular el número de combinaciones de n elementos tomados de m en m.

4.2 Sentencia MIENTRAS.

4.2.1 Definición.

En la sentencia PARA, debemos conocer el número exacto de veces que se deben ejecutar las sentencias del bucle. Cuando esto no es posible, deberemos emplear la sentencia MIENTRAS para conseguir ejecutar varias veces las sentencias del bucle. En esta sentencia el número de iteraciones depende de la evaluación de una condición.

La sentencia MIENTRAS tiene la siguiente definición:

MIENTRAS <condición ejecución> HACER <sentencia bucle>

- **<condición ejecución>** es una expresión que puede tomar los valores cierto o falso.

- **<sentencia bucle>** es la sentencia que se debe ejecutar en el caso de que el resultado de la evaluación de <condición ejecución> sea cierto. Puede ser una sentencia compuesta.

4.2.2 Algoritmo ejemplo

```
ALGORITMO media;

VAR i, j, sum : ENTERO

PRINCIPIO
   Inicializa suma y contador de elementos leídos;
   leer (elemento);
   MIENTRAS elemento no valga -1 HACER
      PRINCIPIO
      Cuenta elemento leído;
      Lo suma a la suma de todos los anteriores;
      leer (elemento)
      FIN;
   escribeln (" La media es ", sum DIV i )
FIN.
```

Algoritmo 4.2: hallar la media de un conjunto de valores enteros de entrada acabado en el numero -1.

NOTA: DIV indica que se toma sólo la parte entera de la división.

4.2.3 Programa PASCAL.

```
PROGRAM media;

VAR i, j, sum : INTEGER;

BEGIN
{  Inicializa suma y contador de elementos leídos }
   sum := 0; i := 0;
{  leer (elemento) }
   readln (j) ;
```

```
{  MIENTRAS elemento no valga -1 HACER }
   WHILE j <> -1  DO
      BEGIN
{     Cuenta elemento leído }
      i:= i + 1;
{     Lo suma a la suma de todos los anteriores }
      sum := sum + j;
{     leer  ( elemento ) }
      readln ( j );
      END;
{  escribeln ( " La media  es ", sum DIV i ) }
   writeln  ( " La media  es ", sum DIV i  );
END.
```

Programa PASCAL 4.2 : hallar la media de un conjunto de valores enteros de entrada, acabado en el número -1.

4.2.4 Programa C.

```
/*****************************/
/* PROGRAMA media */
/*****************************/

#include <stdio.h>

main ( )

{
   int i, j, sum;

/* Inicializa suma y contador de elementos leídos */
   sum = 0;
   i = 0;
/* leer (elemento) */
   scanf ( "%d", &j );
/* MIENTRAS elemento no valga -1 HACER COMIENZO*/
   while ( j != -1 ) {
/*     Cuenta elemento leído */
       i ++ ;
/*     Lo suma a la suma de todos los anteriores */
```

```
        sum + = j;
/*      leer ( elemento ) */
        scanf("%d", &j);
/*  FIN */
    }
/*  escribeln ( " La media es ", sum DIV i ) */
    printf ( "La media es %d\n", sum / i );
}
```

Programa C 4.2: hallar la media de un conjunto de valores enteros de entrada, acabado en el número -1.

NOTA: En C, al realizar la división entre dos enteros, el resultado da directamente un entero y, por tanto, no hace falta emplear una operación especial.

4.2.5 Ejercicios de aplicación.

a).- Calcular la suma de los 20 primeros números multiplos de 7.

b).- Calcular la suma de los 20 primeros números múltiplos de 7 o de 5.

c).- Convertir un valor entero positivo a cualquier base de nmeración igual o inferior a 10.

d).- Imprimir los términos de la serie de Fibonacci menores que 10.000 ($a_i = a_{i-1} + a_{i-2}$, $a_0=a_1=1$).

4.3 Sentencia REPETIR.

4.3.1 Definición.

En la sentencia MIENTRAS, la condición de ejecución del bucle se verifica antes de haber ejecutado éste por primera vez. Por tanto, es necesario inicializar las variables empleadas para determinar su ejecución antes de entrar por primera vez en éste, lo cual puede tener como consecuencia que el bucle no llegue a ejecutarse nunca. Si queremos evitar ambas cosas, es decir:

* tener que inicializar las variables de control del bucle y
* ejecutar al menos una vez el bucle,

debemos emplear la sentencia REPETIR. En esta sentencia, el número de iteraciones depende de la evaluación de una condición, como en la MIENTRAS, pero esta evaluación se realiza después de cada ejecución del bucle.

La sentencia REPETIR tiene la siguiente definición:

REPETIR < sentencia bucle > HASTA < condición salida >, donde:

- **<condición salida>** es una expresión que puede tomar los valores cierto o falso.
- **<sentencia bucle>** es la sentencia que se debe volver a ejecutar en el caso de que el resultado de la evaluación de <condición salida> sea falso. Puede ser una sentencia compuesta.

4.3.2 Algoritmo ejemplo.

Como ejemplo de este tipo de sentencia emplearemos un algoritmo con el mismo objetivo que el del apartado anterior, cambiando la estructura MIENTRAS por REPETIR. De esta forma, además, podremos comparar los resultados al emplear una u otra.

Para mantener la similitud entre ambas descripciones algorítmicas, haremos que ésta tenga las mismas restricciones que la anterior, es decir, que permita una lista vacía de números, y que ésta acabe en "-1".

```
ALGORITMO media_R;
VAR i, j, sum : ENTERO

PRINCIPIO
   Inicializa suma y contador de elementos leídos;
   REPETIR
      PRINCIPIO
      leer (elemento);
      Cuenta elemento leído;
      Lo suma a la suma de todos los anteriores
      FIN;
   HASTA elemento =  -1;
   Restar el último elemento "-1" y descontarlo
   escribeln ( "La media  es ", sum DIV i  )
FIN.
```

Algoritmo 4.3: hallar la media de un conjunto de valores enteros de entrada acabado en el numero -1.

NOTA: DIV indica que se toma sólo la parte entera de la división.

4.3.3 Programa PASCAL.

```
PROGRAM media;

VAR i, j, sum : INTEGER;

BEGIN
{  Inicializa suma y contador de elementos leídos }
   sum := 0; i := 0;
{  REPETIR }
   REPEAT
      BEGIN
{     leer (elemento) }
      readln ( j );
{     Cuenta elemento leído }
      i := i + 1;
{     Lo suma a la suma de todos los anteriores }
      sum := sum + j;
      END;
{  HASTA elemento = -1 }
   UNTIL j =  -1;
{  Restar el último elemento "-1" y descontarlo }
   sum := sum - j;            i := i - 1;
{  escribeln (" La media  es ", sum DIV i  ) }
   writeln  (" La media  es ", sum DIV i  );
END.
```

Programa PASCAL 4.3 : hallar la media de un conjunto de valores enteros de entrada, acabado en el número -1.

4.3.4 Programa C.

```
/*************************/
/* PROGRAMA media */
/*************************/

#include <stdio.h>
```

```
main ( )

{
   int i, j, sum;

/* Inicializa suma y contador de elementos leídos */
   sum = 0 ;
   i = 0 ;
/* REPETIR */
   do {
/*     leer (elemento) */
       scanf ( "%d", &j ) ;
/*     Cuenta elemento leído;*/
       i ++ ;
/*     Lo suma a la suma de todos los anteriores */
       sum + = j ;
/* HASTA elemento = -1 */
   } while ( j != -1 ) ;
/* Restar el último elemento "-1" y descontarlo */
   sum - = j ; i -- ;
/* escribeln ( " La media es ", sum DIV i ) */
   printf ( "La media es %d\n", sum / i ) ;
}
```

Programa C 4.3 : hallar la media de un conjunto de valores enteros de entrada, acabado en el número -1.

NOTA: Obsérvese que en C, la condición de la sentencia equivalente a la de REPETIR (do/while) indica cuándo debe repetirse el bucle, y no cuándo se debe salir de él, es decir, lo contrario del lenguaje algorítmico o PASCAL.

4.3.5 Ejercicios de aplicación.

Como ejercicios para practicar el uso de esta sentencia, se pueden emplear los mismos enunciados de la sentencia MIENTRAS, tal como se ha hecho en el algoritmo ejemplo.

5. Tipos de datos.

Al declarar una variable, se especifica tanto su nombre, como el tipo al que pertenece. El **nombre** sirve para distinguirla de cualquier otra variable empleada en el algoritmo. El **tipo** indica la forma en que debe almacenarse el valor de la variable en el ordenador, de la cual dependerá el rango de valores que ésta podrá tomar, así como su precisión.

Podemos considerar tres clases generales de tipos de datos:

- **tipos de datos elementales**: son aquéllos predefinidos en el lenguaje, es decir, aquellos que el ordenador ya sabe como almacenar.

- **tipos de datos definidos por el usuario**: son aquéllos que el ordenador no conoce previamente, por esta razón el usuario debe indicar la forma en que se deben almacenar los posibles valores de las variables de esos tipos.

- **tipos de datos estructurados**: Aquéllos en los que, a diferencia de los tipos anteriores, cada variable puede almacenar más de un dato simultáneamente.

En este capítulo nos centraremos fundamentalmente en las dos primeras clases de tipos de datos. De los tipos de datos estructurados nos ocuparemos más adelante, cuando dispongamos de más herramientas de diseño de algoritmos.

5.1 Carácter.

5.1.1 Definiciones.

- Tipo CAR (CHAR en Pascal y C): representa un conjunto ordenado y finito de caracteres. A cada carácter le corresponde un número natural.

- De todas las operaciones que podemos realizar con variables de tipo CAR, destacaremos dos: ORD y CHR, que permiten conocer la relación existente entre el conjunto de caracteres y los números naturales.
  * ORD(<carácter>) nos da como resultado el número de orden de ese <carácter> en la tabla empleada para codificarlo en el ordenador.
  * CHR(<número>) nos da como resultado el carácter que ocupa la posición <número> en la tabla empleada para codificarlo en el ordenador.

5.1.2 Algoritmo ejemplo.

```
ALGORITMO mayúscula

VAR  c: CAR;
     diferencia : ENTERO;

PRINCIPIO
   Calcular la diferencia de orden entre una letra en minúscula y en mayúscula
   Leer ( c );
   MIENTRAS c no sea un ' . ' HACER
      PRINCIPIO
      escribir ( carácter que ocupa la posición de c menos diferencia )
      leerln ( c );
      FIN;
FIN.
```

Algoritmo 5.1: convierte una serie de letras minúsculas, acabadas en '.' , a mayúsculas.

5.1.3 Programa PASCAL.

```
PROGRAM mayusculas;

VAR  c : CHAR;
      diferencia : INTEGER;

BEGIN

{  Calcular la diferencia de orden entre una letra en minúscula y en mayúscula }
   diferencia := ord ( 'a' ) - ord ( 'A' );
```

```
{  Leer ( c ) }
   readln ( c ) ;
{  MIENTRAS  c no sea un ' . ' HACER }
   WHILE c <> ' . ' DO
      BEGIN
{     escribir ( carácter que ocupa la posición de c menos diferencia) }
      writeln ( chr ( ord ( c ) - diferencia ) );
{     leerln ( c) }
      readln ( c );
      END
   END.
```

Programa PASCAL 5.1: convertir una serie de caracteres en minúscula, acabados por el caracter '.', a caracteres en mayúscula.

5.1.4 Programa C.

```
/**********************************/
/*  PROGRAMA mayusculas  */
/**********************************/

#include <stdio.h>

main ( )
{
   char c;
   int diferencia;

/* Calcular la diferencia de orden entre una letra en minúscula y en mayúscula */
   diferencia =  ( 'a' ) - ( 'A' );
/* Leer ( c ) */
   scanf ( "%c", &c );
/* MIENTRAS  c no sea un ' . ' HACER  COMIENZO*/
   while ( c != ' . ' ) {
/*    escribir (carácter que ocupa la posición de c menos diferencia) */
      printf ( "%c", ( ( c ) - diferencia ) );
/*    leerln (c) */
      scanf ( "%c", &c );
/* FIN */
   }
```

```
/*FIN */
}
```

Programa C 5 : convertir una serie de caracteres en minúscula, acabados por el caracter '.', a caracteres en mayúscula.

NOTA: En C el operador distinto (a<>b) tiene la forma: a != b (literalmente: no (!) igual (=)).

5.1.5 Ejercicios de aplicación.

a).- Contar el número de apariciones de una letra en un texto acabado en punto, leído como dato.

b).- Contar el número de líneas de un texto acabado en punto.

c).- Lectura de un carácter y determinación de su tipo: alfabético, numérico, blanco o signo de puntuación.

5.2 Numéricos.

5.2.1 Definiciones.

Hasta ahora hemos estado empleando un tipo de dato ENTERO para definir las variables, que podían tomar valores numéricos, pero éste no es el único tipo de datos con el que las podemos codificar en el ordenador:

- ENTERO: permite almacenar cualquier número entero, entre -MAXENT y MAXENT-1, donde MAXENT depende del ordenador que estemos empleando, aunque generalmente es 2^{15} (32.768).

- REAL: permite almacenar algunos números reales comprendidos entre MINREAL y MAXREAL, el 0, y el mismo número de negativos. MINREAL y MAXREAL dependen del ordenador que estemos empleando.

- Con los números reales se pueden realizar multitud de funciones. Remitimos al lector a los manuales correspondientes del compilador que esté empleando, y aquí tan sólo citaremos algún ejemplo:
  - * aritméticas: SQRT (raíz cuadrada), SQR (cuadrado), ABS (valor absoluto), etc.
  - * trigonométricas: SIN (seno), ARCTAN (arco tangente), etc.

5.2.2 Algoritmo ejemplo.

```
ALGORITMO reales;

VAR i: ENTERO;
   sum: REAL;

PRINCIPIO
   Inicializa sumatoria;
   PARA i := 1 HASTA 10 HACER
      Suma a sumatoria el término i-ésimo;
   escribe ( "La suma es ", sumatoria );
FIN.
```

Algoritmo 5.2: calcular sumatoria (1 / i) para i = 1,10.

5.2.3 Programa PASCAL.

```
PROGRAM reales;

VAR i : INTEGER; sum : REAL;

BEGIN

{  Inicializa sumatoria }
   sum := 0.0;

{  PARA i:=1 HASTA 10 HACER }
   FOR i := 1 TO 10 DO
{     Suma a sumatoria el término i-ésimo }
      sum := sum + 1 / i;
{  escribe ( "La suma es ", sumatoria ) }
   write (" La suma es ", sum );

END .
```

Programa PASCAL 5.2: calcular sumatoria (1 / i) para i = 1,10.

5.2.4 Programa C.

```
/**************************/
/* PROGRAMA reales */
/**************************/

#include <stdio.h>

main ( )
{
   int      i;
/* sum:     REAL */
   float    sum;

/* Inicializa sumatoria */
   sum = 0;
/* PARA i := 1 HASTA 10 HACER */
   for ( i= 1; i <= 10; i++ )
/*     Suma a sumatoria el término i-ésimo */
       sum = sum + ( 1. / i );
/* escribe ( "La suma es ", sumatoria ) */
   printf ( "\nLa suma es %f", sum );
}
```

Programa C 5.2: calcular sumatoria (1 / i) para i = 1,10.

NOTA: El término i-ésimo se calcula como 1./i para forzar la realización de la división entre reales, porque i es entero y, como la división se realiza antes que la suma, si hiciésemos 1/i, sería una división entre enteros y, el resultado sería también entero, es decir 0, salvo para i=1.

5.2.5 Ejercicios de aplicación.

a).- Resolver la ecuación: $ax^2 + bx + c = 0$; con a <> 0.

b).- Leer los lados de un rectángulo y determinar el lado del menor cuadrado que lo contiene.

c).- Determinar la raíz cuadrada entera de un número entero positivo mediante la ley de recurrencia del algoritmo de Newton:

raíz (0) = 1; raíz (i+1) = (raíz (i) + n / raíz (i)) / 2

d).- Calcular el número e.

5.3 Booleano.

5.3.1 Definiciones.

Este tipo de dato está predefinido en Pascal, pero no lo está en C. No obstante, lo hemos incluido en este capítulo por su sencillez y utilidad de uso.

- El tipo de dato booleano puede tomar tan solo dos valores: cierto o falso.
- Con las variables de este tipo se pueden realizar operaciones lógicas: Y, O, NO.
- Las operaciones de relación (=, >, >=, <, <=, <>) dan un resultado de este tipo. (En C estos operadores son: ==, >, >=, <, <=, !=).

5.3.2 Algoritmo ejemplo.

```
ALGORITMO reales_2;

VAR i: ENTERO;
    sumatoria, épsilon: REAL;
    fin: BOOLEAN;
PRINCIPIO
   escribir ( 'Introduzca la constante de error : ' );
   leer ( épsilon );
   Inicializamos variables del bucle: sumatoria, fin, contador término ( i );
   MIENTRAS NO fin HACER
   PRINCIPIO
      Suma el término i-ésimo a sumatoria;
      Cuenta el término;
      SI término i-ésimo es menor que épsilon
            ENTONCES final del bucle;
   FIN;
   escribir ( " La suma es ", sumatoria );
FIN.
```

Algoritmo 5.3: calcular sumatoria (1 / i) para i = 1 .. infinito.

El algoritmo descrito calcula la sumatoria de la serie de los inversos de los números naturales. El número de términos realmente sumados lo determina el error con el que queramos obtener la citada suma, el cual es función del valor del último término sumado, que en nuestro caso deberá ser menor que una cierta cantidad muy pequeña, leída del terminal al principio de la ejecución del algoritmo, y llamada épsilon.

5.3.3 Programa PASCAL.

```
PROGRAM reales;

VAR  i : INTEGER; sum, epsilon : REAL;
       fin : BOOLEAN;

BEGIN
   writeln ( 'Introduzca la constante de error : ' );
   readln ( epsilon );
{  Inicializamos variables del bucle: sumatoria, fin, contador término ( i )  }
   fin := FALSE;
   sum := 0.0;  i := 1;
{  MIENTRAS NO fin HACER }
   WHILE NOT fin  DO
      BEGIN
{     Suma el término i-ésimo a sumatoria }
      sum := sum + 1/i;
{     Cuenta el término }
      i := i + 1;
{     SI término i-ésimo es menor que épsilon  ENTONCES  final del bucle }
      IF  1 / i  < epsilon THEN fin := TRUE;
      END
   write ( " La suma es ", sum );
   END.
```

Programa PASCAL 5.3: calcular sumatoria (1 / i) para i = 1,infinito, con error < épsilon (constante real muy pequeña).

5.3.4 Programa C.

```
/*****************************/
/* PROGRAMA reales_2*/
/*****************************/
```

```
#include <stdio.h>

/* Definimos la variable de tipo booleano */
typedef int boolean;
#define TRUE 1
#define FALSE 0

main ( )

{
   int i;
   float sum, epsilon;
   boolean fin;

/* escribir ( 'Introduzca la constante de error : ') */
   printf ( "Introduzca la constante de error: " );
/* leer ( épsilon ) */
   scanf("%f", &epsilon);
/* Inicializamos variables del bucle: sumatoria, fin, contador término (i) */
   fin = FALSE;
   sum = 0;
   i = 1;
/* MIENTRAS NO fin HACER  COMIENZO*/
   while ( ! fin ) {
/*     Suma el término i-ésimo a sumatoria */
       sum = sum + 1./ i;
/*     Cuenta el término */
       i ++;
/*     SI término i-ésimo es menor que épsilon  ENTONCES  final del bucle */
       if ( ( 1./ i ) < epsilon )  fin = TRUE;
/* FIN */
   }
   printf ( "\nLa suma es %f", sum );
}
```

Programa C 5.3: calcular sumatoria (1 / i) para i = 1, infinito, con error < épsilon (constante real muy pequeña).

NOTA: En C nos vemos obligados a definir un tipo de variable BOOLEAN, tal como se ve en el ejemplo.

5.3.5 Ejercicios de aplicación.

a).- Determinar si un número x es primo.

b).- Lectura de una hora expresada en horas: minutos: segundos, y determinar si es válida.

5.4 Tipos definidos por el usuario.

5.4.1 Definiciones.

Como hemos visto en el programa C del ejemplo anterior, en ocasiones es conveniente definir unos tipos de datos, que se almacenarán en la memoria del ordenador como alguno de los datos elementales vistos hasta ahora, pero que tan sólo pueden tomar un subconjunto de los valores de éstos. El usuario puede especificar este subconjunto de valores mediante dos métodos distintos:

- En los denominados **tipos enumerados**, el usuario "enumera" los valores que una variable de este tipo podrá tomar. Es decir, indica explícitamente todos los valores que una variable de este tipo puede tomar. Se definen mediante sentencias de la forma:

  TIPO = (<lista de valores>)

  Donde: <lista de valores> se define como: <valor>[,<valor>]$_0^n$

  Por ejemplo:

  TIPO SEMANA (lunes, martes, miércoles, jueves, viernes, sábado, domingo);

  Algoritmo 5.4: declaración de un tipo enumerado.

- En los denominados **tipos subrango**, el usuario simplemente indica los límites inferior y superior del conjunto de valores posibles para el tipo que se está definiendo. El tipo de datos elemental del que se toma el subrango de valores, debe ser un **tipo ordenado**, es decir, que se conozca fácilmente el **predecesor** y el **sucesor** de cualquier valor posible dentro de ese tipo (los enteros y los caracteres son ordenados, pero los reales no lo son). Los tipos subrango se definen mediante sentencias de la forma:

  TIPO = < mínimo > .. < máximo >

  Por ejemplo:

```
CONSTANTE MAXENT = 32.767;
TIPO decimal =   0 .. 9;
     letra =     'A' .. 'Z';
     Mes =       1 .. 31;
     Positivo =  0 .. MAXENT;
```

Algoritmo 5.5: declaración de un tipo subrango.

La utilidad de estos tipos de datos se manifiesta en los programas escritos en lenguajes como el Pascal, que pueden verificar la validez de los valores adquiridos por las variables de uno de estos tipos durante la ejecución del programa, con lo cual facilitan la detección de errores en la ejecución de éste, consecuencia de los cuales las variables toman valores inadmisibles.

En otros lenguajes, en los que estas comprobaciones no se realizan durante la ejecución del programa, tan sólo sirve para detectar algunos errores en la escritura del algoritmo, si mezclamos variables de tipos distintos en una misma sentencia de asignación. Y en cualquier caso, la práctica de las definiciones de estos tipos de datos contribuye a mejorar sensiblemente la legibilidad de los programas y, por tanto, su inteligibilidad por los programadores que deban interpretarlos en un momento dado.

5.4.2 Algoritmo ejemplo.

```
ALGORITMO sumar_horas;

TIPO diasemana = ( lunes, martes, miercoles, jueves, viernes, sabado, domingo );
VAR  dia, horas :  ENTERO;
     diasem :      diasemana ;

PRINCIPIO
{  prescindimos de fiestas y suponemos que el primer dia del año es lunes }
   Inicializamos variables del bucle;
   PARA cada día del año HACER
   PRINCIPIO
      CASO diasem DE
            lunes, miercoles, viernes: horas := horas + 8;
            martes, jueves :           horas := horas + 7 ;
            sabado :                   horas := horas + 4 ;
            domingo : ;
      FINCASO;
```

```
        diasem := diasem siguiente;
    FIN;
FIN.
```

Algoritmo 5.6: suma las horas laborables del año de una empresa.

NOTA: La jornada laboral de dicha empresa es de 8 horas los lunes, miércoles y viernes; de 7 horas los martes y jueves, y de 4 horas los sábados.

5.4.3 Programa PASCAL.

```
PROGRAM sumar_horas ;

TYPE diasemana = ( lunes, martes, miercoles, jueves, viernes, sabado, domingo ) ;
VAR  dia, horas :  INTEGER ;
     diasem :      diasemana ;

BEGIN
{  Prescindimos de fiestas y suponemos que el primer día del año es lunes }
{  Inicializamos variables del bucle }
   horas := 0; diasem := lunes ;

{  PARA cada día del año HACER }
   FOR dia := 1 TO 365 DO
   BEGIN
      CASE diasem OF
            lunes, miercoles, viernes :  horas := horas + 8 ;
            martes, jueves :             horas := horas + 7 ;
            sabado :                     horas := horas + 4 ;
            domingo : ;
      END ;

{     diasem := diasem siguiente }
      IF diasem = domingo
            THEN  diasem := lunes
            ELSE  diasem := succ ( diasem ) ;
   END
END.
```

Programa PASCAL 5.4 : sumar las horas laborables del año de una empresa.

NOTA1: La jornada laboral de dicha empresa es de 8 horas los lunes, miércoles y viernes; de 7 horas los martes y jueves, y de 4 horas los sábados .

NOTA2: Obsérvese que para calcular el siguiente diasem se distinguen dos casos, según sea el siguiente de los valores dados al enumerarlos en la definición del tipo, o debamos empezar nuevamente por el primero al acabarse una semana.

5.4.4 Programa C.

```
/*************************/
/* PROGRAMA sumarhoras */
/*************************/

#include <stdio.h>

main ( )
{
   typedef enum  { LUNES, MARTES, MIERCOLES, JUEVES,
                   VIERNES, SABADO, DOMINGO } diasemana;
   int dia, horas;
   diasemana diasem;

/* Prescindimos de fiestas y suponemos que el primer día del año es lunes */
/* Inicializamos variables del bucle */
   horas = 0; diasem = LUNES ;
/* PARA cada día del año HACER */
   for ( dia = 1; dia <= 365; dia++ ) {
/*     CASO diasem DE */
       switch ( diasem ) {
          case LUNES: case MIERCOLES: case VIERNES:  horas = horas + 8; break;
          case MARTES:case JUEVES:  horas = horas + 7 ;break;
          case SABADO:  horas = horas + 4 ;break;
          case DOMINGO:
       };
/*     diasem := diasem siguiente */
       if ( diasem == DOMINGO )
                   diasem = LUNES;
       else        diasem = succ ( diasem ) ;
   };
 }
```

Programa C 5.4: sumar las horas laborables del año de una empresa.

NOTA: Obsérvese que, entre cada lista de valores correspondiente a cada CASO de la sentencia SWITCH, se ha intercalado la sentencia break, cuyo resultado es evitar que, después de encontrarse un valor igual al de la variable de la estructura (diasem en este caso), se siga comparando ésta con los valores contenidos en el resto de las listas de la estructura.

5.4.5 Ejercicios de aplicación.

a).- Pasar una fecha expresada en día, mes y año, a la fecha inmediatamente siguiente.

b).- Descomponer en billetes y monedas un importe determinado.

5.5 Vectores.

5.5.1 Definiciones.

Un vector es una estructura de datos en la que podemos almacenar un determinado número de datos, todos ellos del mismo tipo, ordenados en una sola dimensión.

Declaración:

< tipovector > = MATRIZ [< tipoíndice >] DE < tipoelemento >

- **<tipovector>** es el nombre del nuevo tipo vectorial. Una vez definido el tipo, pueden declararse variables de ese tipo. También pueden declararse variables de tipo vector directamente sin nombre de tipo explícito. Esta segunda opción (abreviada) para la definición de vectores, la emplearemos cuando sólo tengamos que definir vectores en una sentencia del algoritmo o programa.

- **[<tipoíndice>]** es, en general, un subrango de los enteros, [<límite inferior> .. <límite superior>], pero puede ser cualquier tipo ordenado (subrango o enumerado). Indica el tamaño de la estructura, es decir, el número de variables del mismo tipo que se alinean dentro del vector.

- **<tipoelemento>** es el tipo de todos los elementos del vector, el tipo básico, que puede a su vez ser de cualquier tipo (elemental, estructurado, o definido por el usuario). Un vector ocupa tantas posiciones de memoria como resulte de multiplicar las ocupadas por una variable de **tipoelemento** por el número de elementos del vector, indicado por **[tipoíndice]**.

- Un tipo especial de vector es la **"cadena de caracteres" (string),** que en algunos lenguajes, como el PASCAL, es un tipo de dato elemental, es decir, definido en el lenguaje. Prácticamente todos los lenguajes disponen de funciones especiales para manejar este tipo de datos.

  Con los vectores podemos realizar operaciones a dos niveles:

- con todo el vector: copia de una variable vectorial sobre otra de su mismo tipo (declarada en la misma sentencia o mediante el mismo nombre de tipo definido por el usuario).

- accediendo a un elemento del vector, mediante referencias de la forma:

  <nombrevector> [<índice>]

  donde **<nombrevector>** es el nombre de la variable de tipo vector, e **[<índice>]** indica el elemento concreto del vector con el que queremos operar.

  Con los elementos de un vector podemos realizar cualquier operación permitida con variables de su tipo, es decir, del tipo básico del vector.

5.5.2 Algoritmo ejemplo.

En una primera aproximación, el algoritmo para normalizar un vector, es decir, para conseguir que el módulo de un vector sea 1, es el siguiente:

```
Leer_vector
Calcular módulo
normalizar_vector
escribir_vector
```

Como las acciones en las que se ha expresado este algoritmo son aún demasiado complejas para poderlas traducir directamente a un lenguaje de programación, vamos a refinarlas hasta que su descripción corresponda a una sentencia válida en un lenguaje de programación de alto nivel. El resultado es el siguiente:

```
ALGORITMO normalizar;

CONST maxelem = 14 ;
VAR vector : MATRIZ [ 1..maxelem ] DE REAL;
    j : ENTERO;
   vmodulo := REAL;
```

```
PRINCIPIO
{  Leer_vector: }
   PARA j := 1 HASTA maxelem HACER  leer ( vector [ j ] ) ;
{  Calcular módulo: }
   Inicializar suma de cuadrados:  vmodulo := 0.;
   PARA j := 1 HASTA maxelem HACER
      Acumular a vmodulo el elemento j de vector al cuadrado;
   módulo := raíz cuadrada de suma de cuadrados ( vmodulo );
{  normalizar vector: }
   PARA i := 1 HASTA maxelem HACER
{     normalizar elemento i de vector: }
      vector [ i ] := vector [ i ] / vmodulo ;
{  escribir_vector: }
   PARA j := 1 HASTA maxelem HACER
      escribir ( vector [ j ] : 6 : 3 ) ;
FIN.
```

Algoritmo 5.7: normalización del módulo de un vector.

5.5.3 Programa PASCAL

```
PROGRAM normalizar;

CONST maxelem = 14 ;
VAR  vector : ARRAY [ 1..maxelem ] OF REAL;
       j  : INTEGER ;
       vmodulo : REAL ;

BEGIN
{  Leer_vector: }
   FOR j := 1 TO maxelem DO  read  ( vector [ j ]  : 6 : 3 ) ;
{  Calcular módulo: }
{  Inicializar suma de cuadrados: vmodulo := 0. }
   vmodulo := 0.0 ;

   FOR j :=1  TO maxelem DO
{     Acumular a vmodulo el elemento j de vector al cuadrado }
      vmodulo := vmodulo + sqr ( vector [ j ] ) ;
{  módulo := raíz cuadrada de suma de cuadrados (vmodulo) }
   vmodulo := sqrt ( vmodulo ) ;
```

```
{  normalizar_vector: }
   FOR j := 1 TO maxelem DO
{      normalizar elemento i de vector:}
       vector [ j ] := vector [ j ] / vmodulo ;
{  escribir_vector: }
   FOR j := 1 TO maxelem DO write ( vector [ j ] : 6 : 3 ) ;
END .
```

Programa PASCAL 5.5: normalizar un vector de reales con el criterio de hacer su módulo igual a la unidad.

5.5.4 Programa C.

```
/*******************************/
/* PROGRAMA normalizar */
/*******************************/

#include <stdio.h>
#include <math.h>

#define MAXELEM  14

main ( )
{
   float  vector [ MAXELEM ];
   int    j;
   float  vmodulo;

/* Leer_vector:*/
   for ( j = 1 ; j <= MAXELEM ; j++ ) scanf ( "%f", &vector [ j - 1] );

/* Calcular módulo:*/
/* Inicializar suma de cuadrados: vmodulo := 0.;*/
   vmodulo =  0.0;

   for ( j = 1 ; j <= MAXELEM ; j++ )
/*     Acumular a vmodulo el elem. j de vector al cuadrado*/
       vmodulo = vmodulo + sqrt ( vector [ j - 1 ] );
/* módulo := raíz cuadrada de suma de cuadrados ( vmodulo ) */
   vmodulo =  sqrt ( vmodulo );
```

```
/* normalizar_vector:*/
   for ( i = 1 ; i <= MAXELEM ; i++ )
/*     normalizar elemento i de vector:*/
       vector [ i - 1 ] = vector [ i - 1 ] / vmodulo;
/* escribir_vector:*/
   for ( j = 1 ; j <= MAXELEM ; j++ )
       printf ("\n%f",vector [ j - 1 ] );
}
```

Programa C 5.5: normalizar un vector de reales con el criterio de hacer su módulo igual a la unidad.

5.5.5 Ejercicios de aplicación

a).- Programa que lee dos números de hasta 20 cifras y los suma. Los números estarán separados por espacios en blanco y/o cambios de línea.

b).- Cálculo de los polinomios derivados de uno original hasta el de grado 0. (Incluye lectura del polinomio original y escritura de los resultados).

c).- Dibujar un Diagrama de Barras Verticales: se debe leer la lista de las longitudes de las barras del diagrama, que acaba con un valor negativo.

d).- Calcular la moda de un conjunto de números. La moda es el valor más repetido de la lista.

5.6 Matrices

5.6.1 Definiciones

Por matriz entendemos una estructura de datos en la que podemos almacenar tantos datos como queramos, todos ellos del mismo tipo, el tipo básico, ordenados en cualquier número de dimensiones. Podemos decir que es una generalización del tipo vector, aunque, como veremos, es mucho más lógico pensar que los vectores son un caso particualr de una matriz.

Las matrices se declaran empleando una sintaxis muy similar a la de los vectores, añadiendo simplemente tantos rangos de valores como dimensiones tenga la estructura matricial que estemos definiendo:

<tipomatriz> = MATRIZ [<tipoíndice>[, <tipoíndice>]$_0^n$] **DE <tipoelemento>**

Así pues, una matriz bidimensional (de dos dimensiones) tendrá dos rangos de valores <tipoíndice>, una tridimensional tres, etc., todos ellos separados por comas.

Esta declaración es equivalente a la empleada en Pascal, en la que un vector es un caso particular de matriz unidimensional. En C, en cambio, una matriz bidimensional es un vector de vectores, es decir, un vector cuyos elementos son vectores, esto es, una generalización del tipo vector. Pero ésta es una sutileza, que no tiene consecuencias en la referenciación convencional de los elementos de una matriz. A continuación veremos un par de ejemplos en los que se muestra la diferencia entre estas dos formas de declarar una matriz:

Ejemplo 1: Matriz bidimensional:

```
TYPE coordenada = ( X , Y , Z ) ;
VAR segmento : ARRAY [ 1 .. 2 , coordenada ] OF REAL ;

segmento [ 1, X ] := 3.45 ;
read ( segmento [ 2 , Z ] ) ;
```

Ejemplo 2: Matriz como vector de vectores:

```
TYPE coordenada = ( X , Y , Z ) ;
   punto = ARRAY [coordenada ] OF real ;
VAR segmento : ARRAY [ 1 .. 2 ] OF punto ;

segmento [ 1 ] [ X ] := 3.45 ;
read ( segmento [ 2 ] [ Z ] ) ;
```

5.6.2 Algoritmo ejemplo.

A continuación vamos a ilustrar la utilización de la estructura de datos tipo matriz mediante el cálculo del producto de dos matrices.

Sean las dos matrices bidimensionales A (nmaxfa, nmaxca) y B (nmaxfb, nmaxcb), siendo:

nmaxfa: Número máximo de filas de la matriz A.
nmaxca: Número máximo de columnas de la matriz A.
nmaxfb: Número máximo de filas de la matriz B.
nmaxcb: Número máximo de columnas de la matriz B.

Para que dos matrices A y B se puedan multiplicar es necesario que el número de columas de la matriz A sea igual al número de filas de la matriz B, es decir:

nmaxca = nmaxfb = nmax

La matriz producto C (nmaxfa, nmaxcb) se calcula mediante la siguiente expresión:

$$C(i,j) = \sum_{k=1..nmax} A(i,k) * B(k,j) \qquad 1 \le i \le nmaxfa \quad y \quad 1 \le j \le nmaxcb$$

```
ALGORITMO ProductodeMatrices;

BEGIN
   Lectura de los datos de la matriz A;
   Lectura de los datos de la matriz B;
   SI número última columna matriz A <> número última fila matriz B
      ENTONCES Escribir "Las matrices no se pueden multiplicar"
      SINO COMIENZO { Cálculo del producto de matrices A y B }
            PARA i := 1 HASTA última fila matriz A HACER
               PARA j := 1 HASTA última columna matriz B HACER
{                 Cálculo del elemento de la matriz C ( i, j ) }
                  PARA k := 1 HASTA última columna matriz A HACER
                        C ( i, j ) := Acumular el producto A ( i, k ) * B ( k, j );
            FIN;
   Escritura de la matriz producto C;
END.
```

Algoritmo 5.8: cálculo del producto de dos matrices.

5.6.3 Programa PASCAL.

```
{ Cálculo del producto de dos matrices }

PROGRAM ProductoMatrices ;

CONST    nmaxfa = 10; nmaxca = 20 ;
         nmaxfb = 20; nmaxcb = 30;

VAR      matriza: ARRAY [ 1..nmaxfa, 1..nmaxca ] OF INTEGER;
         matrizb: ARRAY [ 1..nmaxfb, 1..nmaxcb ] OF INTEGER;
         matrizc: ARRAY [ 1..nmaxfa, 1..nmaxcb ] OF INTEGER;
```

```
            i, j, k : INTEGER ;

BEGIN
{   Lectura de los datos de la matriz A }
{   Lectura de los datos de la matriz B }
    IF nmaxca <> nmaxfb
        THEN writeln ( "Las matrices no se pueden multiplicar  ")
        ELSE BEGIN  { Cálculo del producto de las matrices A y B }
                writeln ( "El producto de matrices es " );
{               PARA i := 1 HASTA última fila matriz A HACER }
                FOR i := 1 TO nmaxfa DO
{                   PARA j := 1 HASTA última columna matriz B HACER }
                    FOR j := 1 TO nmaxcb DO
                        BEGIN
                        matrizc [ i, j ] := 0;
{                       Cálculo del elemento de la matriz C ( i, j ) }
{                       PARA k := 1 HASTA última columna matriz A ó última fila matriz B
HACER}
                        FOR k := 1 TO nmaxca DO
{                           C ( i, j ) := Acumular el producto A ( i, k ) * B ( k, j ) }
                            matrizc [ i, j ] := matrizc [ i, j ] + matriz [ i, k ] * matrizb [ k, j ];
                        END;
                END;
{   Escritura de la matriz producto C }
END.
```

Programa PASCAL 5.6: cálculo del producto de dos matrices.

5.6.4 Programa C

```
/*************************************/
/* PROGRAMA producto_matrices */
/*************************************/

#include <stdio.h>

#define NMAXFA = 10;
#define NMAXCA = 20;
#define NMAXFB = 20;
#define NMAXCB = 30;
```

```
{
   int matriza [ NMAXFA ] [ NMAXCA ];
   int matrizb [ NMAXFB ] [ NMAXCB ];
   int matrizc [ NMAXFA ] [ NMAXCB ];
   int i, j, k;

/* Lectura de los datos de la matriz A */
/* Lectura de los datos de la matriz B */
   if ( nmaxca <> nmaxfb )
      printf ( "Las matrices no se pueden multiplicar " );
   else {  /* Cálculo del producto de las matrices A y B */
/*    PARA i := 1 HASTA última fila matriz A HACER */
      for ( i = 0; i < nmaxfa; i = i + 1 ) {
/*       PARA j := 1 HASTA última columna matriz B HACER COMIENZO */
         for ( j = 0; j < nmaxcb; j = j + 1 ) {
            matrizc [ i ] [ j ] = 0;
/*          Cálculo del elemento de la matriz C ( i, j ) */
/*          PARA k := 1 HASTA última columna matriz A ó última fila matriz B HACER */
            for ( k = 0; k < nmaxca; k = k + 1 )
/*             C ( i, j ) := Acumular el producto A ( i, k ) * B ( k, j ) */
               matrizc [ i ] [ j ] = matrizc [ i ] [ j ] + matriz [ i ] [k ] * matrizb [ k ] [ j ];
         };
      };
   };
/* Escritura de la matriz producto C */
}
```

Programa C 5.6: cálculo del producto de dos matrices.

5.6.5 Ejercicios de aplicación.

a).- Lectura de una matriz y determinación de sus puntos de silla: elementos cuyo valor es el mínimo de su fila y el máximo de su columna.

b).- Programa para calcular el determinante de una matriz 3 * 3 .

c).- Programa para resolver un sistema de tres ecuaciones mediante la Regla de Cramer.

6. Metodología de diseño.

6.1 Definiciones.

Hasta ahora hemos propuesto la resolución de problemas muy sencillos, pues el objetivo era simplemente practicar la utilización de las acciones estructuradas permitidas en la descripción de algoritmos. En la vida real, los problemas no son tan sencillos, y en este capítulo veremos la forma de reducir esos problemas complejos a un nivel de dificultad equivalente al de los vistos hasta ahora, es decir, a un nivel de dificultad que sepamos resolver.

Para conseguirlo se pueden aplicar dos técnicas generales:

- **diseño descendente**, consistente en ir **descomponiendo** sucesivamente el algoritmo general en algoritmos progresivamente más sencillos.

- **diseño ascendente**, consistente en ir **componiendo** el algoritmo general a partir de algoritmos sencillos al principio, y progresivamente más complicados.

Este ultimo método es el dual del anterior, y se emplea combinado con el anterior. Para hacerlo, cuando descompongamos un algoritmo complicado en algoritmos más sencillos, intentaremos que algunos de éstos coincidan con los algoritmos más sencillos que ya sabemos describir o tenemos descritos en problemas anteriores. En las próximas secciones de este capítulo veremos con detalle cómo aplicar estos dos métodos de diseño.

6.1.1 Diseño descendente: Descomposición.

Por diseño descendente entendemos una técnica de diseño de algoritmos, que nos permite **aproximarnos** sucesivamente a la solución definitiva del problema que pretendemos resolver.

Cuando la dificultad del problema no hace evidente la descripción de la solución definitiva, se hace necesaria la realización de una **primera descripción** del algoritmo, en la

que descompondremos el problema planteado en un **reducido número de acciones**, más sencillas. Con ello, de hecho, estamos descomponiendo el problema de describir una acción complicada, en tantos problemas como acciones hayamos empleado en su primera descripción, a cada una de las cuales corresponderá un algoritmo más sencillo que el original.

Esta descomposición se puede realizar empleando dos técnicas fundamentales:

- **funcional o modular**, en la que cada una de las acciones en las que hemos descompuesto el problema original se corresponde con una función relativamente independiente, que además podremos utilizar en la resolución de otros problemas similares. Por ejemplo, en la descripción de un problema de tratamiento de vectores, podemos emplear funciones para leer o escribir el vector, que definiremos una sola vez, y que podremos emplear siempre que las necesitemos. También podemos definir funciones más complicadas, como ordenar los datos de un vector, o multiplicar o sumar dos de ellos, etc. A esta forma de descomposición también se la denomina modular, porque cada función puede ser en realidad un módulo, es decir, un conjunto de programas y funciones empleados en la descripción de la función constituyente del módulo.

- **secuencial**, cuando la descomposición se reduce a enumerar la **lista de acciones** que se deberán ejecutar **en secuencia**, para conseguir el objetivo final del algoritmo propuesto.

6.1.2 Programación modular.

Como ya se ha dicho, las descripciones de acciones complejas realizadas mediante acciones más sencillas pueden constituir módulos, al traducir los algoritmos a programas. Cuando empleamos esta técnica de descomponer, no tan sólo el algoritmo en funciones o módulos, sino que esta descomposición se refleja en el programa resultante, que también queda descompuesto en módulos, decimos que estamos empleando un método de **programación modular**. En este método de programación, las acciones descritas mediante algoritmos más sencillos pueden traducirse a programas independientemente del resto del algoritmo, constituyendo un **proceso** o **procedimiento**. En un módulo o programa podemos tener uno o más procesos o procedimientos.

Dedicaremos el próximo capítulo al estudio de la forma de declarar los procedimientos. En este capítulo tan solo nos centraremos en la técnica de descomposición de un algoritmo en módulos o funciones. En ocasiones, la complejidad del problema a resolver requiere el empleo de técnicas gráficas para ilustrar la descomposición del algoritmo en módulos o procesos. En la siguiente figura (Fig. 6.1) podemos ver un ejemplo

genérico de la forma de descomponer un procedimiento principal (PP) en varios procesos (P1 .. P3) y éstos a su vez en sub-procesos SP11 .. SP33).

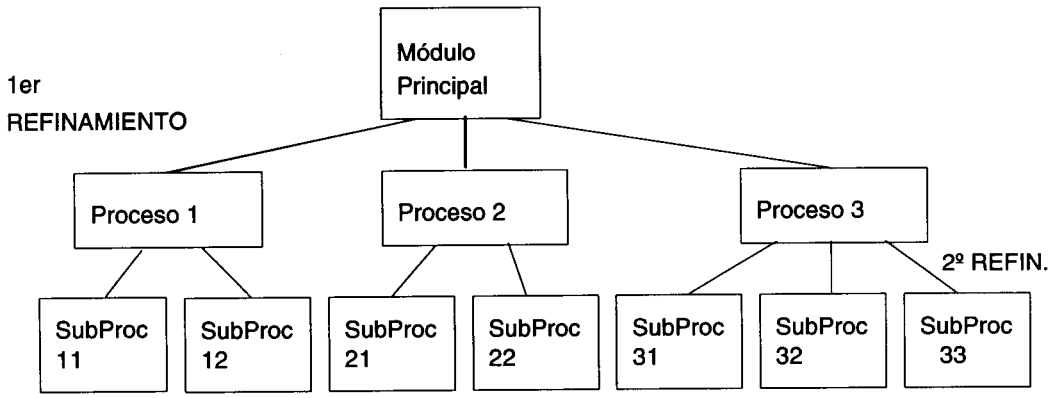


Fig. 6.1: descomposición modular de un proceso.

La descomposición mostrada en esta figura es de tipo secuencial, pero existen una serie de convenciones para indicar otros tipos de sentencias estructuradas, tal como muestran las figuras siguientes Fig. 6.2 y Fig. 6.3.

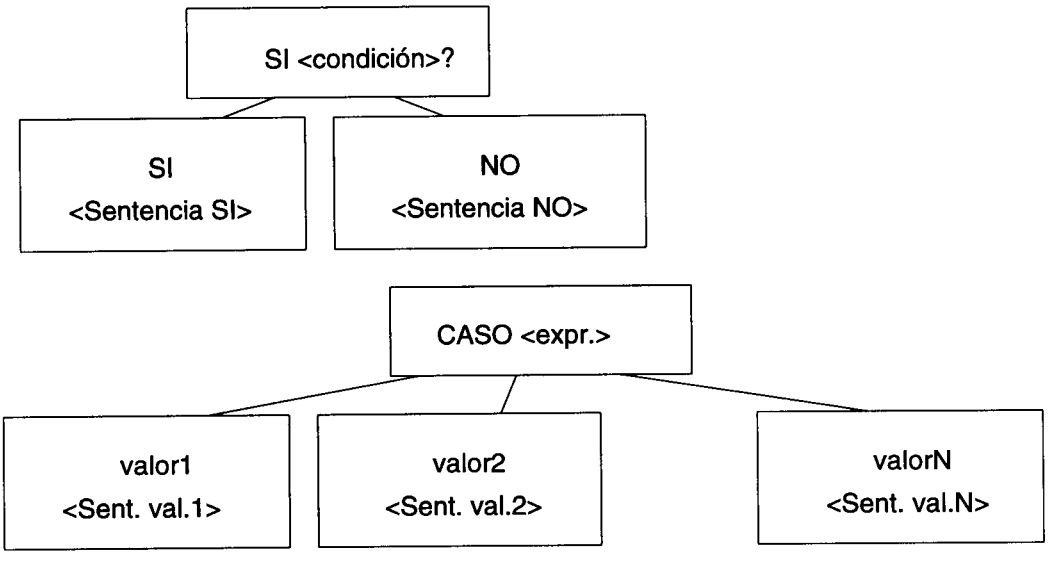


Fig. 6.2: descomposición modular de las sentencias selectivas.

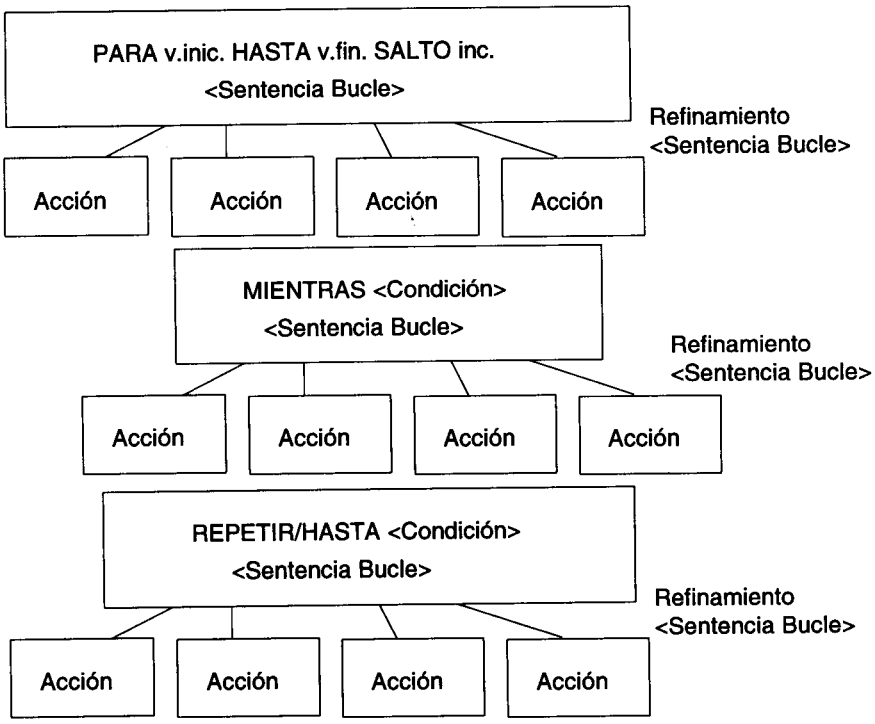


Fig. 6.3: descomposición modular de las sentencias iterativas.

6.1.3 Refinamiento.

En los pasos sucesivos del diseño descendente, llamados **refinamientos**, las acciones en las que se habrá ido descomponiendo el problema original, serán descritas (**refinadas**) a su vez, en acciones cada vez más sencillas, hasta llegar al nivel de detalle de las acciones empleadas en los sencillos algoritmos utilizados como ejemplo en los capítulos precedentes.

Esta metodología de diseño se denomina **descendente**, porque en cada descripción sucesiva del algoritmo se "desciende" en el nivel de detalle de la descripción. Este "descenso" en el nivel de descripción se consigue descomponiendo (refinando) las acciones "compuestas" de las primeras descripciones en sentencias estructuradas (iteraciones, selecciones) o en secuencias de sentencias, que "explican" los pasos a seguir para conseguir realizar la acción compuesta.

6.1.4 Módulo.

Con todo lo dicho, podemos abordar la tarea de definir un módulo de una forma más rigurosa, diciendo que un módulo es una acción refinable, es decir, demasiado compleja para ser realizada directamente por el ordenador, sin ser previamente desglosada en acciones más elementales.

Como toda acción o sentencia de un algoritmo, un módulo tiene un solo punto de entrada y uno solo de salida, y su ejecución se descompone en un número finito de acciones más sencillas.

La ejecución de un módulo no tiene por qué ser siempre idéntica, pues entonces no correspondería a la definición estricta de un programa, sino que en general dependerá de unos datos denominados **parámetros** formales:

- Los parámetros de los que depende la ejecución del algoritmo se denominan **parámetros de entrada**. Una característica importante de los parámetros de entrada es que la ejecución del módulo no puede modificar su valor original. Para que esto pueda suceder es preciso que el programa principal, que referencia (invoca) este módulo, le entregue como parámetros de entrada los valores actuales de las variables de las que debe depender la ejecución del módulo. Estos valores actuales de los parámetros de entrada se denominan **argumentos**.

- En contraposición a éstos existen los denominados **parámetros de salida**, a través de los cuales el módulo entrega los resultados de su ejecución, es decir, que después de ejecutarse el módulo su valor podrá haber sido modificado por éste. Para que esto pueda suceder es preciso que el programa principal, que referencia (invoca) este módulo, le entregue como parámetro de salida una referencia de las variables en las que el módulo debe entregar el resultado de su ejecución. En general, los parámetros de salida también pueden utilizarse como parámetros de entrada, en realidad, son **parámetros de entrada/salida**.

Al traducir un algoritmo a programa, un módulo puede traducirse de varias formas, según las posibilidades ofrecidas por el lenguaje. Así, podemos emplear:

- **procedimiento**: es un módulo que puede tener un número cualquiera de parámetros de entrada y/o salida.

- **función**: es un caso particular de procedimiento, en el que uno de los parámetros es exclusivamente de salida y su valor se puede emplear directamente en una expresión, tal como sucede con las funciones matemáticas convencionales.

- **unidad**: Unit, es el nombre que reciben los módulos en el Turbo Pascal.

En el próximo capítulo veremos ejemplos de realización de procedimientos y funciones, pero en éste nos limitaremos a ir traduciendo las acciones más complejas de los algoritmos en otras más sencillas.

6.2 Algoritmo ejemplo.

Para introducir al lector en la metodología descendente emplearemos un ejemplo muy sencillo, en el que refinaremos tan solo una acción, un par de veces.

Esquema inicial de un algoritmo para calcular la suma de los números primos comprendidos entre dos límites dados por el usuario en cada ejecución del algoritmo:

```
ALGORITMO suma_primos;

VAR mínimo, máximo, suma: ENTERO;

PRINCIPIO
   leer ( mínimo, máximo );
   Inicializar suma;
   PARA todos los números comprendidos entre mínimo y máximo HACER
      SI el número es primo  ENTONCES sumarlo a suma;
   escribir ( suma )
FIN.
```

Algoritmo 6.1: primer refinamiento del algoritmo para calcular la suma de los números primos comprendidos entre dos límites, que se leen como datos.

En la figura 6.4 se muestra la descomposición modular inicial del algoritmo anterior.

Como refinamiento de la sentencia **"SI el número es primo ENTONCES ..."**, emplearemos la propia definición de número primo:

SI número no tiene ningún divisor (factor) distinto de 1 o él mismo ENTONCES...

Ahora debemos refinar la comprobación de que **número** "no tiene ningún divisor...". Para ello, deberemos describir una iteración en la que se vaya verificando si alguno de los números (factor) comprendidos entre 2 y el propio **número** es divisor de éste:

```
Inicializamos la comparación con factor := 2;
MIENTRAS número no sea divisible por factor HACER factor := factor + 1;
```

SI factor = número ENTONCES...

*Algoritmo 6.2: refinamiento de la acción de determinar **SI un número es primo**.*

NOTA: Obsérvese que dentro de la sentencia MIENTRAS tan sólo se incrementa factor de 1 en 1 y que, por tanto, siempre encontraremos un factor, tal que el resto de dividir **número** por éste (factor) sea 0. Si número tiene divisores, será el menor de éstos y, si no los tiene, será el propio **número**. Por tanto, si el divisor de **número** más pequeño encontrado (distinto de 1) es el propio **número**, éste es primo.

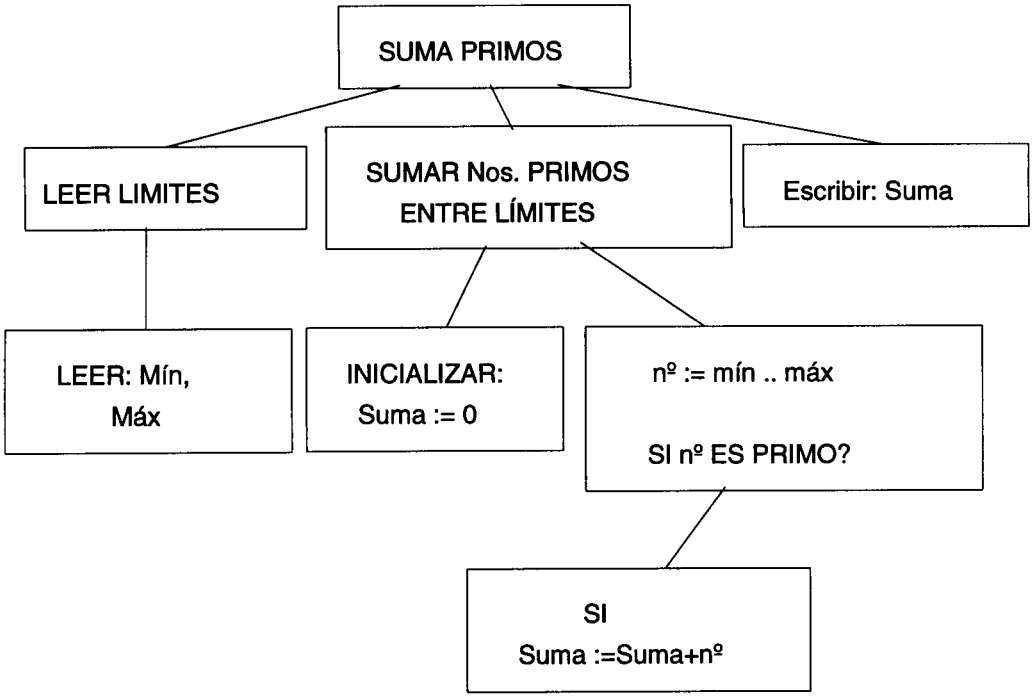


Figura 6.4: primera descomposición modular del algoritmo para calcular la suma de los números primos comprendidos entre dos límites, que se leen como datos.

Este algoritmo va comprobando todos y cada uno de los números comprendidos entre 2 y **número**, hasta encontrar uno que sea divisor de **número**. Evidentemente este algoritmo se puede mejorar, reduciendo el número de comprobaciones. Para ello podemos realizar dos modificaciones:

- la primera es muy sencilla y consiste en dividir por dos el número de comprobaciones a ralizar, simplemente no comprobando los números pares (excepto del 2). Para ello, el bucle PARA debe empezar por 3 y saltar de 2 en 2.

- la segunda es algo más compleja y consiste en reducir las comprobaciones hasta la raíz cuadrada de **número**, pues si éste tiene divisores enteros, el menor de ellos será como máximo la raíz cuadrada del mismo, correspondiendo al caso peor, es decir, al que número sea cuadrado perfecto y que sólo tenga un divisor, precisamente su raíz cuadrada.

La primera modificación implica que el bucle para detectar si número tiene algún divisor, sólo tiene que ejecutarse para los números impares. La segunda modificación implica que al final del bucle factor nunca será igual a número, tanto si hemos encontrado un divisor de éste como si no. Todo ello hace que sea más sencillo emplear una variable booleana **primo**, en la que memoricemos si se cumple alguna de las condiciones que hacen que número no sea primo. De esta forma, la verificación de si número es primo se reducirá a comprobar si el valor de esta variable es cierto o falso.

El nuevo refinamiento de **"SI número es primo ..."** es:

```
SI número es par y no es 2
    ENTONCES Primo := FALSO
    SINO ver si número tiene divisor entero:
        Inicializar bucle: Factor := 3; Primo := CIERTO;
        REPETIR
            SI número es divisible por Factor
                ENTONCES Primo := FALSO
                SINO Factor := Factor + 2;
        HASTA Factor > sqrt ( número ) o Primo = FALSO;
SI Primo ENTONCES ...
```

*Algoritmo 6.3: refinamiento de la acción de determinar **SI un número es primo**.*

NOTA: "SI Primo ENTONCES" es equivalente a "SI Primo = CIERTO ENTONCES".

En la figura siguiente (Fig. 6.5) se muestra la descomposición modular detallada de la acción de determinar **si un número es primo** correspondiente al algoritmo anterior.

A continuación veremos la descripción del nivel más detallado del algoritmo completo:

```
ALGORITMO suma_primos;
VAR mínimo,máximo,suma : ENTERO;
```

```
PRINCIPIO
   leer ( mínimo, máximo );
   Inicializar suma := 0;
   PARA todos los números comprendidos entre mínimo y máximo HACER
   PRINCIPIO
      SI número es par y no es 2
            ENTONCES primo := FALSO
            SINO ver si número tiene divisor entero:
                  PRINCIPIO
                        Inicializar bucle: factor := 3;  primo := CIERTO;
                        REPETIR
                              SI número es divisible por Factor
                                    ENTONCES primo := FALSO
                                    SINO factor := factor + 2;
                        HASTA factor > sqrt ( número ) o primo = FALSO;
```

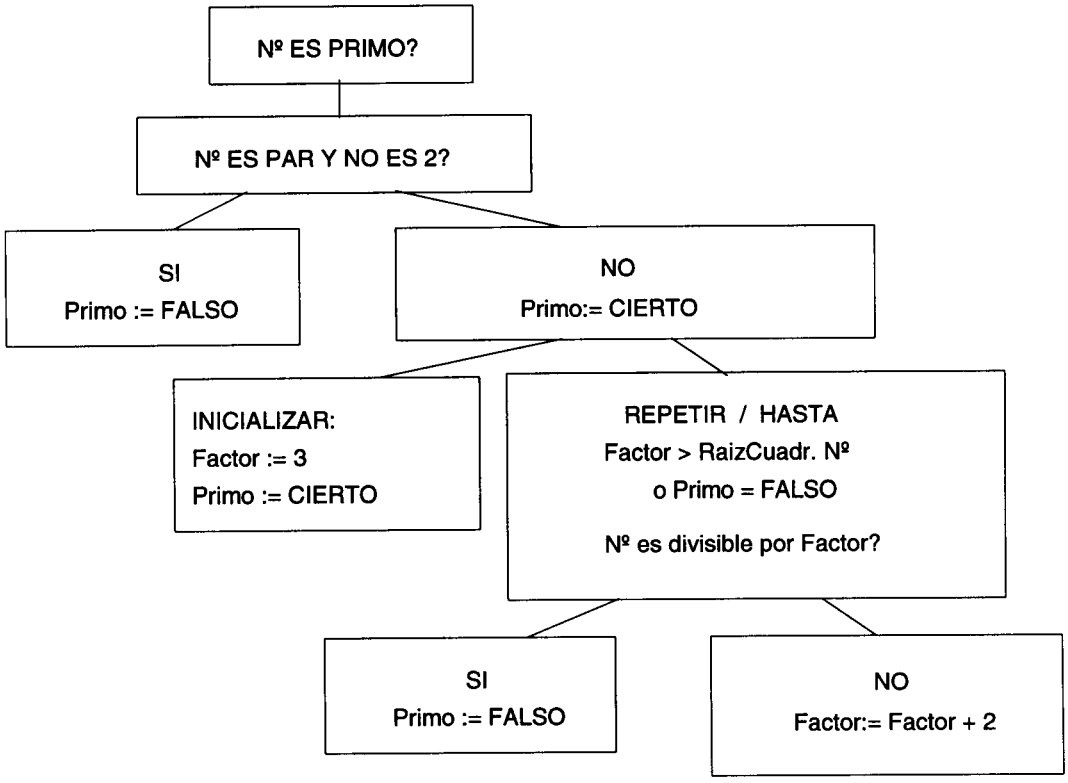


Figura 6.5: descomposición modular detallada de la acción de determinar ***SI un número es primo.***

```
                FIN;
        SI Primo ENTONCES sumar número a suma;
    FIN;
    escribir ( suma )
FIN.
```

Algoritmo 6.4: algoritmo detallado para calcular la suma de los números primos comprendidos entre dos límites.

NOTA: Las acciones consideradas demasiado complejas, se han refinado mediante una lista de acciones más sencillas (abierta con ":").

6.3 Programa PASCAL.

A partir de este punto de esta publicación, tanto la mayor complejidad de los algoritmos descritos, como los conocimientos ya adquiridos por el lector, recomiendan reducir los comentarios de los programas en los que se traducen los algoritmos ejemplo, a los mínimos imprescindibles para la comprensión de éstos.

En general, evitaremos poner como comentario una acción del algoritmo que se ha traducido en una sola sentencia del programa resultante, pues se supone que el nivel de comprensión de ambas es el mismo. Solo cuando la acción del algoritmo tenga una descripción semánticamente esclarecedora la incluiremos en el programa. Por ejemplo las sentencias de "inicialización de sentencias de iteración", o cuando no sea evidente la razón por la que realizamos aquella sentencia, sin conocer su descripción a nivel de algoritmo.

```
PROGRAM suma_primos;

TYPE positivo = 0..maxint;
VAR  numero : positivo;
     minimo,maximo: positivo;
     suma , factor : positivo;
     primo: BOOLEAN;

BEGIN
   readln ( minimo, maximo );
   suma := 0;
{  PARA todos los números entre mínimo y máximo HACER }
   FOR  numero := minimo TO maximo  DO
   BEGIN
{      SI el número es par y no es 2 ENTONCES  }
```

```
        IF even ( numero ) AND ( numero <> 2 )
                THEN primo := FALSE
{               SINO ver si número tiene divisor entero: }
                ELSE BEGIN
{                       Inicializar bucle: }
                        factor := 3;
                        primo := TRUE;
                        REPEAT
{                               SI número es divisible por Factor}
                                IF ( numero MOD factor ) = 0
                                        THEN  primo := FALSE
                                        ELSE  factor := factor + 2
{                       HASTA Factor > sqrt ( número ) o Primo = FALSO }
                        UNTIL factor > sqrt ( numero ) OR primo = FALSE
                        END;
                IF primo THEN suma := suma + numero;
    END;
    write ( " La suma es ", suma ) ;
END.
```

Programa PASCAL 6.1: suma de los números primos comprendidos entre dos límites.

6.4 Programa C.

```
/*********************/
/* PROGRAMA primos */
/*********************/

#include <stdio.h>
#include <math.h>

typedef int boolean;
#define TRUE 1
#define FALSE 0

main ( )
{
    int     numero, minimo, maximo, suma, factor;
    boolean primo;
```

```
    scanf ( "%d %d", &minimo, &maximo );
    suma = 0;
/*  PARA todos los números entre mínimo y máximo HACER*/
    for ( numero = minimo; numero <= maximo; numero++ ) {
/*      SI el número es par y no es 2 ENTONCES */
        if ( ( fmod ( numero, 2 ) == 0 ) && ( numero != 2 ) )
                primo = FALSE
/*      SINO ver si número tiene divisor entero: */
        else {
/*              Inicializar bucle: */
                factor = 3;
                primo = TRUE;
                do {
/*                      SI número es divisible por Factor*/
                        if ( fmod ( numero, factor ) == 0 )
                                primo = FALSE;
                        else
                                factor = factor + 2;
/*              HASTA Factor > sqrt(número) o Primo = FALSO */
                }while ( ( factor <= sqrt ( numero ) ) && primo != FALSE );
        }
        if ( primo == TRUE )
                suma = suma + numero ;
    } printf ( "La suma es %d", suma );
}
```

Programa C 6.1: suma de los números primos comprendidos entre dos límites.

6.5 Ejercicios de aplicación.

a).- Obtener la descomposición en factores primos de un número dado.

b).- Reducir a fracciones simples una serie de fracciones. Los datos son parejas de valores que representan el numerador y el denominador, respectivamente, de cada fracción. El final de los datos se marca con un número negativo. Imprimir la fracción original y la simplificada.

c).- Calcular si un número entero no negativo leído como dato tiene algún amigo. Se dice que dos números son amigos si cada uno de ellos es igual a la suma de los divisores del otro. Si un número es igual a la suma de sus divisores (será amigo de sí mismo) se

dice que es un número perfecto. El 1 es divisor de cualquier número, pero un número no se considera divisor de sí mismo. Los datos contienen un número por línea. El último dato es negativo.

7. Procedimientos y funciones.

7.1 Definiciones.

En el capítulo anterior hemos visto cómo descomponer un algoritmo en acciones complejas que, a su vez, requerían un refinamiento en otras más sencillas. Estas acciones, que podemos calificar de "refinables", pueden ser directamente desglosadas en el algoritmo general, obteniendo al final un algoritmo "monolítico", o bien puede dejarse su declaración al margen del algoritmo general, contribuyendo de esta forma a mejorar la legibilidad de éste, y haciendo que el algoritmo final conserve la estructura seguida para su diseño.

En este último caso, diremos que las acciones, cuyo refinamiento se deja al margen del algoritmo principal, constituyen **procedimientos** o **funciones**. Los **procedimientos** simplemente realizan una acción sobre unos determinados parámetros. Las **funciones**, además, calculan un valor, que se puede emplear directamente en una expresión, como cualquier variable. Con esta técnica podemos construir nuestras propias bibliotecas de funciones y procedimientos, en las que almacenaremos las que consideremos que nos pueden servir en la redacción de futuros algoritmos. Además, todos los compiladores ofrecen una biblioteca de funciones y procedimientos creados por el fabricante, además de las predefinidas en el lenguaje. Nosotros ya estamos acostumbrados a utilizar algunas de éstas, como: leer (read, scanf), escribir (write, printf), raíz cuadrada (sqrt), cuadrado (sqr), etc.

El algoritmo con el que se describe un procedimiento o función tiene una estructura igual a la de cualquiera de los algoritmos hasta ahora descritos, con su cabecera, sus declaraciones de variables y su cuerpo, y con las acciones en las que se refina. Las variables, procedimientos y funciones declarados dentro de un procedimiento se denominan **locales**, para distinguirlos de los declarados a nivel del algoritmo global, **variables globales**. La diferencia fundamental respecto a un algoritmo general (no empleable como procedimiento) radica en la posibilidad de intercambiar información con el algoritmo que invoca la acción representada por este procedimiento, **algoritmo invocante**. Este intercambio de información se realiza a través de unas variables declaradas en la cabecera del procedimiento o función, junto al nombre de éste. Estas variables se denominan los **parámetros** del procedimiento o función, y el algoritmo invocante las puede asociar a

cualquiera de sus variables del mismo tipo que estos parámetros. Las variables con las que el algoritmo general invoca el procedimiento o función se denominan **argumentos** de la invocación. Los parámetros pueden ser de dos tipos:

VALOR: son variables cuyo valor, al iniciarse la ejecución del algoritmo, ha sido fijado por el algoritmo invocante. Éste transfiere al procedimiento, como **parámetro de entrada**, el **valor** de una de sus variables, la que ha sido seleccionada como argumento de entrada.

REFERENCIA: son variables cuyo valor, al final de la ejecución del procedimiento o función, puede ser utilizado por el algoritmo invocante. Estas variables no tienen porque haber sido inicializadas por el algoritmo invocante, pues su utilidad es entregar los resultados de la ejecución del algoritmo a la SALIDA de éste. No obstante, puede ser que alguna de estas variables tome un valor inicial del algoritmo invocante, en cuyo caso será un parámetro de ENTRADA/SALIDA. Para que todo esto sea posible, el algoritmo invocante debe transferir al procedimiento o función una **referencia** del argumento de salida, es decir, la **dirección de memoria** en la que se deben almacenar los valores de ese parámetro del procedimiento o función, para que el algoritmo invocante pueda usarlos a través del argumento.

Siempre conviene transferir los vectores y matrices por referencia, pues pueden ser estructuras de datos muy grandes (muchos valores), que se deberían copiar íntegramente a una estructura local del procedimiento si empleásemos el mecanismo de paso de parámetros por valor. Algunos lenguajes de programación, como el C, siempre transfieren las matrices por referencia, aunque el programador no lo indique explícitamente.

Así pues, podemos resumir diciendo que dentro del cuerpo del algoritmo de un procedimiento o función se pueden referenciar como variables: las **locales** del procedimiento y los parámetros de éste. Algunos lenguajes de programación permiten también referenciar variables globales, pero constituye una técnica que sólo aporta dudosas ventajas en algunos casos, y que siempre constituye una técnica de programación peligrosa, porque puede provocar los denominados **efectos colaterales**, de consecuencias difícilmente previsibles.

7.2 Algoritmo ejemplo.

Para este capítulo hemos seleccionado un ejemplo en el que podremos ver la utilidad de convertir los refinamientos de las acciones complejas en procedimientos y funciones.

El algoritmo propuesto calcula la longitud media de las palabras de una frase acabada en punto, contando las palabras que va encontrando y sumando sus longitudes. La media se calculará simplemente dividiendo estas dos cantidades.

```
ALGORITMO long_media_palabras;

{Este programa calcula la longitud media de las palabras en una frase acabada en un punto }

CONST    maxlong = 50;
TIPO     palabras = MATRIZ [ 1..maxlong ] de CAR;
VAR      frase: palabras;
         i, long, sum_long , media, num_long : ENTERO;

PRINCIPIO
   Leer Frase;
   Inicializar Bucle:
   num_long := 0;
   sum_long := 0;
   Puntero a carácter de frase: i := 1;
   MIENTRAS Quede palabra en frase, a partir del carácter i, de long > 0 HACER
      PRINCIPIO
         Incrementar num_long;
         Acumular long a sum_long
      FIN;
   Calcular la media de las longitudes acumuladas;
   Escribir ( 'suma longitudes ', sum_long, 'numero longitudes ', num_long );
   Escribir ( 'la longitud media es ', media )
FIN.
```

Algoritmo 7.1: algoritmo general para el cálculo de la longitud media de las palabras de una frase acabada en punto.

En este algoritmo hay varias acciones que requieren un refinamiento, porque su complejidad no permite su traducción directa a un lenguaje de programación como Pascal. El refinamiento de la inicialización del bucle ya lo hemos indicado en la primera descripción del algoritmo, pues su sencillez no requiere mayor explicación. Así pues, comenzaremos con el refinamiento de la acción para la lectura de una frase, acabada en punto:

```
REFINAMIENTO Leer Frase;

VAR CuentaDeCaracteres : ENTERO;

PRINCIPIO
   CuentaDeCaracteres := 0;
   REPETIR
```

```
        Incrementar CuentaDeCaracteres;
        Leer siguiente carácter de frase;
    HASTA Carácter leído = '.';
FIN.
```

Algoritmo 7.2: refinamiento de la acción de leer una frase acabada en punto.

La traducción de este algoritmo a un lenguaje de programación aún no es inmediata, pero el siguiente nivel de refinamiento requiere tener en cuenta la estructura de datos en la que se almacenarán los caracteres de la frase: una matriz de caracteres, como ya se ha anticipado en la definición de tipos del algoritmo general. Teniendo esto en cuenta, el refinamiento anterior puede convertirse a una forma menos legible para nosotros, pero más facilmente interpretable por un ordenador:

```
REFINAMIENTO Leer Frase;

VAR CuentaDeCaracteres : ENTERO;

PRINCIPIO
    CuentaDeCaracteres := 0;
    REPETIR
        Incrementar CuentaDeCaracteres;
        Leer ( frase [ CuentaDeCaracteres ] ) ;
    HASTA frase [ CuentaDeCaracteres ] = '.' ;
FIN.
```

Algoritmo 7.3: refinamiento de la acción de leer una frase acabada en punto.

En el problema que nos ocupa, este algoritmo tan sólo es necesario ejecutarlo una vez, pero la lectura de una frase es una acción muy frecuente, por lo que nos interesará convertir el refinamiento de esta acción en un procedimiento. Para ello basta con que indiquemos en la cabecera del algoritmo los parámetros de los que dependerá cada una de sus ejecuciones, pues las variables empleadas exclusivamente en el refinamiento de esta acción, que en el procedimiento serán variables locales, ya las hemos declarado. En este caso, el único parámetro del procedimiento será la única variable compartida con el resto de las acciones del programa, es decir, la matriz de caracteres en la que se deben ir dejando almacenados los datos leídos:

```
PROCEDIMIENTO Leer Frase ( Texto: Palabras ) ;
```

```
VAR CuentaDeCaracteres : ENTERO;

PRINCIPIO
   CuentaDeCaracteres := 0;
   REPETIR
      Incrementar CuentaDeCaracteres;
      Leer ( Texto [ CuentaDeCaracteres ] ) ;
   HASTA Texto [ CuentaDeCaracteres ] = ' . ' ;
FIN.
```

Algoritmo 7.4: procedimiento que realiza la acción de leer una frase acabada en punto.

La siguiente acción del algoritmo general que requiere un refinamiento es la de determinar si quedan palabras en la frase. Esta acción se emplea para determinar si el bucle debe ejecutarse alguna vez más, en función del resultado de la acción. En el algoritmo general ya se ha indicado que el bucle se ejecutará si la longitud encontrada es mayor que 0. En el refinamiento de esta acción, por tanto, deberemos calcular también la longitud de la palabra siguiente de la frase, pues este dato es necesario dentro del bucle. Además, dentro de esta acción se va incrementando el valor de i, de forma que, al invocar esta acción, siempre apunte al principio de la siguiente palabra de frase, o al punto final, si ya se han analizado todas las palabras de frase (no quedan palabras). Para conseguir que esta condición se cumpla al comenzar la ejecución de la acción, debe cumplirse ya al finalizar su ejecución, pues no hay ninguna otra acción en el algoritmo que modifique i. Por ello, debemos seguir incrementando i hasta que apunte al principio de la palabra siguiente, esto es, "saltar los blancos" que separan el final de la palabra que acabamos de procesar del principio de la siguiente. El primer refinamiento de esta acción puede ser el siguiente:

```
REFINAMIENTO Quede palabra en frase, a partir del carácter i, de long > 0;

VAR queda :BOOLEANO;

PRINCIPIO { i apunta al primer carácter de una palabra o al punto final }
   LongitudPalabraSiguiente := 0;
   MIENTRAS el carácter i de frase pertenezca a una palabra HACER
      PRINCIPIO { i apunta a un carácter de una palabra }
         Incrementar i { para que apunte al carácter siguiente de frase };
         Incrementar la LongitudPalabraSiguiente
      FIN; { i no apunta a un carácter de una palabra }
   Saltar blancos;
{  i apunta al primer carácter de una palabra o al punto final }
```

```
   SI LongitudPalabraSiguiente > 0
      ENTONCES Queda palabra en frase := CIERTO
      SINO Queda palabra en frase := FALSO
FIN.
```

Algoritmo 7.5: refinamiento de la acción de determinar si queda palabra en frase, que empiece en la posición apuntada por i de una frase acabada en punto.

NOTA: Entre llaves { } se han indicado las **condiciones invariantes** que debe cumplir i, siempre que se ejecute el algoritmo, para mostrar la necesidad de la acción "saltar blancos".

Como en el caso de la acción anterior, este refinamiento puede ser descrito de una forma más fácilmente inteligible para un compilador, si introducimos la referencia explícita a los datos en los que se almacena la información:

```
REFINAMIENTO Quede palabra en frase, a partir del carácter i, de long > 0;

PRINCIPIO
   LongitudPalabraSiguiente := 0;
   MIENTRAS frase [ i ] no sea ni 'blanco' ni 'punto' HACER
      PRINCIPIO
         Incrementar i { para que apunte al carácter siguiente de frase };
         Incrementar la LongitudPalabraSiguiente
      FIN;
   Saltar blancos;
   SI LongitudPalabraSiguiente > 0
      ENTONCES Queda palabra en frase := CIERTO
      SINO Queda palabra en frase := FALSO
FIN.
```

Algoritmo 7.6: refinamiento de la acción de calcular la longitud de una palabra, en el que se explícita la referencia a los elementos de la estructura de datos.

En el algoritmo general, esta acción constituye la condición de ejecución del bucle en el que se van acumulando las longitudes de las palabras de la frase. Si en el programa final hacemos aparecer otro bucle, que como hemos visto es necesario para calcular la longitud de cada palabra, los dos bucles consecutivos restarán legibilidad al algoritmo. Por ello es

recomendable dejar la descripción de las acciones, en las que se desdobla la acción del cálculo de la longitud, fuera del algoritmo principal, es decir, como una función. En este caso nos interesa que sea una función, para que su resultado pueda ser utilizado directamente en una expresión (Quede palabra... valga CIERTO). Para convertir el refinamiento realizado en función, hay que especificar los parámetros de los que depende su ejecución y las variables locales, como en los procedimientos y, además, hay que indicar el tipo de la función, es decir, el del dato que se va a devolver como resultado. Además, como en el lenguaje de programación C, el valor devuelto por la función debe indicarse explícitamente en una sentencia especial denominada "return", como veremos en la traducción del algoritmo que estamos desarrollando, vamos a acostumbrarnos a emplear una variable local dentro de la función para realizar los cálculos intermedios, antes de obtener el valor final devuelto por la función y en una sentencia de asignación explícita asignaremos el valor final de esa variable local al nombre de la función. Esta técnica puede ser también útil en la traducción del algoritmo a Pascal, pues las operaciones realizadas con variables locales son mucho más rápidas que las realizadas con el nombre de la función. En este caso concreto, no hay que realizar operaciones intermedias con la variable booleana que devuelve esta función, por lo que el comentario realizado no se ilustra con este ejemplo en concreto, pero ya tendremos ocasiones de ponerlo en práctica en los próximos capítulos. La modificación que sí se ha realizado en esta nueva version del refinamiento de la acción, es la de sustituir la sentencia condicional del final del algoritmo por una simple sentencia de asignación, pues al tener que asignar simplemente un valor booleano a una variable, en función del resultado de una comparación, podemos asignar directamente este resultado a la variable booleana, y en la traducción a C nos ahorraremos dos sentencias "return". Así pues, la expresión de la acción de calcular la longitud de la palabra como acción es la siguiente:

```
FUNCION QuedaPalabra ( texto: palabras; VAR i, LongitudPalabraSiguiente: ENTERO):
BOOLEANO;

PRINCIPIO
   LongitudPalabraSiguiente := 0;
   MIENTRAS frase [ i ] no sea ni 'blanco' ni 'punto' HACER
      PRINCIPIO
         Incrementar i { para que apunte al carácter siguiente de frase };
         Incrementar LongitudPalabraSiguiente ;
      FIN;
   Saltar blancos;
   QuedaPalabra := LongitudPalabraSiguiente > 0
FIN.
```

Algoritmo 7.7: procedimiento que realiza la ación de leer una frase acabada en punto.

Puesto que las acciones que hemos refinado se han convertido en procedimientos y funciones, el algoritmo final para el cálculo de la longitud media de las palabras de la frase podría ser directamente el algoritmo dado como primer refinamiento. A pesar de ello vamos a repetirlo, indicando explícitamente los argumentos con los que se invocan los procedimientos y funciones:

```
ALGORITMO LongMediaPalabras;

{Este programa calcula la longitud media de las palabras en una frase acabada en un punto}

CONST    maxlong = 50;
TIPO     palabras = MATRIZ [ 1..maxlong ] de CAR;
VAR      frase: palabras;
         i, long, NumeroDeLongitudes, media, SumaDeLongitudes : ENTERO;

PRINCIPIO
   Leer Frase ( frase );
   Inicializar Bucle:
      NumeroDeLongitudes := 0;
      SumaDeLongitudes := 0;
   Puntero a carácter de frase: i := 1;

   MIENTRAS QudaPalabra ( frase, i, long ) HACER
      PRINCIPIO
         Incrementar NumeroDeLongitudes;
         Acumular long a SumaDeLongitudes
      FIN;
   media := SumaDeLongitudes / NumeroDeLongitudes;
   Escribir ( 'suma longitudes ', SumaDeLongitudes, 'numero longitudes',
            NumeroDeLongitudes);
   Escribir ( 'la longitud media es ', media )
FIN.
```

Algoritmo 7.8: algoritmo general para el cálculo de la longitud media de las palabras de una frase acabada en punto.

Una vez introducidos los procedimientos y funciones, los algoritmos generales ya podrán incluir, desde el principio, las invocaciones de éstos, de una forma más formal, como en esta última version del algoritmo del ejemplo.

7.3 Programa PASCAL.

```
PROGRAM LongMediaPalabras;

{ Este programa calcula la longitud media de las palabras en una frase acabada en un punto }

CONST     maxlong=50;
TYPE      palabras = ARRAY [1..maxlong] of CHAR;
VAR       frase: palabras;
          i, long, num_long, media, suma_long: INTEGER;

PROCEDURE  leer_frase ( VAR texto: palabras ) ;
VAR i: INTEGER;
BEGIN
{   CuentaDeCaracteres := 0 }
    i := 0;
{   REPETIR Incrementar CuentaDeCaracteres }
    REPEAT
        i := i + 1;
{       Leer Texto [ CuentaDeCaracteres ] }
        read ( texto [ i ] )
{   HASTA Texto[ CuentaDeCaracteres] = ' . ' }
    UNTIL texto [ i ] = ' . ' ;
    readln ;
END;

FUNCTION QuedaPalabra ( VAR texto: palabras; VAR i, long: INTEGER ): BOOLEAN;

BEGIN
{   LongitudPalabraSiguiente := 0;}
    long := 0;
{   MIENTRAS frase [ i ] no sea ni ' blanco'  ni ' punto' }
    WHILE ( texto [ i ] <> ' ' ) AND ( texto [ i ] <> ' . ' ) DO
    BEGIN
{       Incrementar i ( para que apunte al carácter siguiente de frase ) }
        i := i + 1;
{       Incrementar LongitudPalabraSiguiente }
        long := long + 1
    END;
{   Saltar blancos }
    WHILE ( texto [ i ] = ' ' )  DO i := i + 1;
{   QuedaPalabra := LongitudPalabraSiguiente > 0}
```

```
    QuedaPalabra := long > 0
END;

BEGIN {Comienza el programa principal}
    leer_frase ( frase );
{   Inicializar Bucle:}
{       NumeroDeLongitudes := 0 }
{       SumaDeLongitudes := 0 }
{       Puntero a carácter de frase: i := 1 }
    i := 1; num_long := 0; suma_long := 0;
{   MIENTRAS QuedaPalabra ( frase, i, long ) }
    WHILE QuedaPalabra ( frase, i, long ) DO
    BEGIN
{       Incrementar NumeroDeLongitudes;}
        num_long := num_long + 1;
{       Acumular long a SumaDeLongitudes}
        suma_long := suma_long + long
    END;
    media := suma_long DIV ( num_long );
    writeln ( 'suma longitudes = ', suma_long, 'numero longitudes = ', num_long ) ;
    writeln ( 'la longitud media es ', media )
END.
```

Programa PASCAL 7.1: cálculo de la longitud media de las palabras de una frase.

7.4 Programa C.

```
/*****************************************/
/* PROGRAMA long_media_palabras */
/*****************************************/

/*Este programa calcula la longitud media de las palabras en una frase acabada en un punto*/

#include <stdio.h>

#define TRUE 1
#define FALSE 0
#define MAXLONG 50
```

```
typedef     int boolean;
typedef     char palabras [ MAXLONG ];
            palabras frase ;
            int i, long, num_long, media, suma_long;

/* PROCEDIMIENTO LeerFrase ( texto: palabras ) */
void
leer_frase ( palabras  texto ) ;
{
   int i ;

/*  CuentaDeCaracteres := 0 */
   i = 0;
/*  REPETIR Incrementar CuentaDeCaracteres */
   do {
      i =  i + 1;
/*    Leer Texto [ CuentaDeCaracteres ] */
      scanf ( &texto [ i - 1] ); }
/*  HASTA Texto [ CuentaDeCaracteres ] =  '.'  */
   } while ( ! ( texto [ i - 1] ==  '.' ) ) ;
   scanf ( );
}

/* FUNCION QuedaPalabra ( texto: palabras; VAR i, LongitudPalabraSiguiente: ENTERO ):
BOOLEANO; */

boolean
queda_palabra  ( palabras texto, int *i, int *long)

{
   boolean valor_devolucion ;

/*  LongitudPalabraSiguiente := 0 */
   ( *long ) = 0;
/*  MIENTRAS frase [ i ] no sea ni ' blanco'  ni ' punto'  HACER  PRINCIPIO */
   while ( ( texto [ ( *i ) - 1 ] !=  ' ' ) && ( texto [ ( *i ) - 1 ] !=  '.' ) )  {
/*    Incrementar i (para que apunte al carácter siguiente de frase) */
      ( *i ) = ( *i ) + 1;
/*    Incrementar LongitudPalabraSiguiente  */
      ( *long ) = ( *long ) + 1;
   };
/*  Saltar espacios en blanco  */
```

```
    while ( ( texto [ ( *i ) - 1 ] == ' ' ) ) ( *i ) = ( *i ) + 1;
/*  QuedaPalabra := LongitudPalabraSiguiente > 0*/
    valor_devolucion = ( *long ) > 0;
    return ( valor_devolucion ) ;
}

main ( )

{
    leer_frase ( frase ) ;
/*  Inicializar Bucle: */
/*      NumeroDeLongitudes := 0 */
/*      SumaDeLongitudes := 0 */
/*      Puntero a carácter de frase: i := 1 */
    i = 1; num_long = 0; suma_long = 0;
/*  MIENTRAS QuedaPalabra ( frase, i, long )  HACER PRINCIPIO */
    while ( ( queda_palabra ( frase, &i, &long ) ) ) {
/*      Incrementar NumeroDeLongitudes */
        num_long = num_long + 1 ;
/*      Acumular long a SumaDeLongitudes */
        suma_long = suma_long + long ;
/*  FIN */
    } ;
    media = suma_long  /  ( num_long ) ;
    printf ( "suma longitudes = %d", " numero longitudes = %d", suma_long, num_long );
    printf ( "la longitud media es %d\n", media ) ;
}
```

Programa C 7.1: cálculo de la longuitud media de las palabras de una frase acabada en un punto.

Obsérvese que en C los parámetros de procedimientos y funciones cuyo valor se transfiere por referencia se indican precediendo el nombre de la variable de un operador. Al invocarlo desde el programa principal, se le pasa la "dirección del argumento" (ej. &i, &long), para que el procedimiento o función la pueda usar para dejar en la posición de memoria, referenciada por ella, el resultado de salida. Dentro del procedimiento o función, los parámetros de salida (de los que se ha recibido la dirección), se manejan indirectamente, es decir, mediante la fórmula "contenido de ..." (ej. *i, *long), tanto en su declaración en la cabecera, como en las expresiones en las que aparezcan dentro del cuerpo de la función. De esta forma, siempre recordaremos que las modificaciones realizadas sobre estas

variables serán accesibles desde el programa invocante. Los vectores y matrices, siempre se transfieren por referencia, por eso no se indica explícitamente la indirección.

7.5 Ejercicios de aplicación.

a).- Calcular el tamaño de la palabra más larga de una frase almacenada en un vector y acabada con un punto.

b).- Contar el número de veces que se repite una determinada palabra clave en una frase acabada en un punto.

c).- Contar el número de veces que aparece una subcadena (substring) de caracteres en una frase acabada en un punto.

d).- Dado un vector de n palabras clave, separadas por blancos y dos frases acabadas por un punto, contar el número de palabras clave que aparecen en las dos frases.

8. Tipos de datos estructurados.

Los tipos de datos estructurados, como ya hemos dicho al definir los tipos de datos, son aquellos que nos permiten almacenar dentro de una sola variable más de un valor simultáneamente. Esta facultad se puede materializar de tres formas distintas, en tres familias de tipos de datos estructurados:

vectores y **matrices**, en los que se almacena un determinado número de elementos de un tipo básico fijo. En la declaración de un vector o matriz, precisamos el número máximo de elementos que podremos almacenar en él, y el tipo de estos elementos.

registros, en los que se almacena también un determinado número de valores, cada uno de ellos de un tipo potencialmente distinto. En la declaración de un registro precisamos los elementos de que consta y el tipo de cada uno de ellos.

ficheros, en los que se almacena un número indeterminado de elementos, todos ellos del mismo tipo. Es como un vector en cuya declaración no se limita el número máximo de elementos que puede almacenar. Esta indeterminación en el tamaño de la estructura de un fichero, que lo distingue de los vectores, está intrínsecamente ligada a su otra característica esencial, diferenciadora de todos los otros tipos de datos, y es el hecho de albergar los datos en la memoria secundaria del ordenador (disco o cinta magnética), en vez de la memoria principal (semiconductores).

Al introducir los tipos de datos ya hemos descrito ampliamente los vectores y matrices, pues su conocimiento nos ha permitido desarrollar ejemplos suficientemente complejos para poner en práctica la metodología de diseño y los procedimientos y funciones. En este capítulo, por tanto, nos vamos a centrar en los otros dos tipos de datos estructurados: registros y ficheros.

8.1 Registros.

8.1.1 Definiciones.

Como ya hemos dicho, en un registro podemos almacenar datos de cualquier combinación de los tipos estudiados hasta el momento. Los componentes de un registro se denominan **campos** y, dada la irregularidad de la estructura, cada uno de ellos tiene un nombre y tipo propios, que se deben hacer constar en la declaración del registro:

< TipoRegistro > **REGISTRO** [< campo > : < TipoCampo > ;]$_1^n$ **FIN**

< TipoRegistro >, el identificador del tipo de dato que estamos definiendo, en este caso el de un registro.

< campo >, el nombre de cada uno de los campos que componen el registro que estamos definiendo.

< TipoCampo >, el tipo correspondiente a cada uno de los campos del registro. Puede ser de cualquiera de los tipos elementales o estructurados definidos hasta ahora.

De la misma forma que, para referenciar un elemento de una matriz debemos citar el nombre de ésta y los subíndices del elemento referenciado, para operar con un campo de un registro debemos indicar el nombre de éste y el del campo con el que queremos operar, separados por un punto (.).

8.1.2 Algoritmo ejemplo.

Para ilustrar el uso de los registros, vamos a desarrollar un sencillo algoritmo, que simplemente lee los datos personales de un conjunto de personas y después los escribe ordenadamente, con cada uno de los datos encolumnados.

El algorimo no presenta ninguna dificultad, pero ilustra la posibilidad de mezclar en la definición de un tipo de dato matrices y registros:

```
ALGORITMO registro ;

CONST  MAX = 20;
TIPO   cadena = MATRIZ [ 1..MAX ] DE CAR ;
       tablapersonas =
       REGISTRO
          numpersonas : ENTERO;
```

```
            persona : MATRIZ [ 1..100 ]  DE
            REGISTRO
               nombre : cadena;
               apellido : cadena;
               dni : ENTERO;
            FIN;
         FIN;
VAR      tabla : tablapersonas ;
         k : ENTERO;

PRINCIPIO
   Leer el número de personas a introducir en la tabla;
   PARA cada persona HACER  leer sus datos personales;
   PARA cada persona HACER  escribir sus datos personales;
FIN.
```

Algoritmo 8.1: lectura y escritura de unas fichas de datos personales.

En este nivel de descripción de algoritmo no se aprecia la forma de operar con los campos de un registro. Para ello debemos descender un nivel en el proceso de refinar las acciones de este algoritmo:

```
Leer los datos personales de la persona k-ésima :
   leer cadena nombre de la persona k-ésima ;
   leer cadena apellido de la persona k-ésima ;
   leer dni de la persona k-ésima ;
```

Este mismo nivel de refinamiento se puede describir en un lenguaje más formal, empleando las referencias explícitas a los elementos de la matriz y sus campos:

```
Leer los datos personales de la persona k-ésima :
   leer cadena ( tabla. persona [ k ]. nombre ) ;
   leer cadena ( tabla. persona [ k ]. apellido ) ;
   leer ( tabla. persona [ k ]. dni ) ;
```

La acción leer cadena se podría refinar así:

```
PROCEDIMIENTO leercadena ( dato : cadena ) ;

VAR i: ENTERO ;
```

```
PRINCIPIO
   inicializar i := 0;
   REPETIR
      incrementar i ;
      leer ( dato [ i ] ) ;
   HASTA dato leído sea blanco, o estructura cadena esté llena
   rellenar de blancos el resto de cadena;
FIN.
```

La acción de escribir los datos personales puede igualmente descomponerse en la escritura de cada uno de los campos del registro, y la escritura de los campos de tipo cadena puede realizarse mediante un procedimiento análogo al empleado para la lectura:

```
PROCEDIMIENTO escribircadena  ( dato : cadena ) ;

VAR i: ENTERO ;

PRINCIPIO
   PARA i := 1 HASTA  MAX  HACER  escribir ( dato [ i ] ) ;
   escribir blancos de separación con el campo siguiente;
FIN.
```

Con todos estos refinamientos y procedimientos, el algoritmo general podría quedar más o menos:

```
ALGORITMO registro ;

CONST  MAX = 20;
TIPO   cadena = MATRIZ [ 1..MAX ] DE CAR;
       tablapersonas =
       REGISTRO
          numpersonas : ENTERO;
          persona : MATRIZ [ 1..100 ]  DE
          REGISTRO
             nombre : cadena;
             apellido : cadena;
             dni : ENTERO;
          FIN;
       FIN;
VAR    tabla : tablapersonas ;
       k : ENTERO;

PRINCIPIO
```

```
   leer el número de personas a introducir en la tabla
   PARA k := 1 HASTA número de personas HACER
      PRINCIPIO
      leercadena ( tabla. persona [ k ]. nombre ) ;
      leercadena ( tabla. persona [ k ]. apellido ) ;
      leer ( tabla. persona [ k ]. dni ) ;
      FIN ;
   PARA k := 1 HASTA número de personas HACER
      PRINCIPIO
      escribircadena ( tabla. persona [ k ]. nombre ) ;
      escribircadena ( tabla. persona [ k ]. apellido ) ;
      escribirln ( tabla. persona [ k ]. dni ) ;
      FIN;
FIN.
```

Algoritmo 8.2: lectura y escritura de unas fichas de datos personales.

8.1.3 Programa PASCAL.

```
PROGRAM  registro ;

CONST  MAX = 20;
TYPE   cadena = ARRAY [ 1..MAX ] OF CHAR;
       tablapersonas =
       RECORD
          numpersonas : INTEGER ;
          persona : ARRAY [ 1..100 ] OF
          RECORD
             nombre : cadena;
             apellido : cadena;
             dni : INTEGER ;
          END;
       END;
VAR    tabla : tablapersonas ;
       k : INTEGER ;

PROCEDURE leer ( VAR dato : cadena ) ;

VAR  i, j : INTEGER ;
```

```
BEGIN
   i := 0 ;
   REPEAT
      i := i + 1 ;
      read (dato [ i ] ) ;
{  HASTA dato leído sea blanco, o estructura cadena esté llena }
   UNTIL ( dato [ i ] = ' ' ) OR ( i = MAX ) ;
{  rellenar de blancos el resto de cadena}
   FOR j := i + 1 TO MAX  DO dato [ j ] := ' '
END ;

PROCEDURE escribir ( VAR dato : cadena) ;

VAR i : INTEGER ;

BEGIN
   FOR i := 1 TO MAX DO write (dato [ i ] ) ;
{  escribir blancos de separación con el campo siguiente }
   write ( ' ' ) ;
END ;

BEGIN       { Comienzo del programa principal }
{  leer el número de personas a introducir en la tabla}
   write (  'Introducir numero de personas : ' ) ;
   readln ( tabla. numpersonas ) ;
{  PARA k:= 1 hasta número de personas HACER PRINCIPIO}
   FOR k := 1 TO tabla.numpersonas DO
      BEGIN
{        leercadena ( tabla. persona [ k ]. nombre ) }
      leer ( tabla. persona [ k ]. nombre ) ;
{        leercadena ( tabla. persona [ k ]. apellido ) }
      leer ( tabla. persona [ k ]. apellido ) ;
{        leer ( tabla. persona [ k ]. dni ) }
      readln ( tabla. persona [ k ]. dni ) ;
      END;
{  PARA k:= 1 hasta número de personas HACER PRINCIPIO }
   FOR k := 1 TO tabla.numpersonas DO
   WITH tabla.persona [ k ] DO
      BEGIN
{        escribircadena ( tabla. persona [ k ]. nombre ) }
      escribir ( nombre ) ;
{        escribircadena ( tabla. persona [ k ]. apellido ) }
```

```
        escribir ( apellido ) ;
{       escribirln ( tabla. persona [ k ]. dni ) }
        writeln ( dni );
        END;
END.
```

Programa PASCAL 8.1: rellenado e impresión de unas fichas de datos personales.

Obsérvese que en Pascal es posible evitar la escritura del nombre completo del campo de un registro, usando la sentencia:

WITH < NombreRegistro > < sentencia >

Con ella, todas las referencias a campos del registro <NombreRegistro> se pueden realizar sin necesidad de explicitar el nombre de éste, pues se da por defecto. Como veremos en el apartado siguiente, esta facilidad no existe en otros lenguajes, como el C.

8.1.4 Programa C.

```
/*****************************/
/* PROGRAMA registro */
/*****************************/

#include <stdio.h>
#define  MAX  20

typedef  char cadena [ 20 ] ;
typedef  struct {
             int numpersonas ;
             struct {
                 cadena nombre ;
                 cadena apellido ;
                 int dni ;
             } persona [ 100 ] ;
         } tablapersonas ;

tablapersonas tabla;
int k;

void
```

```
leer (cadena dato)
{
   int i, j ;

   i = 0 ;
   do {
      i = i + 1 ;
      scanf ( "%c", &dato [ i - 1 ] ) ;
/* HASTA dato leído sea blanco, o estructura cadena esté llena */
   } while ( ! ( ( dato [ i - 1 ] == ' ' ) || ( i == MAX ) ) ) ;
/* rellenar de blancos el resto de cadena*/
   for ( j = i + 1 ; j <= MAX ; j++ ) dato [ j - 1 ] = ' ' ;
}

void
escribir ( cadena dato )
{
   int i;

   for ( i = 1 ; i <= MAX ; i++ ) printf ( "%c", dato [ i - 1 ] ) ;
/* Escribir blancos de separación entre campos */
   printf ( " " ) ;
}

main ( )
{
/* leer el número de personas a introducir en la tabla */
   printf ( "Introducir numero de personas : " ) ;
   scanf ("%d", &tabla. numpersonas ) ;
/* PARA k:= 1 hasta número de personas HACER PRINCIPIO*/
   for ( k = 1 ; k <= tabla .numpersonas ; k++ ) {
/*    leercadena ( tabla. persona [ k ]. nombre ) */
      leer ( tabla. persona [ k - 1 ]. nombre ) ;
/*    leercadena ( tabla. persona [ k ]. apellido ) */
      leer ( tabla. persona [ k - 1 ]. apellido ) ;
/*    leer ( tabla. persona [ k ]. dni ) */
      scanf ( "%d", &tabla. persona [ k -1 ]. dni ) ;
/* FIN */
   }
/* PARA k:= 1 hasta número de personas HACER PRINCIPIO*/
   for ( k = 1 ; k <= tabla. numpersonas ; k++ ) {
/*    escribircadena ( tabla. persona [ k ]. nombre ) */
```

```
        escribir ( tabla. persona [ k - 1 ]. nombre ) ;
/*      escribircadena ( tabla. persona [ k ]. apellido ) */
        escribir ( tabla. persona [ k - 1 ]. apellido ) ;
/*      escribirln ( tabla. persona [ k ]. dni ) */
        printf ( "%d\n", tabla. persona [ k - 1 ]. dni ) ;
/*  FIN */
    }
}
```

Programa C 8.1: rellenado e impresión de unas fichas de datos personales.

8.1.5 Ejercicios de aplicación.

a).- Lectura y ordenación de una serie de fechas (día, mes y año).

b).- Obtener la suma de una serie de polinomios representados simbólicamente. Por ej:

$2x^4 + x^3 - 9x + 23$

Cada polinomio ocupa una línea.

c).- Modificar el algoritmo dado como ejemplo en este capítulo para que, una vez introducidas todas las fichas de personas, vaya solicitando apellidos, de uno en uno, y escribiendo las fichas de las personas cuyo apellido coincida con el que se acaba de leer.

8.2 Ficheros.

8.2.1 Definiciones.

Como ya hemos dicho, los ficheros son unas estructuras de datos de tamaño indefinido, es decir, desconocido al comienzo del programa. Por esta razón, no todos los datos contenidos en este tipo de estructura de datos están almacenados en la memoria principal del ordenador, sino que tan sólo uno de los elementos de la estructura es accesible en cada momento de la ejecución del algoritmo, mediante una sentencia especial que transfiere ese elemento a una variable convencional del mismo tipo, realizando lo que denominamos una **lectura**. Análogamente, cuando queremos almacenar un valor en uno de los elementos del fichero, debemos realizar una operación de **escritura** en el mismo, con lo cual transferimos el contenido de una variable convencional (memoria principal) al elemento

del fichero (la memoria secundaria). Es como si tuviésemos una **ventana** a través de la cual podemos acceder (leer o escribir) a uno de los elementos del fichero. Esta ventana a través de la cual podemos ver y modificar el contenido del fichero se denomina **tampón** o **buffer** de entrada/salida. Los elementos que componen un fichero se suelen denominar **registros**, pues en general lo son, aunque también pueden ser variables elementales (carácter, real, etc.).

Según la forma en que se pueda desplazar esta ventana sobre los elementos del fichero, tendremos distintos tipos de ficheros:

secuenciales: cada vez que se realiza una operación de lectura o escritura de un elemento del fichero, la ventana avanza una posición para colocarse sobre el elemento siguiente de la estructura. De esta forma, todos los elementos del fichero se van leyendo o escribiendo automáticamente uno detrás de otro, en secuencia. Un caso particular de los ficheros secuenciales son los **ficheros de texto**, cuyos elementos son caracteres imprimibles y para los que la mayoría de compiladores ofrecen funciones de tratamiento especiales como, por ejemplo, la de detección de final de línea, que vale cierto cuando el último carácter leído es el último de una línea de texto, es decir, cuando la ventana se encuentra sobre un carácter de "retorno de carro" (<CR>).

de acceso directo: son aquéllos en los que el programador realiza manualmente el desplazamiento de la ventana sobre los elementos del fichero, es decir, controlado directamente por el algoritmo. Este modo de acceso permite emplear los ficheros como los vectores, leyendo o escribiendo datos en sus elementos en cualquier orden, aleatoriamente.

Cuando declaramos un fichero, en realidad estamos declarando una variable estructurada, en la que se almacenan todos los datos necesarios para acceder al fichero (nombre de éste, situación de la ventana, operaciones permitidas, etc.). Entre éstos se encuentra un campo del tipo de los elementos del fichero, que el sistema de gestión de ficheros del ordenador asociará a la ventana de comunicación entre nuestro programa y el fichero. Esta asociación se realiza mediante una instrucción de **apertura** del fichero, tras la cual podremos operar con los elementos de éste, a través de la ventana, hasta que realicemos la operación complementaria de **cierre** del fichero, que disocia la variable y el fichero.

Como resumen podemos decir que con los ficheros podemos realizar las siguientes operaciones:

ABRIR: asocia una variable del algoritmo a un fichero del ordenador. Además de estos dos parámetros, con esta operación debemos indicar el tipo de acciones que vamos a realizar sobre los elementos del fichero:

Lectura: **Abrir** (< **IdFichero** >, **l** , " < **NombreFichero** > "),

< **IdFichero** >, identificador lógico del fichero, es el nombre de la variable que lo representa en el algoritmo, a través de la cual accedemos al tampón (ventana) del fichero.

l, indica que el fichero se abre para leer su contenido. En algunos casos, si el fichero no existe nos dará un error.

< **NombreFichero** >, es el nombre completo del fichero que queremos abrir, incluyendo el nombre del dispositivo (disco) en el que se encuentra, y el camino de acceso al subdirectorio en el que se encuentre, de no coincidir con los valores por defecto en el momento de ejecutarse el programa.

Escritura: **Abrir** (< **IdFichero** >, **e**, " < **NombreFichero** >"),

e, indica que el fichero se abre para escribir en él. Si existe un fichero <NombreFichero>, destruye su contenido, para empezar a escribirlo desde el primer elemento.

Añadido: **Abrir** (< **IdFichero** >, **a** , " < **NombreFichero** > "),

a, indica que el fichero se abre para escribir en él, pero a partir del último dato que contenga, con lo cual, si ya existía, no se borra su contenido.

En Pascal esta acción se traduce en dos sentencias:

assign (< **IdFichero** >, " < **NombreFichero** > "),

Para asignar un fichero de la memoria secundaria al identificador. La segunda sentencia es distinta, según el tipo de operaciones que queramos realizar sobre el fichero:

reset (< **IdFichero** >) Para leerlo a partir del principio.

rewrite (< **IdFichero** >) Para escribirlo a partir del principio. Pone la marca de final de fichero en el primer elemento, con lo que queda borrado su contenido si el fichero ya existía.

append (< **IdFichero** >) Para añadir, es decir, escribirlo a partir del final.

En C la traducción es directa:

< **IdFichero** > = **fopen** ("< **NombreFichero** >" ,< **modo** >)

donde < **modo** > indica el modo de acceso:

rb = lectura.

r = lectura de texto.

wb = escritura.

w = escritura de texto.

r+b = lectura/escritura.

r+ = lectura/escritura texto sin borrar el contenido previo.

w+ = lectura/escritura texto borrando el contenido previo.

a+b = añadido.

a = añadido de texto.

CERRAR: disocia el < **IdFichero** > del fichero < **NombreFichero** >. Después de esta operación ya no se podrán realizar operaciones de lectura o escritura sobre <**NombreFichero** >. Esta operación tiene la forma:

Cerrar (< **IdFichero** >)

La traducción de esta acción a Pascal o C es inmediata: **close** (< **IdFichero** >)

LEER: copia el contenido del registro del fichero sobre el que se encuentra la ventana (tampón) a la variable que le pasemos como parámetro.

leer (< **IdFichero** >, < **variable** >)

La traducción a Pascal es inmediata:

read (< **IdFichero** >, < **variable** >)

La traducción a C es un poco más compleja, pues como en la declaración de la variable de tipo fichero no se ha indicado el tipo de sus elementos, hay que indicar su tamaño al acceder a ellos (**sizeof** (< **var** >)). Además, el procedimiento permite leer más de un elemento (< **NúmeroElementos** >) de una vez:

fread (&< **var** >, **sizeof** (< **var** >), < **NúmeroElementos** >, < **IdFichero** >)

ESCRIBIR: copia el contenido de la variable que le pasemos como parámetro al registro del fichero sobre el que se encuentra la ventana, a través del tampón.

escribir (< **IdFichero** > , < **variable** >)

Su traducción a Pascal y C es dual a la de lectura, respectivamente:

write (< **IdFichero** > , < **variable** >)

fwrite (&< **var** > , **sizeof** (< **var** >) , < **NúmeroElementos** > , < **IdFichero** >)

BORRAR: borra del directorio del dispositivo de memoria secundaria (disco) toda referencia al fichero < **NombreFichero** >. Después de esta operación no se puede volver a abrir el fichero borrado ni, por tanto, acceder a su contenido. Para poder borrar un fichero, debe estar cerrado.

Borrar (< **NombreFichero** >)

Su traducción a Pascal y C es respectivamente:

assign (< **IdFichero** >, < **NombreFichero** >) ; **erase** (< **IdFichero** >)

remove (< **NombreFichero** >)

FinalDeFichero (**FDF**): Es una función que devuelve el valor cierto si la ventana del fichero se encuentra sobre el registro siguiente al último lleno (escrito), y falso en

caso contrario. En un fichero secuencial, después de que esta función se hace cierta, no podemos seguir leyendo del fichero.

En Pascal tiene la forma: **eof** (< **IdFichero** >) : boolean.

En C tiene la forma: int **feof** (< **IdFichero** >). Devuelve cero (FALSO) si no detecta fin de fichero y otro valor (CIERTO) si lo detecta.

Los ficheros de texto, además, en algunos lenguajes de programación, como el Pascal, permiten realizar algunas acciones específicas, que por su utilidad puede interesarnos incluir en nuestros algoritmos, aunque otros lenguajes, como el C, no las soporten y tengamos que traducirlas en acciones más sencillas. Estas acciones son:

LEERLN: readln (< **IdFichero** > , < **variable** >). Lee la < **variable** > en el fichero <**IdFichero** > y después avanza la ventana hasta el siguiente final de línea.

ESCRIBIRLN: writeln (< **IdFichero** > , < **variable** >). Escribe la < **variable** > en el fichero <**IdFichero** > y después un final de línea.

FinalDeLinea (FDL): **eol** (< **IdFichero** >) : BOOLEAN.

Los ficheros de acceso directo siempre deben abrirse para lectura/escritura, pues se supone que la aleatoriedad del acceso tiene por finalidad ir leyendo y modificando los datos leídos. Para acceder a una posición concreta, desde el punto de vista algorítmico, utilizaremos una sola acción para aproximar al máximo la sintaxis a la de manejo de vectores (una sola acción para acceder a un elemento del fichero), pero tanto en Pascal como en C hay que invocar dos sentencias complementarias para hacerlo: mover la ventana a la posición <pos>, y leer o escribir con las sentencias normales:

LEERPOS (< **IdFichero** > , < **pos** > , < **var** >):

En Pascal tiene la forma:

seek (< **IdFichero** > , < **pos** > - 1) ;

read (< **IdFichero** > , < **variable** >)

En C tiene la forma:

fseek (< **IdFichero** > , **sizeof** (< **var** >) * (< **pos** > - 1) , **SEEK_SET**) ;

fread (&< **var** > , **sizeof** (< **var** >) , < **NúmeroElementos** > , < **IdFichero** >)

ESCRIBIRPOS (< **IdFichero** > , < **pos** > , < **var** >) :

En Pascal tiene la forma:

seek (< **IdFichero** > , < **pos** > - 1) ;

write (< **IdFichero** > , < **variable** >)

En C tiene la forma:

fseek (< **IdFichero** > , **sizeof** (< **var** >) * (< **pos** > - 1) , **SEEK_SET**) ;

fwrite (&< **var** > , **sizeof** (< **var** >) , < **NúmeroElementos** > , < **IdFichero** >)

8.2.2 Algoritmo ejemplo: fichero secuencial.

Como ejemplo de fichero secuencial vamos a desarrollar un algoritmo para escribir una cantidad cualquiera de números en un fichero, y luego sumarlos según los vamos leyendo. Como el algoritmo en sí no presenta ninguna dificultad, y su utilidad es simplemente la de ilustrar el uso de las sentencias de manejo de los ficheros secuenciales, no será necesaria la profusión de comentarios y niveles de descripción de los algoritmos de los últimos capítulos:

```
ALGORITMO fichero_suma;

VAR  lista : FICHERO de ENTERO;
       k, suma, cantidad : ENTERO;
PRINCIPIO
   abrir ( lista, e, " lista.lis " ) ;
   leer la cantidad de números a sumar ;
   grabar en lista los números del 1 a cantidad ;
   cerrar ( lista ) ;
   Obtener la suma de los números grabados:
      abrir ( lista, l, " lista.lis " ) ;
      inicializar: suma := 0;
      MIENTRAS queden números HACER
            PRINCIPIO
            Leer ( lista, k ) ;
            Sumar k a suma
            FIN;
   cerrar lista;
   escribir suma
FIN.
```

Algoritmo 8.3: escribe una cantidad de números en un fichero y luego calcula su suma.

8.2.3 Programa PASCAL: fichero secuencial.

```
PROGRAM ficheros ;

VAR  lista : FILE OF INTEGER;
       k , suma, cantidad : INTEGER ;


BEGIN
{   abrir ( lista, e, " lista.lis " ) }
    assign ( lista , " lista.lis " ) ;
    rewrite ( lista ) ;
{   leer la cantidad de números a sumar;}
    read ( cantidad ) ;

{   grabar en lista los números del 1 a cantidad ;}
    FOR k := 1 TO cantidad DO write ( lista , k ) ;
{   Obtener la suma de los números grabados:}
{       cerrar ( lista ) }
{       abrir ( lista, l, " lista.lis " ) }
    reset ( lista ) ;
    suma := 0 ;

{   MIENTRAS queden números HACER PRINCIPIO}
    WHILE NOT eof ( lista ) DO
       BEGIN
       read ( lista , k ) ;
       suma := suma + k ;
       END ;
    close ( lista ) ;
    writeln ( suma )
END.
```

Programa PASCAL 8.2: escribe una cantidad de números en un fichero y luego calcula su suma.

Obsérvese que en este caso no es necesario cerrar y reabrir a continuación el fichero "lista", tal como requiere el algoritmo, pues en Pascal basta con invocar la función rewrite para que la ventana de acceso al fichero deje el sitio de éste en el que se encuentre, y se sitúe sobre el primer dato registrado en el fichero.

8.2.4 Programa C: fichero secuencial.

```
/*******************************/
/* PROGRAMA ficheros */
/*******************************/

#include <stdio.h>

main ( )

{
   FILE  *lista;
   int    k, suma, cantidad;

/* abrir ( lista, e, " lista.lis " ) */
   lista = fopen ( "lista.lis", "w" ) ;
/* leer la cantidad de números a sumar */
   scanf  ( "%d", &cantidad ) ;
/* grabar en lista los números del 1 a cantidad */
   for ( k = 1; k <= cantidad; k++ ) fprintf ( lista, "%d\n", k ) ;
/* cerrar ( lista ) */
   fclose ( lista ) ;
/* Obtener la suma de los números grabados:*/
/* abrir ( lista, l, " lista.lis " ) */
   lista = fopen ( "lista.lis", "r" ) ;
   suma = 0 ;

/* MIENTRAS queden números HACER PRINCIPIO*/
   while ( !feof  ( lista ) ) {
/*     Leer (lista, k);*/
       fscanf ( lista, "%d" , &k ) ;
/*     Sumar k a sum a*/
       suma = suma + k ;
    }
/* cerrar ( lista ) */
   fclose ( lista ) ;
/* escribir suma */
   printf ( "La suma de los %d primeros números es: %d", cantidad, suma )
 }
```

Programa C 8.2: grabación y lectura de un fichero para el cálculo de la suma de los números en él contenidos.

8.2.5 Ejercicios de aplicación: fichero secuencial.

a).- Mezcla de dos ficheros de números reales, ordenados de menor a mayor, sobre un tercer fichero, de forma que éste resulte también ordenado. El algoritmo consiste en leer un dato de cada fichero, y repetir: escribir el menor de los dos sobre el fichero de salida, y leer el dato siguiente del fichero al que pertenecía el dato escrito, hasta que no queden más datos en uno de los ficheros; para terminar escribiendo el resto del fichero no acabado en el fichero de salida.

b).- Dado un fichero cuyos elementos son registros del tipo:

```
cliente REGISTRO
        Nombre : cadena;
        dirección REGISTRO
                calle : cadena;
                número : entero;
                población : cadena;
                distrito : entero
        FIN;
FIN
```

Se pide generar otro fichero en el que tan sólo aparezcan los datos de los clientes cuyo distrito postal esté comprendido entre dos valores, que se leerán del teclado.

c).- Dado un fichero en el que cada línea contiene todos los coeficientes de un polinomio, ordenados de a_0 hasta a_n, incluidos los ceros de los términos inexistentes, siendo n el grado del polinomio. Se pide realizar el algoritmo que genere otro fichero, de iguales características al mencionado, en el que cada línea, i, contenga los coeficientes del polinomio resultante de sumar las líneas i e i+1 del polinomio de entrada.

8.2.6 Algoritmo ejemplo: fichero de texto.

Ordenar las palabras de un fichero de texto. Cada frase del fichero está en una linea diferente. Cada palabra es una secuencia de letras precedida por una secuencia de asteriscos, que indica la posición que deberá ocupar la palabra dentro de la frase ordenada. Imprimir un fichero de texto con las frases ordenadas, sin asteriscos, con un solo espacio en blanco entre cada palabra y un punto al final de la última. Es decir que las líneas:

```
***la****frase******ordenada*****correctamente**es*Esta
*Y***la****segunda**ésta*****línea
```

Deberá ser impresa correctamente ordenada, es decir:

Esta es la frase correctamente ordenada.
Y ésta la segunda línea.

La figura 8.1 muestra la descripción gráfica de la estructura de los ficheros de entrada y salida. La metodologia de descomposición modular descrita en el capítulo 6 se basa en el método de Jackson para la descripción de estructuras de datos. La técnica es la misma: secuencias y alternativas en la misma línea, y refinamientos en líneas inferiores. La

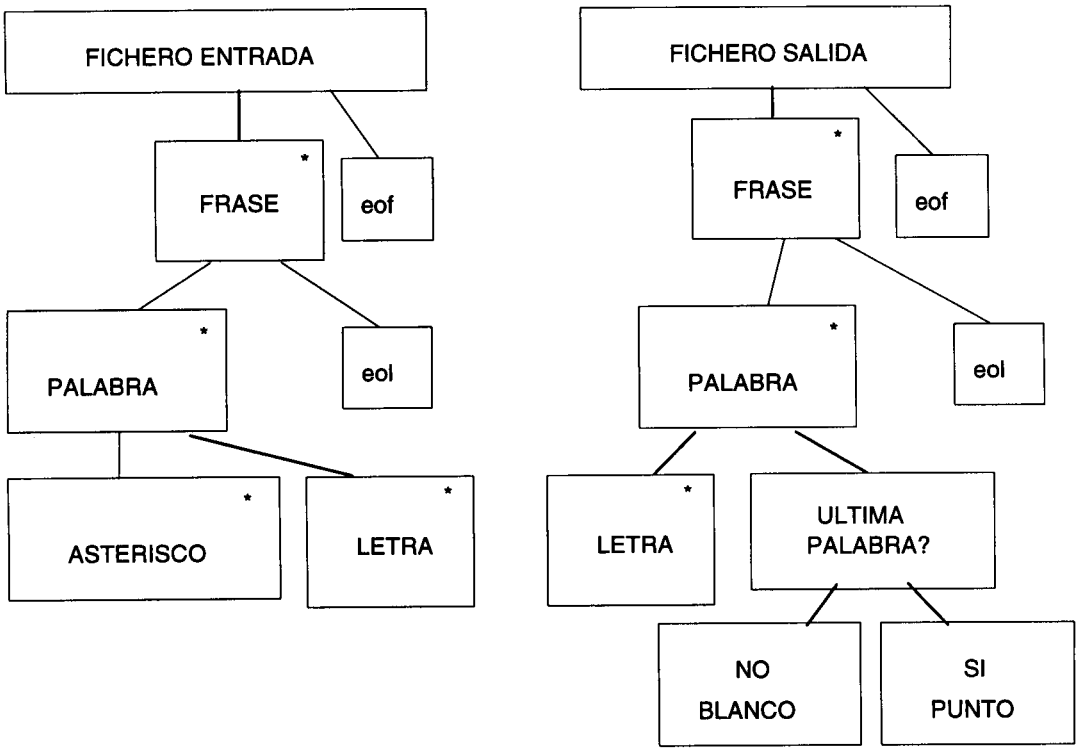


Fig 8.1: descripción gráfica según el método de Jackson de la estructura de los ficheros de entrada y salida.

única diferencia radica en la forma de expresar las apariciones de una componente (campo) más de una vez, que se indica con un " * " en la caja que lo representa. Así pues, en la figura 8.1 leeremos que: "el fichero entrada se compone de un número indeterminado de componentes del tipo *frase*, cuyo final se reconoce por la aparición de la componente de tipo *eof*. Cada frase se compone de un número indeterminado de elementos del tipo *palabra*, seguido de un elemento del tipo *eol*.....". Si en la especificación del fichero se

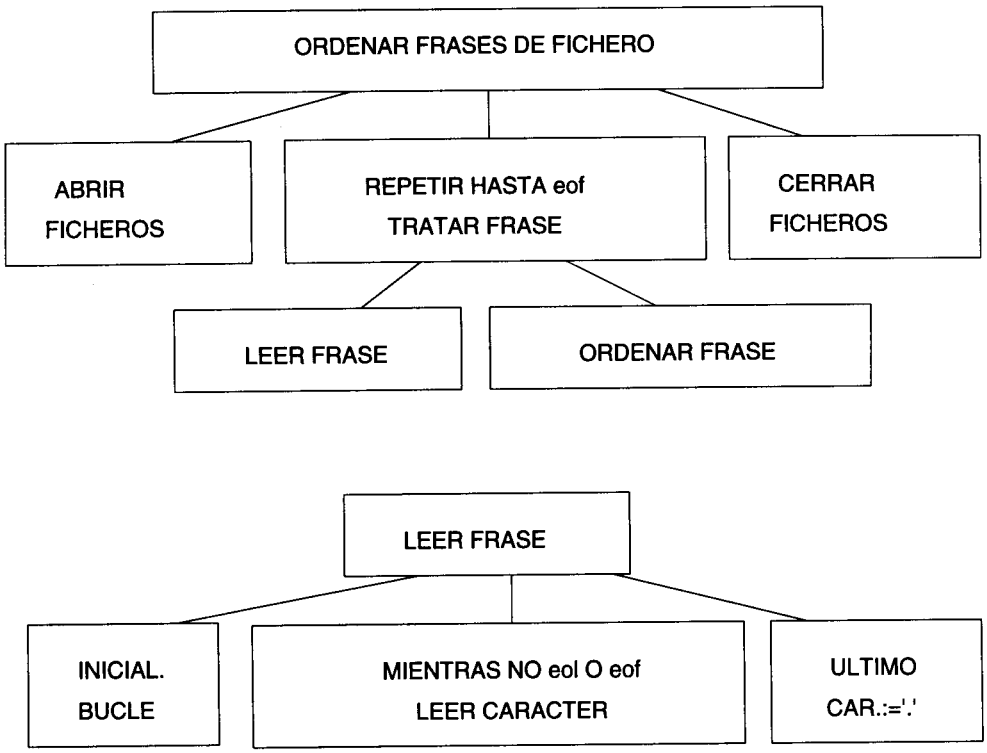


Fig 8.2: a) descomposición modular del algoritmo: ordena palabras de frase. b) descomposición modular de la acción: leer frase.

indicase el número exacto de componentes de un tipo determinado, entonces indicaríamos este número en el cajón, en vez del " * ".

Esquema inicial del algoritmo:

```
abrir ficheros ;
REPETIR
    leer siguiente frase ;
    ordenar frase ;
HASTA fin_de_fichero
cerrar ficheros ;
```

Algoritmo 8.4: primer refinamiento de la acción: ordenar palabras de frases.

Refinamiento de la acción ordenar frase. Esquema inicial:

```
Inicializar contador de palabras y puntero a frase:
num_palabras := 0 ; i := 1 ;
MIENTRAS no fin de frase HACER
   PRINCIPIO
   almacenar_palabra ( orden, frase, i ) ;
   contar palabra;
   FIN;
escribir_frase ( orden, salida, frase, num_palabras ) ;
```

Algoritmo 8.5: refinamiento de la acción: ordenar una frase.

Refinamiento del algoritmo de la acción Almacenar_palabra:

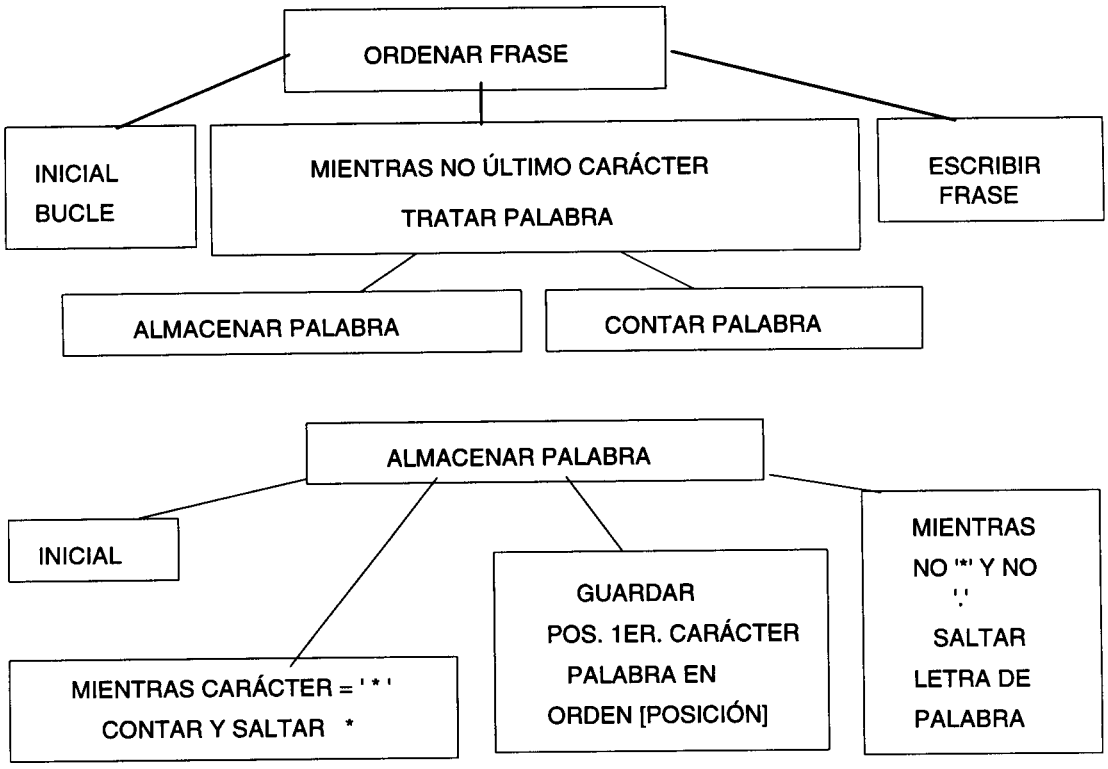


Fig 8.3: a) descomposición modular de la acción: ordenar frase. b) descomposición modular de la acción:almacenar palabra.

```
PROCEDIMIENTO Almacenar_palabra ( orden, frase, situación ) ;
```

```
PRINCIPIO
{  Situación ya está apuntando al primer asterisco de la palabra}
   Inicializa número de orden := 0;
   MIENTRAS situación apunte a un asterisco HACER
                  avanzar situación y cuenta * en número de orden;
   Guardar situación en vector de número de orden;
   Salta letras de palabra, incrementando puntero situación;
{  Situación ya apunta a primer * de palabra siguiente o al punto final }
FIN.
```

Algoritmo 8.6: refinamiento de la acción: almacenar palabra.

8.2.7 Programa PASCAL: fichero de texto.

El programa en Pascal que realiza el algoritmo anterior incluye dos procedimientos, de los que no se ha dado el algoritmo por ser de codificación evidente: el de leer_frase y escribir_frase. A continuación se describen los tres procedimientos utilizados por el programa principal.

```
PROCEDURE leer_frase ( VAR entrada : TEXT ; VAR linea : frase ) ;

VAR i : INTEGER ;

BEGIN
   i := 0 ;
   REPEAT
      i := i + 1;
      read ( entrada, linea [ i ] ) ;
{  HASTA fdl ( entrada ) O fdf ( entrada ) }
   UNTIL eoln ( entrada ) OR eof ( entrada )
{  Prepara la lectura de la línea siguiente }
   readln ( entrada ) ;
{  Coloca el punto indicador del final de línea, después del último carácter leído }
   linea [ i + 1 ] := '.' ;
END;

PROCEDURE almacenar_palabra (  VAR orden : vector ; VAR linea : frase;
                               VAR i:  INTEGER ) ;
{ Situación ( i ) ya está apuntando al primer asterisco de la palabra }
```

```
VAR  num : INTEGER ;

BEGIN
{   Inicializa núm. de orden := 0 }
    num := 0 ;
{   MIENTRAS situación (i) apunte a un asterisco HACER }
    WHILE linea [ i ] = '*' DO
    BEGIN
{       avanza situación y cuenta * en núm. de orden;}
        i := i + 1 ; num := num + 1 ;
    END;
{   Guarda situación en vector de num. de orden;}
    orden [ num ] := i ;
{   Salta letras de palabra, incrementando puntero situación;}
    WHILE ( linea [ i ] <> '*' ) AND ( linea [ i ] <> '.' )  DO  i := i + 1 ;
{   Situación ya apunta a primer * de palabra siguiente o al punto final}
END ;

PROCEDURE escribir_frase ( VAR orden: vector; VAR salida: TEXT;
                           VAR linea: frase; num: INTEGER);

VAR i , j : INTEGER ;

BEGIN
{   PARA cada palabra HACER}
    FOR i := 1 TO num DO
    BEGIN
{       Obtener posición de su primer carácter}
        j := orden [ i ] ;
{       Escribir todos los caracteres de la palabra}
        WHILE ( linea [ j ] <> '*' ) AND ( linea [ j ] <>  '.' ) DO
              BEGIN
              write ( salida , linea [ j ]  ) ;
              j := j + 1 ;
              END
    END;
{   SI la palabra es la última}
    IF i := num
{       ENTONCES escribe el punto final y el final de línea }
        THEN writeln ( salida, '.' )
{       SINO escribe blanco separador de palabras }
        ELSE write ( salida , ' ' )
```

```
END;
```

Programa Pascal 8.3: procedimientos para la lectura, ordenación y escritura de una frase.

```
PROGRAM ordena_palabras ;

CONST     maxcar = 80 ;
          maxpalabras = 10 ;
TYPE      frase = ARRAY [ 1..maxcar ] OF CHAR ;
          vector = ARRAY [ 1..maxpalabras ] OF INTEGER ;
VAR       entrada , salida : TEXT ;
          linea : frase ;
          orden : vector ;
          num_palabras, i : INTEGER ;

PROCEDURE  leer_frase ( VAR entrada : TEXT ;  VAR linea : frase ) ;
. . .
PROCEDURE almacenar_palabra ( VAR orden : vector  ; VAR linea : frase ;
                              VAR i : INTEGER ) ;
. . .
PROCEDURE escribir_frase ( VAR orden : vector  ; VAR salida : TEXT ;
                           VAR linea : frase ;  num : INTEGER ) ;
. . .

BEGIN     { Comienzo del programa principal }
{  abrir ficheros }
   assign ( entrada , " desordenado.lis" ) ;
   assign ( salida , " ordenado.lis" ) ;
   reset ( entrada ) ; rewrite (salida ) ;
   REPEAT
{     leer siguiente frase ;}
      leer_frase ( entrada , linea ) ;
{     ordenar frase:}
{     Inicializar contador de palabras y puntero a frase:}
      num_palabras := 0 ; i := 1 ;
{     MIENTRAS no fin de frase HACER  }
      WHILE linea [ i ] <>  ' . ' DO
            BEGIN
            almacenar_palabra ( orden, linea , i  ) ;
            num_palabras := num_palabras + 1 ;
            END;
```

```
      escribir_frase ( orden , salida , linea , num_palabras ) ;
   UNTIL eof ( entrada );
   close ( entrada ) ; close (salida ) ;
END .
```

Programa Pascal 8.4: ordena las frases de un fichero de texto.

8.2.8 Programa C: fichero de texto.

```
/*********************************/
/* PROGRAMA ord_pal */
/*********************************/

#include <stdio.h>
#define MAXCAR 200
#define MAXPALABRAS 30

typedef    char frase [ MAXCAR ] ;
typedef    int vector [ MAXPALABRAS ] ;

void
leer_frase (FILE *entrada, frase linea )

{
   int i;

   i = 0;
/* REPETIR*/
   do {
      i++ ;
      fscanf ( entrada, "%c", &linea [ i - 1 ] ) ;
/* HASTA fdl ( entrada ) O fdf ( entrada ) */
   } while ( ( linea [ i - 1 ] != '\n' ) && ( linea [ i - 1 ] != EOF ) ) ;
/* Coloca el punto indicador del final de línea, después del último carácter leído */
   linea [ i - 1 ] = '.' ;
}

void
almacenar_palabra ( vector orden, frase linea, int *i )
```

```
/* Situación (i) ya está apuntando al primer asterisco de la palabra */
{
   int num;

/* Inicializa núm. de orden := 0;*/
   num = 0 ;
/* MIENTRAS situación apunte a un asterisco HACER */
   while ( linea [ ( *i ) - 1 ] == ' * ' ) {
/*    avanza situación y cuenta * en núm. de orden */
      ( *i ) = ( *i ) + 1; num = num + 1;
   }
/* Guarda situación en vector de num. de orden;*/
   orden [ num - 1 ] = ( *i ) ;
/* Salta letras de palabra, incrementando puntero situación */
   while ( ( linea [ ( *i ) - 1 ] != ' * ' ) && ( linea [ ( *i ) - 1 ] != ' . ' ) ) ( *i ) = ( *i ) + 1 ;
/* Situación ya apunta a primer * de palabra siguiente o al punto final */
}


void
escribir_frase ( vector orden, FILE *salida, frase linea, int num )

{
   int i, j ;

/* PARA cada palabra HACER*/
   for ( i = 1 ; i <= num ; i++ ) {
/*    Obtener posición de su primer carácter */
      j = orden [ i - 1 ] ;
/*    Escribir todos los caracteres de la palabra */
      while ( ( ( linea [ j - 1 ] != ' * ' ) && ( linea [ j - 1 ] != ' . ' ) && ( linea [ j - 1 ] != ' / ' ) ) )
{
            fprintf ( salida, "%c" , linea [ j - 1 ] ) ; j = j + 1;
      }
/*    SI la palabra es la última */
      if ( i := num )
/*          ENTONCES escribe el punto final y el final de línea */
            fprintf ( salida, "\n" ) ;
/*    SINO escribe blanco separador de palabras */
      else
            fprintf ( salida, " " ) ;
   }
```

```
}

main ( )
{
   FILE     *entrada, *salida;
   frase    linea;
   vector   orden;
   int      num_palabras, i;

/* abrir ficheros ;*/
   entrada = fopen ( "ord-pal.dat" , "r" ) ;
   salida = fopen ( "ord-sal.dat" , "w" ) ;
/* REPETIR*/
   do {
/*    leer siguiente frase */
      leer_frase ( entrada, linea ) ;
/*    ordenar frase */
/*    Inicializar contador de palabras y puntero a frase: */
      num_palabras = 0 ;
      i = 1 ;
/*    MIENTRAS no fin de frase HACER PRINCIPIO */
      while ( linea [ i - 1 ] != ' . ' ) {
            almacenar_palabra ( orden, linea, &i ) ;
            num_palabras++ ;
      }
      escribir_frase ( orden, salida, linea, num_palabras ) ;
/* HASTA fin_de_fichero */
   } while ( !feof ( entrada ) ) ;
   fclose ( salida ) ;
   fclose ( entrada ) ;
}
```

Programa C 8.4: ordena las frases de un fichero de texto.

8.2.9 Ejercicios de aplicación: fichero de texto.

a).- Traducir un texto a "pseudo-latín" a base de cambiar cada palabra poniendo la primera letra al final y terminándola en "um ".

b).- Reconocer frases palindromes. Cada frase ocupa una línea del fichero. La salida ha de ser por pantalla y debe consistir en escribir cada frase acompañada por la respuesta correspondiente.

9. Algoritmos básicos.

En este capítulo vamos a estudiar algunos problemas clásicos identificados dentro del ámbito del proceso de datos y que están ampliamente estudiados debido a su utilización para resolver un gran abanico de problemas.

Para ello hemos seleccionado tres tipos de problemas a modo de ejemplo y sin el ánimo de ser exhaustivos. El criterio que hemos seguido para seleccionar los ejemplos ha sido por una parte que sean algoritmos sencillos y que respondan a problemas muy comunes y a ser posible ingenieriles. De acuerdo con estos criterios estudiaremos: métodos de ordenación de un conjunto de datos, métodos de búsqueda en un conjunto de datos que puede estar ordenado o desordenado y finalmente presentaremos algunos algoritmos numéricos para resolver algunos de los problemas matemáticos más usuales.

9.1 Métodos de ordenación.

El problema de la ordenación de un conjunto de datos según un cierto criterio es uno de los más utilizados dentro del proceso de datos, y está ampliamente estudiado en la bibliografia. A lo largo del tiempo se han desarrollado gran cantidad de algoritmos de ordenación con el objeto de optimizar algún aspecto en relación a los anteriores y mejorar su rendimiento desarrollando algoritmos más o menos complejos.

La problemática de la ordenación, además de tener interés por sí misma, tiene también su importancia para facilitar la búsqueda en un conjunto de datos. Tal y como estudiaremos en el apartado siguiente, se han desarrollado una familia de algoritmos de búsqueda que incluyen la ordenación como fase previa.

La elección del tipo de algoritmo de ordenación más apropiado depende, entre otros criterios, de las características de la estructura de los datos. De hecho existen dos grandes familias de métodos de ordenación en función de dicho criterio, los métodos de ordenación internos y los externos. Los primeros se ocupan de estudiar la ordenación de datos almacenados en alguna estructura de datos, que se almacena en la memoria principal del ordenador como, por ejemplo, vectores y matrices. Los métodos de ordenación externos se ocupan de estudiar la problemática específica que comporta el hecho de tener los datos a

ordenar almacenados en memoria auxiliar, como es el caso de los ficheros, sobre todo en el caso de ficheros secuenciales.

En este apartado nos vamos a ocupar exclusivamente de los métodos de ordenación internos, donde la estructura de datos en la que se almacenan los datos a ordenar será un vector. Dentro de los métodos internos vamos a estudiar únicamente aquellos métodos que ordenan sobre la propia estructura de datos, es decir, que no requieren de la utilización de memoria adicional.

Existen tres tipos de métodos de ordenación internos en función del método en el que se basa el algoritmo para realizar la ordenación:

Ordenación por inserción.

Ordenación por selección.

Ordenación por intercambio.

Dentro de estos tres tipos de métodos existen bastantes algoritmos conocidos que responden a esta clasificación; nosotros hemos seleccionado únicamente un algoritmo representativo de cada grupo para ilustrar el método.

9.2 Ordenación por inserción.

9.2.1 Definición.

El método de ordenación por inserción consiste en dividir el conjunto de datos a ordenar en dos subconjuntos que corresponden, respectivamente, al subconjunto de datos ya ordenados y el subconjunto que contiene los datos pendientes de ordenar. Cada paso del algoritmo consiste en incrementar en un elemento el subconjunto ordenado y decrementar el desordenado, hasta que todos los elementos pertenezcan al subconjunto ordenado y el subconjunto desordenado esté vacío.

Conjunto inicial de datos a ordenar: a(1), a(2), ...,a(n).

En un momento dado del algoritmo:

subconjunto ordenado: a(1), a(2),.....,a(i-1).

subconjunto desordenado: a(i), a(i+1),.....,a(n).

Cada paso consiste en insertar el elemento a(i) del conjunto desordenado en el lugar apropiado del subconjunto ordenado. Para insertar a(i) en el lugar apropiado, es necesario desplazar todos los elementos que no cumplen el criterio de ordenación (ordenación creciente o decreciente).

A continuación, en la Fig 9.1, ilustraremos el método de inserción directa sobre un ejemplo numérico donde el criterio de ordenación es el creciente.

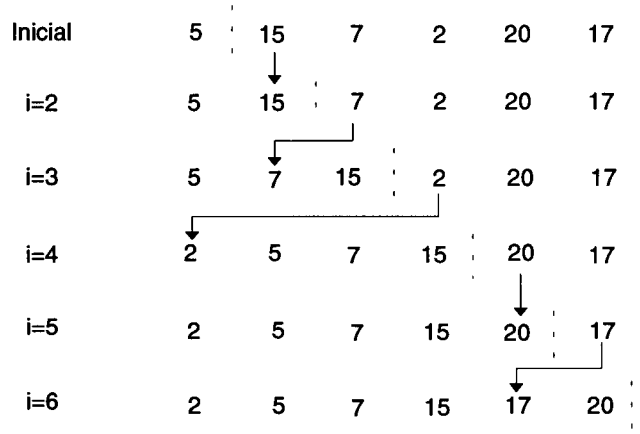


Fig 9.1: método de ordenación por inserción directa.

Existen otros algoritmos de ordenación basados en la idea de la inserción entre ellos citaremos:

La ordenación por inserción binaria: consiste en mejorar el procedimiento de inserción introduciendo un método para encontrar directamente el lugar de inserción, aprovechando el hecho que el conjunto de datos en el que se debe insertar está ya ordenado. El algoritmo de inserción binaria consiste en utilizar como método de búsqueda del lugar de inserción la búsqueda binaria o dicotómica, que estudiaremos en el apartado de métodos de búsqueda.

Otro algoritmo basado en la inserción es el método Shell, que consiste en ordenar el vector mediante la ordenación de subvectores. Cada subvector se construye mediante aquellos elementos que distan entre sí una constante, empezando por los más distantes hasta llegar al vector compuesto por los elementos que distan una posición, es decir, todo el vector. Por esta razón, al método de Shell se le denomina también método de inserción con incrementos decrecientes.

9.2.2 Algoritmo ejemplo.

```
ALGORITMO inserción;

COMIENZO
   Introducir los datos a ordenar en el vector;
   DESDE i := 2 HASTA maxelem HACER
      COMIENZO
      x := vector [ i ] ;
      Insertar x en el lugar apropiado de vector [ 1 ].....vector [ i ] ;
      FIN
FIN
```

Algoritmo 9.1: primer refinamiento del método de ordenación por inserción.

Refinamiento:

```
Insertar x en el lugar apropiado de vector[1]...vector[i]:
Inicializar índice: j := i - 1;
MIENTRAS x y vector [ j ] no estén en el orden correcto y j > 0 HACER
   COMIENZO
   Desplazar una posición a la derecha ;
   vector [ j + 1 ] := vector [ j ] ;
   Actualizar índice: j := j - 1 ;
   FIN
vector [ j + 1 ] := x
```

Algoritmo 9.2: refinamiento de la acción ***insertar x en el lugar apropiado de vector [1]...vector [i].***

9.2.3 Programa PASCAL.

```
PROGRAM inserción;

CONST    maxelem = 14 ; {Número de elementos a ordenar}
VAR      vector: ARRAY [ 1..maxelem ] OF INTEGER;
         j, x: INTEGER;

BEGIN
{ Introducir los datos a ordenar en el vector }
```

```
FOR i := 1 TO maxelem DO
   BEGIN
   readln ( x ) ;
   vector [ i ] := x ;
   END;
FOR i := 2 TO maxelem DO
   BEGIN
   x := vector [ i ] ;
{  Insertar x en el lugar apropiado de vector [ 1 ].....vector [ i ] }
{  Inicializar índice:} j := i - 1;
{  MIENTRAS x y vector [ j ] no estén en el orden correcto y j > 0 HACER }
   WHILE ( x < vector [ j ] ) AND ( j > 0 ) DO
      BEGIN
{     Desplazar una posición a la derecha }
      vector [ j + 1] := vector [ j ] ;
{     Actualizar índice } j := j - 1 ;
      END;
   vector [ j + 1 ] := x;
   END
END.
```

Programa PASCAL 9.1: ordenación ascendente de un vector de números enteros mediante el método de inserción directa.

9.2.4 Programa C

```
#include <stdio.h>

#define MAXELEM 14  /*Número de elementos a ordenar*/

main ( )
{
   int vector [ MAXELEM ] ;
   int i, j, x;

/* Introducir los datos a ordenar en el vector */
   for ( i = 0; i < MAXELEM; i++ ) {
      printf ( "elemento %d: ", i ) ;
      scanf ( "%d", &vector [ i ] );
   }
```

```
    for ( i = 1; i < MAXELEM; i++ ) {
        x = vector [ i ] ;
/*      Insertar x en el lugar apropiado de vector [ 1 ].....vector [ i ] */
/*      Inicializar índice: */ j = i - 1;
/*      MIENTRAS x y vector [ j ] no estén en el orden correcto y j > 0 HACER  */
        while ( ( x < vector [ j ] ) && ( j >= 0 ) ) {
/*              Desplazar una posición a la derecha */
                vector [ j + 1 ] = vector [ j ] ;
/*              Actualizar índice */ j-- ;
        }
        vector [ j + 1 ] = x ;
    }
}
```

Programa C 9.1: ordenación ascendente de un vector de números enteros mediante el método de inserción directa.

9.3 Ordenación por selección.

9.3.1 Definición.

El método de ordenación por selección directa consiste también en dividir el conjunto de datos a ordenar en dos subconjuntos, el ordenado y el que todavía está desordenado. En cada pasada, el algoritmo selecciona el elemento que cumple el criterio de ordenación (creciente o decreciente) del subconjunto desordenado y lo coloca en el subconjunto ordenado. El algoritmo prosigue hasta que el subconjunto desordenado no consta de ningún elemento.

Conjunto inicial de datos a ordenar: a(1), a(2), ...a(n-1)

En un momento dado del algoritmo:

subconjunto ordenado: a(1), a(2),.....,a(i-1)

subconjunto desordenado: a(i), a(i+1).....,a(n-1).

Si el criterio de ordenación es el creciente, en cada pasada el algoritmo selecciona el elemento menor del subconjunto desordenado a(i), a(i+1),.....,a(n-1), y lo intercambia por el elemento a(i), de forma que pasa a engrosar el subconjunto ordenado. El método prosigue hasta que el subconjunto desordenado no consta de ningún elemento.

A continuación, en la Fig 9.2, ilustraremos el método de selección directa sobre un ejemplo numérico donde el criterio de ordenación es el creciente.

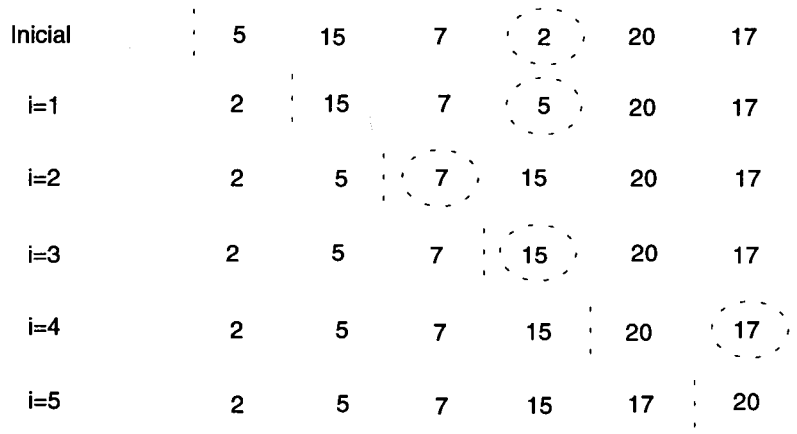


Fig 9.2: método de ordenación por selección directa.

Existen otros métodos de ordenación basados en la idea de la selección, entre los cuales citaremos el método del montículo o Heapsort. Este método se basa en ordenar el conjunto de datos según un árbol invertido, de forma que a lo largo del algoritmo cada elemento desciende por el árbol hasta el lugar que le corresponde según el criterio de ordenación. Por esta razón, al método del Heapsort también se le conoce con el nombre de ordenación mediante criba por hundimiento.

9.3.2 Algoritmo ejemplo.

```
ALGORITMO selección;

COMIENZO
Introducir los datos a ordenar en el vector;
DESDE i := 1 HASTA maxelem - 1 HACER
   COMIENZO
      Calcular minind: índice del elemento mínimo entre vector [ i ] y vector [ maxelem-1 ];
      Intercambiar vector [ minind ] con vector [ i ];
   FIN
FIN
```

Algoritmo 9.3: primer refinamiento del método de ordenación por selección.

Refinamientos:

Calcular minind: índice del elemento mínimo entre vector [i] y vector [maxelem - 1]
Inicializar índice mínimo: minind := i;
DESDE primer elemento subconjunto desordenado HASTA maxelem HACER
  SI elemento j-ésimo < elemento minind ENTONCES Actualizar minind ;

Algoritmo 9.4: refinamiento de la acción ***calcular minind....***

Intercambiar vector [minind] con vector [i]:
tmp := vector [minind] ;
vector [minind] := vector [i] ;
vector [i] := tmp ;

Algoritmo 9.5: refinamiento de la acción ***intercambiar vector [minind] con vector [i].***

9.3.3 Programa PASCAL

```
PROGRAM selección;
CONST    maxelem = 14 ;
VAR      vector: ARRAY [ 1..maxelem ] OF INTEGER;
         i, minind, j, tmp, x: INTEGER;

BEGIN
{Introducir los datos a ordenar en el vector}
FOR i:=1 TO maxelem DO
   BEGIN
   readln ( x ) ;
   vector [ i ] := x ;
   END;
FOR i := 1 TO maxelem-1 DO
{  Calcular minind: índice del elemento mínimo entre vector [ i ] y vector [ maxelem - 1 ] }
   BEGIN
{  Inicializar índice mínimo: }    minind := i ;
{  DESDE primer elemento subconjunto desordenado HASTA maxelem HACER }
   FOR j := i + 1 TO maxelem DO
{      SI elemento j-ésimo < elemento minind ENTONCES Actualizar minind  }
```

```
        IF (vector [ j ] < vector [ minind ] ) THEN minind := j ;
{   Intercambiar vector [ minind ] con vector [ i ] }
    tmp := vector [ minind ] ;
    vector [ minind ] := vector [ i ] ;
    vector [ i ] := tmp
    END
END.
```

Programa PASCAL 9.2: ordenación ascendente de un vector de números enteros mediante el método de selección directa.

9.3.4 Programa C.

```
#include <stdio.h>

#define MAXELEM 14

main ( )
{
    int vector [ MAXELEM ] ;
    int i, j, x, minind, tmp;

/*  Introducir los datos a ordenar en el vector*/
    for ( i = 0; i < MAXELEM; i++ ) {
        printf ( "elemento %d: ", i ) ;
        scanf ( "%d", &vector [ i ] ) ;
    }
/*  Método de ordenación por selección directa */
    for ( i = 0; i < MAXELEM - 1; i++ ) {
/*      Calcular minind: índice del elemento mínimo entre vector[i] y vector[maxelem-1]*/
/*      Inicializar índice mínimo */  minind := i ;
/*      DESDE primer elemento subconjunto desordenado HASTA maxelem HACER */
        for ( j = i + 1; j < MAXELEM ; j++ )
/*              SI elemento j-ésimo < elemento minind ENTONCES Actualizar minind  */
                if ( vector [ j ] < vector [ minind ] ) minind = j ;
/*      Intercambiar vector[minind] con vector[i]*/
        tmp = vector [ minind ] ;
        vector [ minind ] = vector [ i ] ;
        vector [ i ] = tmp ;
```

```
    }
}
```

Programa C 9.2: ordenación ascendente de un vector de números enteros mediante el método de selección directa.

9.4 Ordenación por intercambio.

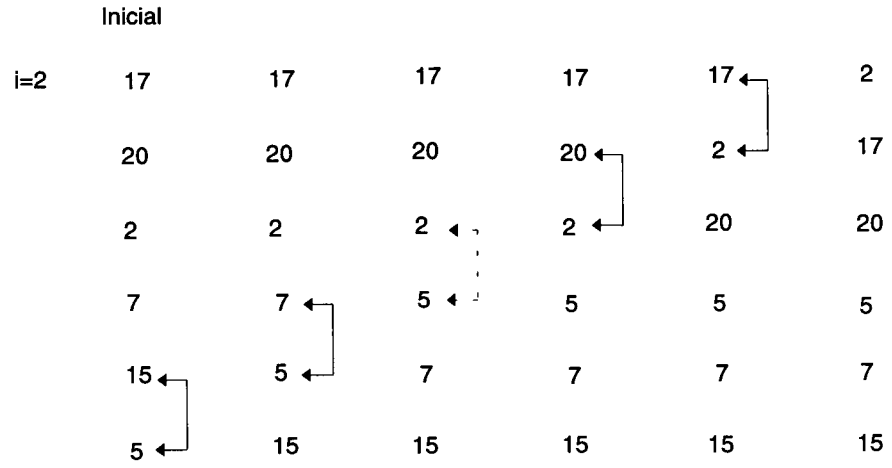
9.4.1 Definición.

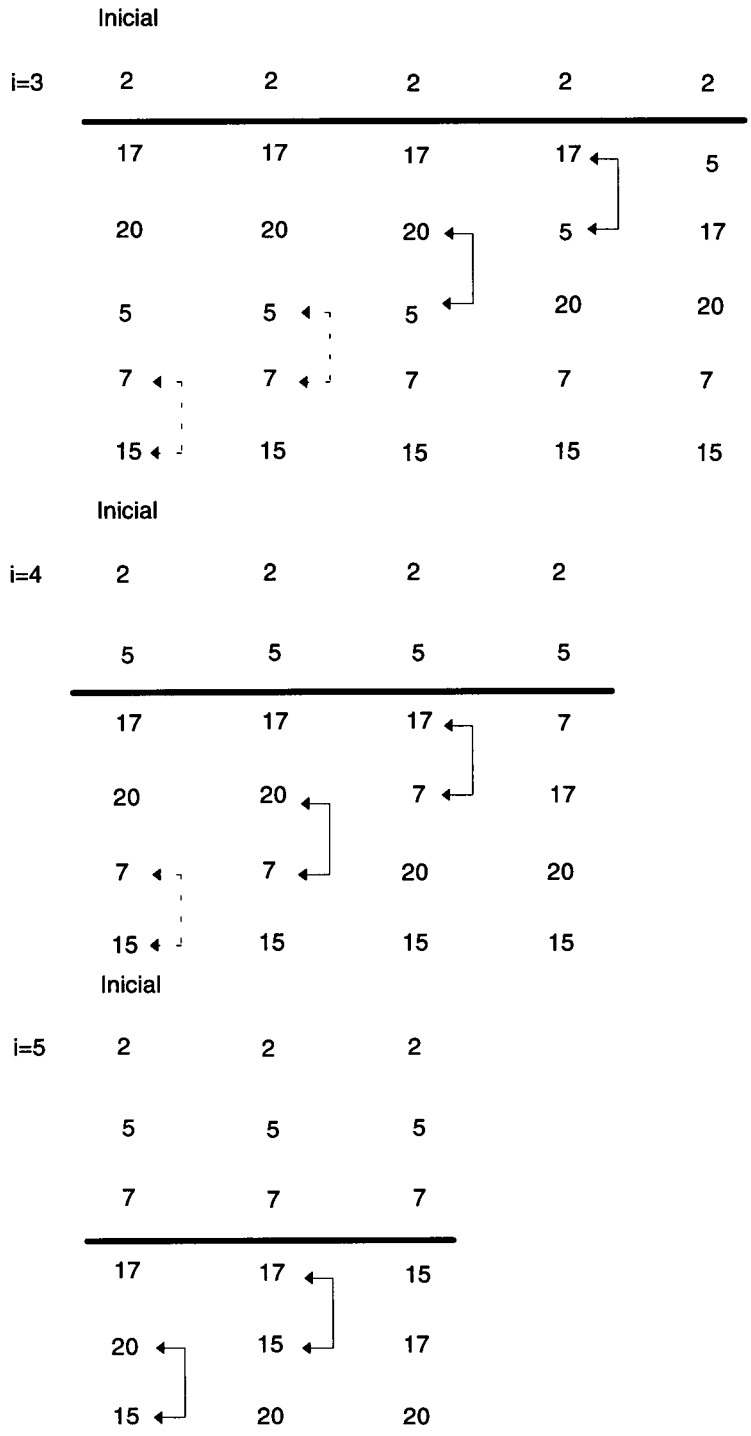
Vamos a describir dos métodos de ordenación basados en la idea del intercambio que son:

- método de la burbuja
- método rápido o Quicksort.

El método de la burbuja consiste en comparar pares de elementos adyacentes e intercambiarlos en el caso de que no estén bien situados según el criterio de ordenación utilizado. Para cada pasada del algoritmo conseguimos que el elemento menor, si la ordenación es creciente, suba como una burbuja hasta su lugar definitivo, pasando de esta forma a engrosar en un elemento el subconjunto ordenado. El método prosigue hasta que el subconjunto desordenado no consta de ningún elemento.

A continuación, en la Fig 9.3, ilustraremos mediante un ejemplo numérico el método de la burbuja donde el criterio de ordenación es el creciente.





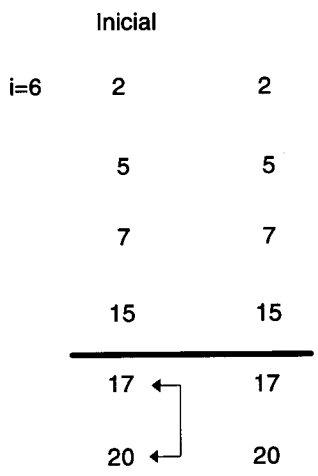


Fig 9.3: Método de ordenación de la burbuja.

El método de ordenación rápido o Quicksort es también un método de ordenación basado en el intercambio, pero en este caso los intercambios no se realizan entre elementos adyacentes, sino distanciados lo más posible. El método Quicksort está basado en la idea de partir el vector en dos particiones delimitadas por un elemento "central" tales que, al final del tratamiento de una partición, la partición izquierda contenga sólo elementos menores al "central" y la derecha mayores al "central". El método consiste en realizar particiones de las particiones hasta que consten de un solo elemento.

El tratamiento para una partición consiste en inspeccionar los elementos empezando por los dos extremos y progresando hacia el interior hasta encontrar parejas de elementos, uno de cada mitad, que no estén en la mitad adecuada según el criterio de ordenación utilizado. En la Fig 9.4 se muestra, mediante un gráfico, el tratamiento para una partición cualesquiera.

En el algoritmo que se muestra en el apartado siguiente, se ha escogido como elemento central el que está situado físicamente en la mitad de la partición. Se puede demostrar que la eficiencia del método aumenta considerablemente si se escoge la mediana como elemento central en las particiones. El algoritmo del método de ordenación Quicksort, que figura en el apartado siguiente, es recursivo. Se dice que un procedimiento es recursivo cuando puede llamarse a sí mismo; éste es el método que se utiliza en este algoritmo para realizar las particiones de las particiones.

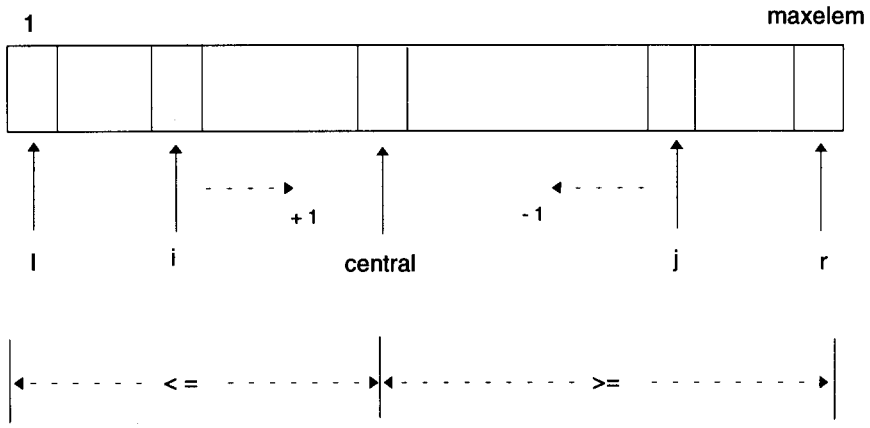


Fig 9.4: tratamiento de una partición del método de ordenación Quicksort.

9.4.2 Algoritmo ejemplo

1- Ordenación por intercambio: método de la burbuja.

```
ALGORITMO burbuja ;

PRINCIPIO
Introducir los datos a ordenar en el vector;
DESDE i:=2 HASTA maxelem HACER
    Ascender el elemento menor de vector[i]..vector[maxelem] al lugar que le corresponde;
FIN.
```

Algoritmo 9.6: primer refinamiento del método de ordenación de la burbuja.

Refinamiento:

```
Ascender el elemento menor de vector[i]..vector[maxelem] al lugar que le corresponde
DESDE i := 2 HASTA maxelem HACER
    DESDE j := maxelem DEC-HASTA i HACER
        Intercambiar vector [ j - 1 ] con vector [ j ] ;
```

Algoritmo 9.7: refinamiento de la acción ***ascender el elemento menor de vector[i].....***

2- Ordenación por intercambio: método Quicksort.

Algoritmo del tratamiento de una partición del método de ordenación Quicksort

```
ALGORITMO partición;

COMIENZO
Inicializar los índices de la partición;
Calcular el elemento central;
REPETIR
   Buscar en partición izquierda un elemento mayor que el central;
   Buscar en partición derecha un elemento menor que el central;
   SI existen ENTONCES intercambiarlos y avanzar los índices;
HASTA que se crucen los índices de la partición;
FIN.
```

Algoritmo 9.8: tratamiento de una partición del método de ordenación Quicksort.

Ordenación ascendente de un vector de números enteros mediante el método Quicksort recursivo

```
ALGORITMO Quicksort;

PROCEDIMIENTO partición ( l, r: entero ) ;
   VAR i, j, central: entero;
   COMIENZO
      Inicializar índices;
      Calcular el elemento central;
      REPETIR { Tratar la partición }
            Buscar partición izquierda elemento > central;
            Buscar partición derecha elemento < central;
            SI existen ENTONCES intercambiarlos y avanzar índices;
      HASTA que se crucen los índices de la partición;
      SI queda partición izquierda llamar recursivamente a partición;
      SI queda partición derecha llamar recursivamente a partición;
   FIN;

COMIENZO
   Inicializar primera partición;
FIN.
```

Algoritmo 9.9: ordenación ascendente de un vector de números enteros mediante el método Quicksort recursivo.

9.4.3 Programa PASCAL.

```
PROGRAM burbuja;

CONST     maxelem = 14;

VAR       vector: ARRAY [ 1..maxelem ] OF INTEGER;
          i, j, temp: INTEGER;

BEGIN
{Introducir los datos a ordenar en el vector }
FOR i := 1 TO maxelem DO
   BEGIN
   readln ( x ) ;
   vector [ i ] := x ;
   END;
{Método de ordenación de la burbuja}
FOR i:=2 TO maxelem DO
{Ascender el elemento menor de vector [ i ]..vector [ maxelem ] al lugar que le corresponde}
   BEGIN
   FOR j := maxelem DOWNTO i DO
      IF vector [ j - 1 ] > vector [ j ] THEN
{           Intercambiar vector [ j - 1 ] con vector [ j ] }
            BEGIN
            temp := vector [ j - 1 ] ;
            vector [ j - 1 ] := vector [ j ] ;
            vector [ j ] := temp;
            END
   END
END.
```

Programa PASCAL 9.3: ordenación ascendente de un vector de números enteros mediante el método de la burbuja.

A continuación se presenta el programa correspondiente al método de ordenación Quicksort recursivo:

```
PROGRAM Quicksort;

CONST     maxelem = 15;
```

```
VAR         vector: ARRAY [ 1..maxelem ] OF INTEGER;

PROCEDURE particion ( l, r:  INTEGER ) ;
VAR  i, j, central, temp: INTEGER;

BEGIN
{   Inicializar índices y calcular el elemento central }
    i := l; j := r; central := vector [ ( i + j ) DIV 2 ] ;
    REPEAT  {Partición}
{       Buscar en la partición derecha un elemento menor que el central}
        WHILE vector [ i ] < central DO i := i + 1;
{       Buscar en partición izquierda un elemento mayor que el central}
        WHILE central < vector [ j ]  DO j := j - 1 ;
              IF i <= j THEN     BEGIN
{                                Intercambiar vector [ i ] con vector [ j ] }
                                 temp := vector [ i ] ;
                                 vector [ i ] := vector [ j ] ;
                                 vector [ j ] := temp ;
{                                Avanzar los índices }
                                 i := i + 1 ; j := j - 1 ;
                                 END;
{   Repetir hasta que se crucen los índices }
    UNTIL i > j ; { Fin partición }
{   Si queda partición izquierda llamar recursivamente a partición }
    IF l < j THEN particion ( l, j ) ; { Partición izquierda }
{   Si queda partición derecha llamar recursivamente a partición }
    IF i < r THEN particion ( i, r ) ; { Partición derecha }
END; { Fin Partición }

BEGIN
{   Inicializar primera partición }
    particion ( 1, maxelem ) ;
END. { Fin Quicksort }
```

Programa PASCAL 9.4: ordenación ascendente de un vector de números enteros: método Quicksort recursivo.

9.4.4 Programa C.

```
#include <stdio.h>
```

```
#define MAXELEM 14

main ( )
{
   int vector [ MAXELEM ] ;
   int i, j, tmp;

/* Introducir los datos a ordenar en el vector*/
   for ( i = 0 ; i < MAXELEM ; i++ ) {
      printf ( "elemento %d: ", i ) ;
      scanf ( "%d", &vector [ i ] ) ;
   }
   for ( i = 1; i < MAXELEM - 1; i++ )
/* Ascender el elemento menor de vector[i]..vector[maxelem] al lugar que le corresponde*/
      for ( j = MAXELEM - 1; j >= i ; j-- )
            if ( vector [ j - 1 ] > vector [ j ] ) {
                  /*Intercambiar vector [ j - 1 ] con vector [ j ] */
                  tmp = vector [ j - 1 ] ;
                  vector [ j - 1 ] = vector [ j ] ;
                  vector [ j ] = tmp ;
            }
}
```

Programa C 9.3: ordenación ascendente de un vector de números enteros: método de la burbuja.

A continuación se presenta el programa C correspondiente al método de ordenación Quicksort recursivo:

```
#include <stdio.h>

#define MAXELEM 14

/*Comienzo de la función de tratamiento de una partición*/
void
particion ( int l, r )
{
      int i, j, central, tmp;

/*    Inicializar índices y calcular el elemento central */
```

```
        i = l ;
        j = r ;
        central = vector [ ( i + j ) / 2 ] ;
        do {  /* Partición */
/*              Buscar en partición derecha un elemento menor que el central */
                while  ( vector [ i ] < central )  i++;
/*              Buscar en partición izquierda un elemento mayor que el centra l*/
                while ( central < vector [ j ] )  j-- ;
                if ( i <= j )  {
/*                      Intercambiar vector [ i ] con vector [ j ]*/
                        tmp = vector [ i ] ;
                        vector [ i ] = vector [ j ] ;
                        vector [ j ] = tmp ;
/*                      Avanzar los índices*/
                        i++ ;
                        j-- ;
                }
/*      Repetir hasta que se crucen los índices*/
        } while ( ! ( i > j ) ) ; /* Fin partición */
/*      Si queda partición izquierda llamar recursivamente a partición*/
        if ( l < j )  particion ( l , j ) ;  /* Partición izquierda */
/*      Si queda partición derecha llamar recursivamente a partición*/
        if ( i < r ) particion ( i , r ) ;  /*Partición derecha*/
} /* Fin Partición*/

/*Comienzo del programa principal */
main ( )
{

   int     vector [ MAXELEM ] ;

/*  Introducir los datos a ordenar en el vector*/
   for ( i = 0 ; i < MAXELEM ; i++ )  {
      printf ( "elemento %d: ", i ) ;
      scanf ( "%d", &vector [ i ] ) ;
   }
   particion  ( 1, maxelem ) ;  /* Inicializar primera partición */
} /*Fin Quicksort*/
```

Programa C 9.4: ordenación ascendente de un vector de números enteros mediante el método Quicksort recursivo.

10. Métodos de búsqueda.

La operación de buscar un dato dentro de un conjunto dado es una de las operaciones más frecuentes dentro del proceso de datos y, por lo tanto, hemos creído conveniente incluir su estudio en este apartado. Como en el caso de la ordenación, la búsqueda ha sido desde siempre un problema muy estudiado y existen una gran variedad de algoritmos diferentes que resuelven el problema con mayor o menor fortuna.

Clasificaremos los algoritmos de búsqueda en función de si el conjunto de datos sobre el que se extiende la búsqueda está ordenado o no. En el caso de utilizar alguno de los métodos de búsqueda sobre un conjunto de datos ordenado es necesario, previamente a la búsqueda propiamente dicha, someter al conjunto de datos a un proceso de ordenación utilizando, por ejemplo, alguno de los algoritmos estudiados en el apartado anterior.

En este apartado vamos a estudiar únicamente dos algoritmos de búsqueda, que son los más sencillos:

- búsqueda secuencial,

- búsqueda binaria o dicotómica.

10.1 Búsqueda secuencial.

10.1.1 Definición.

El método de búsqueda secuencial es el más sencillo y se debe utilizar si el conjunto de datos sobre el que se extiende la búsqueda no está ordenado siguiendo criterio alguno. Como su nombre indica, consiste en recorrer el conjunto de datos uno a uno y comparar cada dato con el valor buscado hasta encontrarlo, si es que existe, o hasta el último dato sin éxito en la búsqueda, en el caso de que no exista.

10.1.2 Algoritmo ejemplo.

ALGORITMO secuencial;

PRINCIPIO
  Introducir el valor a buscar;
  Recorrer secuencialmente el vector sobre el que se realiza la búsqueda y comparar cada elemento con el valor buscado;
FIN.

Algoritmo 10.1: búsqueda secuencial de un valor x en un vector de maxelem números enteros.

10.1.3 Programa PASCAL.

```
PROGRAM busc_sec_desord;

CONST    maxelem = 15;
VAR      vector: ARRAY [ 1..maxelem ] OF INTEGER;
         i, x: INTEGER;

BEGIN
{Introducir el valor a buscar }
readln ( "El valor a buscar es" , x ) ;
i := 1;
{Recorrer secuencialmente el vector sobre el que se realiza la búsqueda y comparar cada elemento con el valor buscado}
WHILE NOT  ( ( vector [ i ] = x ) OR ( i = n ) DO i := i + 1 ;
IF vector [ i ] = x
   THEN writeln ( "El valor buscado tiene la componente i = " , i )
   ELSE writeln ( "El valor " , x, " no existe en el vector" ) ;
END.
```

Programa PASCAL 10.1: búsqueda secuencial de un valor x en un vector de maxelem números enteros no ordenados.

Vamos a modificar el algoritmo anterior para el caso de que el conjunto de datos esté ordenado de forma creciente:

```
PROGRAM busc_sec_ord;

CONST    maxelem = 15;
VAR      vector: ARRAY [ 1..maxelem ] OF INTEGER;
         i, x: INTEGER;

BEGIN
{Introducir el valor a buscar }
readln ( "El valor a buscar es" , x ) ;
i := 1;
{Recorrer secuencialmente el vector sobre el que se realiza la búsqueda y comparar cada elemento con
el valor buscado}
WHILE ( vector [ i ] < x ) AND ( i < n ) DO i := i + 1 ;
IF vector [ i ] = x
   THEN writeln ( "El valor buscado tiene la componente i = " , i )
   ELSE writeln ( "El valor ", x, " no existe en el vector " ) ;
END.
```

Programa PASCAL 10.2: búsqueda secuencial de un valor x en un vector de maxelem números enteros ordenados de forma creciente.

10.1.4 Programa C.

```
/*Búsqueda secuencial de un valor x en un vector de maxelem números enteros no ordenados*/

#include <stdio.h>

#define MAXELEM 14

main ( )
{
   int vector [ MAXELEM ] ;
   int i, x ;

/*  Introducir los valores sobre los que realizará la búsqueda */
   for ( i = 0; i < MAXELEM ; i++ ) {
      printf ( "elemento %d: ", i ) ;
      scanf ( "%d", &vector [ i ] ) ;
   }
/*  Introducir el valor a buscar */
```

```
    printf ( "El  valor a buscar es:" ) ;
    scanf ( "%d", &x ) ;
/*  Recorrer secuencialmente el vector sobre el que se realiza la búsqueda y comparar cada elemento
    con el valor buscado*/
    i = 0;
    while ( ! ( ( vector [ i ] == x )  | |  ( i == MAXELEM - 1 ) ) ) i++;
    if ( vector [ i ] == x )
        printf ( "El valor buscado tiene la componente i =  %d\n ",  i ) ;
    else
        printf ( "El valor %d no existe en el vector", x ) ;
}
```

Programa C 10.1: búsqueda secuencial de un valor x en un vector de maxelem números enteros no ordenados.

```
#include <stdio.h>

#define MAXELEM 14

main ( )
{
    int         vector [ MAXELEM ] ;
    int         i, x ;

    for ( i = 0; i < MAXELEM ; i++ )  {
        printf ( "elemento %d: " ,  i ) ;
        scanf ( "%d" , &vector [ i ] ) ;
    }
    printf ( "El  valor a buscar es:" ) ;
    scanf ( "%d" , &x ) ;

    i = 0 ;
    while ( ( vector [ i ] < x  &&  i < MAXELEM - 1 ) )  i++;
    if ( vector [ i ] == x )
        printf ( "El valor buscado tiene la componente i =  %d\n" ,  i ) ;
    else
        printf ( "El valor %d no existe en el vector" , x ) ;
}
```

Programa C 10.2: búsqueda secuencial de un valor x en un vector de maxelem números enteros ordenados de forma creciente.

10.2 Búsqueda binaria o dicotómica.

10.2.1 Definición.

En el caso de buscar sobre un conjunto ordenado, es posible optimizar el criterio de búsqueda gracias a esa ordenación. Estos métodos de búsqueda son muy eficientes, sobre todo para grandes cantidades de datos; es por esta razón que se suelen utilizar incluso sobre conjuntos de datos no ordenados, sometiéndolos a un proceso de ordenación previo a la búsqueda propiamente dicha. Uno de estos métodos, el más popular y sencillo de diseñar, es el método de búsqueda binaria o dicotómica, que estudiaremos en este apartado.

La búsqueda binaria o dicotómica se basa en aprovechar el hecho de que se trata de datos ordenados para extender la búsqueda únicamente a aquella porción de los datos donde, si el dato existe es seguro que se encuentra. En cada pasada del algoritmo se va reduciendo la zona donde buscar el dato hasta que se da con él, en el caso de que exista, o la zona a buscar queda reducida a cero, en el caso de que no exista el valor buscado.

En la Fig 10.1, ilustraremos el método de búsqueda binaria sobre un gráfico.

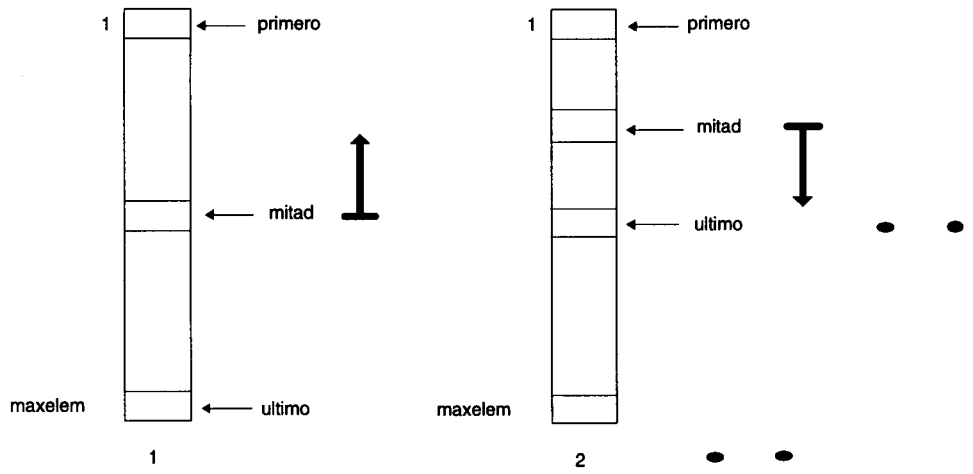


Fig 10.1: búsqueda binaria o dicotómica.

10.2.2 Algoritmo ejemplo.

ALGORITMO dicotomica;

```
CONST     maxelem = 15;
VAR       vector: MATRIZ [ 1..maxelem ] DE ENTERO;
          primero, ultimo, mitad: ENTERO;

PRINCIPIO
Inicializar punteros primero y último;
REPETIR
   Calcular puntero mitad;
   Averiguar en que mitad hay que seguir buscando x;
   Actualizar los punteros primero y último;
HASTA encontrar x o fin de la búsqueda;
Escribir si la búsqueda ha tenido éxito o no;
FIN.
```

Algoritmo 10.2: búsqueda de un valor x en un vector de maxelem números enteros ordenado de menor a mayor mediante búsqueda binaria o dicotómica.

10.2.3 Programa PASCAL.

```
PROGRAM dicotomica ;

CONSTmaxelem = 15;
VAR  vector: ARRAY [ 1..maxelem ] OF INTEGER;
     primero, ultimo, mitad: INTEGER;

BEGIN
{Inicializar punteros primero y último }
primero := 1 ;
ultimo := maxelem ;
REPEAT
{  Calcular puntero mitad }
   mitad := ( primero + ultimo ) DIV 2 ;
{  Averiguar en que mitad hay que seguir buscando x }
   IF x > vector [ mitad ]
{     Actualizar los punteros primero y último }
      THEN primero := mitad + 1
      ELSE ultimo := mitad - 1 ;
UNTIL ( vector [ mitad ] = x )  OR  ( primero > ultimo )
{Escribir si la búsqueda ha tenido éxito o no }
```

```
IF vector [ mitad ] = x
   THEN writeln ( "El valor " , x , " corresponde a i = " , mitad )
   ELSE writeln ( "El valor " , x , " no existe" )
END.
```

Programa PASCAL 10.3: búsqueda de un valor x en un vector de maxelem números enteros ordenado de menor a mayor mediante búsqueda binaria o dicotómica.

10.2.4 Programa C.

```
#include <stdio.h>

#define MAXELEM 14

main ( )
{
   int vector [ MAXELEM ] ;
   int i, x, primero, ultimo, mitad;

/* Introducir los valores sobre los que se realizará la búsqueda */
   for ( i = 0; i < MAXELEM; i++ ) {
      printf ( "elemento %d: " , i ) ;
      scanf ( "%d" , &vector [ i ] ) ;
   }
   printf ( "El valor a buscar es: " ) ;
   scanf ( "%d" , &x ) ;
/* Inicializar punteros primero y último */
   primero = 0 ;
   ultimo = MAXELEM - 1 ;
   do {
/*    Calcular puntero mitad */
      mitad = ( primero + ultimo ) / 2 ;
/*    Averiguar en que mitad hay que seguir buscando x */
      if ( x > vector [ mitad ] )
/*    Actualizar los punteros primero y último */
            primero = mitad + 1 ;
      else
            ultimo = mitad - 1 ;
   } while ( !( ( vector [ mitad ] == x ) || ( primero > ultimo ) ) ) ;
```

```
/* Escribir si la búsqueda ha tenido éxito o no */
    if ( vector [ mitad ] == x )
        printf ( "El valor buscado tiene la componente %d\n" , mitad ) ;
    else
        printf ( "El valor %d no existe en el vector\n" , x ) ;
}
```

Programa C 10.3: búsqueda de un valor x en un vector de maxelem números enteros ordenado de menor a mayor mediante búsqueda binaria o dicotómica.

11. Métodos numéricos.

Existen una serie de problemas científicos y tecnológicos que, debido a su complejidad, no tienen una solución analítica y, por lo tanto, no pueden ser resueltos utilizando los recursos matemáticos tradicionales. Este tipo de problemas, gracias a la aparición de los computadores, pueden resolverse mediante el cálculo numérico.

En este apartado vamos a estudiar algunos de los problemas clásicos en ingeniería y para cada uno de ellos hemos seleccionado un ejemplo algorítmico que resuelve el problema con sencillez. En concreto estudiaremos los casos siguientes:

- interpolar el valor de una función usando polinomios de Lagrange,

- cálculo de las raíces de una función mediante el método de Newton-Raphson,

- integración númerica de una función mediante el método del trapecio.

- resolución de sistemas de ecuaciones: método iterativo de Gauss-Seidel.

11.1 Interpolación: polinomios de Lagrange.

11.1.1 Definición.

Cuando tenemos los valores de una función en una serie de puntos, pero no disponemos de su expresión analítica, es necesario utilizar métodos de interpolación numérica para calcular el valor de la función en cualquier otro punto. Podemos clasificar los métodos de interpolación en función de varios criterios como, por ejemplo, si el conjunto de puntos de los que se dispone son equidistantes o no, y también en función del tipo de función utilizada para aproximar la función original (recta, parábola, etc.).

En este apartado vamos a estudiar un método de integración numérica basado en la utilización de un polinomio de un grado dado para aproximar la función. Este método únicamente es válido para el caso de que los puntos este

én irregularmente espaciados y además utiliza todos los puntos para interpolar el valor de la función en un punto diferente al conjunto de puntos original.

La expresión algebraica para la interpolación numérica del valor de una función $f[x_1,x_2,.....,x_{nmax})$ en un punto k mediante polinomios de Lagrange de grado nmax es la siguiente:

$$Pn(k) = \sum_{i=1..nmax} Li(k) * f(xi)$$

$$Li(k) = \prod_{j=1..nmax\ y\ i<>j} (k - xj) / (xi - xj)$$

11.1.2 Algoritmo ejemplo.

```
ALGORITMO Lagrange;
PRINCIPIO
   Entrada de los puntos: Vector x;
   Entrada valores de la función en esos puntos: Vector función;
   Entrada del punto en el que se desea interpolar: k;
   Calcular el valor de la función en el punto k según: Polinomios Lagrange;
   Escribir el valor de la función en el punto k;
FIN.
```

Algoritmo 11.1: interpolar el valor de una función en un punto k mediante un polinomio de Lagrange de grado nmax.

11.1.3 Programa PASCAL.

```
PROGRAM Lagrange;

CONST  nmax=20;
VAR    i, j, k: INTEGER;
       sum, prod: REAL;
       funcion: ARRAY [ 1..nmax ] OF REAL;
       x: ARRAY [ 1..nmax ] OF INTEGER;

BEGIN
{  Entrada de los puntos: Vector x}
```

```
{  Entrada valores de la función en esos puntos: Vector función}
{  Entrada del punto en el que se desea interpolar: k}
   readln ("Punto en el que se desea interpolar = ",k);
   prod := 1.0;
   sum := 0.0;
{  Calcular el valor de la función en el punto k: Polinomios de Lagrange}
   FOR i := 1 TO nmax DO
      BEGIN
      FOR j := 1 TO nmax DO
            IF i <> j THEN prod := prod * ( ( k - x [ j ] ) / ( x [ i ] - x [ j ] ) );
      sum := sum + prod * funcion ( i ) ;
      END;
{  Escribir el valor de la función en el punto k }
   writeln ( "El valor de la función en el punto ", k," es ", sum ) ;
END.
```

Programa PASCAL 11.1: interpolar el valor de una función en un punto k mediante un polinomio de Lagrange de grado nmax.

11.1.4 Programa C.

```
#include <stdio.h>

#define NMAX 20

main ( )
{
   int i, j, k, tmp ;
   float sum, prod ;
   float funcion [ NMAX ] ;
   int x [ NMAX ] ;
/*  Entrada de los puntos: Vector x*/
/*      Entrada valores de la función en esos puntos Vector función*/
   for ( i = 0; i < NMAX;  i++ ) {
      printf ( "punto %d: ", i ) ;
      scanf ( "%d", &x [ i ] ) ;
      printf ( "valor a %d: ", x [ i ] ) ;
      scanf ( "%f", &funcion [ i ] ) ;
   }
```

```
/* Entrada del punto en el que se desea interpolar: k */
   printf ( "punto en el que se desea interpolar: " ) ;
   scanf ( "%d", &k ) ;

   prod = 1.0;
   sum = 0.0;
/* Calcular el valor de la función en el punto k: Polinomios de Lagrange*/
   for ( i = 0; i < NMAX; i++ ) {
      for ( j = 0; j < NMAX; j++)
            if ( i != j )
                  prod* = ( ( k - x [ j ] ) / ( float ) ( x [ i ] - x [ j ] ) ) ;
      sum+ = prod * funcion [ i ] ;
   }
/* Escribir el valor de la función en el punto k */
   printf ( "el valor de la función en punto %d es %f\n", k, sum ) ;
}
```

Programa C 11.1: interpolar el valor de una función en un punto k mediante un polinomio de Lagrange de grado nmax.

11.2 Raices de funciones: Método de Newton-Raphson.

11.2.1 Definición.

El problema del cálculo de las raíces de una función f(x) consiste en encontrar el valor de x para el cual se cumple que f(x) = 0. Existen un gran número de métodos numéricos que resuelven este problema; nosotros vamos a estudiar uno de los más conocidos, que es el método de Newton-Raphson. Este método sólo es válido para raíces reales, si la función tiene raíces complejas es necesario utilizar otro tipo de métodos. El método es iterativo y está basado en aproximarse en cada pasada del algoritmo a la raíz de la función, a partir de una solución inicial. El método para calcular el nuevo valor de x en cada pasada se basa en el cálculo de la tangente a la función en ese punto. Esto hace que este método sólo se pueda utilizar para aquellos casos en que es posible calcular la derivada primera de la función.

$$x_{i+1} = x_i - f(x_i) / f'(x_i).$$

En la siguiente Fig 11.1 se ilustra gráficamente el método de Newton-Raphson para el cálculo de raíces de una función.

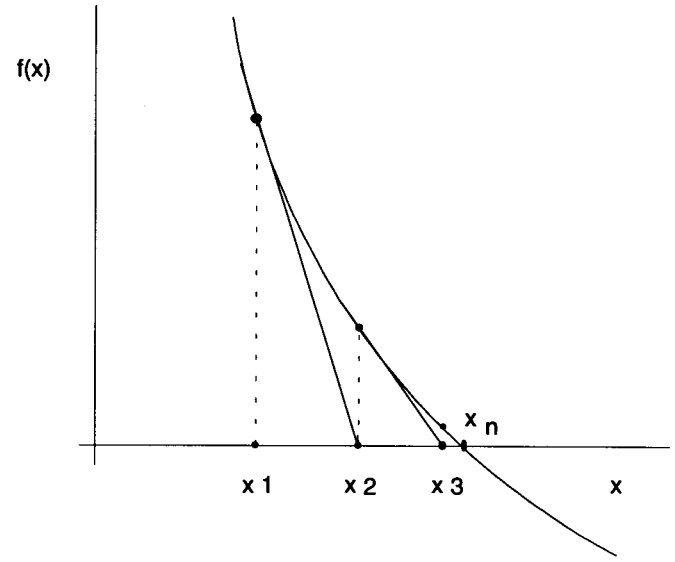


Fig 11.1: cálculo de la raiz de una función: método de Newton-Raphson.

Como en todo método iterativo es necesario verificar el criterio de convergencia. En el algoritmo y los programas de los siguientes apartados se limita el número máximo de iteraciones (nmax) que se aceptan para que el valor de x converja hacia cero y se fija el error permitido en la solución (épsilon).

11.2.1 Algoritmo ejemplo.

```
ALGORITMO Newton;

PRINCIPIO
Primera aproximación de la solución;
Cálculo de la función, fx y su derivada primera, dfx;
   MIENTRAS la función no se acerce suficientemente a cero HACER
      PRINCIPIO
      Calcular fx y dfx;
      Calcular nuevo x según método Newton-Raphson;
      FIN;
SI converge  ENTONCES Escribir el valor de la raíz
             SINO Indicar que no converge para nmax iteraciones;
```

FIN.

Algoritmo 11.2: encontrar las raíces de una función mediante el método de Newton-Raphson.

11.2.3 Programa PASCAL.

```
PROGRAM Newton;

CONST     epsilon = 1.0E-6;
          nmax = 100;
VAR  k: INTEGER;
     x: REAL;

{Cálculo de la función, fx y su derivada primera, dfx}
PROCEDURE funcion ( x: REAL; VAR fx, dfx: REAL ) ;
BEGIN
   fx :=  x * x  - 2;
   dfx := 2 * x;
END;

BEGIN
x := 2.0 ; { Primera aproximación de la solución }
k := 1;
funcion ( x, fx, dfx ) ;
WHILE (abs ( fx ) >= epsilon ) AND ( k < nmax ) DO
   BEGIN
{  Calcular fx y dfx }
   funcion ( x, fx, dfx ) ;
{  Calcular nuevo x según método Newton-Raphson }
   x := x - ( fx / dfx );
   k := k + 1;
   END;
{Averiguar si la solución converge}
IF k < nmax THEN writeln ( x )
             ELSE writeln ( "No converge con ", nmax, " iteraciones" ) ;
END.
```

Programa PASCAL 11.2: encontrar las raíces de una función mediante el método de Newton-Raphson.

11.2.4 Programa C.

```
#include <stdio.h>
#include <math.h>

#define EPSILON 1.0E-6
#define NMAX 100;

/*Cálculo de la función: fx */
float
funcion ( float x )
{
   return  x * x - 2;
}

/*Cálculo la derivada primera de la función: dfx*/
float
derivada ( float x )
{
   return  2 * x;
}
/*Comienzo del programa principal */
main ( )
{
   int k;
   float x, fx, dfx;

   x = 2.0 ; /*Primera aproximación de la solución */
   k = 1;
   fx = funcion ( x ) ;
   dfx = derivada ( x ) ;

   while ( ( sqrt ( fx * fx ) >= EPSILON ) && ( k < NMAX ) ) {
/*      Calcular fx y dfx */
      fx = funcion ( x );
      dfx = derivada ( x );
/*      Calcular nuevo x según método Newton-Raphson */
      x - = fx / dfx;
      k ++;
   }
/*  Averiguar si la solución converge*/
   if ( k < NMAX )
```

```
        printf ( "La raíz es: %f\n", x ) ;
    else
        printf ( "no converge con %d iteraciones\n", NMAX ) ;
}
```

Programa C 11.2: encontrar las raíces de una función mediante el método de Newton-Raphson.

11.3 Integración numérica: Método del trapecio.

11.3.1 Definición.

El método de integración numérica mediante la regla del trapecio se basa en descomponer la integral a calcular en una serie de secciones iguales, y calcular la suma acumulada del valor de la integral en cada una de dichas secciones. En cada sección se utiliza una recta para aproximar el valor de la función. El método es válido únicamente para el caso en que las secciones sean equidistantes y disten:

$$Dx = (b - a) / n,$$

siendo a y b los límites de integración, y n el número de secciones en las que se divide la integral.

En definitiva, la expresión que nos permite calcular la integral de la función mediante el método del trapecio es la siguiente:

$$Integral = [f(a) + 2^* \sum_{i=1..n-1} f(i) + f(b)] * Dx / 2.$$

En la Fig 11.2 se muestra, mediante un gráfico, la diferencia entre la aproximación a la integral calculada mediante el método del trapecio (zona sombreada) y la integral real.

Es posible mejorar la aproximación del valor de la integral utilizando el método de integración numérica de Simpson, que está basado en utilizar una parábola para aproximar el valor de la función en cada una de las secciones en las que se descompone la integral. En este caso la expresión sería la siguiente:

$$Int\_Simpson = [f(a) + 4^* \sum_{i=1..n-1 \text{ y i impar}} f(i) + 2^* \sum_{i=1..n-2 \text{ y i par}} f(b)] * Dx / 3$$

siendo:

Dx = (b-a) / n
n = número de secciones en que se divide la integral.

11.3.2 Algoritmo ejemplo.

```
ALGORITMO Trapecio;

PRINCIPIO
   Inicializar los límites de integración y el número de secciones;
   Calcular el valor del intervalo dx;
   Calcular los valores de la función en los límites de integración;
   DESDE i := 1 HASTA número de secciones - 1 HACER
      Calcular la integral de la sección i-ésima y acumular;
   Calcular el valor de la integral según el método del trapecio: la integral de cada sección
   más el valor de la función en los límites;
   Escribir el valor de la integral;
FIN
```

Algoritmo 11.3: integrar una función f (x) entre a y b mediante el método del trapecio.

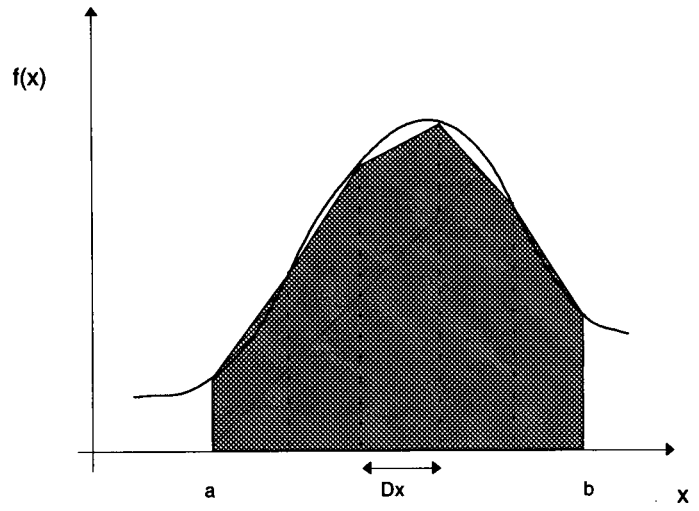


Fig 11.2: integración numérica mediante el método del trapecio.

11.3.3. Programa PASCAL.

```
PROGRAM Trapecio;

VAR  integral, inferior, superior: REAL;
       n: INTEGER;

{Función fx }
FUNCTION fx ( x: REAL ) : REAL ;
BEGIN
   fx := 1.0 / x ;
END;

{Cálculo de la integral de la función fx mediante el método del trapecio }
PROCEDURE Trapecio ( inferior, superior: REAL; n: INTEGER; var integral: REAL ) ;

VAR  i: INTEGER; dx, x, suma, suma: REAL;

BEGIN
{   Calcular el valor del intervalo dx }
   dx := ( superior - inferior ) / n ;
{   Calcular los valores de la función en los límites de integración }
   suma1 := fx ( inferior ) + fx ( superior ) ;
   suma := 0.0 ;
   FOR i := 1 TO n - 1 DO
      BEGIN
      x := inferior + i * dx ;
      suma := suma + fx ( x ) ;
      END;
   integral := ( suma1 + 2 * suma ) * dx * 0.5 ;
END;

BEGIN
{   Inicializar los límites de integración y el número de secciones }
   inferior := 1.0 ; superior := 9.0 ; n := 10 ;
{   Calcular la integral de la función mediante el método del trapecio }
   Trapecio ( inferior, superior, n, integral );
{   Escribir el valor de la integral }
   writeln ( "La integral es ", integral : 9 : 5 )
END.
```

Programa PASCAL 11.3: integrar una función f(x).

11.3.4 Programa C.

```
#include <stdio.h>
#include<math.h>

/*Función fx */
float
funcion ( float x )
{
   return  1.0 / x ;
}

/*Cálculo de la integral de la función fx mediante el método del trapecio */
float
trapecio ( float  inferior, float  superior, int  n )
{
   int i ;
   float dx, x, suma, suma1;

/*  Calcular el valor del intervalo dx */
   dx = ( superior - inferior ) / ( float ) n ;
/*  Calcular los valores de la función en los límites de integración */
   suma1 = funcion ( superior ) + funcion ( inferior) ;
   suma = 0.0 ;
   for ( i=1; i <= n - 1; i++ ) {
      x = inferior + i * dx ;
      suma+ = funcion ( x ) ;
   }
   return ( suma1 + 2 * suma ) * dx * 0.5 ;
}

main ( )
{
   float inferior, superior, integral;
   int n;

/*  Inicializar los límites de integración y el número de secciones */
   inferior = 1.0;
   superior = 29.0;
   n = 10;
/*  Calcular la integral de la función mediante el método trapecio */
   integral = trapecio ( inferior, superior, n ) ;
```

```
/* Escribir el valor de la integral */
   printf ( "la integral es %f\n", integral ) ;
}
```

Programa C 11.3: integrar una función f(x) entre a y b mediante el método del trapecio.

11.4 Resolución de sistemas de ecuaciones: método de Gauss-Seidel.

11.4.1 Definición.

Existen una gran número de métodos numéricos para resolver sistemas de ecuaciones que podemos clasificar en métodos iterativos y no iterativos o directos. Los métodos no iterativos se basan en calcular directamente la solución utilizando algunas de la técnicas algebraicas para resolver sistemas de ecuaciones. El inconveniente que presentan estos métodos es el error de redondeo acumulado, sobre todo si el número de ecuaciones es grande. Algunos de los métodos directos más utilizados para resolver sistemas de ecuaciones son:

- método de Cramer basado en el cálculo de determinantes,
- métodos de eliminación de Gauss y Gauss-Jordan ambos basados en utilizar técnicas de pivotación de la matriz de coeficientes.

El método de Gauss-Seidel es iterativo, es decir, a partir de una solución inicial para el vector de las variables incógnitas se obtiene una nueva aproximación en cada pasada del algoritmo, hasta que dispongamos de una aproximación a la solución suficientemente válida. El problema de los métodos iterativos es asegurar la convergencia hacia la solución, cada método tiene su criterio de convergencia que debe ser verificado previamente.

A continuación vamos a describir el método iterativo de Gauss-Seidel para resolver un sistema de n ecuaciones.

Dado el siguiente sistema de n ecuaciones:

$$a_{11}\,x_1 + a_{12}\,x_2 + \ldots\ldots + a_{1n-1}\,x_{n-1} + a_{1n}\,x_n = u_1$$

$$a_{21}\,x_1 + a_{22}\,x_2 + \ldots\ldots + a_{2n-1}\,x_{n-1} + a_{2n}\,x_n = u_2$$

$$\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots$$

$$a_{n1}\,x_1 + a_{n2}\,x_2 + \ldots\ldots + a_{nn-1}\,x_{n-1} + a_{nn}\,x_n = u_n$$

Donde:

$[a_{ij}]$ es la matriz de coeficientes $(1 \leq i,,j \leq n)$

$[x_i]$ es el vector de las variables incógnitas $(1 \leq i \leq n)$

$[u_i]$ es el vector de los términos independientes $(1 \leq i \leq n)$

El sistema se basa en despejar en cada ecuación i-ésima la variable x_i de la siguiente forma:

$$x_1 = 1 / a_{11} (u_1 - a_{12} x_2 - a_{13} x_3 - \ldots\ldots - a_{1n} x_n)$$

$$x_2 = 1 / a_{22} (u_2 - a_{21} x_1 - a_{22} x_2 - \ldots\ldots - a_{2n} x_n)$$

$$\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots$$

$$x_n = 1 / a_{nn} (u_n - a_{n1} x_1 - a_{n2} x_2 - \ldots\ldots - a_{nn-1} x_{n-1})$$

Se puede demostrar que el criterio de convergencia en el caso del método de Gausss-Seidel es que la matriz de coeficientes del sistema de ecuaciones sea diagonalmente predominante, es decir, para todo i se debe cumplir:

$$a_{ii} > \Sigma_{(j=1..n,\, i \neq j)} | a_{ij} | \quad (1 \leq i,,j \leq n)$$

Si llamamos k a la iteración k-ésima, en general la expresión sería la siguiente:

$$x_{i,k} = 1 / a_{ii} (u_i - a_{i1} x_{1,k} - a_{i2} x_{2,k} - \ldots\ldots - a_{ii-1} x_{i-1,k} - a_{ii+1} x_{i+1,k-1} - \ldots\ldots - a_{in} x_{n,k-1})$$

Siendo:

$x_{i,k}$ la aproximación de la variable x_i de la iteración k-ésima

$x_{i,k-1}$ la aproximación de la variable x_i de la iteración (k-1)-ésima

11.4.2 Algoritmo ejemplo.

ALGORITMO Gauss-Seidel;

COMIENZO
  Introducir matriz coeficientes a [i, j] con 1 ≤ i, j ≤ n ;

```
   Introducir vector términos independientes u [ i ] con 1 ≤ i ≤ n ;
   Introducir la solución inicial x [ i ] con 1 ≤ i ≤ n ;
   REPETIR {bucle para cada iteración}
      máximo := 0; {máxima diferencia entre x [ i ] de iteraciones consecutivas }
      PARA i := 1 HASTA n HACER {bucle para cada ecuación }
         COMIENZO
            SI i ≠ 1 ENTONCES Sumar los términos 1,....., i - 1 ;
            SI i ≠ n ENTONCES Sumar los términos i + 1,....., n ;
            Calcular x [ i ]_aproximado ;
            SI | x [ i ]_aproximado - x [ i ] | > maximo
                  ENTONCES maximo := | x [ i ]_aproximado - x[ i ] |;
            x [ i ] := x [ i ]_aproximado;
         FIN;
   HASTA número máximo de iteraciones ó máximo < error_permitido
FIN.
```

Algoritmo 11.4: resolver un sistema de n ecuaciones mediante el método iterativo de Gauss-Seidel.

NOTA: Si a pesar de verificar el criterio de convergencia del método de Gauss-Seidel el programa no converge, puede ser debido al número máximo de iteraciones o al error permitido para la solución.

11.4.3 Programa PASCAL.

```
PROGRAM Gauss_Seidel;

CONST    n = 10 ; { n es el número de ecuaciones }
         kmax = 50 ; { kmax es el número máximo de iteraciones }
         error = 1.0E-6 ; {error permitido en la solución}

TYPE     matriz =  ARRAY [ 1..n, 1..n ] OF REAL;
         vector = ARRAY [ 1..n ] OF REAL;
VAR      a: matriz; { a [ i, j ] es la matriz de coeficientes }
         {u [ i ] es el vector de términos independientes y x [ i ] el vector de variables incógnitas}
         u,x: vector;
         i, j, k, m:  INTEGER;
         maximo, suma, aproximado: REAL;

BEGIN
{   Leer la matriz de coeficientes a [ i, j ],  1 ≤   i, j ≤   n }
```

```
{  Leer el vector de términos independientes u [ i ], 1 ≤  i ≤  n }
{  Leer la solución inicial x[ i ], 1 ≤  i ≤  n }
   k := 1; { k es el número de iteración }
   REPEAT        { bucle para cada iteración }
{     maximo = mayor diferencia entre dos aproximaciones consecutivas de x [ i ] }
      maximo := 0;
      writeln ( "Número de iteración = ", k ) ;
      FOR i := 1 TO n DO     { bucle para cada ecuación}
         BEGIN
           suma := 0 ;
           IF i <> 1 THEN     { términos 1,.., i - 1 }
              BEGIN
                 m := i - 1; j := 1;
                 WHILE j <= m DO
                    BEGIN
                       suma := suma + a [ i, j ] * x [ j ] ;
                       j := j + 1;
                    END;
              END;     { fin de términos 1,...,i - 1 }
           IF i <> n THEN     {términos i + 1,...., n }
              BEGIN
                 j := i + 1;
                 WHILE j <=  n DO
                    BEGIN
                       suma := suma + a [ i, j ] * x [ j ];
                       j := j + 1;
                    END;
              END;     { fin de términos i + 1,...., n }
           aproximado := ( u [ i ] - suma ) / a [ i, i ] ;
           IF ABS ( aproximado - x [ i ] ) > maximo)
                       THEN maximo := ABS ( aproximado - x [ i ] ) ;
           x [ i ] := aproximado;
         END; { fin bucle para cada ecuación}
{     Imprimir vector x [ i ], 1 ≤  i ≤  n  para iteración k-ésima}
      k := k + 1; { Incrementar el número de iteración }
   UNTIL (k > kmax) OR (maximo < error) { Fin bucle para cada iteración }
END.
```

Programa PASCAL 11.4: resolver un sistema de n ecuaciones mediante el método iterativo de Gauss-Seidel.

11.4.4 Programa C.

```
#include <stdio.h>
#include <math.h>

#define N 10  /* N es el número de ecuaciones */
#define KMAX 50 /* KMAX es el número máximo de iteraciones */
#define ERROR 1.0E-6  /* error permitido en la solución */

typedef double matriz [ N ] [ N ];
typedef vector [ N ];

main ( )
{
   matriz   a;
   vector   u, x;
   int      i, j, k, m;
   double   maximo, suma, aproximado;

/* Leer la matriz de coeficientes a [ i, j ], 1 ≤  i, j ≤  n */
/* Leer el vector de términos independientes u [ i ], 1 ≤  i ≤  n */
/* Leer la solución inicial x[ i ], 1 ≤  i ≤  n */
   k = 1;  /* k es el número de iteración */
   do {     /* bucle para cada iteración */
/*    maximo = mayor diferencia entre dos aproximaciones consecutivas de x [ i ] */
      maximo = 0;
      printf ( "Número de iteración =  %d\n", k ) ;
      for ( i = 0; i < N; i++ )  {  /*bucle para cada ecuación*/
            suma = 0 ;
            if ( i != 0 ) {         /* términos 1 ,....., i - 1 */
                  m = i - 1; j = 0 ;
                  while ( j <= m ) {
                        suma + = a [ i ] [ j ] * x [ j ] ;
                        j ++ ;
                  }
            }                       /* fin términos  1 ,....., i - 1 */
            if  ( i != N - 1 )  {   /* términos  i + 1 ,....., N - 1 */
                  j = i + 1 ;
                  while ( j <= N ) {
                        suma + = a [ i ] [ j ] * x [ j ] ;
                        j++;
                  }
```

```
            }                   /* fin términos i + 1 ,....., N - 1*/
            aproximado = ( u [ i ] - suma ) / a [ i ] [ i ] ;
            if  ( fabs ( aproximado - x [ i ] ) > maximo )
                                maximo = fabs (aproximado - x [ i ] ) ;
            x [ i ] = aproximado ;
        }           /* fin de bucle para cada ecuación */
/*      Imprimir vector x [ i ] , 0 ≤  i ≤  N - 1 para la iteración k-ésima */
        k++;  /*Incrementar el número de iteración*/
        } while ( ( k <= KMAX ) && ( maximo >= ERROR ) );/*fin bucle para cada iteración*/
}
```

Programa C 11.4: resolver un sistema de n ecuaciones mediante el método iterativo de Gauss-Seidel.

12. Tipos de datos avanzados.

En los apartados precedentes hemos estudiado tipos de datos simples estándar como enteros, reales o carácteres, y tipos de datos estructurados como vectores, matrices, registros o ficheros. La mayor parte de estos tipos de datos son directamente utilizables mediante la mayor parte de lenguajes de programación de alto nivel. En los programas de usuario necesitaremos utilizar estructuras de datos adaptadas al problema que se pretende resolver y lo más probable es que el programador deba diseñarlas a medida. De todas formas existen una serie de estructuras de datos muy utilizadas en un gran número de aplicaciones que merecen un tratamiento especial, como son: pilas, colas, listas, árboles, grafos, etc. En este capítulo, que hemos llamado tipos de datos avanzados vamos a estudiar algunas de estas estructuras, en concreto, veremos pilas y colas estáticas. Para cada una de ellas, y a nivel algorítmico, definiremos la estructura de datos y las operaciones permitidas sobre cada una de ellas. Construiremos sendas representaciones de la estructura y de las operaciones utilizando los lenguajes de programación Pascal y C.

12.1 Estructura de datos tipo PILA.

12.1.1 Definición.

La estructura de datos tipo pila consiste en un conjunto ordenado de elementos iguales de forma que únicamente es posible añadir o suprimir elementos en una posición fija llamada cabeza de la pila. La estructura pila es de tipo LIFO (Last In, First Out), ya que el último elemento introducido en la pila será el primer elemento que se suprimirá. La estructura de tipo pila que diseñaremos será un registro que constará de dos campos: un campo reservado para los elementos de la pila que almacenaremos en un vector y el otro para indicar la cabeza de la pila.

Definiremos una serie de operaciones que es posible realizar sobre una variable de tipo pila, como son:

- InicializarPila: crear una pila vacía.
- PilaVacía: averiguar si una pila está vacía.
- Pila Llena: averiguar si una pila está llena.

- PonerPila: añadir un elemento a la cabeza de la pila.
- SacarPila: sacar el elemento de la cabeza de la pila.

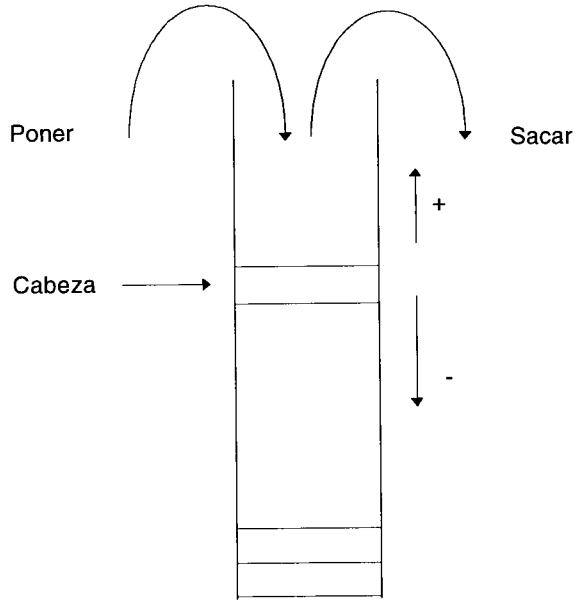


Figura 12.1: estructura de datos tipo PILA (LIFO).

InicializarPila (Pila)

Función:	Inicializar la pila a estado vacío
Entrada:	Pila a inicializar
Salida:	Pila inicializada
Precondiciones:	Ninguna
Postcondiciones:	Pila vacía

PilaVacía (Pila)

Función:	Averiguar si la pila está vacía
Entrada:	Pila a comprobar
Salida:	Valor booleano que indica si la pila está vacía
Precondiciones:	Ninguna
Postcondiciones:	Indica si la pila está vacía

PilaLlena (Pila)

Función:	Averiguar si la pila está llena
Entrada:	Pila a comprobar
Salida:	Valor booleano que indica si la pila está llena

Precondiciones: Ninguna
Postcondiciones: Indica si la pila está llena

PonerPila (Pila, NuevoElemento)
Función: Añadir NuevoElemento a la pila
Entrada: Pila y elemento a añadir (Pila, NuevoElemento)
Salida: Pila con NuevoElemento añadido (Pila)
Precondiciones: Pila no está llena
Postcondiciones: Pila convenientemente actualizada

SacarPila (Pila, ElementoSacado)
Función: Sacar ElementoSacado de la pila
Entrada: Pila (Pila)
Salida: Pila y elemento sacado (Pila, ElementoSacado)
Precondiciones: Pila no está vacía
Postcondiciones: Pila convenientemente actualizada

12.1.2 Algoritmo ejemplo.

```
PROCEDIMIENTO InicializarPila ( VAR Pila: TipoPila ) ;
PRINCIPIO
   Inicializar la cabeza de la pila a cero;
FIN;
```

Algoritmo 12.1: operación de inicializar una pila estática.

```
FUNCION PilaVacía ( Pila: TipoPila ) ;
PRINCIPIO
   Pila está vacía si la cabeza de la pila es igual a cero;
FIN;
```

Algoritmo 12.2: operación de averiguar si una pila estática está vacía.

```
FUNCION PilaLlena ( Pila: TipoPila );
PRINCIPIO
   Pila está llena si la cabeza de la pila vale el número máximo de elementos de la cola;
FIN;
```

Algoritmo 12.3: operación de averiguar si una pila estática está llena.

```
PROCEDIMIENTO PonerPila  ( VAR Pila: TipoPila; NuevoElemento: TipoElemento );
```

```
PRINCIPIO
   Incrementar el apuntador a la cabeza de la pila;
   Añadir NuevoElemento en la cabeza de la pila;
FIN;
```

Algoritmo 12.4: operación de añadir un elemento en una pila estática.

```
PROCEDIMIENTO SacarPila ( VAR Pila: TipoPila; VAR ElementoSacado: TipoElemento );
PRINCIPIO
   Almacenar en ElementoSacado la cabeza de la pila;
   Decrementar el apuntador a la cabeza de la pila;
FIN;
```

Algoritmo 12.5: operación de sacar un elemento de una pila estática.

12.1.3 Programa PASCAL.

```
{Paquete pila estática}

CONST MaxPila = 100;
TYPE
TipoIndice = 1.. MaxPila;
TipoPila = RECORD
   Elementos: ARRAY [ TipoIndice ] OF TipoElemento;
   Cabeza: TipoIndice
END;

PROCEDURE InicializarPila ( VAR Pila: TipoPila );
{Inicializar pila a estado vacío}

BEGIN
   Pila.Cabeza := 0
END; {Fin InicializarPila}
```

Programa PASCAL 12.1: procedimiento de inicializar una pila estática.

```
FUNCTION PilaVacia ( Pila: TipoPila ): Boolean;
{Devuelve verdad si la pila esta vacía y falso en caso contrario}

BEGIN
```

```
   PilaVacía := Pila.Cabeza = 0
END; {Fin PilaVacía}
```

Programa PASCAL 12.2: función para averiguar si una pila estática está vacía.

```
FUNCTION PilaLlena ( Pila: TipoPila ): Boolean;
{Devuelve verdad si la pila está llena y falso en caso contrario}

BEGIN
   PilaLlena := Pila.Cabeza = MaxPila
END; {Fin PilaLlena}
```

Programa PASCAL 12.3: función para averiguar si una pila estática está llena.

```
PROCEDURE PonerPila ( VAR Pila: TipoPila; NuevoElemento: TipoElemento );
{Añade NuevoElemento a la cabeza de la pila.Supone que la pila no está llena}

BEGIN
{  Incrementar el apuntador a la cabeza de la pila}
   Pila.Cabeza := Pila.Cabeza + 1;
{  Añadir NuevoElemento en la cabeza de la pila}
   Pila.Elementos [ Pila.Cabeza ] := NuevoElemento
END; {Fin PonerPila}
```

Programa PASCAL 12.4: procedimiento para añadir un elemento en una pila estática.

```
PROCEDURE SacarPila ( VAR Pila: TipoPila; VAR ElementoSacado: TipoElemento );
{Quita el elemento cabeza de la pila y lo devuelve en ElementoSacado. Supone que la pila no está
vacía}

BEGIN
{  Almacenar en ElementoSacado la cabeza de la pila}
   ElementoSacado := Pila.Elementos [ Pila.Cabeza ];
{  Decrementar el apuntador a la cabeza de la pila}
   Pila.Cabeza := Pila.Cabeza - 1
END; {Fin SacarPila}
```

Programa PASCAL 12.5: procedimiento para sacar un elemento de una pila estática.

12.1.4 Programa C.

```
/*Paquete pila estática*/

#include <stdio.h>
#define MAXPILA 100

typedef enum {TRUE =1, FALSE = 0} boolean;
typedef int tipo_info;
typedef struct {
    tipo_info elementos [ MAXPILA ];
    int       cabeza;
} tipo_pila;

/*Inicializar pila*/
tipo_pila
inicializar_pila ( void )
/*Inicializar pila a estado vacío*/
{
    tipo_pila       pila;

    pila.cabeza = 0;
    return pila;
}/*Fin inicializar pila*/
```

Programa C 12.1: función de inicializar una pila estática.

```
/*Pila vacía*/
boolean
pila_vacía ( tipo_pila pila )
/*Devuelve verdad si la pila está vacía y falso en caso contrario*/
{
    return ( pila.cabeza == 0 );
}/*Fin pila vacía*/
```

Programa C 12.2: función para averiguar si una pila estática está vacía.

```
/*Pila llena*/
boolean
pila_llena ( tipo_pila pila )
```

```
/*Devuelve verdad si la pila está llena y falso en caso contrario*/
{
   return ( pila.cabeza  ==  MAXPILA - 1 );
}/*Fin pila llena*/
```

Programa C 12.3: función para averiguar si una pila estática está llena.

```
/*Poner pila*/
tipo_pila
poner_pila ( tipo_pila  pila, tipo_info  nuevo_elemento )
/*Añade NuevoElemento a la cabeza de la pila. Supone que la pila no está llena*/
{
/*  Incrementar el apuntador a la cabeza de la pila*/
   pila.cabeza  =  pila.cabeza + 1;
/*  Añadir NuevoElemento en la cabeza de la pila*/
   pila.elementos [ pila.cabeza ] =  nuevo_elemento;
   return pila;
}/*Fin poner pila*/
```

Programa C 12.4: función para añadir un elemento a una pila estática.

```
/*Sacar pila*/
tipo_pila
sacar_pila ( tipo_pila  pila, tipo_info  *punt_elemen_sacado )
/*Quita el elemento cabeza de la pila y lo devuelve en ElementoSacado. Supone que la pila no está
vacía*/
{
/*  Almacenar en ElementoSacado la cabeza de la pila*/
   *punt_elemen_sacado  =  pila.elementos [ pila.cabeza ];
/*  Decrementar el apuntador a la cabeza de la pila*/
   pila.cabeza  =  pila.cabeza - 1;
   return  pila;
}/*Fin sacar pila*/
```

Programa C 12.5: función para sacar un elemento de una pila estática.

12.2 Estructura de datos de tipo COLA.

12.2.1 Definición.

La estructura de datos tipo cola consiste en un conjunto ordenado de elementos del mismo tipo en el que sólo podemos introducir elementos en la cola por un extremo (Cola.Poner) y sacarlos por el extremo contrario (Cola.Sacar). La estructura cola es del tipo FIFO (First In, First Out), es decir, el primer elemento que ha entrado en la cola será el primero en salir. En este apartado vamos a diseñar una cola circular que se caracteriza por que el elemento siguiente al último es el primero. En la Figura 12.2 se muestra gráficamente la estructura de una cola circular de MaxCola elementos.

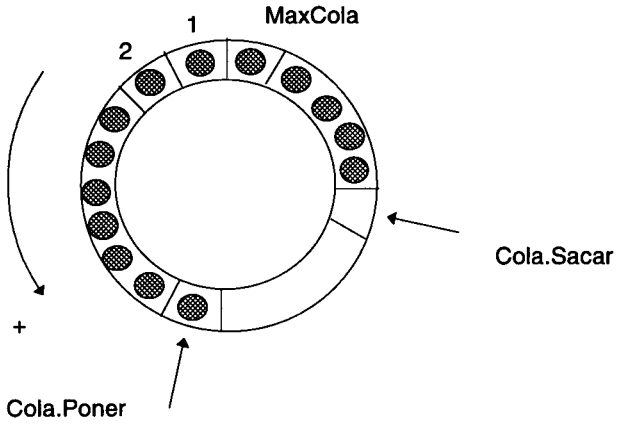


Figura 12.2: estructura de datos tipo cola circular (FIFO).

En este caso la estructura tipo cola que definiremos deberá constar de tres campos. Un campo contendrá los elementos de la cola (campo de información), otro almacenará el valor del índice para poner nuevos elementos en la cola (Cola.Poner) y, finalmente, el tercer campo contendrá el valor del índice para sacar un elemento de la cola (Cola.Sacar).

En el caso de colas circulares es necesario distinguir entre las dos situaciones siguientes: cola llena y cola vacía. En las Figuras 12.3 y 12.4 se muestra de forma gráfica el valor de los punteros Cola.Poner y Cola.Sacar en los dos casos anteriormente citados, es decir, cola vacía y cola llena.

Es necesario resaltar que únicamente se pueden utilizar MaxCola -1 elementos, ya que este hecho permite distinguir las dos situaciones de la cola que nos interesan, es decir, cola llena y cola vacía.

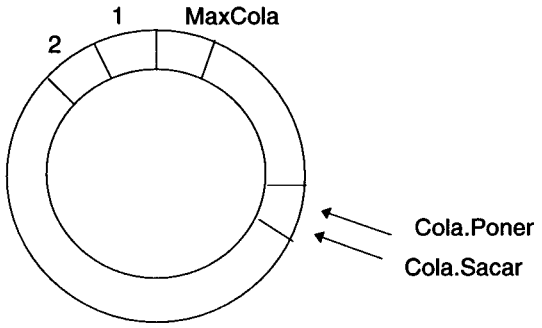


Figura 12.3: cola circular vacía.

Definiremos una serie de operaciones que es posible realizar sobre una variable de tipo cola como son:

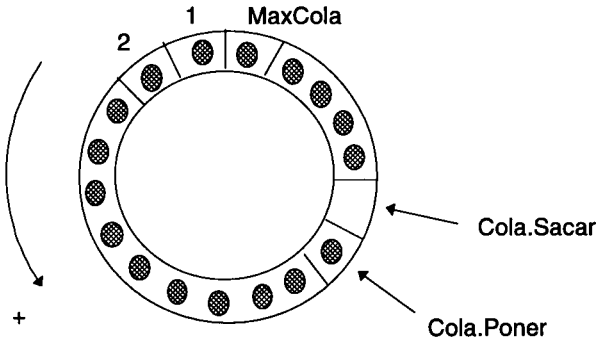


Figura 12.4: cola circular llena.

- InicializarCola: crear una cola vacía.
- ColaVacía: averiguar si una cola está vacía.
- ColaLlena: averiguar si una cola está llena.
- PonerCola: añadir un elemento en la cola.
- SacarCola: sacar un elemento de la cola.

InicializarCola (Cola)

Función:	Inicializar una cola a estado vacío
Entrada:	Cola a inicializar
Salida:	Cola inicializada
Precondiciones:	Ninguna

Postcondiciones: Cola vacía

ColaVacía (Cola)
Función: Averiguar si la cola está vacía
Entrada: Cola a comprobar
Salida: Valor booleano que indica si la cola está vacía
Precondiciones: Ninguna
Postcondiciones: Indica si la cola está vacía

ColaLlena (Cola)
Función: Averiguar si la cola está llena
Entrada: Cola a comprobar
Salida: Valor booleano que indica si la cola está llena
Precondiciones: Ninguna
Postcondiciones: Indica si la cola está llena

PonerCola (Cola, NuevoElemento)
Función: Añadir NuevoElemento a la cola
Entrada: Elemento a añadir y cola (NuevoElemento, Cola)
Salida: Cola con NuevoElemento añadido (Cola)
Precondiciones: Cola no está llena
Postcondiciones: Cola convenientemente actualizada

SacarCola (Cola, ElementoSacado)
Función: Sacar ElementoSacado de la cola
Entrada: Cola (Cola)
Salida: Cola y ElementoSacado (Cola, ElementoSacado)
Precondiciones: Cola no está vacía
Postcondiciones: Cola convenientemente actualizada

12.2.2 Algoritmo ejemplo.

{Paquete cola estática}

```
PROCEDIMIENTO InicializarCola ( VAR Cola: TipoCola );
PRINCIPIO
   Inicializar puntero de poner y de sacar a un mismo valor, por ejemplo el número de
   elementos de la cola circular;
FIN;
```

Algoritmo 12.6: inicializar una cola circular.

```
FUNCION ColaVacía ( Cola: TipoCola ): Booleano;
PRINCIPIO
   Cola está vacia si el índice de sacar y el de poner coinciden;
FIN;
```

Algoritmo 12.7: averiguar si una cola circular está vacía.

```
FUNCION ColaLlena ( Cola: TipoCola );
PRINCIPIO
   Cola está llena si la siguiente posición en la que deberíamos poner está ocupada, es
   decir, coincide con la de sacar;
FIN;
```

Algoritmo 12.8: averiguar si una cola circular está llena.

```
PROCEDIMIENTO PonerCola ( VAR Cola: TipoCola; NuevoElemento: TipoElemento );
PRINCIPIO
   Calcular la posición de la cola donde añadir;
   Añadir NuevoElemento en la cola según indica poner;
FIN;
```

Algoritmo 12.9: añadir un elemento en una cola circular.

```
PROCEDIMIENTO SacarCola ( VAR Cola: TipoCola; ElementoSacado: TipoElemento );
PRINCIPIO
   Calcular la posición de la cola donde sacar;
   Sacar el elemento de la cabeza de la cola según indica el índice de sacar y almacenarlo en
   ElementoSacado;
FIN;
```

Algoritmo 12.10: sacar un elemento de una cola circular.

12.2.3 Programa PASCAL.

```
{Paquete cola estática circular}

CONST MaxCola = 100;
TYPE
TipoIndice = 1..MaxCola;
TipoElemento = INTEGER;
```

```
TipoCola = RECORD
   Elementos:     ARRAY [ TipoIndice ] OF TipoElemento;
   Sacar:         TipoIndice;
   Poner:         TipoIndice;
END;

PROCEDURE InicializarCola ( VAR Cola: TipoCola ) ;
{Inicializar cola a estado vacío}

BEGIN
   Cola.Poner := MaxCola;
   Cola.Sacar := MaxCola
END ; {Fin InicializarCola}
```

Programa PASCAL 12.6: procedimiento para inicializar una cola circular.

```
FUNCTION ColaVacía (Cola: TipoCola): Boolean;
{Devuelve verdad si la cola está vacía y falso en caso contrario}

BEGIN
{  Cola está vacía si el índice de sacar y el de poner coinciden}
   ColaVacía:= Cola.Poner = Cola.Sacar
END;{Fin ColaVacía}
```

Programa PASCAL 12.7: función para averiguar si una cola circular está vacía.

```
FUNCTION ColaLlena (Cola : TipoCola): Boolean;
{Devuelve verdad si la cola está llena y falso en caso contrario}

BEGIN
{  Cola está llena si la siguiente posición en la que deberiamos poner esta llena, es decir, coincide con
   la de sacar}
   ColaLlena :=  Cola.Sacar  =  ( Cola.Poner MOD MaxCola ) + 1
END; {Fin ColaLlena}
```

Programa 12.8: función para averiguar si una cola circular está llena.

```
PROCEDURE PonerCola ( VAR Cola: TipoCola; NuevoElemento: TipoElemento );
{Añadir NuevoElemento en la cola. Supone que la cola no está llena}

BEGIN
{  Calcular la posición de la cola donde añadir}
```

```
    Cola.Poner := (Cola.Poner MOD MaxCola) + 1;
{   Añadir NuevoElemento en la cola según poner}
    Cola.Elementos [ Cola.Poner ] := NuevoElemento
END;{Fin PonerCola}
```

Programa PASCAL 12.9: procedimiento para añadir un elemento en una cola circular.

```
PROCEDURE SacarCola ( VAR Cola: TipoCola; ElementoSacado: TipoElemento ) ;
{Quitar ElementoSacado de la cola. Supone que la cola no está vacía}

BEGIN
{   Calcular la posición de la cola donde sacar}
    Cola.Sacar :=  ( Cola.Sacar MOD MaxCola ) + 1;
{   Sacar el elemento de la cabeza de la cola según indica el índice de sacar y almacenarlo en
    ElementoSacado}
    ElementoSacado :=  Cola.Elementos [ Cola.Sacar ]
END;{Fin SacarCola}
```

Programa PASCAL 12.10: procedimiento para sacar un elemento de una cola circular.

12.2.4 Programa C.

```
/*Paquete cola estática circular*/

#include <stdio.h>
#define MAXCOLA 100

typedef enum {TRUE = 1, FALSE = 0} boolean;
typedef int   tipo_elemen;
typedef struct {
    tipo_elemen     elementos [ MAXCOLA ] ;
    int             poner, sacar;
}tipo_cola;

/*Inicializar cola*/
tipo_cola
inicializar_cola ( void )
/*Inicializar cola a estado vacío*/
```

```
{
   tipo_cola        cola;

   cola.poner = MAXCOLA - 1;
   cola.sacar = MAXCOLA - 1;
   return cola;
}/*Fin inicializar cola*/
```

Programa C 12.6: función para inicializar una cola circular.

```
/*Cola vacía */
boolean
cola_vacía ( tipo_cola cola )
/*Devuelve verdad si la cola está vacía y falso en caso contrario*/
{
/*  Cola está vacía si sacar y poner coinciden*/
   return ( cola.poner == cola.sacar );
}/*Fin cola vacía*/
```

Programa C 12.7: función para averiguar si una cola circular está vacía.

```
/*Cola llena*/
boolean
cola_llena (tipo_cola cola)
/*Devuelve verdad si la cola está llena y falso en caso contrario*/
{
/*  Cola está llena si la siguiente posición en la que deberíamos poner está llena, es decir, coincide con
    la de sacar*/
   return ( cola.sacar == ( cola.poner + 1 ) % MAXCOLA  );
}/*Fin cola llena*/
```

Programa C 12.8: función para averiguar si una cola circular está llena.

```
/*Poner cola*/
tipo_cola
poner_cola ( tipo_cola  cola, tipo_elemen  nuevo_elemento )
/*Añadir NuevoElemento en la cola. Supone que la cola no está llena*/
{
/*  Calcular la posición de la cola donde añadir*/
   cola.poner = ( cola.poner + 1 ) % MAXCOLA;
/*  Añadir NuevoElemento en la cola según indica el índice de poner*/
   cola.elementos [ cola.poner ] = nuevo_elemento;
```

```
    return cola;
}/*Fin poner cola*/
```

Programa C 12.9: función para añadir un elemento en una cola circular.

```
/*Sacar cola*/
tipo_cola
sacar_cola ( tipo_cola cola, tipo_elemen *punt_elem_sacado )
/*Quitar elem_sacado de la cola. Supone que la cola no está vacía*/
{
/*  Calcular la posición de la cola domde sacar*/
    cola.sacar = ( cola.sacar + 1 ) % MAXCOLA;
/*  Almacenar en elem_sacado el elemento de la cola según indica sacar*/
    *punt_elem_sacado = cola.elementos [ cola.sacar ];
    return cola;
}/*Fin sacar cola*/
```

Programa C 12.10: función para sacar un elemento de una cola circular.

13. Tipos de datos dinámicos.

Los tipos de datos estáticos se caracterizan porque su dimensión debe ser especificada a priori por el programador, ya que el compilador debe reservar el espacio de memoria adecuado para su almacenamiento. Existen una gran variedad de aplicaciones en las que no es posible conocer a priori la dimensión de la estructura de datos, en estos casos es necesario utilizar tipos de datos dinámicos. Las estructuras dinámicas se caracterizan porque su dimensión varía dinámicamente en tiempo de ejecución.

Las variables de tipo puntero se utilizan para crear variables dinámicas (es decir, variables que se crean en tiempo de ejecución). El tipo de dato puntero está predefinido en la mayoría de lenguajes de alto nivel. Por ejemplo, en PASCAL se representa mediante el caracter '^' y en lenguaje C se utiliza el carácter '*'.

Ejemplos en lenguaje PASCAL:

```
TYPE TipoPuntero = ^Integer; { TipoPuntero es un puntero a entero }
VAR PunteroEntero : TipoPuntero; { Variable de tipo puntero a entero }
```

Ejemplos en lenguaje C:

```
int             *tipo_puntero; /* tipo_puntero es un puntero a entero */
tipo_puntero    puntero_entero /* Variable de tipo puntero a entero */
```

Para referenciar la variable dinámica apuntada por un puntero se utiliza la siguiente notación:

Lenguaje PASCAL:

```
entero := PunteroEntero^; { entero apuntado por el puntero PunteroEntero }
```

Lenguaje C:

```
entero = &puntero_entero; /* entero apuntado por el puntero puntero_entero */
```

En lenguaje C cuando la variable referenciada por un puntero es de tipo struct, se suele utilizar la siguiente notación para acceder a un campo de dicha estructura:

```
/* Definimos el tipo tipo_estructura compuesto por dos campos: entero y carácter */
typedef  struct  {
              int entero;
              char caracter;
          } tipo_estructura;

/* Definimos la variable puntero_estructura que apunta a una estructura de tipo tipo_estructura */
tipo_estructura     *puntero_estructura

/* Referencia al campo entero de la variable puntero_estructura */
puntero_estructura->entero

/* Referencia al campo caracter de la variable puntero-estructura */
puntero_estructura->caracter
```

Existe una constante predefinida de tipo puntero que se representa mediante la palabra clave **NIL** en lenguaje PASCAL y **NULL** en lenguaje C que se puede utilizar para asignar un valor nulo a una variable tipo puntero. Un puntero de valor nulo (NIL o NULL) no apunta a ningún dato, de forma que se puede utilizar para indicar el final de la estructura dinámica.

En lenguaje PASCAL existe un procedimiento predefinido, llamado **NEW** (variable-puntero), que crea la variable dinámica variable-puntero^ y hace que la variable de tipo puntero variable-puntero apunte a ella. También existe un procedimiento predefinido, llamado **DISPOSE** (variable-puntero), que libera la memoria ocupada por la variable dinámica variable-puntero^ apuntada por la variable de tipo puntero variable-puntero.

En el caso del lenguaje C utilizaremos la función predefinida "**malloc**" para asignar memoria dinámicamente y la función predefinida "**free**" para liberar la memoria.

13.1 Estructura de datos tipo PILA DINÁMICA.

13.1.1 Definición.

Ya hemos descrito la estructura de datos tipo pila en el apartado anterior correspondiente a estructuras de datos avanzadas. El que sea una pila dinámica hace referencia al hecho de utilizar punteros para enlazar los nodos de la pila, de forma que la asignación de memoria al añadir o suprimir nodos puede ser dinámica.

En la Figura 13.1 se muestra la estructura de un nodo de una pila dinámica, que consta de dos campos, el campo que contiene la información propiamente dicha y el campo que contiene el puntero que apunta al siguiente nodo de la pila. En realidad la estructura del nodo es válida para todas las estructuras dinámicas que veremos en este capítulo, es decir, pilas, colas y listas dinámicas.

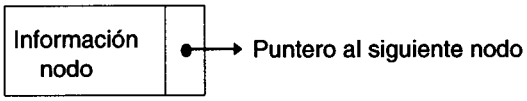


Figura 13.1: estructura de un nodo de una pila dinámica.

Las operaciones definidas sobre la estructura de datos de tipo pila dinámica son las siguientes:

- InicializarPila: crear una pila dinámica vacía.
- PilaVacía: averiguar si una pila dinámica está vacía.
- InsertarPila: insertar un nodo en la cabeza de la pila dinámica.
- SuprimirPila: suprimir un nodo de la cabeza de la pila dinámica.

InicializarPila (Pila)
Función: Inicializar la pila a estado vacío
Entrada: Pila a inicializar
Salida: Pila inicializada
Precondiciones: Ninguna
Postcondiciones: Pila vacía

PilaVacía (Pila)
Función: Averiguar si la pila está vacía
Entrada: Pila a comprobar
Salida: Valor booleano que indica si la pila está vacía
Precondiciones: Pila inicializada
Postcondiciones: Indica si la pila está vacía

InsertarPila (Pila, NuevoElemento)
Función: Añadir NuevoElemento a la pila
Entrada: Pila y elemento a añadir (Pila, NuevoElemento)
Salida: Pila con NuevoElemento añadido (Pila)
Precondiciones: Pila inicializada

Postcondiciones: Pila convenientemente actualizada

SuprimirPila (Pila, ElementoSacado)
Función: Sacar ElementoSacado de la pila
Entrada: Pila sobre la que suprimir un elemento (Pila)
Salida: Pila y elemento sacado (Pila, ElementoSacado)
Precondiciones: Pila no está vacía
Postcondiciones: Pila convenientemente actualizada

NOTA: El resto de procedimientos no son necesarios en el caso de pilas dinámicas.

13.1.2 Algoritmo ejemplo.

```
InicializarPila ( Pila )
PRINCIPIO
   Inicializar pila a valor nulo;
FIN
```

Algoritmo 13.1: inicializar una pila dinámica.

```
PilaVacía ( Pila )
PRINCIPIO
   Pila está vacía si vale nulo;
FIN
```

Algoritmo 13.2: averiguar si una pila dinámica está vacía.

En la Figura 13.2 se muestra gráficamente el proceso de insertar un nodo en la cabeza de una pila dinámica.

```
InsertarPila ( Pila, NuevoElemento )
PRINCIPIO
   Obtener un nuevo nodo y asignarle memoria;
   Colocar en su campo de información el NuevoElemento;
   Insertar el nuevo nodo en la cabeza de la pila;
FIN
```

Algoritmo 13.3: insertar un nodo en la cabeza de una pila dinámica.

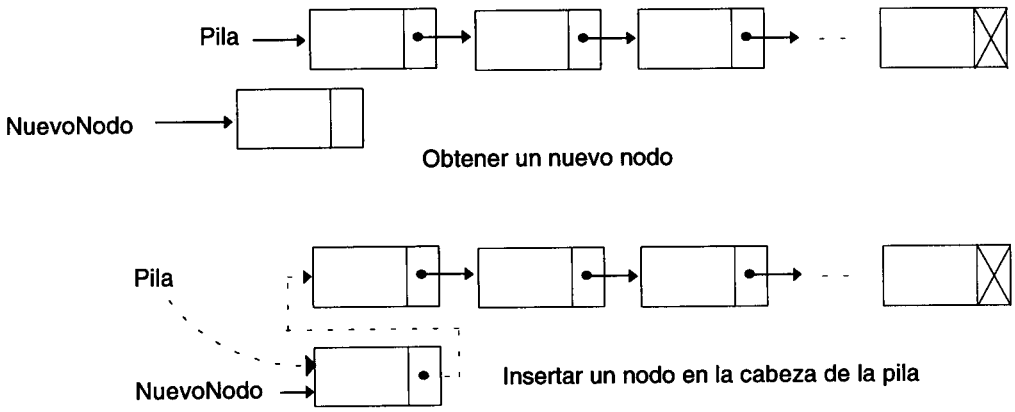


Figura 13.2: insertar un nodo en la cabeza de una pila dinámica.

En la Fig 13.3 se muestra gráficamente el proceso de eliminación de un nodo de la cabeza de una pila dinámica.

```
SuprimirPila ( Pila, ElementoSacado )
PRINCIPIO
   Almacenar el campo de información del nodo de la cabeza de la pila en
   ElementoSacado;
   Suprimir el nodo de la cabeza de la pila;
   Liberar la memoria ocupada por el nodo suprimido;
FIN
```

Algoritmo 13.4: suprimir el nodo de la cabeza de una pila dinámica.

13.1.3 Programa PASCAL.

```
{ Paquete de pila dinámica }
TYPE
   TipoElemento = INTEGER;
   TipoPuntero = ^TipoNodo;
   TipoNodo = RECORD
      Info:        TipoElemento;
      Siguiente:   TipoPuntero
   END;
   TipoPila = TipoPuntero;
```

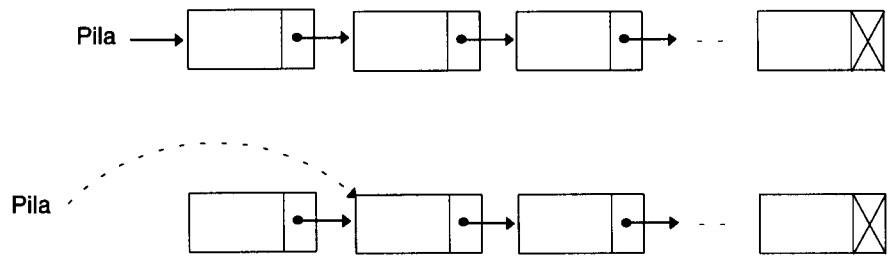
```
VAR
   Pila: TipoPila;

PROCEDURE InicializarPila ( VAR Pila: TipoPila ) ;
{Inicializar pila a estado vacío}
BEGIN
   Pila := NIL
END;{ Fin InicializarPila }
```

Programa PASCAL 13.1: procedimiento de inicializar una pila dinámica.

```
FUNCTION PilaVacía ( Pila: TipoPila ): Boolean ;
{Devuelve verdad si la pila está vacía y falso en caso contrario }
BEGIN
   PilaVacía := Pila = NIL
END;{ Fin PilaVacía }
```

Programa PASCAL 13.2: función para averiguar si una pila dinámica está vacía.

```
PROCEDURE InsertarPila ( VAR Pila: TipoPila; NuevoElemento: TipoElemento );
{Añade NuevoElemento a la cabeza de la pila}
VAR NuevoNodo: TipoPuntero;
BEGIN
{  Obtener un nodo nuevo y colocar en su campo Info NuevoElemento}
   New (NuevoNodo);{Reservar espacio para NuevoElemento en memoria}
   NuevoNodo^.Info := NuevoElemento;
```



Suprimir un nodo de la cabeza de la pila

Figura 13.3: suprimir el nodo de la cabeza de una pila dinámica

```
{  Insertar NuevoNodo en la cabeza de la pila}
   NuevoNodo^.Siguiente := Pila;
   Pila := NuevoNodo
END;{Fin InsertarPila}
```

Programa PASCAL 13.3: procedimiento para insertar un elemento en una pila dinámica.

```
PROCEDURE SuprimirPila (VAR Pila: TipoPila; VAR ElementoSacado: TipoElemento);
{Quita el elemento cabeza de la pila y lo devuelve en ElementoSacado. Supone que la pila no está
vacía}
VAR PunteroAux: TipoPuntero;
BEGIN
{  Almacenar el campo Info del nodo de la cabeza de la pila en ElementoSacado}
   ElementoSacado := Pila^.Info;
{  Suprimir el nodo de la cabeza de la pila}
   PunteroAux := Pila;
   Pila := Pila^.Siguiente;
   Dispose (PunteroAux) {Liberar memoria ocupada por el nodo suprimido}
END; {Fin SuprimirPila}
```

Programa PASCAL 13.4: procedimiento para suprimir un elemento de una pila dinámica.

13.1.4 Programa C.

```
/* Paquete pila dinámica */

#include <stdio.h>
#include <malloc.h>

typedef enum {FALSE = 0, TRUE =1} boolean;
typedef int tipo_info;
typedef struct nodo_pila {
   tipo_info        info;
   struct nodo_pila *siguiente;
}tipo_nodo;

/*Mensaje originado por un malloc erróneo*/
void
```

```
no_memoria ( )
{
   fprintf  ( stderr, "Lo siento no queda memoria\n" ) ;
}/* Fin no_memoria*/
```

Programa C 13.1: función que indica que no se ha resuelto la asignación dinámica de memoria.

```
/*Crear memoria dinámica con un malloc*/
void *
crear_espacio ( )
{
   tipo_nodo       *puntero;

   puntero = ( tipo_nodo * ) malloc ( sizeof *puntero ) ;
/* ¿Puede hacer la asignación dinámica? */
   if ( puntero == NULL ) no_memoria ( );
   return puntero;
}/* Fin crear_espacio*/
```

Programa C 13.2: función para hacer la asignación dinámica de memoria.

```
/*Inicializar pila*/
tipo_nodo *
inicializar_pila (void)
/*Inicializar la pila dinámica a estado vacío*/
{
   return NULL;
}/* Fin inicializar_pila*/
```

Programa C 13.3: función para inicializar una pila dinámica.

```
/*Pila vacía*/
boolean
pila_vacía (tipo_nodo *pila)
/*Devuelve verdad si la pila está vacía y falso en caso contrario*/
{
   return (pila == NULL);
}/* Fin pila_vacía*/
```

Programa C 13.4: función para averiguar si una pila dinámica está vacía.

```
/*Insertar pila*/
tipo_nodo *
insertar_pila (tipo_nodo *pila, tipo_info nuevo_elemento)

/*Añade nuevo_elemento a la cabeza de la pila*/
{
   tipo_nodo        *nuevo_nodo;

/*  Obtener un nuevo nodo*/
   nuevo_nodo = crear_espacio ( );
/*  Colocar en su campo info nuevo_elemento*/
   nuevo_nodo->info = nuevo_elemento;
/*  Insertar nuevo_nodo en la cabeza de la pila*/
   nuevo_nodo->siguiente = pila;
   pila = nuevo_nodo;
   return pila;
}/* Fin insertar_pila*/
```

Programa C 13.5: función para insertar un elemento en una pila dinámica.

```
/*Suprimir pila*/
tipo_nodo *
suprimir_pila (tipo_nodo *pila, tipo_info elemento_sacado)

/*Quita el elemento cabeza de la pila y lo devuelve en elemento_sacado. Supone que la pila no está
vacía*/
{
   tipo_nodo        *puntero_aux;

   /*Almacenar el campo info del nodo de la cabeza de la pila en elemento_sacado*/
   elemento_sacado = pila->info;
   /*Suprimir el nodo de la cabeza de la pila*/
   puntero_aux = pila;
   pila = pila->siguiente;
   /*Liberar la memoria ocupada por el nodo suprimido*/
   free (puntero_aux);
   return pila;
}/* Fin suprimir_pila*/
```

Programa C 13.6: función para añadir un elemento a una pila dinámica.

NOTA: En lenguaje C cuando en un campo de una estructura es necesario hacer referencia a la estructura que se está definiendo, en necesario dotarla de un identificador.

13.2 Estructura de datos de tipo COLA DINÁMICA.

13.2.1 Definición.

Ya hemos estudiado la estructura de datos tipo cola en el apartado anterior. El hecho de que sea una cola dinámica hace referencia a que se utilizan punteros para enlazar los elementos de la cola, de forma que pueda crecer y decrecer en tiempo de ejecución de forma dinámica. En la Figura 13.4 se muestra una representación gráfica de la estructura de datos cola dinámica, en la que se insertan nodos según indica el puntero Cola.Poner y se eliminan nodos según indica el puntero Cola.Sacar.

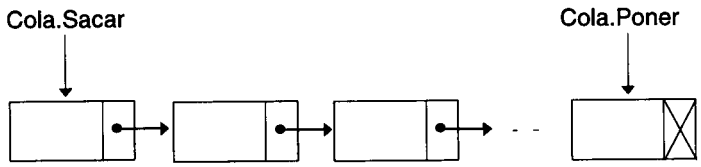


Figura 13.4: estructura de datos cola dinámica.

Las operaciones definidas sobre la estructura de datos de tipo cola dinámica son las siguientes:

- InicializarCola: crear una cola dinámica vacía.
- ColaVacía: averiguar si una cola dinámica está vacía.
- InsertarCola: insertar un nodo en una cola dinámica.
- SuprimirCola: suprimir un nodo en una cola dinámica.

InicializarCola (Cola)

Función:	Inicializar cola a estado vacío
Entrada:	Cola a inicializar
Salida:	Cola inicializada
Precondiciones:	Ninguna
Postcondiciones:	Cola vacía

ColaVacía (Cola)

Función:	Averiguar si la cola está vacía

Entrada: Cola a comprobar
Salida: Valor booleano que indica si cola está vacía
Precondiciones: Cola inicializada
Postcondiciones: Indica si la cola está vacía

InsertarCola (Cola, NuevoElemento)
Función: Añadir NuevoElemento a la cola
Entrada: Cola y elemento a insertar (Cola, NuevoElemento)
Salida: Cola con NuevoElemento insertado (Cola)
Precondiciones: Cola inicializada
Postcondiciones: Cola convenientemente actualizada

SuprimirCola (Cola, ElementoSacado)
Función: Sacar ElementoSacado de la cola
Entrada: Cola sobre la que suprimir un nodo (Cola)
Salida: Cola y elemento suprimido (Cola, ElementoSacado)
Precondiciones: Cola no está vacía
Postcondiciones: Cola convenientemente actualizada

NOTA: El resto de procedimientos no son necesarios en el caso de colas dinámicas

13.2.2 Algoritmo ejemplo.

```
InicializarCola ( Cola )
PRINCIPIO
   Inicializar cola a valor nulo;
FIN
```

Algoritmo 13.5: inicializar una cola dinámica.

```
ColaVacía ( Cola )
PRINCIPIO
   Cola está vacía si los punteros de sacar y poner de la cola tienen valor nulo;
FIN
```

Algoritmo 13.6: averiguar si una cola dinámica está vacía.

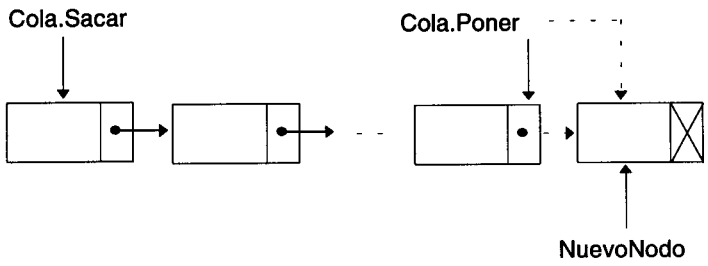


Figura 13.5: insertar un nodo en una cola dinámica.

En la Fig. 13.5 se muestra el proceso de inserción de un nodo en una cola dinámica.

```
InsertarCola ( Cola, NuevoElemento )
PRINCIPIO
   Crear un nuevo nodo y asignarle memoria;
   Poner en su campo Info NuevoElemento;
   Añadir el nuevo nodo a la cola según el puntero de poner;
   SI la cola estaba vacía
      ENTONCES el nuevo nodo pasa a ser el elemento a sacar
      SI NO el nuevo nodo se encadena después del último;
   Hacer que el puntero de poner apunte al nuevo nodo;
FIN
```

Algoritmo 13.7: insertar un elemento en una cola dinámica.

En la Fig 13.6 se ilustra el proceso de suprimir un nodo de una cola dinámica.

```
SuprimirCola ( Cola, ElementoSacado )
PRINCIPIO
```

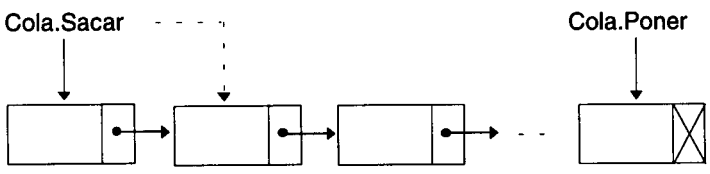


Figura 13.6: suprimir un nodo de una cola dinámica.

```
   Guardar el puntero al nodo a suprimir de la cola;
   Almacenar en ElementoSacado el campo info del nodo apuntado por sacar;
   Suprimir el nodo de la cola apuntado por sacar;
   SI la cola está vacía
      ENTONCES hemos sacado el último nodo;
   Liberar la memoria ocupada por el nodo suprimido;
FIN
```

Algoritmo 13.8: suprimir un elemento de una cola dinámica.

13.2.3 Programa PASCAL.

```
{Paquete cola dinámica}
TYPE
TipoElemento = INTEGER;
TipoPuntero = ^NodoCola;

NodoCola = RECORD
   Info:          TipoElemento;
   Siguiente:     TipoPuntero
END;

TipoCola = RECORD
   Sacar: TipoPuntero;
   Poner: TipoPuntero;
END;

VAR
Cola: TipoCola;

PROCEDURE InicializarCola ( VAR Cola: TipoCola );
{Inicializar cola a estado vacío}
BEGIN
   Cola.Poner := NIL;
   Cola.Sacar := NIL
END;{Fin InicializarCola}
```

Programa PASCAL 13.5: procedimiento para inicializar una cola dinámica.

```
FUNCTION ColaVacía ( Cola: TipoCola ): Boolean;
```

```
{Devuelve verdad si la cola esta vacía y falso en caso contrario}
BEGIN
{  Cola está vacía si índice de sacar y poner no apuntan a ningún nodo}
   ColaVacía := Cola.Sacar = NIL {No hay nada para sacar de la cola}
END;{Fin ColaVacía}
```

Programa PASCAL 13.6: función para averiguar si una cola dinámica está vacía.

```
PROCEDURE InsertarCola ( VAR Cola: TipoCola; NuevoElemento: TipoElemento );
{Añadir NuevoElemento en la cola según indica el puntero Poner}
VAR NuevoNodo: TipoPuntero;
BEGIN
{  Crear nodo nuevo para añadir cola y poner en campo Info NuevoElemento}
   New ( NuevoNodo );{ Reserva espacio de memoria para NuevoElemento}
   NuevoNodo^.Info := NuevoElemento;
   NuevoNodo^.Siguiente := NIL;{ No habrá ningún nodo después del nuevo}
{  Añadir NuevoNodo en la cola según indica el puntero de Poner}
   IF Cola.Poner = NIL   { Cola estaba vacía}
      THEN {NuevoNodo será el primer elemento a sacar}
            Cola.Sacar := NuevoNodo
      ELSE {Cola no estaba vacía, NuevoNodo se encadena después último}
            Cola.Poner^.Siguiente := NuevoNodo;
{  Hacer que el puntero Poner apunte a NuevoNodo}
   Cola.Poner := NuevoNodo { NuevoNodo será último nodo añadido a la cola}
END;{Fin InsertarCola}
```

Programa PASCAL 13.7: procedimiento para añadir un elemento en una cola dinámica.

```
PROCEDURE SuprimirCola ( VAR Cola: TipoCola; ElementoSacado: TipoElemento ) ;
{Saca el elemento de la cola apuntado por Sacar y lo almacena en ElementoSacado. Supone que la
cola no está vacía}
VAR PunteroAux: TipoPuntero;
BEGIN
{  Guardar el puntero al nodo a suprimir de la cola}
   PunteroAux := Cola.Sacar
{  Sacar elemento cola según Cola.Sacar y almacenarlo en ElementoSacado}
   ElementoSacado := Cola.Sacar^.Info;
{  Suprimir el nodo de la cola apuntado por Sacar}
   Cola.Sacar := Cola.Sacar^.Siguiente;
{  El elemento a sacar será el siguiente al sacado}
```

```
    IF Cola.Sacar = NIL {Cola vacía}
        THEN Cola.Poner := NIL;{Hemos sacado el último elemento cola}
    Dispose (PunteroAux) {Libera la memoria ocupada por elemento sacado}
END;{Fin SuprimirCola}
```

Programa PASCAL 13.8: procedimiento para suprimir un elemento de una cola dinámica.

13.2.4 Programa C.

```
/*Paquete cola dinámica*/

#include <stdio.h>
#include <malloc.h>

typedef enum {FALSE = 0, TRUE =1} boolean;
typedef int tipo_info;
typedef struct nodo_cola {
    tipo_info       info;
    struct nodo_cola *siguiente;
}tipo_nodo;

typedef struct {
    tipo_nodo *sacar;
    tipo_nodo *poner;
}tipo_cola;


/*Mensaje originado por un malloc erróneo*/
void
no_memoria ( )
{
    fprintf (stderr, "Lo siento no queda memoria\n");
}/* Fin no_memoria*/
```

Programa C 13.7: función que indica que no se ha resuelto la asignación dinámica de memoria.

```
/*Crear memoria dinámica con un malloc*/
void *
```

```
crear_espacio ( )
{
   tipo_nodo       *puntero;

   puntero = ( tipo_nodo * ) malloc ( sizeof *puntero );
/*  ¿Puede hacer la asignación dinámica de memoria ?*/
   if ( puntero == NULL ) no_memoria ( );
   return puntero;
}/* Fin crear_espacio */
```

Programa C 13.8: función para hacer la asignación dinámica de memoria.

```
/*Inicializar cola*/
tipo_cola
inicializar_cola (void)
{
   tipo_cola cola;

/*  Punteros de poner y sacar de la cola a valor nulo*/
   cola.poner = NULL;
   cola.sacar = NULL;
   return cola;
}*/ Fin inicializar_cola */
```

Programa C 13.9: función que inicializa una cola dinámica.

```
/*Cola vacía*/
boolean
cola_vacía ( tipo_cola cola )
{
/*  Cola está vacía si sacar y poner tienen el valor nulo*/
   return ( cola.sacar == NULL );
}/* Fin cola_vacía*/
```

Programa C 13.9: función para averiguar si una cola dinámica está vacía.

```
/*Insertar cola*/
tipo_cola
insertar_cola ( tipo_cola cola, tipo_info nuevo_elemento )
/*Añadir nuevo-elemento en la cola según indica poner*/
{
   tipo_nodo       *nuevo_nodo;
```

```
/* Crear un nuevo_nodo para añadir en la cola*/
   nuevo_nodo = crear_espacio( );
/* Poner en su campo Info nuevo_elemento*/
   nuevo_nodo->info = nuevo_elemento;
   nuevo_nodo->siguiente = NULL;
/* Añadir nuevo_nodo en la cola según el puntero de poner*/
   if ( cola.poner == NULL ) /*Si cola estaba vacía*/
      cola.sacar = nuevo_nodo; /*nuevo_nodo será primer nodo a sacar*/
   else      /*si no, nuevo_nodo se encadena después del último*/
      cola.poner->siguiente = nuevo_nodo;
/* Hacer que el puntero de poner apunte a nuevo-nodo*/
   cola.poner = nuevo_nodo;
   return cola;
}/*Fin insertar_cola*/
```

Programa C 13.10: función para insertar un elemento en una cola dinámica.

```
/*Suprimir cola*/
tipo_cola
suprimir_cola ( tipo_cola cola, tipo_info elemento_sacado )
/*Saca el nodo de la cola apuntado por sacar y lo almacena en elemento_sacado. Supone que la cola
no está vacía*/
{
   tipo_nodo      *puntero_aux;

/* Guardar el puntero al nodo a suprimir de la cola*/
   puntero_aux = cola.sacar;
/* Almacenar en elemento_sacado el elemento cola apuntado por sacar*/
   elemento_sacado = cola.sacar->info;
/* Suprimir el nodo de la cola apuntado por sacar*/
   cola.sacar = cola.sacar->siguiente;
/* ¿Cola vacía?*/
   if ( cola.sacar == NULL )
      cola.poner = NULL; /*Hemos sacado el último elemento de la cola*/
/* Liberar la memoria ocupada por el nodo suprimido*/
   free ( puntero_aux );
   return cola;
}/*Fin suprimir_cola*/
```

Programa C 13.11: función para suprimir un elemento de una cola dinámica.

13.3 Estructura de datos de tipo LISTA DINÁMICA.

La estructura de datos de tipo lista consiste en un conjunto de datos del mismo tipo de forma que cada elemento de la lista (nodo) consta de dos campos. Un campo contiene la información que almacena la lista y el otro indica cuál es el siguiente nodo de la lista. A la información normalmente contenida en cada nodo de una lista se le añade otra, para enlazar unos nodos con otros. Esta información adicional apunta a otro u otros nodos de la estructura. En general hay dos tipos de listas: las estáticas, en las que esta información adicional son índices a los elementos de un vector (o matriz), cuyos elementos son los nodos de la lista; y las dinámicas, en las que los nodos son referenciados directamente mediante su dirección de memoria, con una variable de tipo puntero, de las soportadas por los lenguajes de programación. El manejo de las estructuras de tipo lista dinámica es mucho más sencillo que el de las estáticas, pues el sistema operativo del ordenador se encarga de gestionar la memoria ocupada por la estructura en cada momento, en vez de tener que hacerlo el usuario.

El hecho de que pueda haber más de un apuntador en cada nodo, permite que los nodos de la estructura se encadenen según criterios distintos. Se dice que una lista es ordenada si sus nodos están encadenados según un criterio de ordenación (ascendente, descendente, numérico, etc.) de su campo de información.

13.3.1 Definición.

La estructura de datos tipo lista dinámica se diferencia de la lista estática, en que se utilizan punteros para enlazar los nodos que forman la lista. En este apartado vamos a estudiar las listas dinámicas ordenadas, en las que los nodos de la lista están ordenados según los campos de información siguiendo un cierto criterio. En la Figura 13.7 se muestra gráficamente la estructura de datos lista dinámica.

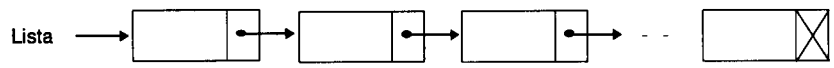


Figura 13.7: estructura de datos tipo lista dinámica.

Las operaciones definidas sobre la estructura de datos de tipo lista ordenada y enlazada mediante punteros (lista dinámica) son las siguientes:

- InicializarLista: crear una lista dinámica vacía.
- ListaVacía. averiguar si una lista dinámica está vacía.
- InsertarLista: insertar un nodo en una lista dinámica.

- SuprimirLista: suprimir un nodo de una lista dinámica.

InicializarLista (Lista)
Función: Inicializar lista a estado vacío
Entrada: Lista a inicializar
Salida: Lista inicializada
Precondiciones: Ninguna
Postcondiciones: Lista vacía

ListaVacía (Lista)
Función: Averiguar si la lista está vacía
Entrada: Lista a comprobar
Salida: Valor booleano que indica si la lista está vacía
Precondiciones: Lista inicializada
Postcondiciones: Indica si la lista está vacía

InsertarLista (Lista, NuevoElemento)
Función: Insertar NuevoElemento a la lista
Entrada: Lista, elemento a insertar (Lista, NuevoElemento)
Salida: Lista con NuevoElemento insertado (Lista)
Precondiciones: Lista inicializada
Postcondiciones: Lista convenientemente actualizada

SuprimirLista (Lista, QuitarElemento)
Función: Suprimir QuitarElemento de la lista
Entrada: Lista sobre la que suprimir un nodo (Lista)
Salida: Lista y elemento suprimido (Lista, QuitarElemento)
Precondiciones: Lista no está vacía
Postcondiciones: Lista convenientemente actualizada

Es posible definir otras estructuras de tipo lista dinámica, con características diferentes de la lista que vamos a desarrollar en este apartado. A continuación dejamos como ejercicio para el lector definir y diseñar las operaciones necesarias para manipular los siguientes tipos de listas dinámicas:

- Lista dinámica circular, es decir, aquélla en la que el último elemento de la lista apunta al primero (Figura 13.8).

- Lista dinámica doblemente enlazada, es decir, aquélla con dos campos de tipo TipoPuntero: el siguiente que apunta al nodo siguiente al actual y el anterior que apunta al nodo anterior al actual (Figura 13.9).

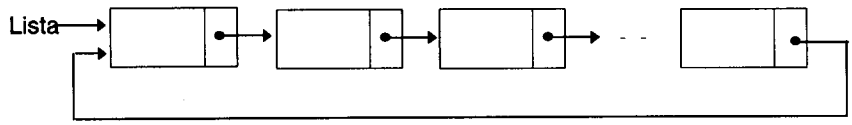


Figura 13.8: lista dinámica circular.

- Lista dinámica circular doblemente enlazada (Figura 13.10).

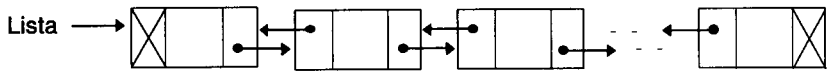
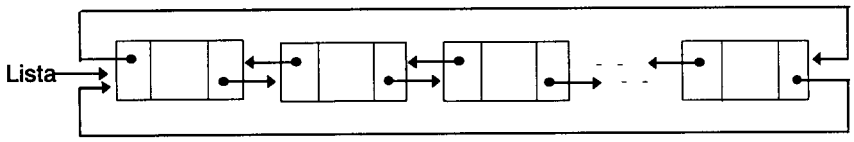


Figura 13.9: lista dinámica doblemente enlazada.

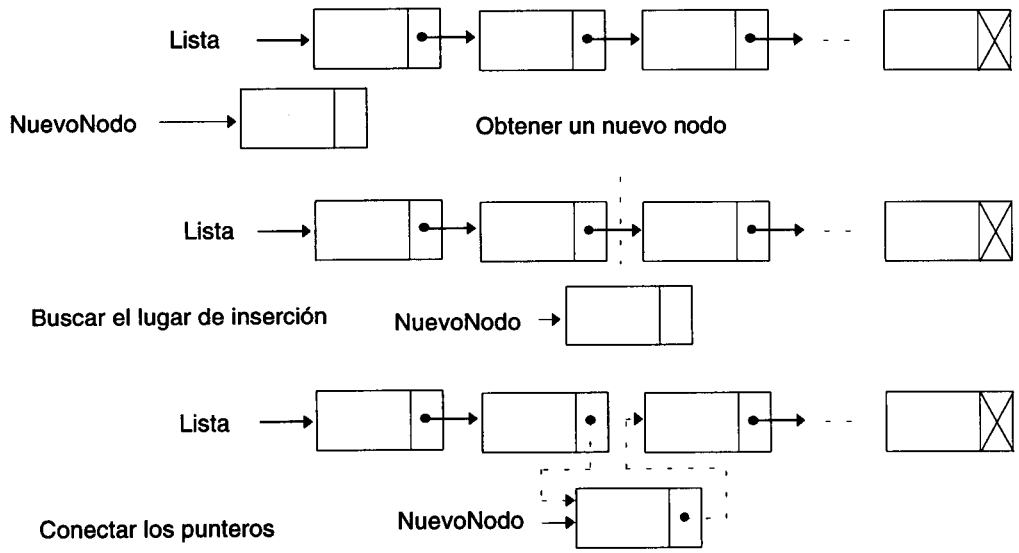
13.3.2 Algoritmo ejemplo.

```
InicializarLista ( Lista )
PRINCIPIO
   Inicializar lista a valor nulo;
FIN
```

Algoritmo 13.9: inicializar una lista dinámica.



Figuar 13.10: lista dinámica circular doblemente enlazada.



Figuar 13.11: insertar un nodo en una lista dinámica.

```
ListaVacía (Lista)
PRINCIPIO
   Lista está vacía si tiene el valor nulo;
FIN
```

Algoritmo 13.10: averiguar si una lista dinámica está vacía.

En la Figura 13.11 se muestra gráficamente el proceso de inserción de un nodo en una lista dinámica ordenada.

```
InsertarLista ( Lista, NuevoElemento )
PRINCIPIO
   Crear un nuevo nodo;
   Llenar el campo de información del nuevo nodo con NuevoElemento;
   SI lista está vacía
      ENTONCES Insertar nuevo nodo en una lista vacía
      SI NO
      PRINCIPIO
            SI NuevoElemento < campo información del primer nodo lista
                  ENTONCES Insertar nuevo nodo antes del primer nodo
                  SI NO
                     PRINCIPIO
```

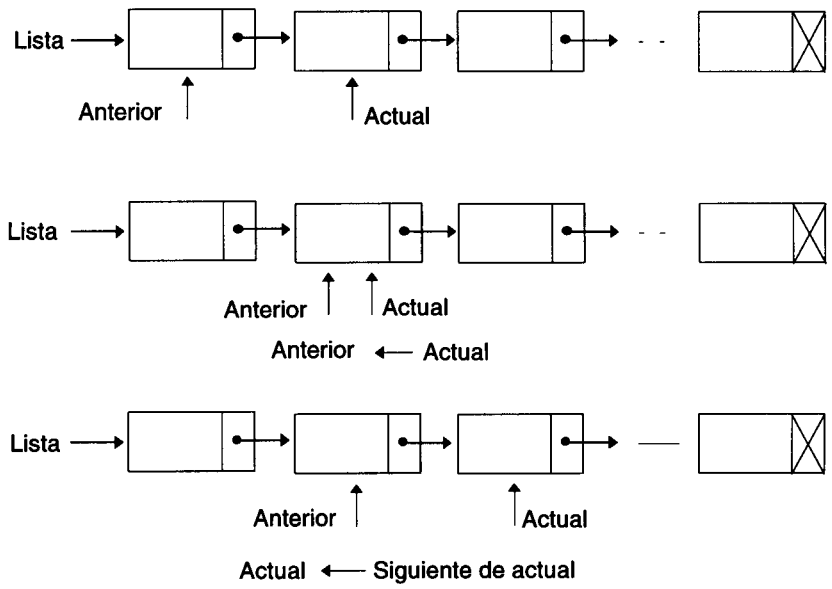


Figura 13.12: recorrido de la lista dinámica para buscar el nodo a suprimir.

```
                Recorrer la lista para buscar lugar inserción;
                Conectar los punteros y materializar inserción;
            FIN ( de insertar en el medio o final de una lista)
        FIN (de insertar en una lista ya existente)
FIN.
```

Algoritmo 13.11: insertar un nodo en una lista dinámica.

En la Figura 13.12 se muestra gráficamente como se realiza el proceso de recorrer una lista dinámica ordenada enlazada con el objeto de buscar el nodo a suprimir. Se utilizan dos punteros, el anterior y el actual con el objeto de poder recomponer los punteros adecuadamente, cuando posteriormente se deba suprimir el nodo. En el caso de disponer de una lista doblemente enlazada no sería necesario utilizar dos punteros para buscar el nodo a suprimir, ya que los nodos de la lista mediante los punteros al nodo anterior y al siguiente permitirían recomponer los punteros adecuadamente para materializar la eliminación del nodo.

La Figura 13.13 se muestra mediante tres gráficos cada uno de los tres casos que se pueden producir al eliminar un nodo de una lista dinámica, es decir, que el nodo a eliminar

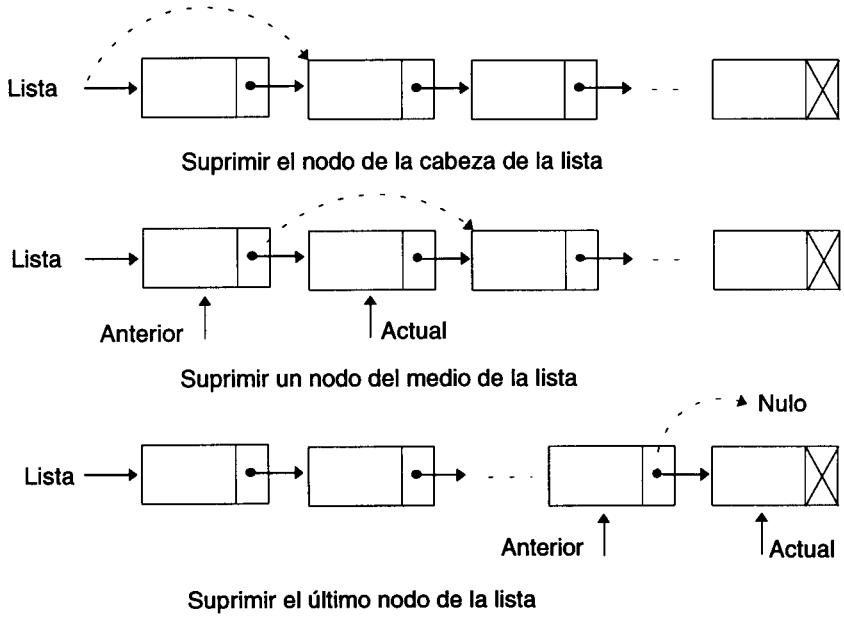


Figura 13.13: suprimir un nodo de una lista dinámica.

sea el primero de la lista, que sea un nodo de la mitad de la lista o que sea el último nodo de la lista.

```
SuprimirLista ( Lista, QuitarElemento )
PRINCIPIO
   Inicializar punteros del nodo actual y anterior;
   Buscar el nodo a suprimir;
   SI es el primer nodo
      ENTONCES Suprimir el primer nodo de la lista
      SI NO Suprimir un nodo del medio de la lista;
   Liberar la memoria ocupada por el nodo suprimido;
FIN
```

Algoritmo 13.12: suprimir un nodo de una lista dinámica.

13.3.3 Programa PASCAL.

{Paquete lista dinámica ordenada}

```
TYPE
```

```
   TipoPuntero = ^TipoNodo;
   TipoLista = TipoPuntero;
   TipoInfo = Integer;
   TipoNodo = RECORD
      Info:        TipoInfo;
      Siguiente:   TipoPuntero
   END;

VAR
   Lista: TipoLista;
   NuevoElemento, QuitaElemento: TipoInfo;

PROCEDURE InicializarLista ( VAR Lista: TipoLista );
{Crear una lista vacía}
BEGIN
   Lista := NIL
END; {Fin InicializarLista}
```

Programa PASCAL 13.9: procedimiento para inicializar una lista dinámica.

```
FUNCTION ListaVacía ( Lista: TipoLista ): Boolean;
{Averiguar si una lista está vacía}
BEGIN
   ListaVacía:= Lista = NIL
END;{ Fin ListaVacía}
```

Programa PASCAL 13.10: función para averiguar si una lista dinámica está vacía.

```
PROCEDURE RecorrerLista (Lista: TipoLista);
{Recorrer todos los elementos de la lista}
VAR PunteroAux: TipoPuntero;
BEGIN
   PunteroAux:= Lista; {Puntero para recorrer la lista}
   WHILE PunteroAux <> NIL DO
      BEGIN
         {Imprimir un elemento de la lista}
         Write(PunteroAux^.Info);
         {Avanzar al siguiente elemento}
         PunteroAux:= PunteroAux^.Siguiente
      END;
END;{Fin RecorrerLista}
```

Programa 13.11: procedimiento para recorrer una lista dinámica.

```
PROCEDURE InsertarLista ( VAR Lista: TipoLista; NuevoElemento: TipoInfo );
{Insertar un NuevoElemento en la lista enlazada apuntada por Lista}

VAR NuevoNodo, Puntero: TipoPuntero; SitioEncontrado: Boolean;

BEGIN
   New ( NuevoNodo ); {Crear el nuevo nodo}
   NuevoNodo^.Info := NuevoElemento; {Llenar campo Info del nodo nuevo}
   NuevoNodo^.Siguiente := NIL;{Llenar el campo Siguiente del nodo nuevo}
   IF ListaVacía ( Lista )
      THEN Lista := NuevoNodo {Insertando en una lista vacía}
      ELSE
         BEGIN {Insertando en una lista ya existente}
            IF NuevoElemento < Lista^.Info THEN
                  BEGIN {Insertar antes del primer nodo}
                     NuevoNodo^.Siguiente := Lista;
                     Lista := NuevoNodo
                  END; {Fin de insertar antes del primer nodo}
            ELSE
                  BEGIN {Insertar en medio o al final de la lista}
                     {Recorrer la lista para buscar lugar inserción}
                     Puntero := Lista;
                     SitioEncontrado := FALSE;
                     {Todos los elementos de la lista hasta el nodo al que apunta Puntero son <
                     que NuevoElemento}
                     WHILE (Puntero^.Siguiente<>NIL) AND NOT SitioEncontrado DO

                     {Comparar NuevoElemento con valor nodo que sigue actual}
                     IF NuevoElemento >= Puntero^.Siguiente^.Info
                        THEN {Seguir buscando}
                              Puntero := Puntero^.Siguiente
                        ELSE {Insertar después del nodo apuntado por Puntero}
                              SitioEncontrado:= TRUE;
                     {Conectar los punteros y materializar la inserción}
                     {El siguiente al nuevo será primero > que nuevo}
                     NuevoNodo^.Siguiente := Puntero^.Siguiente;
                     {Nodo nuevo será siguiente al último < que él}
                     Puntero^.Siguiente := NuevoNodo
                  END {Fin de insertar en medio o al final lista}
         END {Fin de insertar en una lista ya existente}
```

```
END; {Fin de InsertarLista}
```

Programa PASCAL 13.12: procedimiento para insertar un nodo en una lista dinámica.

```
PROCEDURE SuprimirLista ( VAR Lista: TipoLista; QuitarElemento: TipoInfo );

{Suprimir de la lista apuntada por Lista el nodo cuyo campo Info vale QuitarElemento. Se supone
que existe un nodo con ese valor y es único}

VAR Actual: TipoLista; Anterior: TipoPuntero;

BEGIN
{Inicializar punteros}
   Actual := Lista;
   Anterior := NIL;
   {Búsqueda del nodo a suprimir. QuitarElemento no aparece en la lista antes del nodo apuntado
   por Actual}
   WHILE Actual^.Info <> QuitarElemento DO
      BEGIN
            {Avanzar los dos punteros}
            Anterior := Actual;
            Actual := Actual^.Siguiente
      END;
   {Comprobar si se ha suprimido el primer nodo}
   IF Anterior := NIL
      THEN{Quitar el primer nodo de la lista}
            Lista := Lista^.Siguiente
      ELSE {Quitar un nodo del medio}
            Anterior^.Siguiente := Actual^.Siguiente;
   {Liberar el nodo suprimido para que se pueda reutilizar}
   Dispose ( Actual )
END; {Fin SuprimirLista}
```

Programa PASCAL 13.13: procedimiento para suprimir un nodo de una lista dinámica.

13.3.4 Programa C

```
/*Paquete lista dinámica*/
```

```
#include <stdio.h>
#include <malloc.h>

typedef enum {FALSE = 0, TRUE =1} boolean;
typedef int tipo_info;
typedef struct nodo_lista {
    tipo_info                   info;
    struct nodo_lista           *siguiente;
}tipo_nodo;
/*Mensaje originado por un malloc erróneo*/
void  no_memoria ( )
{
    fprintf ( stderr, "Lo siento no queda memoria\n" );
}/*Fin no_memoria*/
```

Programa C 13.13: función que indica que no se ha resuelto la asignación dinámica de memoria.

```
/*Crear memoria dinámica con un malloc*/
void *
crear_espacio ( )
{
    tipo_nodo       *puntero;

    puntero = ( tipo_nodo * ) malloc ( sizeof *puntero );
    /* ¿Puede hacar la asignación dinámica de memoria?*/
    if ( puntero == NULL ) no_memoria ( );
    return puntero;
}/*Fin crear_espacio*/
```

Programa C 13.14: función para hacer la asignación dinámica de memoria.

```
/*Inicializar lista*/
tipo_nodo *
inicializar_lista ( void )
/*Crear una lista dinámica vacía*/
{
    return NULL;
}/*Fin inicializar_lista*/
```

Programa C 13.15: función para inicializar una lista dinámica.

```
/*Lista vacía*/
boolean
lista_vacia (tipo_nodo *lista)
/*Averiguar si una lista dinámica está vacía*/
{
   return (lista == NULL);
}/*Fin lista_vacia*/
```

Programa C 13.16: función para averiguar si una lista dinámica está vacía.

```
/*Recorrer lista*/
void
recorrer_lista (tipo_nodo *lista)
/*Recorrer todos los nodos de la lista*/
{
   int        i = 0;
   tipo_nodo      *puntero_aux;

   puntero_aux = lista;/* puntero para recorrer la lista*/
   while (puntero_aux != NULL) {
/*    Imprimir un nodo de la lista*/
      printf ("posicion %d de la lista: %d\n", i++, puntero_aux->info);
/*    Avanzar al siguiente nodo*/
      puntero_aux = puntero_aux->siguiente;
   }
}/*Fin recorrer_lista*/
```

Programa C 13.17: función para recorrer una lista dinámica.

```
/*Insertar un nodo en una lista dinámica*/
tipo_nodo *
insertar_lista (tipo_nodo *lista, tipo_info nuevo_elemento)
/*Insertar nuevo_elemento el la lista enlazada apuntada por lista*/
{
   tipo_nodo      *nuevo_nodo, *puntero;
   boolean        sitio_encontrado;

   /*Crear un nuevo nodo*/
   nuevo_nodo = crear_espacio();
   /*Llenar el campo info del nuevo_nodo*/
   nuevo_nodo->info = nuevo_elemento;
   /*Llenar el campo suguiente del nuevo_nodo*/
```

```
    nuevo_nodo->siguiente = NULL;

    if ( lista_vacia ( lista ) )
        lista = nuevo_nodo;/*Insertar en una lista vacía*/
    else  { /*Insertar en una lista ya existente*/
        if ( nuevo_elemento < lista->info ) {
                /*Insertar antes del primer nodo*/
                nuevo_nodo->siguiente = lista;
                lista = nuevo_nodo;
                /*Fin de insertar antes del primer nodo*/
        } else { /*Insertar en el medio o al final de la lista*/
                puntero =  lista;
                sitio_encontrado = FALSE;
                /*Todos los elementos de la lista hasta el nodo al que apunta puntero son < que
                nuevo_elemento*/
                while( puntero->siguiente !=NULL && !sitio_encontrado ) {
                        /*Comparar nuevo_elemento con el valor del nodoque sigue al nodo actual*/
                        if( nuevo_elemento >= puntero->siguiente->info )
                                /*Seguir buscando*/
                                puntero = puntero->siguiente;
                        else /*Insertar después del nodo apuntado por puntero*/
                                sitio_encontrado = TRUE;
                }
                /*Conectar los punteros para materializar la inserción*/
                nuevo_nodo->siguiente = puntero->siguiente;
                puntero->siguiente = nuevo_nodo;
        }/*Fin de insertar en el medio o al final de la lista*/
    } /*Fin de insertar en una lista ya existente*/
    return lista;
}/*Fin insertar un nodo en una lista dinámica*/
```

Programa C 13.18: función para insertar un nodo en una lista dinámica.

```
/*Suprimir un nodo de una lista dinámica*/
tipo_nodo *
suprimir_lista(tipo_nodo *lista, tipo_info quitar_elemento)
/*Suprimir de la lista dinámica el nodo cuyo campo info vale quitar_elemento. Se supone que existe
un nodo con ese valor y es único*/
{
    tipo_nodo               *actual, *anterior;

    /*Inicializar punteros*/
```

```
    actual = lista;
    anterior = NULL;
    /*Búsqueda del nodo a suprimir*/
    while ( actual != NULL && actual->info != quitar_elemento ) {
        /*Avanzar los dos punteros*/
        anterior = actual;
        actual = actual->siguiente;
    }
    /*Comprobar si se ha suprimido el primer nodo*/
    if ( anterior == NULL )
            /*Quitar el primer nodo de la lista*/
            lista = lista->siguiente;
        else
            /*Quitar un nodo del medio*/
            anterior->siguiente = actual->siguiente;
        /*Liberar espacio del nodo suprimido para que se pueda reutilizar*/
        free ( actual );
        }
    return lista;
}/*Fin suprimir un nodo en una lista dinámica*/
```

Programa C 13.19: función para suprimir un nodo de una lista dinámica.

Indice de figuras

Indice de algoritmos

Indice de programas PASCAL

Indice de programas C

Bibliografía.

[ALF92] M. ALFONSECA & A. ALCALA
Programación orientada a objetos.
Ed. Anaya Multimedia, 1992.

[CAS92] J. CASTRO & F. CUCKER & X. MESSEGUER y otros.
Curs de programació.
Ed. McGraw-Hill, 1992.

[DAL86] N. DALE & D. ORSHALICK.
PASCAL.
Ed. McGraw-Hill, 1986.

[DAL89] N. DALE & Susan C. LILLY.
PASCAL y estructuras de datos.
Ed. McGraw-Hill, 1989 (Segunda Edición).

[JOY90] Luis JOYANES
Programación en TURBO PASCAL (Versiones 4.0, 5.0 y 5.5)
Ed. McGraw-Hill, 1990.

[KER91] Brian W. Kerningham & Dennis M. Ritchie
El lenguaje de programación C
Ed. Prentice Hall, 1991 (Segunda Edición).

[REY86] Charles W. REYNOLDS
Program Design and Data Structures in PASCAL
Ed. Wadsworth Publishing Company, 1986.

[SCH90a] Herbert SCHILT
Programación en TURBO C
Ed. McGraw-Hill, 1990 (Segunda edición).

[SCH90b] Herbert SCHILT
TURBO C. Programación avanzada.
Ed. McGraw-Hill, 1990 (Segunda edición).

[SED88] Robert SEDGEWICK
Algorithms.
Ed. Addison-Wesley, 1988 (Second edition).

[STR91] Bjarne STROUSTRUP
The C++ programming language.
Ed. Addison-Wesley, 1991 (Second edition).

[WIR86] N. WIRTH.
Algoritmos + Estructuras de Datos = Programas.
Ed. Castillo, 1986

[WIR90] N. WIRTH.
Algoritmos y estructuras de datos.
Ed. Prentice Hall, 1990.