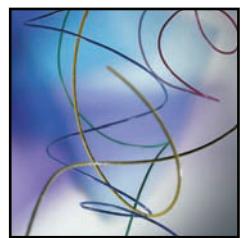
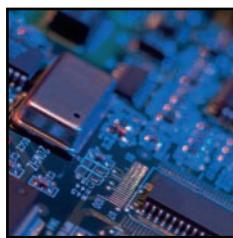


Segunda Edición

Libro de problemas

Fundamentos de programación I



FUNDAMENTOS DE PROGRAMACIÓN

Libro de problemas

Segunda edición

FUNDAMENTOS DE PROGRAMACIÓN. Libro de problemas. Segunda edición

No está permitida la reproducción total o parcial de este libro, ni su tratamiento informático, ni la transmisión de ninguna forma o por cualquier medio, ya sea electrónico, mecánico, por fotocopia, por registro u otros métodos, sin el permiso previo y por escrito de los titulares del Copyright.

DERECHOS RESERVADOS © 2003, respecto a la segunda edición en español, por
McGRAW-HILL/INTERAMERICANA DE ESPAÑA, S. A. U.
Edificio Valrealty, 1.^a planta
Basauri, 17
28023 Aravaca (Madrid)

ISBN: 84-481-3986-0
Depósito legal: M.

Editora: Concepción Fernández Madrid
Asist. editorial: Amelia Nieva
Diseño de cubierta: Design Master DIMA
Preimpresión: Puntographic, S. L.
Impreso en: Fareso, S. A.

IMPRESO EN ESPAÑA - PRINTED IN SPAIN

FUNDAMENTOS DE PROGRAMACIÓN

Libro de problemas

Segunda edición

**Luis Joyanes Aguilar
Luis Rodríguez Baena
Matilde Fernández Azuela**

Departamento de Lenguajes y Sistemas Informáticos e Ingeniería del Software
Facultad de Informática/Escuela Universitaria de Informática
Universidad Pontificia de Salamanca, *campus* Madrid



MADRID • BUENOS AIRES • CARACAS • GUATEMALA • LISBOA • MÉXICO
NUEVA YORK • PANAMÁ • SAN JUAN • SANTAFÉ DE BOGOTÁ • SANTIAGO • SÃO PAULO
AUCKLAND • HAMBURGO • LONDRES • MILÁN • MONTREAL • NUEVA DELHI • PARÍS
SAN FRANCISCO • SIDNEY • SINGAPUR • ST. LOUIS • TOKIO • TORONTO

CONTENIDO

Prólogo	xi
Capítulo 1. Algoritmos y programas	1
1.1. Configuración de una computadora	1
1.2. Lenguajes de programación	2
1.3. Resolución de problemas	3
1.3.1. Fase de resolución del problema	3
1.3.1.1. Análisis del problema	3
1.3.1.2. Diseño del algoritmo	3
1.3.1.3. Verificación de algoritmos	5
1.3.2. Fase de implementación	5
1.4. Ejercicios resueltos	6
Capítulo 2. La resolución de problemas con computadoras y las herramientas de programación	15
2.1. Datos	15
2.1.1. Constantes	16
2.1.2. Variables	16
2.1.3. Expresiones	16
2.1.4. Funciones	17
2.2. Representación de algoritmos	18
2.3. Diagrama de flujo	18
2.4. Diagrama Nassi-Schneiderman	19
2.5. Pseudocódigo	20
2.5.1. Comentarios	20
2.5.2. Palabras reservadas	21
2.5.3. Identificadores	21
2.5.4. Operadores y signos de puntuación	22
2.5.5. Literales	22
2.6. Ejercicios resueltos	22
Capítulo 3. Estructura general de un programa	39
3.1. Estructura de un programa	39
3.2. Estructura general de un algoritmo en pseudocódigo	40
3.3. La operación de asignación	41

3.3.1. Contadores	41
3.3.2. Acumuladores	41
3.3.3. Interruptores	42
3.4. Ejercicios resueltos	42
Capítulo 4. Introducción a la programación estructurada	55
4.1. Programación estructurada	55
4.2. Teorema de Böhm y Jacopini	55
4.3. Control del flujo de un programa	56
4.3.1. Estructura secuencial	56
4.3.2. Estructura selectiva	56
4.3.3. Estructura repetitiva	58
4.3.4. Estructura anidada	60
4.3.5. Sentencias de salto	61
4.4. Ejercicios resueltos	61
Capítulo 5. Subprogramas (subalgoritmos), procedimientos y funciones	79
5.1. Programación modular	79
5.2. Funciones	80
5.2.1. Declaración de funciones	80
5.3. Procedimientos	81
5.3.1. Declaración de procedimientos	81
5.4. Estructura general de un algoritmo	81
5.5. Paso de parámetros	82
5.6. Variables locales y globales	84
5.7. Recursividad	84
5.8. Ejercicios resueltos	85
Capítulo 6. Estructuras de datos (arrays y registros)	105
6.1. Datos estructurados	105
6.2. Arrays (arreglos)	106
6.2.1. Arrays unidimensionales	107
6.2.2. Arrays bidimensionales	107
6.2.3. Recorrido de los elementos del array	108
6.2.4. Arrays como parámetros	109
6.3. Conjuntos	109
6.4. Registros	111
6.4.1. Arrays de registros y arrays paralelos	111
6.5. Ejercicios resueltos	112
Capítulo 7. Las cadenas de caracteres	149
7.1. Cadenas	149
7.2. Operaciones con cadenas	150
7.3. Funciones útiles para la manipulación de cadenas	151
7.4. Ejercicios resueltos	151
Capítulo 8. Archivos (ficheros). Archivos secuenciales	159
8.1. Conceptos generales sobre archivos	159
8.1.1. Jerarquización	160
8.1.2. Clasificación de los archivos según su función	160
8.1.3. Operaciones básicas	160
8.1.4. Otras operaciones usuales	161
8.1.5. Soportes	161

8.2. Flujos	161
8.3. Organización secuencial	161
8.3.1. Archivos de texto	162
8.3.2. Mantenimiento de archivos secuenciales	163
8.4. Ejercicios resueltos	164
Capítulo 9. Archivos directos	185
9.1. Organización directa	185
9.1.1. Funciones de conversión de clave	186
9.1.2. Tratamiento de sinónimos	187
9.1.3. Mantenimiento de archivos directos	187
9.2. Organización secuencial indexada	187
9.3. Modos de acceso	189
9.3.1. Archivos indexados	189
9.4. Ejercicios resueltos	190
Capítulo 10. Ordenación, búsqueda e intercalación	223
10.1. Búsqueda	223
10.1.1. Búsqueda secuencial	223
10.1.2. Búsqueda binaria	224
10.1.3. Búsqueda por transformación de claves	224
10.1.3.1. Funciones de conversión de clave	224
10.1.3.2. Resolución de colisiones	226
10.2. Ordenación	227
10.2.1. Ordenación interna	227
10.2.1.1. Selección	227
10.2.1.2. Burbuja	228
10.2.1.3. Inserción directa	228
10.2.1.4. Inserción binaria	228
10.2.1.5. Shell	229
10.2.1.6. Ordenación rápida	229
10.3. Intercalación	230
10.4. Ejercicios resueltos	232
Capítulo 11. Búsqueda, ordenación y fusión externas (archivos)	239
11.1. Conceptos generales	239
11.2. Búsqueda externa	239
11.3. Fusión	239
11.4. Ordenación externa	240
11.4.1. Partición de archivos	240
11.4.1.1. Partición por contenido	240
11.4.1.2. Partición en secuencias de longitud 1	240
11.4.1.3. Partición en secuencias de longitud N	240
11.4.1.4. Partición en secuencias de longitud N con clasificación interna de dichas secuencias	240
11.4.1.5. Partición según el método de selección por sustitución	241
11.4.1.6. Partición por el método de selección natural	241
11.4.2. Ordenación por mezcla directa	241
11.4.3. Ordenación por mezcla natural	241
11.5. Ejercicios resueltos	242
Capítulo 12. Estructuras dinámicas lineales de datos (listas enlazadas, pilas, colas)	261
12.1. Estructuras dinámicas	261
12.2. Listas	262

12.3.	Pilas	265
12.3.1.	Aplicaciones de las pilas	266
12.4.	Colas	266
12.4.1.	Doble cola	266
12.4.2.	Aplicaciones de las colas	267
12.5.	Ejercicios resueltos	267
Capítulo 13. Estructuras de datos no lineales (árboles y grafos)		307
13.1.	Árboles	307
13.1.1.	Terminología	308
13.1.2.	Árboles binarios	308
13.1.2.1.	Conversión de un árbol general en binario	309
13.1.2.2.	Implementación	310
13.1.2.3.	Recorridos de un árbol binario	311
13.1.2.4.	Árbol binario de búsqueda	312
13.2.	Grafos	312
13.2.1.	Terminología	313
13.2.1.	Representación de los grafos	313
13.3.	Ejercicios resueltos	315
Capítulo 14. Recursividad		333
14.1.	Concepto y tipos de recursividad	333
14.2.	Uso adecuado de la recursividad	334
14.3.	Métodos para la resolución de problemas que utilizan recursividad	335
14.4.	Ejercicios resueltos	336
Capítulo 15. Introducción a la Programación Orientada a Objetos		357
15.1.	Mecanismos de abstracción	357
15.1.1.	Funciones y procedimientos	357
15.1.2.	Módulos	358
15.1.3.	Tipos datos abstractos	358
15.2.	Modelado del mundo real: clases y objetos	358
15.2.1.	Atributos	359
15.2.2.	Comportamiento	360
15.2.3.	Identidad	360
15.2.4.	Paso de mensajes	360
15.3.	El enfoque orientado a objetos	360
15.4.	Clases	363
15.4.1.	Declaración de clases	363
15.5.	Representación gráfica de una clase en UML	364
15.5.1.	Atributos	364
15.5.2.	Operaciones	365
15.5.3.	Representación gráfica de una clase	366
15.5.4.	Notación de objetos	366
15.5.5.	Reglas para encontrar clases en el análisis	367
15.6.	Responsabilidad de una clase	368
15.7.	Declaración de objetos	369
15.8.	Los miembros de un objeto	369
15.9.	Constructores	370
15.10.	Acceso a los miembros de un objeto, visibilidad y encapsulamiento	370
15.11.	Resumen	371
15.12.	Ejercicios resueltos	371

Capítulo 16. Relaciones: Asociación, Generalización, Herencia	379
16.1. Relaciones entre clases	379
16.2. Asociaciones	379
16.3. Agregaciones	382
16.3.1. Composición	383
16.4. Jerarquía de clases: Generalización y Especialización	384
16.5. Clases abstractas	392
16.6. Polimorfismo	394
16.7. Ejercicios resueltos	395
Apéndice A. Especificaciones del lenguaje algorítmico UPSAM. Versión 2.0	407
A.1. Elementos del lenguaje	407
A.1.1. Identificadores	407
A.1.2. Comentarios	407
A.1.3. Tipos de datos estándar	407
A.1.4. Constantes de tipos de datos estándar	408
A.1.5. Operadores	409
A.2. Estructura de un programa	409
A.2.1. Declaración de tipos de datos estructurados	409
A.2.2. Declaración de constantes	410
A.2.3. Declaración de variables	410
A.2.4. Biblioteca de funciones	411
A.2.5. Procedimientos de entrada/salida	411
A.2.6. Instrucción de asignación	412
A.3. Estructuras de control	412
A.3.1. Estructuras selectivas	412
A.3.2. Estructuras repetitivas	412
A.4. Programación modular	413
A.4.1. Cuestiones generales	413
A.4.2. Procedimientos	413
A.4.3. Funciones	414
A.5. Archivos	414
A.5.1. Archivos secuenciales	414
A.5.2. Archivos de texto	415
A.5.3. Archivos directos	416
A.5.4. Consideraciones adicionales	417
A.6. Variables dinámicas	417
A.7. Programación orientada a objetos	418
A.7.1. Cables y objetos	418
A.7.2. Atributos	419
A.7.3. Métodos	420
A.7.4. Herencia	421
A.8. Palabras reservadas	422
Apéndice B. Bibliografía	425
Índice	431

PRÓLOGO

La iniciación de un estudiante —de *informática, de ingeniería de sistemas, de ciencias de la computación*, o de cualquier otra rama de las *ciencias e ingeniería*— en las técnicas de programación del siglo XXI requiere no sólo del aprendizaje clásico del diseño de algoritmos y de la comprensión de las técnicas estructuradas, sino también de técnicas orientadas a objetos. Por esta circunstancia, esta 2.^a edición ha introducido como gran novedad un capítulo completo de *recursividad*, que muchos lectores de la primera edición nos habían solicitado, así como dos capítulos, lo más completos posibles, sobre propiedades y técnicas de *programación orientada a objetos (POO)*.

La **POO** se ha convertido en un paradigma importante en todos los campos de las Ciencias de la Computación y, por ello, es importante enseñar programación OO desde los primeros cursos de programación. Este libro pretende iniciar al lector en la programación orientada a objetos, enseñándole las técnicas básicas con el objetivo fundamental de poder aprender en una segunda etapa y de modo riguroso a programar con un enfoque orientado a objetos y con ayuda de algún lenguaje OO tal como C++, Java o C#, e incluso Visual Basic o mejor VB .Net.

Para conseguir los objetivos del libro utilizaremos fundamentalmente el lenguaje algorítmico, con formato de pseudocódigo, herramienta ya probada y experimentada, no sólo en la primera edición de esta obra, sino también en las tres ediciones de la obra complementaria *Fundamentos de programación*, y muy utilizada en numerosas universidades y centros de formación de todo el mundo.

La clave para desarrollar software es aplicar el concepto de abstracción en el diseño e implementación de proyectos software. En base a ello, se busca también enseñar a resolver problemas utilizando diversos niveles de abstracción, y enseñar cómo visualizar y analizar los problemas en niveles diferentes.

El libro contiene los temas más importantes de la programación tradicional tales como estructuras de control, funciones, estructuras de datos, métodos de ordenación y búsqueda, ... junto con los conceptos fundamentales de orientación a objetos tales como clases, objetos, herencia, relaciones, etc.

OBJETIVOS DEL LIBRO

El libro pretende enseñar a programar utilizando conceptos fundamentales tales como:

1. *Algoritmos* (conjunto de instrucciones programadas para resolver una tarea específica).
2. *Datos* (una colección de datos que se proporcionan a los algoritmos que se han de ejecutar para encontrar una solución: los datos se organizarán en *estructuras de datos*).
3. *Objetos* (el conjunto de datos y algoritmos que los manipulan, encapsulados en un tipo de dato nuevo conocido como *objeto*).

4. *Clases* (tipos de objetos con igual estado y comportamiento, o dicho de otro modo, los mismos atributos y operaciones).
5. *Estructuras de datos* (conjunto de organizaciones de datos para tratar y manipular eficazmente datos homogéneos y heterogéneos).
6. *Temas avanzados* (archivos, recursividad y ordenaciones/búsquedas avanzadas).

Los dos primeros aspectos, algoritmos y datos, han permanecido invariables a lo largo de la corta historia de la informática/computación, pero la *interrelación* entre ellos sí que ha variado y continuará haciéndolo. Esta interrelación se conoce como *paradigma de programación*.

Así pues y en resumen, los objetivos fundamentales de esta obra son: *introducción a la programación estructurada, estructuras de datos y programación orientada a objetos* utilizando un lenguaje algorítmico **UPSAM 2.0** que utiliza como herramienta fundamental el *pseudocódigo*, aunque también se enseñan las herramientas tradicionales tales como *diagramas de flujo* y *diagramas N-S*.

EL LIBRO COMO HERRAMIENTA DOCENTE

La experiencia de los autores desde hace muchos años con obras muy implantadas en el mundo universitario, pero en este caso y de modo especial, la primera edición de **Fundamentos de programación. Libro de problemas** nos ha llevado a mantener la estructura de esta obra, actualizando el contenido que se prevé para los estudiantes del actual siglo XXI y con un lenguaje de programación como es el *lenguaje algorítmico UPSAM 2.0* que pretende contener la sintaxis y las estructuras gramaticales de lenguajes modernos como Java y C# o los ya clásicos C y C++, sin olvidar los populares Pascal o FORTRAN. Por ello, en el contenido de la obra hemos tenido en cuenta no sólo las directrices de los planes de estudio españoles de *ingeniería informática* (antigua licenciatura en informática) y *ciencias de la computación*, sino también de ingenierías tales como *industriales, telecomunicaciones, agrónomos o minas*, o las más recientes incorporadas en España, como *ingeniería en geodesia, ingeniería química o ingeniería telemática*. Nuestro conocimiento del mundo educativo latinoamericano nos ha llevado a pensar también en las carreras de *ingeniería de sistemas computacionales* y las *licenciaturas en informática y en sistemas de información*, como se las conoce en Latinoamérica.

Por todo lo anterior, el contenido del libro intenta seguir un programa estándar de un primer curso de introducción a la programación y, según situaciones, un segundo curso de programación de nivel medio, en asignaturas tales como *Metodología de la programación, Fundamentos de programación, Introducción a la programación,...* El contenido del libro abarca los citados programas y comienza con la *introducción a los algoritmos y a la programación*, para llegar a *estructuras de datos y objetos*. Por esta circunstancia, la estructura del curso no ha de ser secuencial en su totalidad, sino que el profesor/maestro y el alumno/lector podrán estudiar sus materias en el orden que consideren más oportuno.

Se trata de describir los *dos paradigmas* más populares en el mundo de la programación: el *procedimental* y el *orientado a objetos*. Los cursos de programación en sus niveles inicial y medio están evolucionando para aprovechar las ventajas de nuevas y futuras tendencias en ingeniería de software y en diseño de lenguajes de programación, específicamente diseño y programación orientada a objetos. Algunas facultades y escuelas de ingenieros, junto con la nueva formación profesional (*ciclos formativos de nivel superior*) en España y en Latinoamericana, están introduciendo a sus alumnos en la programación orientada a objetos, inmediatamente después del conocimiento de la programación estructurada, e incluso en ocasiones antes.

El contenido del libro se ha pensado en el desarrollo de dos cuatrimestres o semestres (según la terminología americana) y siguiendo los descriptores (temas centrales) recomendados por el Consejo de Universidades de España para los planes de estudio de ingeniería en informática, ingeniería técnica en informática e ingeniería técnica en informática de gestión, así como en asignaturas tales como introducción a la programación y fundamentos de programación de carreras como ingeniero e ingeniero técnico de telecomunicaciones, ingeniería telemática, ingeniería industrial y carreras afines. Así mismo y dado nuestro conocimiento de numerosas universidades latinoamericanas, se ha pensado en carreras de ingeniería de sistemas.

No podíamos dejar de lado las recomendaciones de la más prestigiosa organización de informáticos del mundo, ACM, sobre todo pensando en nuestros lectores de Latinoamérica. Se estudió en su momento los borradores de la *Curricula de Computer Science* de modo que tras su publicación el 15 de diciembre de 2001 del *Computing Curricula 2001 Computer Science*. Como lógicamente no se podía seguir todas sus directrices al pie de la letra, del cuerpo de conocimiento se optó por seguir del modo más aproximado posible las materias, *Programming Fundamentals* (PF) *Algorithms and Complexity* (AL) y *Programming Languages* (PL).

Uno de los temas más debatidos en la educación en informática o en ciencias de la computación (*Computer Sciences*) es el rol de la programación en el currículo introductorio. A través de la historia de la disciplina —como fielmente reconoce en la introducción del Capítulo 7 relativo a cursos de introducción, ACM en su Computing Curricula 2001— la mayoría de los cursos de introducción a la informática se han centrado principalmente en el desarrollo de habilidades o destrezas de programación. La adopción de un curso de introducción a la programación proviene de una serie de factores prácticos e históricos incluyendo los siguientes:

- La programación es una técnica esencial que debe ser dominada por cualquier estudiante de informática. Su inserción en los primeros cursos de la carrera asegura que los estudiantes tengan la facilidad necesaria con la programación para cuando se matriculan en los cursos de nivel intermedio y avanzado.
- La informática no se convirtió en una disciplina académica hasta después que la mayoría de las instituciones ha desarrollado un conjunto de cursos de programación introductorias que sirvan a una gran audiencia.
- El modelo de programación del currículum del 78 de la ACM definía a estos cursos como «Introducción a la Programación» y se les denominó CS1 y CS2. Hoy día se le sigue denominando así después de la publicación de los currículos del 91 y del 91 y del 2001.
- Los programas de informática deben enseñar a los estudiantes cómo usar bien al menos un lenguaje de programación. Además, se recomienda que los programas en informática deben enseñar a los estudiantes a ser competentes en lenguajes y que hagan uso de al menos dos paradigmas de programación. Como consecuencia de estas ideas, el currículo 2001 de la ACM contempla la necesidad de conceptos y habilidades que son fundamentales en la práctica de la programación con independencia del paradigma subyacente. Como resultado de este pensamiento, el área de fundamentos de programación incluye unidades sobre conceptos básicos de programación, estructuras de datos básicas y procesos algorítmicos.

La fluidez en un lenguaje de programación es requisito para el estudio de las ciencias de la computación, pero la dificultad de elegir el lenguaje siempre es una dificultad más a añadir a la ya de por sí difícil misión del maestro o profesor. Por esta razón, ya en el lejano año de 1986 cuando publicamos la primera edición del libro *Fundamentos de programación* y tras la experiencia de sus ediciones sucesivas y sus ya casi 18 años (la mayoría de edad en casi todos los países del mundo), la apuesta que hicimos de utilizar un lenguaje algorítmico, seguimos manteniéndola, y en esta ocasión hemos podido ofrecer la versión 2.0 del lenguaje UPSAM utilizada y probada en numerosas universidades españolas y americanas.

Este libro se ha escrito pensando en que pudiera servir de referencia, guía de estudio y sobre todo como herramientas de prácticas de *introducción a la programación*, con una segunda parte que, a su vez, sirviera como continuación, y de *introducción a las estructuras de datos* y a la *programación orientada a objetos*; todos ellos utilizando un *pseudolenguaje* o *lenguaje algorítmico* como lenguaje de programación. El objetivo final que busca es no sólo describir la sintaxis de dicho lenguaje, sino y, sobre todo, mostrar las características más sobresalientes del lenguaje a la vez que se enseñan técnicas de programación estructurada y orientada a objetos. Así pues, los objetivos fundamentales son:

- Énfasis fuerte en el análisis, construcción y diseño de programas.
- Un medio de resolución de problemas mediante técnicas de programación.
- Una introducción a la informática y a las ciencias de la computación usando algoritmos y el lenguaje algorítmico, basado en pseudocódigo.

En resumen, este es un libro diseñado para *enseñar a programar utilizando un lenguaje algorítmico* con la herramienta de sintaxis, del pseudocódigo, y con una versión probada y actualizada UPSAM 2.0 cuya primera versión vio la luz en la primera edición de la obra base de este libro *Fundamentos de programación*, en el año 1986. Así, se tratará de enseñar las técnicas clásicas y avanzadas de programación estructurada, junto con técnicas orientadas a objetos. La programación orientada a objetos no es la panacea universal del programador del siglo XXI, pero le ayudará a realizar tareas que, de otra manera, serían complejas y tediosas.

El contenido del libro trata de proporcionar soporte a un año académico completo (dos semestres o cuatrimestres), alrededor de 24 a 32 semanas, dependiendo lógicamente de su calendario y planificación. Los nueve primeros capítulos pueden comprender el primer semestre y los restantes capítulos pueden impartirse en el segundo semestre. Lógicamente la secuencia y planificación real dependerá del maestro o profesor que marcará y señalará, semana a semana, la progresión que él considera lógica. Si es un estudiante autodidacta, su propia progresión vendrá marcada por las horas que dedique al estudio y al aprendizaje con la computadora, aunque no debe variar mucho del ritmo citado al principio de este párrafo.

LIBRO COMPLEMENTARIO

Aunque este libro ha sido escrito como una obra práctica de aprendizaje para la *introducción a la programación de computadoras* y no se necesita más conocimientos que los requeridos para la iniciación a los estudios universitarios o de formación profesional en carreras de ingeniería o licenciatura en informática, ingeniería de sistemas, telecomunicaciones o industriales, y estudios de formación profesional en las ramas tecnológicas, también ha sido escrito pensando en ser libro complementario de la obra *Fundamentos de programación* (Joyanes, 3.^a ed. McGraw-Hill, 2003). Para ello el contenido del libro coincide casi en su totalidad con los diferentes capítulos de la citada obra. De esta manera, se pueden estudiar conjuntamente ambos libros con lo que se conseguirá no sólo un aprendizaje más rápido sino y sobre todo mejor formación teórico-práctica y mayor rigor académico y profesional en la misma. La parte teórica de este libro es suficiente para aprender los problemas y ejercicios de programación resueltos y proponer su propias soluciones, sobre la base de que muchos ejercicios propuestos en el libro de teoría ofrecen la solución en este libro de problemas.

ORGANIZACIÓN DEL LIBRO

El libro, aunque no explícitamente, se puede considerar dividido a efectos de organización docente por parte de profesores (maestros) y alumnos o de lectores autodidactas, en cuatro partes que unidas constituyen un curso completo de programación. Dado que el conocimiento es acumulativo, los primeros capítulos proporcionan el fundamento conceptual para la comprensión y aprendizaje de algoritmos, y una guía a los estudiantes a través de ejemplos y ejercicios sencillos; y los capítulos posteriores presentan de modo progresivo la programación en pseudocódigo en detalle, tanto en el paradigma procedimental como en el orientado a objetos. El contenido detallado del libro es el siguiente:

Capítulo 1. Algoritmos y programas. Presenta una breve descripción del concepto y propiedades fundamentales de los algoritmos y de los programas. Introduce al lector/estudiante en los elementos básicos de un programa, tipos de datos, operaciones básicas, etc. soportadas por la mayoría de los lenguajes de programación.

Capítulo 2. Resolución de problemas con computadoras. Se muestran los métodos fundamentales para la resolución de problemas con computadoras y las herramientas de programación necesarias para ello. Se describen las etapas clásicas utilizadas en la resolución de problemas, así como las herramientas clásicas tales como pseudocódigo, diagrama de flujo y diagrama N-S.

Capítulo 3. Estructura general de un programa. Se analiza la estructura general de un programa y sus elementos básicos. Se introduce el concepto de flujo de control y se incluyen los primeros problemas planteados de cierta complejidad y su resolución con algunas de las herramientas descritas en el Capítulo 2.

Capítulo 4. Introducción a la programación estructurada. Enseña la organización y estructura general de un programa, así como su creación y proceso de ejecución. Se enseñan las técnicas de programación utilizadas en programación modular y en programación estructurada; se amplía el concepto de flujo de control desarrollado en el Capítulo 3 y se muestran una gran cantidad de ejemplos y ejercicios aclaratorios de las estructuras clásicas empleadas en programación estructurada: secuenciales, selectivas y repetitivas.

Capítulo 5. Subprogramas. Se estudia el importante concepto de subprograma (*subalgoritmo*), procedimiento y función. Los temas descritos incluyen los conceptos de argumentos y parámetros, así como su correspondencia y la comunicación entre el programa principal y los subprogramas, y entre ellos entre sí. El método de invocación de un subprograma junto con los valores devueltos en el caso de las funciones son otro de los temas considerados en el capítulo. Se analizan también los problemas planteados por el uso inadecuado de variables globales conocidos como *efectos laterales*. Se hace una introducción al concepto de recursividad que se estudiará en profundidad en el Capítulo 14.

Capítulo 6. Estructuras de datos. El concepto de *array* (tabla, lista, vector o matriz) o *arreglo* (como se conoce al término en inglés, prácticamente en toda Latinoamérica). El procesamiento de arrays de una y dos dimensiones, y multidimensionales, se analiza junto con el recorrido de los elementos pertenecientes a dicho *array*. De igual forma se describe el concepto de registro y array de registros.

Capítulo 7. Cadenas de caracteres. Se examinan los conceptos de datos tipo carácter y tipo cadena (*string*) junto con su declaración e inicialización. Se introducen conceptos básicos de manipulación de cadenas, lectura y asignación junto con operaciones básicas tales como longitud, concatenación, comparación, conversión y búsqueda de caracteres y cadenas.

Capítulo 8. Archivos secuenciales. Describe el concepto de archivo externo, su organización, almacenamiento y recuperación en dispositivos externos tales como discos y cintas. Los diferentes tipos de archivos y operaciones típicas que se realizan con ellos son tema central del capítulo, así como las organizaciones secuenciales.

Capítulo 9. Archivos directos. Este capítulo describe las organizaciones de datos y archivos aleatorios o de acceso directo, así como las indexadas, junto con los algoritmos de manipulación de los mismos.

Capítulo 10. Ordenación, búsqueda e intercalación externa. Dos de las operaciones más importantes que se realizan en algoritmos y programas de cualquier entidad y complejidad son: *ordenación de elementos de una lista o tabla y búsqueda de un elemento en dichas listas o tablas*. Los métodos básicos más usuales de búsqueda y ordenación se describen en el capítulo. Así mismo se explica el concepto de análisis de algoritmos de ordenación y búsqueda.

Capítulo 11. Búsqueda, ordenación y fusión externa. Se describen las operaciones básicas sobre archivos en almacenamiento externo.

Capítulo 12. Estructuras lineales de datos. Se examinan las estructuras de datos lineales fundamentales, tales como listas enlazadas, pilas y colas. Las ideas abstractas de pila y cola se describen en el capítulo. Pilas y colas se pueden implementar de diferentes maneras, bien con vectores (*arrays*) o con listas enlazadas. Una lista enlazada es una estructura de datos que mantiene una colección de elementos, pero el número de ellos no se conoce por anticipado o varía en un amplio rango. La lista enlazada se compone de elementos que contienen un valor y un puntero. El capítulo describe los fundamentos teóricos y las operaciones que se pueden realizar en la lista enlazada. También se describen los distintos tipos de listas enlazadas tales como doblemente enlazadas y circulares.

Capítulo 13. Estructuras no lineales de datos. Los árboles son otro tipo de estructura de datos dinámica y no lineal. Las operaciones básicas en los árboles junto con sus operaciones fundamentales se estudian en este capítulo.

Capítulo 14. Recursividad. El importante concepto de **recursividad** (propiedad de una función de llamarla a sí misma) se introduce en el capítulo junto con algoritmos complejos de ordenación y búsqueda en los que además se estudia su eficiencia.

Capítulo 15. Introducción a la programación orientada a objetos. Nuestra experiencia en la enseñanza de la programación orientada a objetos a estudiantes universitarios data de finales de la década de los 80. En esta década transcurrida, los primitivos y básicos conceptos de orientación a objetos se siguen manteniendo desde el punto de vista conceptual y práctico, tal y como se definieron hace más de 10 años. Hoy, sin embargo, la programación orientada a objetos es una clara realidad y por ello cualquier curso de intro-

ducción a la programación aconseja incluir un pequeño curso de orientación a objetos que puede impartirse como un curso independiente, como complemento de la programación estructurada o como parte de un curso completo de introducción a la programación orientación a objetos. En este capítulo se describen con ejemplos prácticos los principios básicos y fundamentales del paradigma de programación orientada a objetos. Los conceptos de objetos, clases y mensajes, así como los conceptos iniciales de **UML** como Lenguaje Unificado de Modelado.

Capítulo 16. Relaciones: Asociación, generalización y herencia. Los objetos y clases se relacionan entre sí mediante relaciones que luego son implementadas en un lenguaje de programación. En este capítulo se estudian las relaciones más importantes y sus notaciones gráficas en el lenguaje de modelado **UML**.

APÉNDICES

En todos los libros dedicados a la enseñanza y aprendizaje de técnicas de programación es frecuente incluir apéndices de temas complementarios a los explicados en los capítulos anteriores. Estos apéndices sirven de guía y referencia de elementos importantes del lenguaje y de la programación de computadoras. Sólo se ha incluido la **Guía de sintaxis del lenguaje algorítmico propuesto UPSAM 2.0**. Si el alumno necesita conocer la gramática y sintaxis, así como las estructuras fundamentales de lenguajes de programación, le recomendamos consulte nuestro libro complementario de teoría *Fundamentos de programación*. En dicha obra encontrará guías completas de los lenguajes de programación C, C++, Pascal, Java y C#, junto con otro tipo de información práctica complementaria, que consideramos le podrán ser de gran utilidad en su formación en lenguajes de programación.

AGRADECIMIENTOS

Un libro nunca es fruto único del autor, sobre todo si el libro está concebido como libro de texto y autoaprendizaje, y en este caso fundamentalmente de prácticas, y pretende llegar a lectores y estudiantes de informática y de computación, y, en general, de ciencias e ingeniería, así como autodidactas en asignaturas tales como programación (introducción, fundamentos, avanzada, etc.). Esta obra no es una excepción a la regla y son muchas las personas que nos han ayudado a terminarla. En primer lugar, deseamos agradecer a nuestros colegas de la *Universidad Pontificia de Salamanca en el campus de Madrid*, y en particular del *Departamento de lenguajes y sistemas informáticos e ingeniería de software* de la misma que, desde hace muchos años, nos ayudan y colaboran en la impartición de las diferentes asignaturas del departamento y sobre todo en la elaboración de los programas y planes de estudio de las mismas. A todos ellos les agradecemos públicamente su apoyo y ayuda.

Así mismo queremos destacar que las especificaciones del lenguaje **UPSAM 2.0** que dispone de un compilador experimental han sido escritas por el citado departamento, probado y experimentado desde su creación y sometido a revisiones y actualizaciones cada curso académico. En este caso se presenta la *versión 2.0* publicada en la *3.ª edición* del libro complementario de este libro práctico, *Fundamentos de programación*, y en el prólogo del mismo están relacionados todos los profesores que hemos contribuido desde el primer borrador aparecido en 1986. A todos ellos les agradecemos muy sinceramente todo su apoyo y su trabajo. Sin ese esfuerzo no se hubiera podido publicar esta última edición de **UPSAM 2.0**.

Además de a nuestros compañeros en la docencia y a nuestros alumnos, no puedo dejar de agradecer, una vez más, a mi editora —y sin embargo amiga— Concha Fernández, las constantes muestras de afecto y comprensión que siempre tiene con nuestras obras. Esta ocasión como no era menos, tampoco ha sido una excepción.

Naturalmente —y aunque ya los hemos citado anteriormente—, no podemos dejar de agradecer a nuestros numerosos alumnos, estudiantes y lectores, en general, españoles y latinoamericanos, que, continuamente nos aconsejan, critican y nos proporcionan ideas para mejoras continuas de nuestros libros. Sin todo lo que hemos aprendido, seguimos aprendiendo y seguiremos aprendiendo de ellos y sin su aliento continuo, sería prácticamente imposible para nosotros terminar nuevas obras y en especial este libro. De modo muy espe-

cial deseamos expresar nuestro agradecimiento a tantos y tantos colegas de universidades españolas y latinoamericanas que apoyan nuestra labor docente y editorial. Nuestro más sincero reconocimiento y agradecimiento a todos: alumnos, lectores, colegas, profesores, maestros, monitores y editores. Sabemos que siempre estaremos en deuda con vosotros. Nuestro consuelo es que vuestro apoyo nos sigue dando fuerza en esta labor académica y que allá por donde vamos, siempre mostramos ese agradecimiento a esa inmensa ayuda que la comunidad académica nos presta. Gracias, una vez más por vuestra ayuda.

En Madrid, verano del 2003.

LOS AUTORES

ALGORITMOS Y PROGRAMAS

@Udf]bWdU'fUhQE'dlfUei Y`Ug`dYf gcbUg`UdfYbXUb`Yb[i U`Yg`XY`dfc[fUa UQE`Yg`i h`]nUf`UWa di ! hUXcfU`Wa c`i bU\YffUa]YbhU`Yb`UfYgc`i WQE`XY`dfcVYa Ug`@UfYgc`i WQE`XY`i b`dfcVYa UYl][Y U`a Ybcg`cg`g][i]YbhYg`dUgcg

% 8YZ]b]WQE`c`Ubz`]glg`XY`dfcVYa U`
 & 8]gY`c`XY`U[cf]ha c`c`a fhcXc`dUfUfYgc`j Yf`c"
 ' Hf`UbgZcf a WQE`XY`U[cf]ha c`Yb`i b`dfc[fUa U`
 (" 9^YWWQE`mj U]XUWQE`XY`dfc[fUa U`

I bc`XY`cg`cV`Yh]j cg`Zi bXUa YbhU`Yg`XY`YghY`]Vfc`Yg`Y`UdfYbX]nUY`mX]gY`c`XY`U[cf]ha cg`9ghY Wdjh`i `c`]bhfcXi W`U`Y`Wcf`Yb`Y`WbWdhc`XY`U[cf]ha c`midfc[fUa U`

1.1. CONFIGURACIÓN DE UNA COMPUTADORA

Una computadora es un dispositivo electrónico utilizado para procesar información y obtener resultados. Los componentes físicos que constituyen la computadora, junto con los dispositivos que realizan las tareas de entrada y salida, se conocen con el término *hardware*. En su configuración física más básica una computadora está constituida por una Unidad Central de Proceso (**UCP**)*, una Memoria Principal y unas Unidades de Entrada y Salida. Las funciones desempeñadas por cada una de estas partes son las siguientes:

- Las unidades de Entrada/Salida se encargan de los intercambios de información con el exterior.
- La memoria principal almacena tanto las instrucciones que constituyen los programas como los datos y resultados obtenidos; para que un programa se ejecute, tanto él como sus datos, se deben situar en memoria central. Sin embargo, la información almacenada en la memoria principal se pierde cuando se desconecta la computadora de la red eléctrica y, además, dicha memoria es limitada en capacidad. Por estas razones, los programas y datos necesitan ser transferidos a *dispositivos de almacenamiento secundario*, que son dispositivos periféricos que actúan como medio de soporte permanente. La memoria caché es una memoria intermedia, de acceso aleatorio muy rápido entre la UCP y la memoria principal, que almacena los datos extraídos más frecuentemente de la memoria principal.

* CPU, Central Processing Unit.

- La UCP es la encargada de la ejecución de los programas almacenados en memoria. La UCP consta de Unidad de Control (UC), que se encarga de ejecutar las instrucciones, y la Unidad Aritmético Lógica (UAL), que efectúa las operaciones.

El *bus del sistema* es una ruta eléctrica de múltiples líneas —clasificables como líneas de direcciones, datos y control— que conecta la UCP, la memoria y los dispositivos de entrada/salida. Los *dispositivos de entrada/salida* son heterogéneos y de características muy variadas; por esta razón cada dispositivo o grupo de ellos cuenta con *controladores* específicos que admiten órdenes e instrucciones muy abstractas que les puede enviar la UCP y se encargan de llevarlas a cabo generando microórdenes para gobernar a los periféricos y liberando a la UCP de estas tareas; en síntesis, un *controlador* es un programa que enlaza un dispositivo de la computadora y el sistema operativo que controla la misma. Los dispositivos de entrada/salida pueden producir *interrupciones*, que son sucesos que se producen inesperadamente pero a los que generalmente se debe atender inmediatamente, por lo que la UCP abandona momentáneamente las tareas que estaba ejecutando para atender a la interrupción.

En cuanto a las operaciones que debe realizar el hardware, éstas son especificadas por una lista de instrucciones, llamadas *programas*, o *software*. El software se divide en dos grandes grupos: software del sistema y software de aplicaciones. El *software del sistema* es el conjunto de programas indispensables para que la máquina funcione, se denominan también programas del sistema, mientras que los programas que realizan tareas concretas, nóminas, contabilidad, análisis estadístico, etc. se denominan programas de aplicación o *software de aplicaciones*.

1.2. LENGUAJES DE PROGRAMACIÓN

Un programa se escribe en un lenguaje de programación y los lenguajes utilizados se pueden clasificar en:

- Lenguajes máquina.
- Lenguajes de bajo nivel (ensamblador).
- Lenguajes de alto nivel.

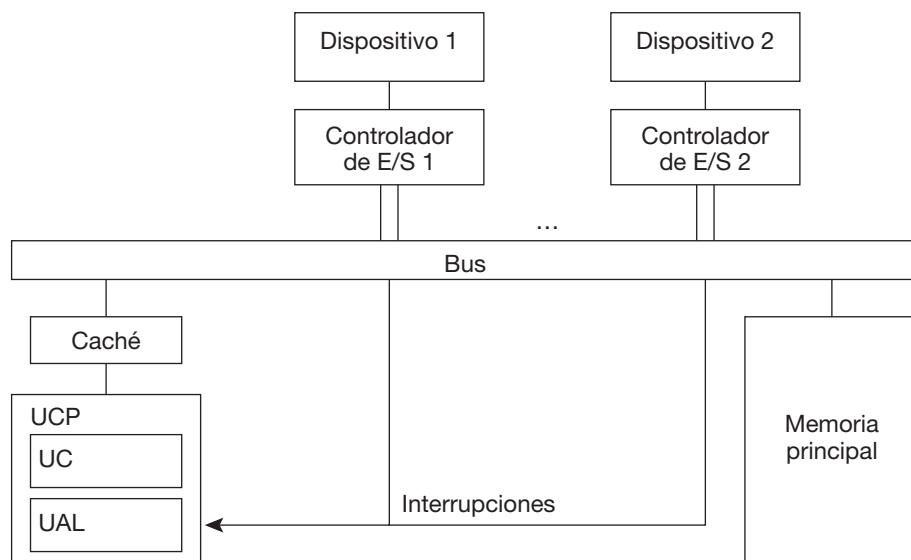


Figura 1.1. Organización física de una computadora.

Los *lenguajes máquina* proporcionan instrucciones específicas para un determinado tipo de *hardware* y son directamente inteligibles por la máquina.

El *lenguaje ensamblador* se caracteriza porque sus instrucciones son mucho más sencillas de recordar, aunque dependen del tipo de computadora y necesitan ser traducidas a lenguaje máquina por un programa al que también se denomina ensamblador.

Los *lenguajes de alto nivel* proporcionan sentencias muy fáciles de recordar, que no dependen del tipo de computadora y han de traducirse a lenguaje máquina por unos programas denominados compiladores o intérpretes. Los programas escritos en un lenguaje de alto nivel se llaman programas fuente y el programa traducido programa objeto o código objeto. El código objeto queda ligado al hardware y al sistema operativo. Casos especiales en cuanto a esto son *Java* y los lenguajes soportados por *.NET Framework*. En Java el archivo que se genera al compilar se llama *bytecode* y puede ejecutarse en cualquier plataforma donde esté instalado un determinado software denominado máquina virtual (*Java Virtual Machine*). La máquina virtual interpreta el *bytecode* para la plataforma, hardware y sistema operativo, en el que se está ejecutando el programa. Los lenguajes soportados por *.NET Framework*, como *C#*, se comportan de forma similar a Java y compilan por defecto a un *código intermedio*, denominado *Intermediate Language* (*ILCommon Language Runtime*, CLR).

1.3. RESOLUCIÓN DE PROBLEMAS

Dos fases pueden ser identificadas en el proceso de creación de un programa:

- Fase de resolución del problema.
- Fase de implementación (realización) en un lenguaje de programación.

La fase de resolución del problema implica la perfecta comprensión del problema, el diseño de una solución conceptual y la especificación del método de resolución detallando las acciones a realizar mediante un algoritmo.

1.3.1. Fase de resolución del problema

Esta fase incluye, a su vez, el análisis del problema así como el diseño y posterior verificación del algoritmo.

1.3.1.1. Análisis del problema

El primer paso para encontrar la solución a un problema es el análisis del mismo. Se debe examinar cuidadosamente el problema a fin de obtener una idea clara sobre lo que se solicita y determinar los datos necesarios para conseguirlo.

1.3.1.2. Diseño del algoritmo

La palabra algoritmo deriva del nombre del famoso matemático y astrónomo árabe **Al-Khôwarizmi** (siglo IX) que escribió un conocido tratado sobre la manipulación de números y ecuaciones titulado *Kitab al-jabr w'al-mugabala*.

Un algoritmo puede ser definido como la secuencia ordenada de pasos, sin ambigüedades, que conducen a la solución de un problema dado y puede ser expresado en lenguaje natural, por ejemplo el castellano.

Todo algoritmo debe ser:

- **Preciso.** Indicando el orden de realización de cada uno de los pasos.
- **Definido.** Si se sigue el algoritmo varias veces proporcionándole los mismos datos, se deben obtener siempre los mismos resultados.

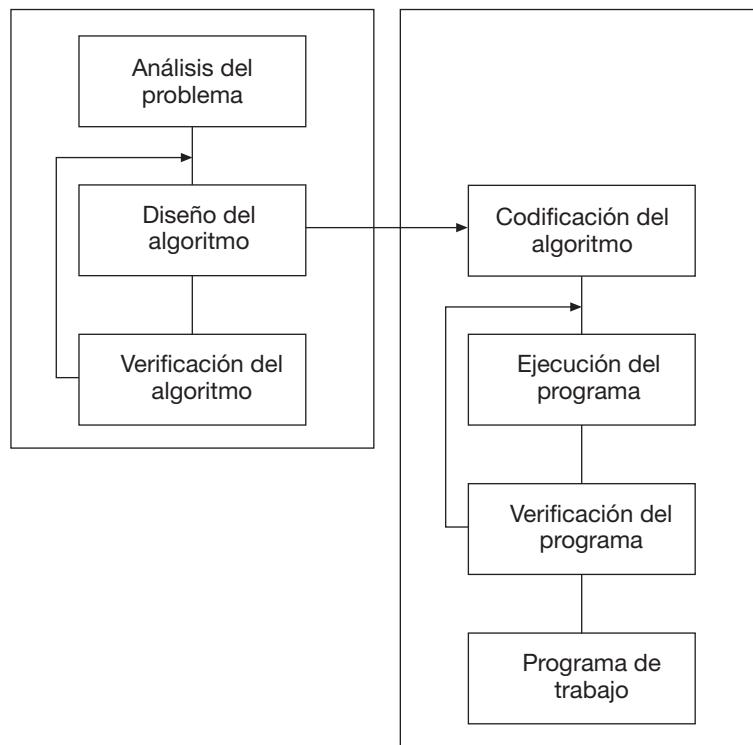


Figura 1.2. Resolución de problemas por computadora.

- **Finito.** Al seguir el algoritmo, éste debe terminar en algún momento, es decir tener un número finito de pasos.

Para diseñar un algoritmo se debe comenzar por identificar las tareas más importantes para resolver el problema y disponerlas en el orden en el que han de ser ejecutadas. Los pasos en esta primera descripción de actividades deberán ser refinados, añadiendo más detalles a los mismos e incluso, algunos de ellos, pueden requerir un refinamiento adicional antes de que podamos obtener un algoritmo claro, preciso y completo. Este método de diseño de los algoritmos en etapas, yendo de los conceptos generales a los de detalle a través de refinamientos sucesivos, se conoce como método descendente (top-down).

En un algoritmo se deben de considerar tres partes:

- **Entrada.** Información dada al algoritmo.
- **Proceso.** Operaciones o cálculos necesarios para encontrar la solución del problema.
- **Salida.** Respuestas dadas por el algoritmo o resultados finales de los cálculos.



Figura 1.3. Análisis del problema.

Como ejemplo imagine que desea desarrollar un algoritmo que calcule la superficie de un rectángulo proporcionándole su base y altura. Lo primero que deberá hacer es plantearse y contestar a las siguientes preguntas:

Especificaciones de entrada

- ¿Qué datos son de entrada?
- ¿Cuántos datos se introducirán?
- ¿Cuáles son datos de entrada válidos?

Especificaciones de salida

- ¿Cuáles son los datos de salida?
- ¿Cuántos datos de salida se producirán?
- ¿Qué precisión tendrán los resultados?
- ¿Se debe imprimir una cabecera?

El algoritmo en el primer diseño se podrá representar con los siguientes pasos:

- Paso 1. Entrada desde periférico de entrada, por ejemplo teclado, de base y altura.
- Paso 2. Cálculo de la superficie, multiplicando la base por la altura.
- Paso 3. Salida por pantalla de base, altura y superficie.

El lenguaje algorítmico debe ser independiente de cualquier lenguaje de programación particular, pero fácilmente traducible a cada uno de ellos. Alcanzar estos objetivos conducirá al empleo de métodos normalizados para la representación de algoritmos, tales como los diagramas de flujo, diagrama Nassi-Schneiderman o pseudocódigo, comentados más adelante.

1.3.1.3. Verificación de algoritmos

Una vez que se ha terminado de escribir un algoritmo es necesario comprobar que realiza las tareas para las que se ha diseñado y produce el resultado correcto y esperado.

El modo más normal de comprobar un algoritmo es mediante su ejecución manual, usando datos significativos que abarquen todo el posible rango de valores y anotando en una hoja de papel las modificaciones que se producen en las diferentes fases hasta la obtención de los resultados. Este proceso se conoce como prueba del algoritmo.

1.3.2. Fase de implementación

Una vez que el algoritmo está diseñado, representado mediante un método normalizado (diagrama de flujo, diagrama N-S o pseudocódigo), y verificado se debe pasar a la fase de codificación, traducción del algoritmo a un determinado lenguaje de programación, que deberá ser completada con la ejecución y comprobación del programa en la computadora.

1.4. EJERCICIOS RESUELTOS

Desarrolle los algoritmos que resuelvan los siguientes problemas:

1.1. Ir al cine.

Análisis del problema

DATOS DE SALIDA: Ver la película

DATOS DE ENTRADA: Nombre de la película, dirección de la sala, hora de proyección

DATOS AUXILIARES: Entrada, número de asiento

Para solucionar el problema, se debe seleccionar una película de la cartelera del periódico, ir a la sala y comprar la entrada para, finalmente, poder ver la película.

Diseño del algoritmo

```

inicio
    //seleccionar la película
    tomar el periódico
    mientras no llegemos a la cartelera
        pasar la hoja
    mientras no se acabe la cartelera
        leer película
        si nos gusta, recordarla
    elegir una de las películas seleccionadas

    leer la dirección de la sala y la hora de proyección

    //comprar la entrada
    trasladarse a la sala
    si no hay entradas, ir a fin
    si hay cola
        ponerse el último
        mientras no lleguemos a la taquilla
            avanzar
            si no hay entradas, ir a fin
    comprar la entrada

    //ver la película
    leer el número de asiento de la entrada
    buscar el asiento
    sentarse
    ver la película
fin

```

1.2. Comprar una entrada para ir a los toros.

Análisis del problema

DATOS DE SALIDA: La entrada

DATOS DE ENTRADA: Tipo de entrada (sol, sombra, tendido, andanada, etc.)

DATOS AUXILIARES: Disponibilidad de la entrada

Hay que ir a la taquilla y elegir la entrada deseada. Si hay entradas se compra (en taquilla o a los reventas). Si no la hay, se puede seleccionar otro tipo de entrada o desistir, repitiendo esta acción hasta que se ha conseguido la entrada o el posible comprador ha desistido.

Diseño del algoritmo

```

inicio
    ir a la taquilla
    si no hay entradas en taquilla
        si nos interesa comprarla en la reventa
            ir a comprar la entrada
        si no ir a fin
    //comprar la entrada

    seleccionar sol o sombra
    seleccionar barrera, tendido, andanada o palco
    seleccionar número de asiento
    solicitar la entrada
    si la tienen disponible
        adquirir la entrada
    si no
        si queremos otro tipo de entrada
            ir a comprar la entrada
fin

```

1.3. Poner la mesa para la comida.

Análisis del problema

DATOS DE SALIDA: La mesa puesta

DATOS DE ENTRADA: La vajilla, los vasos, los cubiertos, la servilletas, el número de comensales

DATOS AUXILIARES: Número de platos, vasos, cubiertos o servilletas que llevamos puestos

Para poner la mesa, después de poner el mantel, se toman las servilletas hasta que su número coincide con el de comensales y se colocan. La operación se repetirá con los vasos, platos y cubiertos.

Diseño del algoritmo

```

inicio
    poner el mantel
    repetir
        tomar una servilleta
    hasta que el número de servilletas es igual al de comensales

    repetir
        tomar un vaso
    hasta que el número de vasos es igual al de comensales

    repetir
        tomar un juego de platos
    hasta que el número de juegos es igual al de comensales

```

```

repetir
    tomar un juego de cubiertos
    hasta que el número de juegos es igual al de comensales

fin

```

1.4. Hacer una taza de té.

Análisis del problema

DATOS DE SALIDA: Taza de té
 DATOS DE ENTRADA: Bolsa de té, agua
 DATOS AUXILIARES: Pitido de la tetera, aspecto de la infusión

Después de echar agua en la tetera, se pone al fuego y se espera a que el agua hierva (hasta que suena el pitido de la tetera). Introducimos el té y se deja un tiempo hasta que está hecho.

Diseño del algoritmo

```

inicio
    tomar la tetera
    llenarla de agua
    encender el fuego
    poner la tetera en el fuego
    mientras no hierva el agua
        esperar
    tomar la bolsa de té
    introducirla en la tetera
    mientras no esté hecho el té
        esperar
    echar el té en la taza
fin

```

1.5. Fregar los platos de la comida.

Análisis del problema

DATOS DE SALIDA: Platos limpios
 DATOS DE ENTRADA: Platos sucios
 DATOS AUXILIARES: Número de platos que quedan

Diseño del algoritmo

```

inicio
    abrir el grifo
    tomar el estropajo
    echarle jabón
    mientras queden platos
        lavar el plato
        aclararlo
        dejarlo en el escurridor
    mientras queden platos en el escurridor
        secar plato
fin

```

1.6. Reparar un pinchazo de una bicicleta.

Análisis del problema

DATOS DE SALIDA: La rueda reparada
 DATOS DE ENTRADA: La rueda pinchada, los parches, el pegamento
 DATOS AUXILIARES: Las burbujas que salen donde está el pinchazo

Después de desmontar la rueda y la cubierta e inflar la cámara, se introduce la cámara por secciones en un cubo de agua. Las burbujas de aire indicarán donde está el pinchazo. Una vez descubierto el pinchazo se aplica el pegamento y ponemos el parche. Finalmente se montan la cámara, la cubierta y la rueda.

Diseño del algoritmo

```

  inicio
    desmontar la rueda
    desmontar la cubierta
    sacar la cámara
    inflar la cámara
    meter una sección de la cámara en un cubo de agua
    mientras no salgan burbujas
      meter una sección de la cámara en un cubo de agua
    marcar el pinchazo
    echar pegamento
    mientras no esté seco
      esperar
    poner el parche
    mientras no esté fijo
      apretar
    montar la cámara
    montar la cubierta
    montar la rueda
  fin

```

1.7. Pagar una multa de tráfico.

Análisis del problema

DATOS DE SALIDA: El recibo de haber pagado la multa
 DATOS DE ENTRADA: Datos de la multa
 DATOS AUXILIARES: Impreso de ingreso en el banco, dinero

Debe elegir cómo desea pagar la multa, si en metálico en la Delegación de Tráfico o por medio de una transferencia bancaria. Si elige el primer caso, tiene que ir a la Delegación Provincial de Tráfico, desplazarse a la ventanilla de pago de multas, pagarla en efectivo y recoger el resguardo.

Si desea pagarla por transferencia bancaria, hay que ir al banco, llenar el impreso apropiado, dirigirse a la ventanilla, entregar el impreso y recoger el resguardo.

Diseño del algoritmo

```

  inicio
    si pagamos en efectivo
      ir a la Delegación de Tráfico
      ir a la ventanilla de pago de multas

```

```

    si hay cola
        ponerse el último
        mientras no lleguemos a la ventanilla
            esperar
        entregar la multa
        entregar el dinero
        recoger el recibo
    si no
        //pagamos por trasferencia bancaria
        ir al banco
        rellenar el impreso
        si hay cola
            ponerse el último
            mientras no lleguemos a la ventanilla
                esperar
            entregar el impreso
            recoger el resguardo
    fin

```

- 1.8.** *Hacer una llamada telefónica. Considerar los casos: a) llamada manual con operador; b) llamada automática; c) llamada a cobro revertido.*

Análisis del problema

Para decidir el tipo de llamada que se efectuará, primero se debe considerar si se dispone de efectivo o no para realizar la llamada a cobro revertido. Si hay efectivo se debe ver si el lugar a donde vamos a llamar está conectado a la red automática o no.

Para una llamada con operadora hay que llamar a la centralita y solicitar la llamada, esperando hasta que se establezca la comunicación. Para una llamada automática se leen los prefijos del país y provincia si fuera necesario, y se realiza la llamada, esperando hasta que cojan el teléfono. Para llamar a cobro revertido se debe llamar a centralita, solicitar la llamada y esperar a que el abonado del teléfono al que se llama dé su autorización, con lo que se establecerá la comunicación.

Como datos de entrada tendríamos las variables que nos van a condicionar el tipo de llamada, el número de teléfono y, en caso de llamada automática, los prefijos si los hubiera. Como dato auxiliar se podría considerar en los casos a y c el contacto con la centralita.

Diseño del algoritmo

```

inicio
    si tenemos dinero
        si podemos hacer una llamada automática
            leer el prefijo de país y localidad
            marcar el número
        si no
            //llamada manual
            llamar a la centralita
            solicitar la comunicación
            mientras no contesten
                esperar
            establecer comunicación
        si no
            //realizar una llamada a cobro revertido
            llamar a la centralita

```

```

    solicitar la llamada
    esperar hasta tener la autorización
    establecer comunicación
fin

```

1.9. Cambiar el cristal roto de la ventana.

Análisis del problema

DATOS DE SALIDA: La ventana con el cristal nuevo
 DATOS DE ENTRADA: El cristal nuevo
 DATOS AUXILIARES: El número de clavos de cada una de las molduras

Diseño del algoritmo

```

inicio
    repetir cuatro veces
        quitar un clavo
        mientras el número de clavos quitados no sea igual al total de clavos
            quitar un clavo
        sacar la moldura
        sacar el cristal roto
        poner el cristal nuevo
    repetir cuatro veces
        poner la moldura
        poner un clavo
        mientras el número de clavos puestos no sea igual al total de clavos
            poner un clavo
        poner el cristal nuevo
fin

```

1.10. Diseñar un algoritmo que imprima y sume la serie de números 3,6,9,12,...,99.

Análisis del problema

Se trata de idear un método con el que obtengamos dicha serie, que no es más que incrementar una variable de tres en tres. Para ello se hará un bucle que se acabe cuando el número sea mayor que 99 (o cuando se realice 33 veces). Dentro de ese bucle se incrementa la variable, se imprime y se acumula su valor en otra variable llamada suma, que será el dato de salida.

No tendremos por tanto ninguna variable de entrada, y sí dos de salida, la que nos va sacando los números de tres en tres (núm) y suma.

Diseño del algoritmo

```

inicio
    asignar a suma un 0
    asignar a núm un 3
    mientras núm <= 99
        escribir núm
        incrementar suma en núm
        incrementar núm en 3
        escribir suma
fin

```

- 1.11.** Diseñar un algoritmo para calcular la velocidad (en metros/segundo) de los corredores de una carrera de 1500 metros. La entrada serán parejas de números (minutos, segundos) que darán el tiempo de cada corredor. Por cada corredor se imprimirá el tiempo en minutos y segundos, así como la velocidad media. El bucle se ejecutará hasta que demos una entrada de 0,0 que será la marca de fin de entrada de datos.

Análisis del problema

DATOS DE SALIDA: v (velocidad media)
 DATOS DE ENTRADA: mm, ss (minutos y segundos)
 DATOS AUXILIARES: distancia (distancia recorrida, que en el ejemplo es de 1500 metros) y tiempo (los minutos y los segundos que ha tardado en recorrerla)

Se debe efectuar un bucle que se ejecute hasta que mm sea 0 y ss sea 0. Dentro del bucle se calcula el tiempo en segundos con la fórmula tiempo = ss + mm * 60. La velocidad se hallará con la fórmula velocidad = = distancia/tiempo.

Diseño del algoritmo

```

  inicio
    asignar a distancia 1500
    leer desde teclado mm,ss
    mientras mm y ss sea alguno de ellos distinto de 0
      asignar a tiempo el resultado de sumar a ss el producto de mm por 60
      asignar a v el cociente entre distancia y tiempo
      escribir mm, ss, v
      leer mm, ss
  fin

```

- 1.12.** Escribir un algoritmo que calcule la superficie de un triángulo en función de la base y la altura.

Análisis del problema

DATOS DE SALIDA: s (superficie)
 DATOS DE ENTRADA: b (base), a (altura)

Para calcular la superficie se aplica la fórmula $S = \text{base} * \text{altura}/2$.

Diseño del algoritmo

```

  inicio
    leer b, a
    asignar a s el cociente de dividir entre dos el producto de b por a
    escribir s
  fin

```

- 1.13.** Escribir un algoritmo que calcule la compra de un cliente en un supermercado.

Análisis del problema

DATOS DE SALIDA: Total
 DATOS DE ENTRADA: Precio, unidades
 DATOS INTERMEDIOS: Importe

Por cada producto de la compra, se leerá el precio del producto y se calculará el importe mediante la expresión precio * unidades. Una vez calculado el importe se acumulará al total que, previamente, se habrá inicializado a 0. El proceso continuará mientras el cliente todavía tenga productos.

Diseño del algoritmo

```
inicio
    asignar 0 a total
    mientras haya productos
        leer precio y unidades
        asignar al importe la expresión precio * unidades
        acumular el importe al total
    escribir total
fin
```


2

LA RESOLUCIÓN DE PROBLEMAS CON COMPUTADORAS

9` dfcZYgcf 'B L_ 'U g'K]fh\z]bj Ybhf 'XY' DgWUz A Gx 'U & mC VYfcbz h]hi 'Gf bc 'XY' gi g'a zg 'ZU a cgcg ']Vfcg '5 'cf]ha cg Z'9ghfi Wm fUg XYXUhg '1 'Dfc[fUa Ug g][b]Z]WbXcbc g'ei YgEc 'gY di YXY''Y! [Uf 'Uf YU]nUf 'i b'Vi Yb'dfc[fUa UWb 'Y' XlgY' c'XY i b'U[cf]ha c'mY' i gc 'XY' Wff YMUg 'Yghfi Wm fUg XY XUhg "9ghY'Wd]hi 'c 'hfUhU XY' 'cg' XUhg m'Uf Ydf YgYbhUWf 'XY U[cf]ha cg'a YX]UbhY' \Yff Ua]Yb! hu'g'XY' dfc[fUa UWf fX]U fUa Ug'XY' Zi '^cz X]U fUa Ug'XY' B !G 'midgYi XcWfX][cl"

2.1. DATOS

Dato es la expresión general que describe los objetos con los cuales opera el algoritmo. El tipo de un dato determina su forma de almacenamiento en memoria y las operaciones que van a poder ser efectuadas con él. En principio hay que tener en cuenta que, prácticamente en cualquier lenguaje y por tanto en cualquier algoritmo, se podrán usar datos de los siguientes tipos:

- **entero.** Subconjunto finito de los números enteros, cuyo rango o tamaño dependerá del lenguaje en el que posteriormente codifiquemos el algoritmo y de la computadora utilizada.
- **real.** Subconjunto de los números reales limitado no sólo en cuanto al tamaño, sino también en cuanto a la precisión.
- **lógico.** Conjunto formado por los valores verdad y falso.
- **carácter.** Conjunto finito y ordenado de los caracteres que la computadora reconoce.
- **cadena.** Los datos (objetos) de este tipo contendrán una serie finita de caracteres, que podrán ser directamente traídos o enviados a/desde la consola.

Es decir, los tipos **entero**, **real**, **carácter**, **cadena** y **lógico** son *tipos predefinidos* en la mayoría de los lenguajes de programación. En los algoritmos para indicar que un dato es de uno de estos tipos se declarará utilizando directamente el identificador o nombre del tipo.

Además los lenguajes permiten al programador definir sus propios tipos de datos. Al definir nuevos tipos de datos hay que considerar tanto sus posibles valores como las operaciones que van a poder ser efectuadas con los datos del mismo. Lo usual es que se definan nuevos tipos de datos agrupando valores de otros tipos definidos previamente o de tipos estándar. Por este motivo se dice que están estructurados. Si todos los valores agrupados fueran del mismo tipo se consideraría a éste como el tipo base del nuevo tipo estructurado.

Los datos pueden venir expresados como constantes, variables, expresiones o funciones.

2.1.1. Constantes

Son datos cuyo valor no cambia durante todo el desarrollo del algoritmo. Las constantes podrán ser literales o con nombres, también denominadas simbólicas.

Las constantes simbólicas o con nombre se identifican por su nombre y el valor asignado. Una constante literal es un valor de cualquier tipo que se utiliza como tal. Tendremos pues constantes:

- **Numéricas enteras.** En el rango de los enteros. Compuestas por el signo (+,-) seguido de una serie de dígitos (0..9).
- **Numéricas reales.** Compuestas por el signo (+,-) seguido de una serie de dígitos (0..9) y un punto decimal (.) o compuestas por el signo (+,-), una serie de dígitos (0..9) y un punto decimal que constituyen la mantisa, la letra E antes del exponente, el signo (+,-) y otra serie de dígitos (0..9).
- **Lógicas.** Sólo existen dos constantes lógicas, **verdad** y **falso**.
- **Carácter.** Cualquier carácter del juego de caracteres utilizado colocado entre comillas simples o apóstrofes. Los caracteres que reconocen las computadoras son dígitos, caracteres alfabéticos, tanto mayúsculas como minúsculas, y caracteres especiales.
- **Cadena.** Serie de caracteres válidos encerrados entre comillas simples.

2.1.2. Variables

Una variable es un objeto cuyo valor puede cambiar durante el desarrollo del algoritmo. Se identifica por su nombre y por su tipo, que podrá ser cualquiera, y es el que determina el conjunto de valores que podrá tomar la variable. En los algoritmos se deben declarar las variables que se van a usar, especificando su tipo. Según la forma para la representación del algoritmo elegida la declaración se hará con una simple tabla de variables o de una forma más rígida, especificando:

```
var
  <tipo_de_dato> : <lista_identificadores_de_variable>
```

donde

<tipo_de_dato> representa un tipo de dato

y

<lista_identificadores_de_variable> deberá ser sustituido por una lista con los nombres de las variables de dicho tipo separados por comas

Cuando se traduce el algoritmo a un lenguaje de programación y se ejecuta el programa resultante, la declaración de cada una de las variables originará que se reserve un determinado espacio en memoria etiquetado con el correspondiente identificador.

La asignación de valor a una variable se podrá hacer en modo interactivo mediante una instrucción de lectura o bien de forma interna a través del operador de asignación.

2.1.3. Expresiones

Una **expresión** es una combinación de operadores y operandos. Los operandos podrán ser constantes, variables u otras expresiones y los operadores de cadena, aritméticos, relacionales o lógicos. Las expresiones se clasifican, según el resultado que producen, en:

- **Numéricas.** Los operandos que intervienen en ellas son numéricos, el resultado es también de tipo numérico y se construyen mediante los operadores aritméticos. Se pueden considerar análogas a las fórmulas matemáticas.

Debido a que son los que se encuentran en la mayor parte de los lenguajes de programación, los algoritmos utilizarán los siguientes operadores aritméticos: menos unario (-), multiplicación (*), división real (/), exponentiación (**), adición (+), resta (-), módulo de la división entera (**mod**) y cociente de la división entera (**div**). Tenga en cuenta que la división real siempre dará un resultado real y que los operadores **mod** y **div** sólo operan con enteros y el resultado es entero.

- **Alfanuméricas.** Los operandos son de tipo alfanumérico y producen resultados también de dicho tipo. Se construyen mediante el operador de concatenación, representado por el operador **ampersand** (&) o con el mismo símbolo utilizado en las expresiones aritméticas para la suma.
- **Booleanas.** Su resultado podrá ser **verdad** o **falso**. Se construyen mediante los operadores relacionales y lógicos. Los operadores de relación son: igual (=), distinto (<>), menor que (<), mayor que (>), mayor o igual (>=), menor o igual (<=). Actúan sobre operandos del mismo tipo y siempre devuelven un resultado de tipo lógico. Los operadores lógicos básicos son: negación lógica (**no**), multiplicación lógica (**y**), suma lógica (**o**). Actúan sobre operandos de tipo lógico y devuelven resultados del mismo tipo, determinados por las tablas de verdad correspondientes a cada uno de ellos.

a	b	no a	a y b	a o b
verdad	verdad	falso	verdad	verdad
verdad	falso	falso	falso	verdad
falso	verdad	verdad	falso	verdad
falso	falso	verdad	falso	falso

El orden de prioridad general adoptado, no común a todos los lenguajes, es el siguiente:

**	Exponentiación
no , -	Operadores unarios
*, /, div , mod , y	Operadores multiplicativos
+, -, o	Operadores aditivos
=, <>, >, <, >=, <=	Operadores de relación

La evaluación de operadores con la misma prioridad se realizará siempre de izquierda a derecha. Si una expresión contiene subexpresiones encerradas entre paréntesis, dichas subexpresiones se evaluarán primero.

2.1.4. Funciones

En los lenguajes de programación existen ciertas funciones predefinidas o internas que aceptan unos argumentos y producen un valor denominado resultado. Como funciones numéricas, se usarán:

Función	Descripción	Tipo de argumento	Resultado
abs (x)	valor absoluto de x	entero o real	igual que el argumento
arctan (x)	arcotangente de x	entero o real	real
cos (x)	coseno de x	entero o real	real
cuadrado (x)	cuadrado de x	entero o real	igual que el argumento
ent (x)	entero de x	real	entero
exp (x)	e elevado a x	entero o real	real

(continúa)

(continuación)

Función	Descripción	Tipo de argumento	Resultado
<code>ln(x)</code>	logaritmo neperiano de x	entero o real	real
<code>log10(x)</code>	logaritmo base 10 de x	entero o real	real
<code>raíz2(x)</code>	ráiz cuadrada de x	entero de x	real
<code>redondeo(x)</code>	redondea x al entero más proximo	real	entero
<code>sen(x)</code>	seno de x	entero o real	real
<code>trunc(x)</code>	parte entera de x	real	entero

Las funciones se utilizarán escribiendo su nombre, seguido de los argumentos adecuados encerrados entre paréntesis, en una expresión.

2.2. REPRESENTACIÓN DE ALGORITMOS

Un algoritmo puede ser escrito en castellano narrativo, pero esta descripción suele ser demasiado propicia y, además, ambigua. Para representar un algoritmo se debe utilizar algún método que permita independizar dicho algoritmo de los lenguajes de programación y, al mismo tiempo, conseguir que sea fácilmente codificable.

Los métodos más usuales para la representación de algoritmos son:

- A. Diagrama de flujo.
- B. Diagrama N-S (Nassi-Schneiderman).
- C. *Pseudocódigo*.

2.3. DIAGRAMA DE FLUJO

Los diagramas de flujo se utilizan tanto para la representación gráfica de las operaciones ejecutadas sobre los datos a través de todas las partes de un sistema de procesamiento de información, diagrama de flujo del sistema, como para la representación de la secuencia de pasos necesarios para describir un procedimiento particular, diagrama de flujo de detalle.

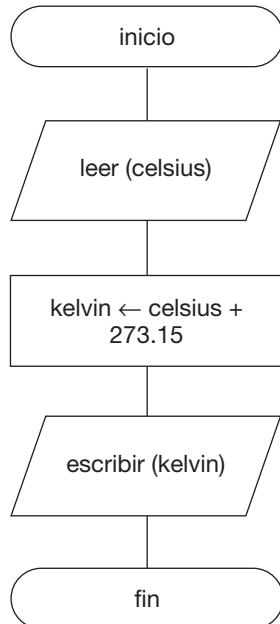
En la actualidad se siguen usando los diagramas de flujo del sistema, pero ha decaído el uso de los diagramas de flujo de detalle al aparecer otros métodos de diseño estructurado más eficaces para la representación y actualización de los algoritmos.

Los diagramas de flujo de detalle son, no obstante, uno de nuestros objetivos prioritarios y, a partir de ahora, los denominaremos simplemente diagramas de flujo.

El diagrama de flujo utiliza unos símbolos normalizados, con los pasos del algoritmo escritos en el símbolo adecuado y los símbolos unidos por flechas, denominadas líneas de flujo, que indican el orden en que los pasos deben ser ejecutados. Los símbolos principales son:

Símbolo	Función
	Inicio y fin del algoritmo
	Proceso
	Entrada/Salida
	Decisión
	Comentario

Resulta necesario indicar dentro de los símbolos la operación específica concebida por el programador. Como ejemplo veamos un diagrama de flujo básico, que representa la secuencia de pasos necesaria para que un programa lea una temperatura en grados Centígrados y calcule y escriba su valor en grados Kelvin.



2.4. DIAGRAMA NASSI-SCHNEIDERMAN

Los diagramas Nassi-Schneiderman, denominados así por sus inventores, (o también N-S, o de Chapin) son una herramienta de programación que favorece la programación estructurada y reúne características gráficas propias de diagramas de flujo y lingüísticas propias de los pseudocódigos. Constan de una serie de cajas contiguas que se leerán siempre de arriba-abajo y se documentarán de la forma adecuada.

En los diagramas N-S las tres estructuras básicas de la programación estructurada, secuenciales, selectivas y repetitivas, encuentran su representación propia. La programación estructurada será tratada en capítulos posteriores.

Símbolo	Tipo de estructura
Sentencia 1	Secuencial
Sentencia 2	
Sentencia n	
[]	Repetitiva de 0 a n veces
[]	Repetitiva de 1 a n veces
[]	Repetitiva n veces
condición si no	Selectiva

El algoritmo que lee una temperatura en grados Celsius y calcula y escribe su valor en grados Kelvin se puede representar mediante el siguiente diagrama N-S:

inicio
leer (celsius)
kelvin ← celsius + 273.15
escribir (kelvin)
fin

2.5. PSEUDOCÓDIGO

El pseudocódigo es un lenguaje de especificación de algoritmos que utiliza palabras reservadas y exige la indentación, o sea sangría en el margen izquierdo de algunas líneas. El pseudocódigo se concibió para superar las dos principales desventajas del diagrama de flujo: lento de crear y difícil de modificar sin un nuevo redibujo. Es una herramienta muy buena para el seguimiento de la lógica de un algoritmo y para transformar con facilidad los algoritmos a programas, escritos en un lenguaje de programación específico. En este libro se escribirán casi todos los algoritmos en pseudocódigo.

En nuestro pseudocódigo utilizaremos palabras reservadas en español. Así, nuestros algoritmos comenzarán con la palabra reservada **inicio** y terminarán con **fin**, constando de múltiples líneas que se sangran o indentan para mejorar la legibilidad. La estructura básica de un algoritmo escrito en pseudocódigo es:

```

algoritmo <identificador_algoritmo>
    // declaraciones, sentencias no ejecutables
inicio
    // acciones, sentencias ejecutables tanto simples como estructuradas
fin

```

Los espacios en blanco entre los elementos no resultan significativos, y las partes importantes se suelen separar unas de otras por líneas en blanco. Para introducir un valor o serie de valores desde el dispositivo estándar y almacenarlos en una o varias variables utilizaremos **leer(<lista_de_variables>)**¹. Con **nombre_de_variable ← <expresión>** almacenaremos en una variable el resultado de evaluar una expresión. Hay que tener en cuenta que una única constante, variable o función, constituyen una expresión. Para imprimir en el dispositivo estándar de salida una o varias expresiones emplearemos **escribir(<lista_de_expresiones>)**².

Los elementos léxicos de nuestro pseudocódigo son: *Comentarios, Palabras reservadas, Identificadores, Operadores y signos de puntuación y Literales*.

2.5.1 Comentarios

Los *comentarios* sirven para documentar el algoritmo y en ellos se escriben anotaciones generalmente sobre su funcionamiento. Cuando se coloque un comentario de una sola línea se escribirá precedido de **//**. Si el comentario es multilínea, lo pondremos entre **{ }**.

```

// Comentario de una línea
{ Comentario que ocupa más
  de una línea }

```

¹ La expresión **<lista_de_variables>** deberá ser sustituida por una lista de variables separadas por comas.

² La expresión **<lista_de_expresiones>** deberá ser sustituida por una lista de expresiones separadas por comas.

2.5.2. Palabras reservadas

Las *palabras reservadas* o *palabras clave* (*Keywords*) son palabras que tienen un significado especial, como: **inicio** y **fin**, que marcan el principio y fin del algoritmo, y las palabras que aparecen en negrita en las estructuras especificadas a continuación. En lugar de las palabras reservadas no deberán utilizarse otras similares, aunque no se distingue entre mayúsculas y minúsculas.

```

Decisión simple:      si <condición> entonces
                           <acciones1>
                           fin_si

Decisión doble:      si <condición> entonces
                           <acciones1>
                           si_no
                           <acciones2>
                           fin_si

Decisión múltiple:   según_sea <expresión_ordinal> hacer
                           <lista_de_valores_ordinales>: <acciones1>
                           .....
                           [si_no      // El corchete indica optionalidad
                           <accionesN>]
                           fin_según

Repetitivas:          mientras <condición> hacer
                           <acciones>
                           fin_mientras

                           repetir
                           <acciones>
                           hasta_que <condición>

                           desde <variable> ← <v_inicial> hasta <v_final>
                           [incremento | decremento <incremento>] hacer
                           <acciones>
                           fin_desde
```

El ejemplo ya citado que transforma grados Celsius en Kelvin, escrito en pseudocódigo quedaría de la siguiente forma:

```

inicio
  leer(celsius)
  kelvin ← celsius + 273.15
  escribir(Kelvin)
fin
```

2.5.3. Identificadores

Identificadores son los nombres que se dan a las constantes simbólicas, variables, funciones, procedimientos, u otros objetos que manipula el algoritmo. La regla para construir un identificador establece que:

- Debe resultar significativo, sugiriendo lo que representa.
- No podrá coincidir con palabras reservadas, propias del lenguaje algorítmico. Como se verá más adelante, la representación de algoritmos mediante pseudocódigo va a requerir la utilización de palabras reservadas.
- Se recomienda un máximo de 50 caracteres.
- Comenzará siempre por un carácter alfabético y los siguientes podrán ser letras, dígitos o el símbolo de subrayado.
- Podrá ser utilizado indistintamente escrito en mayúsculas o en minúsculas.

2.5.4. Operadores y signos de puntuación

Los operadores se utilizan en las expresiones e indican las operaciones a efectuar con los operandos, mientras que los signos de puntuación se emplean con el objetivo de agrupar o separar, por ejemplo . ; o [].

2.5.5. Literales

Los literales son los valores que aparecen directamente escritos en el programa y pueden ser literales lógicos, enteros, reales, de tipo carácter, de tipo cadena y el literal **nulo**.

2.6. EJERCICIOS RESUELTOS

2.1. ¿Cuál de los siguientes datos son válidos para procesar por una computadora?

- | | | |
|-------------|-------------|------------|
| a) 3.14159 | e) 2.234E2 | i) 12.5E.3 |
| b) 0.0014 | f) 12E+6 | j) .123E4 |
| c) 12345.0 | g) 1.1E-3 | k) 5A4.14 |
| d) 15.0E-04 | h) -15E-0.4 | l) A1.E04 |

Serían válidos los datos a, b, c, d, e, f, g y j. Los datos h e i no serían válidos pues el exponente no puede tener forma decimal. El k, no sería correcto pues mezcla caracteres alfabéticos y dígitos, y no se puede considerar una identificador de una variable ya que empieza por un dígito. El dato l aunque mezcla dígitos y caracteres alfabéticos, podría ser un identificador, si admitiéramos el carácter punto como carácter válido para una variable (PASCAL o BASIC lo consideran).

2.2. ¿Cuál de los siguientes identificadores son válidos?

- | | | |
|--------------|-----------------|---------------------|
| a) Renta | e) Dos Pulgadas | i) 4A2D2 |
| b) Alquiler | f) C3PO | j) 13Nombre |
| c) Constante | g) Bienvenido#5 | k) Nombre_Apellidos |
| d) Tom's | h) Elemento | l) NombreApellidos |

Se consideran correctos los identificadores a, b, c, f, h, k y l. El d no se considera correcto pues incluye el apóstrofo que no es un carácter válido en un identificador. Lo mismo ocurre con el e y el espacio en blanco, y el g con el carácter #. El i y el j no serían válidos al no comenzar por un carácter alfabético.

2.3. Escribir un algoritmo que lea un valor entero, lo doble, se multiplique por 25 y visualice el resultado.

Análisis de problema

DATOS DE SALIDA: Resultado (es el resultado de realizar las operaciones)
 DATOS DE ENTRADA: Número (el número que leemos por teclado)

Leemos el número por teclado y lo multiplicamos por 2, metiendo el contenido en la propia variable de entrada. A continuación lo multiplicamos por 25 y asignamos el resultado a la variable de salida `resultado`. También podríamos asignar a la variable de salida directamente la expresión `número*2*25`.

Diseño del algoritmo

```
algoritmo ejercicio_2_3
var
    entero : número, resultado
inicio
    leer (número)
    número ← número * 2
    resultado ← número * 25
    escribir (resultado)
fin
```

- 2.4.** Diseñar un algoritmo que lea cuatro variables y calcule e imprima su producto, su suma y su media aritmética.

Análisis del problema

DATOS DE SALIDA: Producto, suma y media
 DATOS DE ENTRADA: A,b,c,d

Después de leer los cuatro datos, asignamos a la variable `producto` la multiplicación de las cuatro variables de entrada. A la variable `suma` le asignamos su suma y a la variable `media` le asignamos el resultado de sumar las cuatro variables y dividirlas entre cuatro. Como el operador suma tiene menos prioridad que el operador división, será necesario encerrar la suma entre paréntesis. También podríamos haber dividido directamente la variable `suma` entre cuatro.

La variables `a,b,c,d`, `producto` y `suma` podrán ser enteras, pero no así la variable `media`, ya que la división produce siempre resultados de tipo real.

Diseño del algoritmo

```
algoritmo ejercicio_2_4
var
    entero: a,b,c,d,producto,suma
    real: media
inicio
    leer (a,b,c,d)
    producto ← a * b * c * d
    suma ← a + b + c + d
    media ←(a + b + c + d) / 4
    escribir (producto,suma,media)
fin
```

- 2.5.** Diseñar un programa que lea el peso de un hombre en libras y nos devuelva su peso en kilogramos y gramos (Nota: una libra equivale a 0.453592 kilogramos).

Análisis del problema

DATOS DE SALIDA: Kg, gr
 DATOS DE ENTRADA: Peso
 DATOS AUXILIARES: Libra (los kilogramos que equivalen a una libra)

El dato auxiliar libra lo vamos a considerar como una constante, pues no variará a lo largo del programa. Primero leemos el peso. Para hallar su equivalencia en kilogramos multiplicamos éste por la constante libra. Para hallar el peso en gramos multiplicamos los kilogramos entre mil. Como es posible que el dato de entrada no sea exacto, consideraremos todas las variables como reales.

Diseño del algoritmo

```

algoritmo ejercicio_2_5
const
    libra = 0.453592
var
    real: peso,kg,gr
inicio
    leer (peso)
    kg ← peso * libra
    gr ← kg * 1000
    escribir ('Peso en kilogramos: ',kg)
    escribir ('Peso en gramos: ',gr)
fin

```

2.6. Si $A = 6$, $B = 2$ y $C = 3$, encontrar los valores de las siguientes expresiones:

- | | |
|---------------------------|--------------------------------------|
| a) $A - B + C$ | d) $A * B \bmod C$ |
| b) $A * B \text{ div } C$ | e) $A + B \bmod C$ |
| c) $A \text{ div } B + C$ | f) $A \text{ div } B \text{ div } C$ |

Los resultados serían:

- | |
|--|
| a) $(6-2)+3 = 7$ |
| b) $(6*2) \text{ div } 3 = 4$ |
| c) $(6 \text{ div } 2)+3 = 6$ |
| d) $(6*2) \bmod 3 = 0$ |
| e) $6+(2 \bmod 3) = 8$ |
| f) $(6 \text{ div } 2) \text{ div } 3 = 1$ |

2.7. ¿Qué se obtiene en las variables A y B después de la ejecución de las siguientes instrucciones?

```

A ← 5
B ← A + 6
A ← A + 1
B ← A - 5

```

Las variables irán tomando los siguientes valores:

$A \leftarrow 5$, $A = 5$, B indeterminado
$B \leftarrow A + 6$, $A = 5$, $B = 5 + 6 = 11$
$A \leftarrow A + 1$, $A = 5 + 1 = 6$, $B = 11$
$B \leftarrow A - 5$, $A = 6$, $B = 6 - 5 = 1$

Los valores finales serían: $A = 6$ y $B = 1$.

2.8. ¿Qué se obtiene en las variables A , B y C después de ejecutar las siguientes instrucciones?

```

A ← 3
B ← 20
C ← A + B
B ← A + B
A ← B

```

Las variables tomarían sucesivamente los valores:

```
A ← 3      , A = 3      , B y C indeterminados
B ← 20     , A = 3      , B = 20           , C indeterminado
C ← A + B , A = 3      , B = 20           , C = 3 + 20 = 23
B ← A + B , A = 3      , B = 3 + 20 = 23 , C = 23
A ← B      , A = 23     , B = 23           , C = 23
```

Los valores de las variables serían: A = 23, B = 23 y C = 23.

2.9. ¿Qué valor toman las variables A y B tras la ejecución de las siguientes asignaciones?

```
A ← 10
B ← 5
A ← B
B ← A
```

El resultado sería el siguiente:

```
A ← 10 , A = 10 , B indeterminada
B ← 5 , A = 10 , B = 5
A ← B , A = 5 , B = 5
B ← A , A = 5 , B = 5
```

con lo que A y B tomarían el valor 5.

2.10. Escribir las siguientes expresiones en forma de expresiones algorítmicas.

$$a) \frac{M}{N} + 4 \quad b) M + \frac{N}{P - Q} \quad c) \frac{\sin x \cos y}{\tan x}$$

$$d) \frac{M + N}{P - Q} \quad e) \frac{P + \frac{N}{P}}{Q - \frac{r}{5}} \quad f) \frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

- a) M / N + 4
- b) M + N / (P - Q)
- c) (sen(X) + cos(X)) / tan(X)
- d) (M + N) / (P - Q)
- e) (M + N / P) / (Q - R / 5)
- f) ((-B) + raíz2(B**2 - 4*A*C)) / (2*A)

2.11. Se tienen tres variables A, B y C. Escribir las instrucciones necesarias para intercambiar entre sí sus valores del modo siguiente:

B toma el valor de A
 C toma el valor de B
 A toma el valor de C

Nota: sólo se debe utilizar una variable auxiliar que llamaremos AUX.

```
AUX ← A
A ← C
C ← B
B ← AUX
```

2.12. Deducir los resultados que se obtienen del siguiente algoritmo:

```
algoritmo ejercicio_2_12
var
    entero: X, Y, Z
inicio
    X ← 15
    Y ← 30
    Z ← Y - X
    escribir (X,Y)
    escribir (Z)
fin
```

Para analizar los resultados de un algoritmo, lo mejor es utilizar una tabla de seguimiento. En la tabla de seguimiento aparecen varias columnas, cada una con el nombre de una de las variables que aparecen en el algoritmo, pudiéndose incluir además una con la salida del algoritmo. Más abajo se va realizando el seguimiento del algoritmo rellenando la columna de cada variable con el valor que ésta va tomando.

X	Y	Z	Salida
15	30	15	
			15,30
			15

2.13. Encontrar el valor de la variable VALOR después de la ejecución de las siguientes operaciones:

- a) $VALOR \leftarrow 4.0 * 5$
- b) $X \leftarrow 3.0$
 $Y \leftarrow 2.0$
 $VALOR \leftarrow X ** Y - Y$
- c) $VALOR \leftarrow 5$
 $X \leftarrow 3$
 $VALOR \leftarrow VALOR * X$

Los resultados serían:

- a) $VALOR = 20.0$
- b) $VALOR = 7.0$
- c) $VALOR = 15$

Nótese que en los casos a) y b), valor tendrá un contenido de tipo real, ya que alguno de los operandos son de tipo real.

2.14. Determinar los valores de A, B, C y D después de la ejecución de las siguientes instrucciones:

```
algoritmo ejercicio_2_14
var
    entero: A, B, C, D
inicio
    A ← 1
    B ← 4
    C ← A + B
```

```

D ← A - B
A ← C + 2 * B
B ← C + B
C ← A * B
D ← B + D
A ← D + C
si C > D entonces
    C ← A - D
si_no
    C ← B - D
fin_si
fin

```

Realizaremos una tabla de seguimiento:

A	B	C	D
1	4	5	-3
13	9	117	6
123		117	

con lo que A = 123, B = 9, C = 117 y D = 6.

2.15. Escribir un algoritmo que calcule y escriba el cuadrado de 821.

Análisis del problema

DATOS DE SALIDA: Cuadr (almacenará el cuadrado de 821)

DATOS DE ENTRADA: El propio número 821

Lo único que hará este algoritmo será asignar a la variable cuadr el cuadrado de 821, es decir, 821 * 821.

Diseño del algoritmo

```

algoritmo ejercicio_2_15
var
    entero: cuadr
inicio
    cuadr ← 821 * 821
    escribir (cuadr)
fin

```

2.16. Realizar una llamada telefónica desde un teléfono público.

Análisis del problema

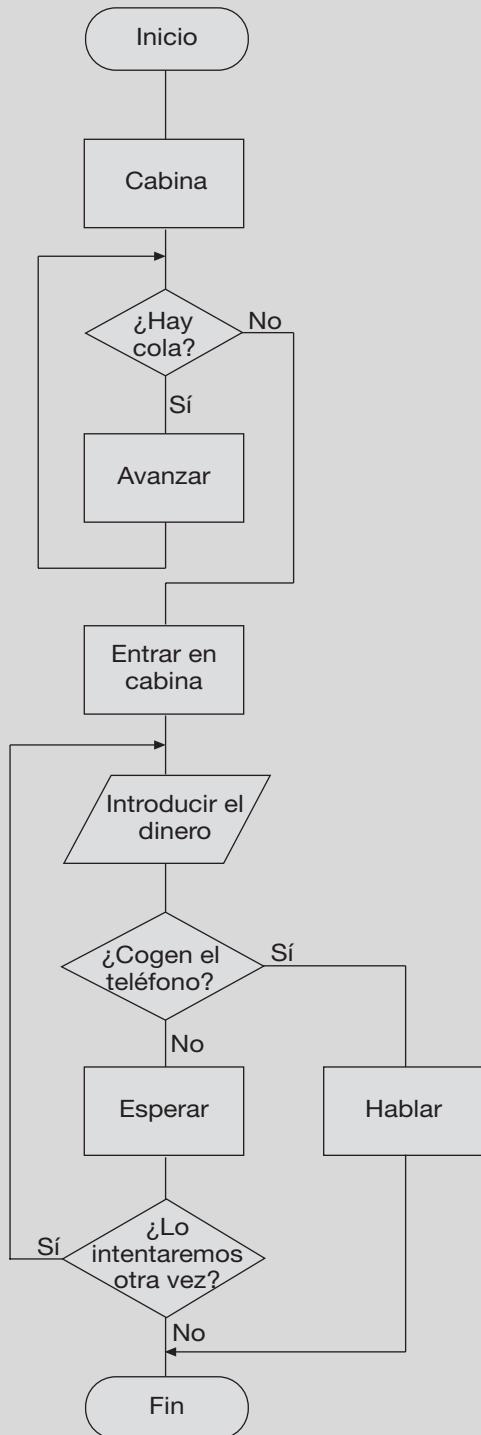
DATOS DE SALIDA: La comunicación por teléfono

DATOS DE ENTRADA: El número de teléfono, el dinero

DATOS AUXILIARES: Distintas señales de la llamada (comunicando, etc.)

Se debe ir a la cabina y esperar si hay cola. Entrar e introducir el dinero. Se marca el número y se espera la señal, si está comunicando o no contestan se repite la operación hasta que descuelgan el teléfono o decide irse.

Diseño del algoritmo



2.17. Realizar un algoritmo que calcule la suma de los enteros entre 1 y 10, es decir $1+2+3+\dots+10$.

Análisis del problema

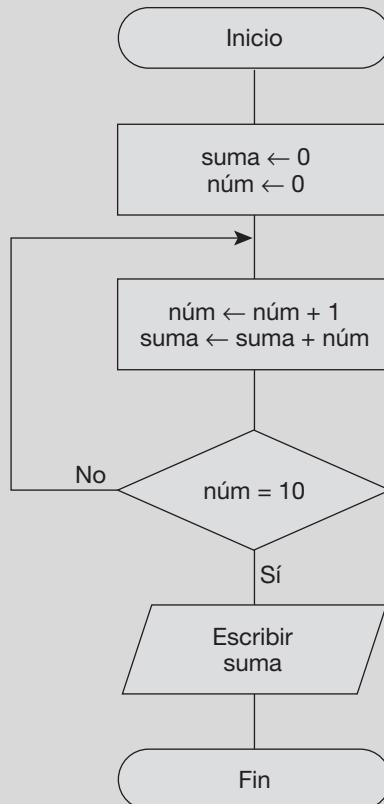
DATOS DE SALIDA: suma (contiene la suma requerida)
 DATOS AUXILIARES: num (será una variable que vaya tomando valores entre 1 y 10 y se acumulará en suma)

Hay que ejecutar un bucle que se realice 10 veces. En él se irá incrementando en 1 la variable num, y se acumulará su valor en la variable suma. Una vez salgamos del bucle se visualizará el valor de la variable suma.

Diseño del algoritmo

TABLA DE VARIABLES:

entero : suma, num



2.18. Realizar un algoritmo que calcule y visualice las potencias de 2 entre 0 y 10.

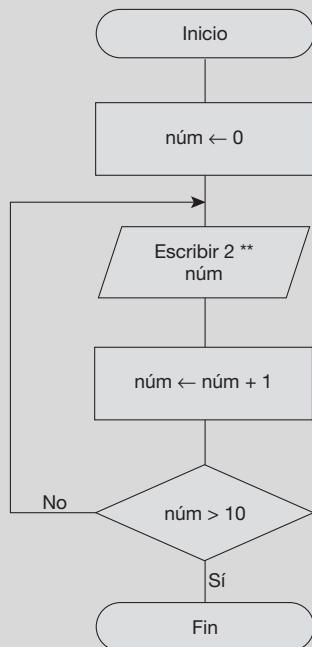
Análisis del problema

Hay que implementar un bucle que se ejecute once veces y dentro de él ir incrementando una variable que tome valores entre 0 y 10 y que se llamará num. También dentro de él se visualizará el resultado de la operación 2^{**}num .

Diseño del algoritmo

TABLA DE VARIABLES:

entero: num



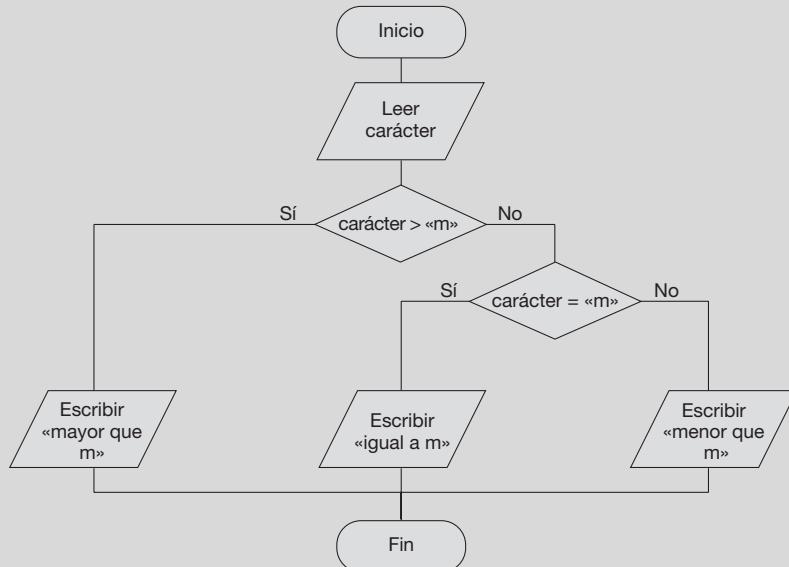
2.19. Leer un carácter y deducir si está situado antes o después de la «m» en orden alfabetico.

Análisis del problema

Como dato de salida está el mensaje que nos dice la situación del carácter con respecto a la «m». Como entrada el propio carácter que introducimos por teclado. No se necesita ninguna variable auxiliar.

Se debe leer el carácter y compararlo con la «m» para ver si es mayor, menor o igual que ésta. Recuerde que para el ordenador también un carácter puede ser mayor o menor que otro, y lo hace comparando el código de los dos elementos de la comparación.

Diseño del algoritmo



2.20. Leer dos caracteres y deducir si están en orden alfabético.**Análisis del problema**

DATOS DE SALIDA: El mensaje que nos indica si están en orden alfabético

DATOS DE ENTRADA: A y B (los caracteres que introducimos por teclado)

Se deben leer los dos caracteres y compararlos. Si A es mayor que B, no se habrán introducido en orden. En caso contrario estarán en orden o serán iguales.

Diseño del algoritmo

```
algoritmo ejercicio_2_20
var
    carácter: a, b
inicio
    leer (a,b)
    si a < b entonces
        escribir('están en orden')
    si_no
        si a = b entonces
            escribir ('son iguales')
        si_no
            escribir ('no están en orden')
        fin_si
    fin_si
fin
```

2.21. Leer un carácter y deducir si está o no comprendido entre las letras I y M ambas inclusive.**Análisis del problema**

DATOS DE SALIDA: El mensaje

DATOS DE ENTRADA: El carácter

Se lee el carácter y se comprueba si está comprendido entre ambas letras con los operadores \geq y \leq .

Diseño del algoritmo

```
algoritmo Ejercicio_2_21
var
    carácter: c
inicio
    leer (c)
    si (c  $\geq$  'I') y (carácter  $\leq$  'M') entonces
        escribir ('Está entre la I y la M')
    si_no
        escribir ('no se encuentra en ese rango')
    fin_si
fin
```

2.22. Averiguar si una palabra es un palíndromo. Un palíndromo es una palabra que se lee igual de izquierda a derecha que de derecha a izquierda, como por ejemplo «radar».

Análisis del problema

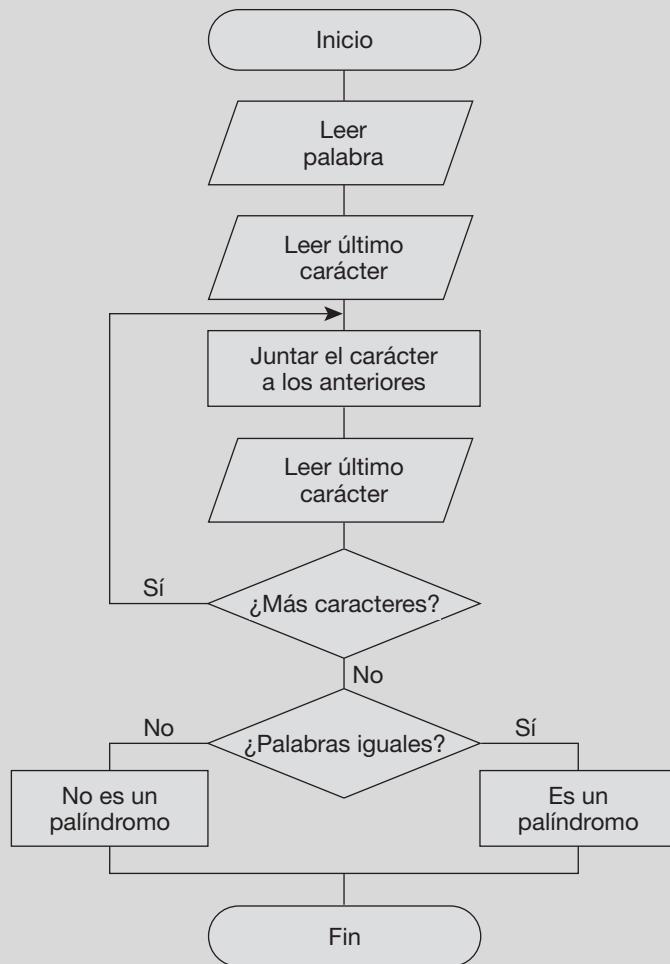
DATOS DE SALIDA: El mensaje que nos dice si es o no un palíndromo

DATOS DE ENTRADA: Palabra

DATOS AUXILIARES: Cada carácter de la palabra, palabra al revés

Para comprobar si una palabra es un palíndromo, se puede ir formando una palabra con los caracteres invertidos con respecto a la original y comprobar si la palabra al revés es igual a la original. Para obtener esa palabra al revés, se leerán en sentido inverso los caracteres de la palabra inicial y se irán juntando sucesivamente hasta llegar al primer carácter.

Diseño del algoritmo



- 2.23. Escribir un algoritmo para determinar el máximo común divisor de dos números enteros por el algoritmo de Euclides.

Análisis del problema

DATOS DE SALIDA: Máximo común divisor (mcd)
DATOS DE ENTRADA: Dos números enteros (a y b)
DATOS AUXILIARES: Resto

Para hallar el máximo común divisor de dos números se debe dividir uno entre otro. Si la división es exacta, es decir si el resto es 0, el máximo común divisor es el divisor. Si no, se deben dividir otra vez los números, pero en este caso el dividendo será el antiguo divisor y el divisor el resto de la división anterior. El proceso se repetirá hasta que la división sea exacta.

Para diseñar el algoritmo se debe crear un bucle que se repita mientras que la división no sea exacta. Dentro del bucle se asignarán nuevos valores al dividendo y al divisor.

Diseño del algoritmo

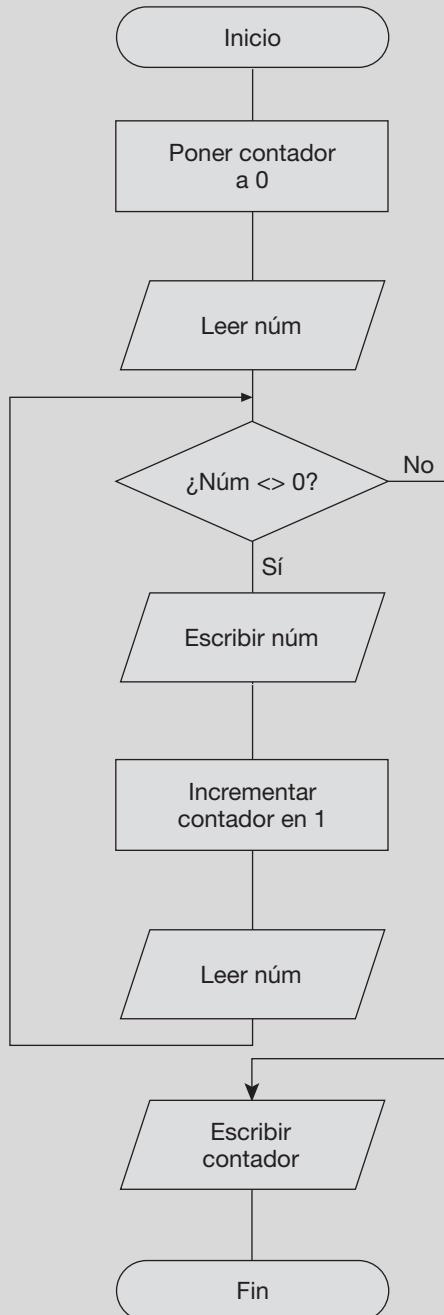
```
algoritmo Ejercicio_2_23
var
    entero: a,b,resto
inicio
    leer (a,b)
    mientras a mod b <> 0 hacer
        resto ← a mod b
        a ← b
        b ← resto
        mcd ← b
    fin_mientras
    escribir (mcd)
fin
```

- 2.24. Diseñar un algoritmo que lea e imprima una serie de números distintos de cero. El algoritmo debe terminar con un valor cero que no se debe imprimir. Finalmente se desea obtener la cantidad de valores leídos distintos de 0.

Análisis del problema

DATOS DE ENTRADA: Los distintos números (núm)
DATOS DE SALIDA: Los mismos números menos el 0, la cantidad de números (contador)

Se deben leer números dentro de un bucle que terminará cuando el último número leído sea cero. Cada vez que se ejecute dicho bucle y antes que se lea el siguiente número se imprime éste y se incrementa el contador en una unidad. Una vez se haya salido del bucle se debe escribir la cantidad de números leídos, es decir, el contador.

Diseño del algoritmo

2.25. Diseñar un algoritmo que imprima y sume la serie de números 3,6,9,12,...,99.

Análisis del problema

Se trata de idear un método con el que obtengamos dicha serie, que no es más que incrementar una variable de tres en tres. Para ello se hará un bucle que se acabe cuando el número sea mayor que 99 (o cuando se rea-

lice 33 veces). Dentro de ese bucle se incrementa la variable, se imprime y se acumula su valor en otra variable llamada suma, que será el dato de salida.

No tendremos por tanto ninguna variable de entrada, y sí dos de salida, la que nos va sacando los números de tres en tres (núm) y suma.

Diseño del algoritmo

```
algoritmo Ejercicio_2_25
var
    entero: núm, suma
inicio
    suma ← 0
    núm ← 3
    mientras núm <= 99 hacer
        escribir (núm)
        suma ← suma + num
        núm ← núm + 3
    fin_mientras
    escribir (suma)
fin
```

2.26. Escribir un algoritmo que lea cuatro números y, a continuación, escriba el mayor de los cuatro.

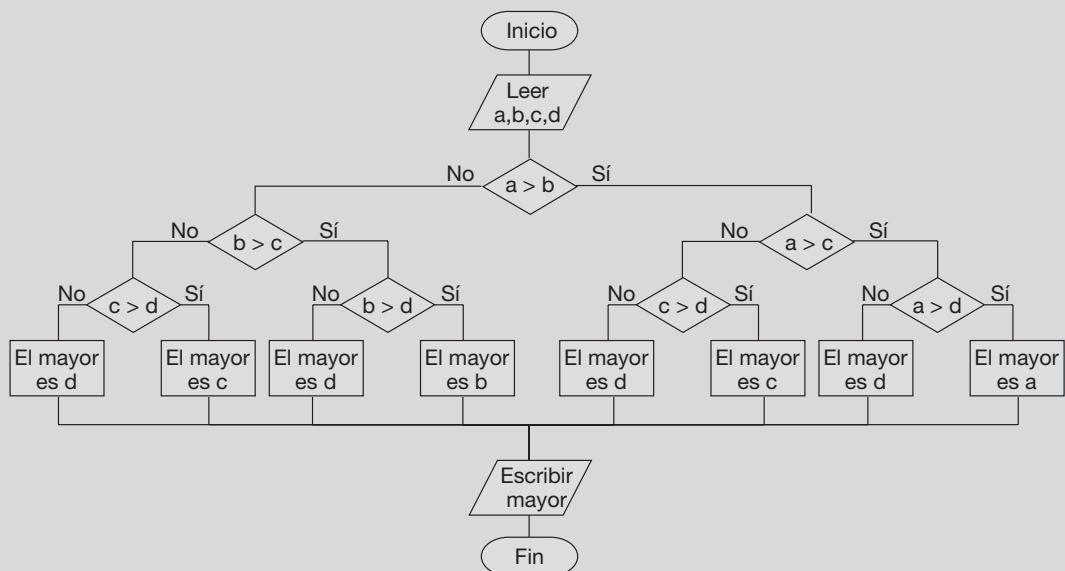
Análisis del problema

DATOS DE SALIDA: Mayor (el mayor de los cuatro números)

DATOS DE ENTRADA: A, b, c, d (los números que leemos por teclado)

Hay que comparar los cuatro números, pero no hay necesidad de compararlos todos con todos. Por ejemplo, si a es mayor que b y a es mayor que c, es evidente que ni b ni c son los mayores, por lo que si a es mayor que d el número mayor será a y en caso contrario lo será d.

Diseño del algoritmo



- 2.27.** Diseñar un algoritmo para calcular la velocidad (en metros/segundo) de los corredores de una carrera de 1.500 metros. La entrada serán parejas de números (minutos, segundos) que darán el tiempo de cada corredor. Por cada corredor se imprimirá el tiempo en minutos y segundos, así como la velocidad media. El bucle se ejecutará hasta que demos una entrada de 0,0 que será la marca de fin de entrada de datos.

Análisis del problema

DATOS DE SALIDA: v (velocidad media)
 DATOS DE ENTRADA: mm, ss (minutos y segundos)
 DATOS AUXILIARES: Distancia (distancia recorrida, que en el ejemplo es de 1500 metros) y $tiempo$ (los minutos y los segundos que ha tardado en recorrerla)

Se debe efectuar un bucle que se ejecute hasta que mm sea 0 y ss sea 0. Dentro del bucle se calcula el tiempo en segundos con la fórmula $tiempo = ss + mm * 60$. La velocidad se hallará con la fórmula $velocidad = distancia / tiempo$.

Diseño del algoritmo

```
algoritmo Ejercicio_2_27
var
  real: distancia, mm, tiempo, ss, v
inicio
  distancia ← 1500
  leer (mm,ss)
  mientras (mm <> 0.0) o (ss <> 0.0) hacer
    tiempo ← ss + mm * 60
    v ← distancia / tiempo
    escribir (mm,ss,v)
    leer (mm,ss)
  fin_mientras
fin
```

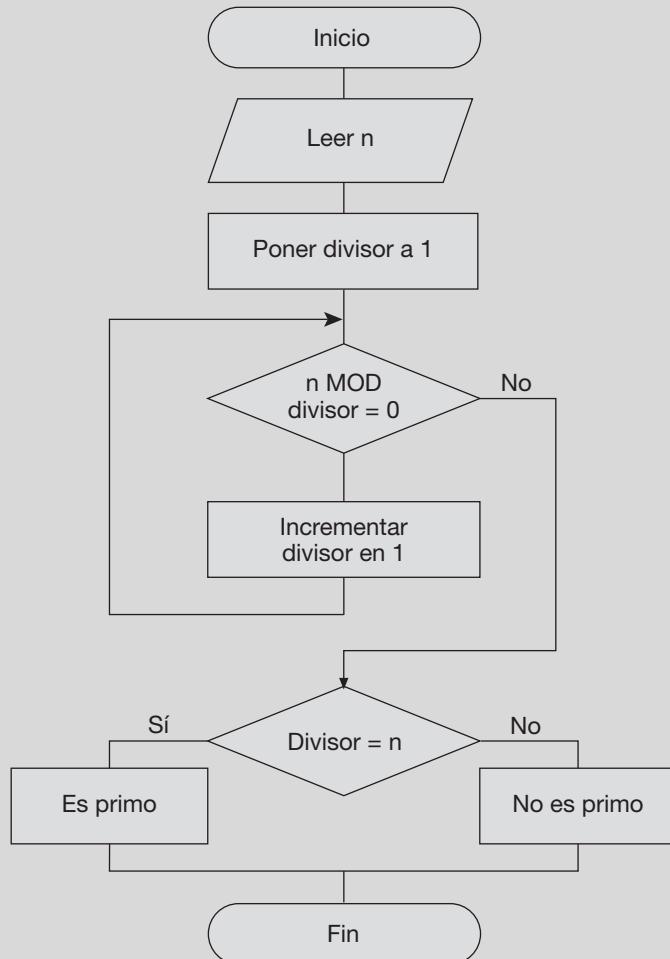
- 2.28.** Diseñar un algoritmo para determinar si un número n es primo (un número primo sólo es divisible por él mismo y por la unidad).

Análisis del problema

DATOS DE SALIDA: El mensaje que nos indica si es o no primo
 DATOS DE ENTRADA: n
 DATOS AUXILIARES: divisor (es el número por el que vamos a dividir n para averiguar si es primo)

Una forma de averiguar si un número es primo es por tanteo. Para ello se divide sucesivamente el número por los números comprendidos entre 2 y n . Si antes de llegar a n encuentra un divisor exacto, el número no será primo. Si el primer divisor es n el número será primo.

Por tanto se hará un bucle en el que una variable (divisor) irá incrementándose en una unidad entre 2 y n . El bucle se ejecutará hasta que se encuentre un divisor, es decir hasta que $n \bmod divisor = 0$. Si al salir del bucle $divisor = n$, el número será primo.

Diseño del algoritmo

2.29. Escribir un algoritmo que calcule la superficie de un triángulo en función de la base y la altura.

Análisis del problema

DATOS DE SALIDA: s (superficie)

DATOS DE ENTRADA: b (base), a (altura)

Para calcular la superficie se aplica la fórmula $s = \text{base} * \text{altura} / 2$.

Diseño del algoritmo

```

algoritmo Ejercicio_2_29
var
  real: s, a, b
inicio
  leer (b,a)
  s ← b * a / 2
  escribir (s)
fin
  
```


3

ESTRUCTURA GENERAL DE UN PROGRAMA

@UXYgW]dWQE`XY`cg YYa Ybhcg Vzg]Wg XY df c[fUa UWQE ei YgY YbWbhf Uzb Yb Wg] hcXcg ``cg`df c!
[fUa Ug`fjbhYff dhcf YgZ WbhUXcf YgZ UWa i UXcf YgZ Ugf Wa c `Ug`bcfa Ug`YYa YbhU Yg`dUFU`U YgW!
hi fU XY`U[cf]ha cg`Yb`dgYi Xc WQX]|| c`Wbgh]hi mYb`Y`WbhYb]Xc`XY`YghY`Wd#hi`c"

3.1. ESTRUCTURA DE UN PROGRAMA

Como ya se ha indicado en otras ocasiones el pseudocódigo es la herramienta más adecuada para la representación de algoritmos. El algoritmo en pseudocódigo debe tener una estructura muy clara y similar a un programa, de modo que se facilite al máximo su posterior codificación. Interesa por tanto conocer las secciones en las que se divide un programa, que habitualmente son:

- La *cabecera*.
- El *cuerpo* del programa:
 - Bloque de declaraciones.
 - Bloque de instrucciones.

La *cabecera* contiene el nombre del programa.

El *cuerpo* del programa contiene a su vez otras dos partes: el bloque de declaraciones y el bloque de instrucciones. En el bloque de declaraciones se definen o declaran las constantes con nombre, los tipos de datos definidos por el usuario y también las variables. Suele ser conveniente seguir este orden.

La *declaración de tipos* suele realizarse en base a los tipos estándar o a otros definidos previamente, aunque también hay que considerar el método directo de enumeración de los valores constituyentes.

El *bloque de instrucciones* contiene las acciones a ejecutar para la obtención de los resultados. Las instrucciones o acciones básicas a colocar en este bloque se podrían clasificar del siguiente modo:

- **De inicio/fín.** La primera instrucción de este bloque será siempre la de inicio y la última la de fin.
- **De asignación.** Esta instrucción se utiliza para dar valor a una variable en el interior de un programa.

- **De lectura.** Toma uno o varios valores desde un dispositivo de entrada y los almacena en memoria en las variables que aparecen listadas en la propia instrucción.
- **De escritura.** Envía datos a un dispositivo de salida.
- **De bifurcación.** Estas instrucciones no realizan trabajo efectivo alguno, pero permiten controlar el que se ejecuten o no otras instrucciones, así como alterar el orden en el que las acciones son ejecutadas. Las bifurcaciones en el flujo de un programa se realizarán de modo condicional, esto es en función del resultado de la evaluación de una condición. El desarrollo lineal de un programa se interrumpe con este tipo de instrucciones y, según el punto a donde se bifurca, podremos clasificarlas en bifurcaciones hacia adelante o hacia atrás. Las representaremos mediante estructuras selectivas o repetitivas.

Además, se recomienda que los programas lleven comentarios.

3.2. ESTRUCTURA GENERAL DE UN ALGORITMO EN PSEUDOCÓDIGO

La representación de un algoritmo mediante pseudocódigo se muestra a continuación. Esta representación es muy similar a la que se empleará en la escritura al programar.

```
algoritmo <nombre_algoritmo>
const
    <nombre_de_constante1> = valor1
    ...
var
    <tipo_de_dato1>: <nombre_de_variable1> [, <nombre_de_variable2>, ....]
    ...
    //Los datos han de ser declarados antes de poder ser utilizados
inicio
    <acción1>
    <acción2>
    //Se utilizará siempre la sangría en las estructuras selectivas y
    //repetitivas.
    .....
    <acciónN>
fin
```

Como se observa, después de la cabecera se coloca el bloque de declaraciones, donde se han declarado constantes con nombre y variables. Para declarar las constantes con nombre el formato ha sido:

```
const
    <nombre_de_constante1> = <valor1>
    ...
```

Al declarar las variables hay que listar sus nombres y especificar sus tipos de la siguiente forma:

```
var
    <tipo_de_dato1>: <lista_de_variables>
    ...
```

En el bloque de instrucciones, marcado por **inicio** y **fin** se sitúan las sentencias ejecutables, por ejemplo las operaciones de cálculo y lectura/escritura. El formato de las operaciones de lectura, cuando el dispositivo es el dispositivo estándar de entrada (teclado) es:

```
leer(<lista_de_variables>)
```

La operación de escritura, cuando los datos los envíamos al dispositivo estándar (pantalla), tiene el siguiente formato:

```
escribir(<lista_de_expresiones>)
```

3.3. LA OPERACIÓN DE ASIGNACIÓN

Esta operación se utiliza para dar valor a una variable en el interior de un algoritmo y permite almacenar en una variable el resultado de evaluar una expresión, perdiéndose cualquier otro valor previo que la variable pudiera tener. Su formato es:

```
<nombre_de_variable> ← <expresión>
```

Una expresión puede estar formada por una única constante, variable o función. La variable que recibe el valor final de la expresión puede intervenir en la misma, con lo que se da origen a contadores y acumuladores.

Se supone que se efectúan conversiones automáticas de tipo cuando el tipo del valor a asignar a una variable es compatible con el de la variable y de tamaño menor. En estos casos se considera que el valor se convierte automáticamente al tipo de la variable. También se interpreta que se efectúa este tipo de conversiones cuando aparecen expresiones en las que intervienen operandos de diferentes tipos.

3.3.1. Contadores

Un *contador* es una variable cuyo valor se incrementa o decremente en una cantidad constante cada vez que se produce un determinado suceso o acción. Los contadores se utilizan en las estructuras repetitivas con la finalidad de contar sucesos o acciones internas del bucle.

Con los contadores deberemos realizar una operación de inicialización y, posteriormente, las sucesivas de *incremento* o *decremento* del contador.

La *inicialización* consiste en asignarle al contador un valor. Se situará antes y fuera del bucle.

```
<nombre_del_contador> ← <valor_de_inicialización>
```

En cuanto a los *incrementos* o *decrementos* del contador, puesto que la operación de asignación admite que la variable que recibe el valor final de una expresión intervenga en la misma, se realizarán a través de este tipo de instrucciones de asignación, de la siguiente forma:

```
<nombre_del_contador> ← <nombre_del_contador> + <valor_constante>
```

dicho *<valor_constante>* podrá ser positivo o negativo. Esta instrucción se colocará en el interior del bucle.

3.3.2. Acumuladores

Son variables cuyo valor se incrementa o decremente en una cantidad determinada. Necesitan operaciones de:

- *Inicialización*

```
<nombre_acumulador> ← <valor_de_inicialización>
```

- *Acumulación*

```
<nombre_acumulador> ← <nombre_acumulador> + <nombre_variable>
```

Hay que tener en cuenta que la siguiente también sería una operación de acumulación:

```
<nombre_acumulador> ← <nombre_acumulador> * <valor>
```

3.3.3. Interruptores

Un interruptor o bandera (*switch*) es una variable que puede tomar los valores **verdad** y **falso** a lo largo de la ejecución de un programa, comunicando así información de una parte a otra del mismo. Pueden ser utilizados para el control de bucles y estructuras selectivas. En este caso también es frecuente que la variable que recibe el valor final de una expresión intervenga en la misma, por ejemplo

En primer lugar el interruptor (*switch*) se inicializa a **verdad** o **falso**

```
<nombre_del_interruptor> ← <valor_de_inicialización>
```

En determinadas condiciones el interruptor comuta

```
<nombre_del_interruptor> ← no <nombre_del_interruptor>
```

3.4. EJERCICIOS RESUELTOS

3.1. Se desea calcular independientemente la suma de los números pares en impares comprendidos entre 1 y 200.

Análisis del problema

El algoritmo no necesitaría ninguna variable de entrada, ya que no se le proporciona ningún valor. Como dato de salida se tendrán dos variables (*sumapar* y *sumaimpar*) que contendrán los dos valores pedidos. Se necesitará también una variable auxiliar (*contador*) que irá tomando valores entre 1 y 200.

Después de inicializar el contador y los acumuladores, comienza un bucle que se ejecuta 200 veces. En ese bucle se controla si el contador es par o impar, comprobando si es divisible por dos con el operador **mod**, e incrementando uno u otro acumulador.

Diseño del algoritmo

```
algoritmo ej_3_1
var
    entero : contador,sumapar,sumaimpar
inicio
    contador ← 0
    sumapar ← 0
    sumaimpar ← 0
repetir
    contador ← contador + 1
    si contador mod 2 = 0 entonces
        sumapar ← sumapar + contador
    si_no
        sumaimpar ← sumaimpar + contador
```

```

fin_si
hasta_que contador = 200
escribir (sumapar,sumaimpar)
fin

```

- 3.2.** Leer una serie de números enteros positivos distintos de 0 (el último número de la serie debe ser el -99) obtener el número mayor.

Análisis del problema

DATOS DE SALIDA: máx (el número mayor de la serie)

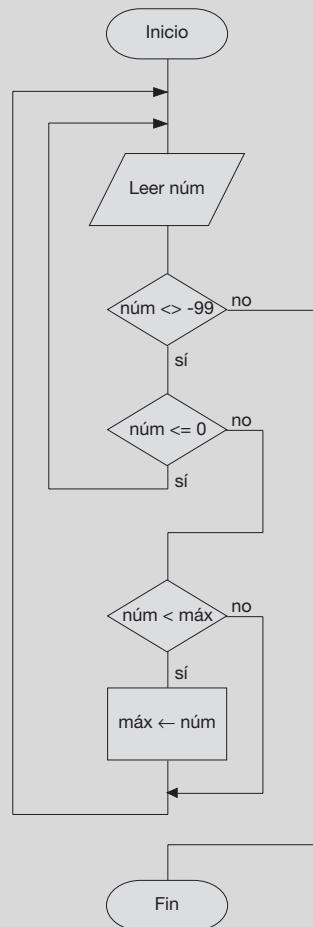
DATOS DE ENTRADA: num (cada uno de los números que introducimos)

Después de leer un número e inicializar máx a ese número, se ejecutará un bucle mientras el último número leído sea distinto de -99. Dentro del bucle se debe controlar que el número sea distinto de 0. Si el número es 0 se vuelve a leer hasta que la condición sea falsa. También hay que comprobar si el último número leído es mayor que el máximo, en cuyo caso el nuevo máximo será el propio número.

Diseño del algoritmo

TABLA DE VARIABLES:

entero : máx, num



3.3. Calcular y visualizar la suma y el producto de los números pares comprendidos entre 20 y 400, ambos inclusive.

Análisis del problema

DATOS DE SALIDA: suma, producto

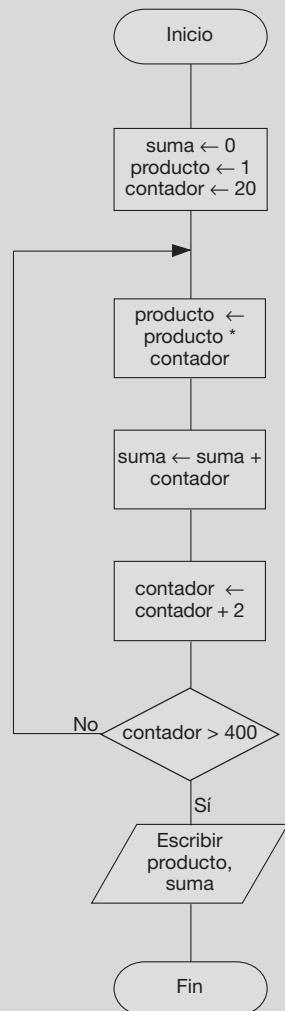
DATOS AUXILIARES: contador

Para solucionar el algoritmo, se deben inicializar los acumuladores suma y producto a 0 y la variable contador a 20 (puesto que se desea empezar desde 20) e implementar un bucle que se ejecute hasta que la variable contador valga 200. Dentro del bucle se irán incrementando los acumuladores suma y producto con las siguientes expresiones: suma \leftarrow suma+contador y producto \leftarrow producto*contador. Una vez realizadas las operaciones se debe incrementar el contador. Como se desea utilizar sólo los números pares, se usará una expresión como contador \leftarrow contador+2.

Diseño del algoritmo

TABLA DE VARIABLES:

entero : contador, suma, producto



3.4. Leer 500 números enteros y obtener cuántos son positivos.**Análisis del problema**

DATOS DE SALIDA: positivos (contiene la cantidad de números positivos introducidos)

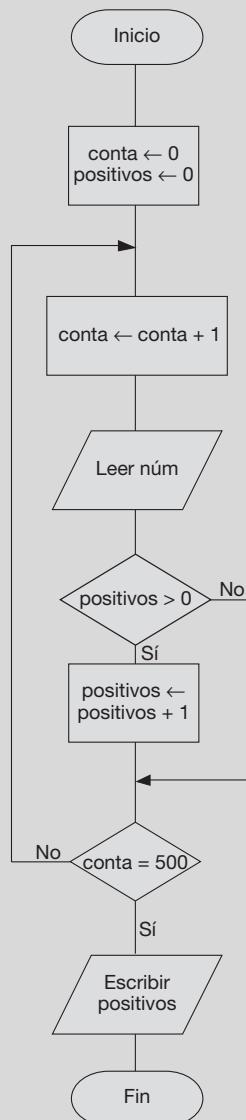
DATOS DE ENTRADA: núm (los números que introducimos)

DATOS AUXILIARES: conta (se encarga de contar la cantidad de números introducidos)

Se diseña un bucle que se ejecute 500 veces, controlado por la variable conta. Dentro del bucle se lee el número y se comprueba si es mayor que 0, en cuyo caso se incrementa el contador de positivos (positivos).

Diseño del algoritmo

TABLA DE VARIABLES:

entero : positivos, núm, conta

- 3.5.** Se trata de escribir el algoritmo que permita emitir la factura correspondiente a una compra de un artículo determinado del que se adquieren una o varias unidades. El IVA (Impuesto de Valor Añadido) a aplicar es del 12% y si el precio bruto (precio de venta + IVA) es mayor de 50.000 pesetas, se aplicará un descuento del 5%.

Análisis del problema

DATOS DE SALIDA: bruto (el precio bruto de la compra, con o sin descuento)

DATOS DE ENTRADA: precio (precio sin IVA), unidades

DATOS AUXILIARES: neto (precio sin IVA), iva (12% del neto)

Después de leer el precio del artículo y las unidades compradas, se calcula el precio neto (`precio * unidades`). Se calcula también el IVA (`neto * 0.12`) y el bruto (`neto + iva`). Si el bruto es mayor que 50.000 pesetas, se le descuenta un 5% (`bruto ← bruto * 0.95`).

Al multiplicar el valor neto por un valor real (0.12) para obtener el IVA, y calcular el bruto a partir del IVA, ambos deben ser datos reales.

Diseño del algoritmo

```
algoritmo ej_3_5
var
    entero : precio,neto,unidades
    real : iva,bruto
inicio
    leer (precio,unidades)
    neto ← precio * unidades
    iva ← neto * 0.12
    bruto ← neto + iva
    si bruto < 50000 entonces
        bruto ← bruto * 0.95
    fin_si
    escribir (bruto)
fin
```

- 3.6.** Calcular la suma de los cuadrados de los 100 primeros números naturales.

Análisis del problema

DATOS DE SALIDA: suma (acumula los cuadrados del número)

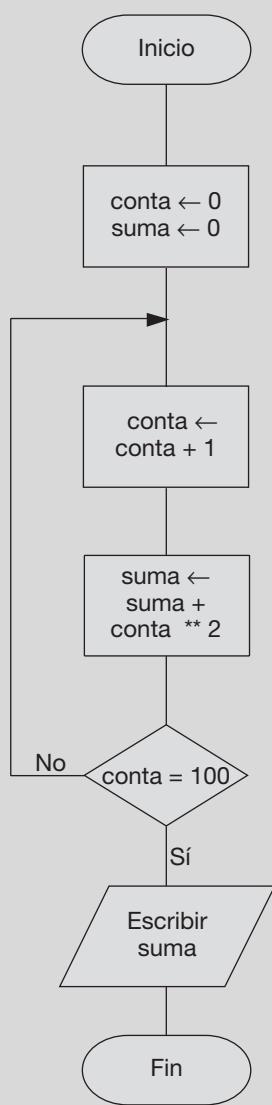
DATOS AUXILIARES: conta (contador que controla las iteraciones del bucle)

Para realizar este programa es necesario un bucle que se repita 100 veces y que se controlará por la variable conta. Dentro de dicho bucle se incrementará el contador y se acumulará el cuadrado del contador en la variable suma.

Diseño del algoritmo

TABLA DE VARIABLES:

entero : suma, conta



3.7. Sumar los números pares del 2 al 100 e imprimir su valor.

Análisis del problema

DATOS DE SALIDA: suma (contiene la suma de los números pares)

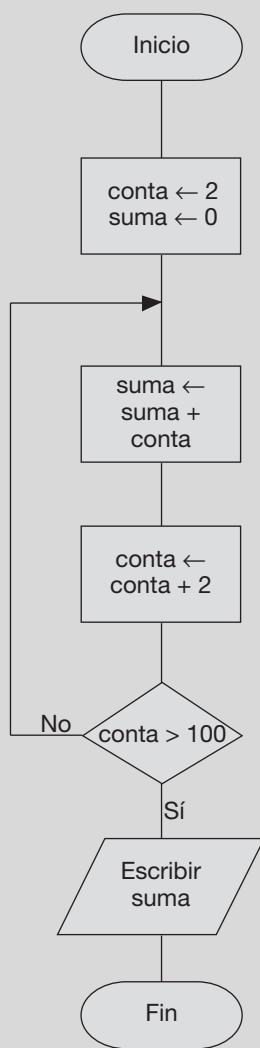
DATOS AUXILIARES: conta (contador que va sacando los números pares)

Se trata de hacer un bucle en el que un contador (conta) vaya incrementando su valor de dos en dos y, mediante el acumulador suma, y acumulando los sucesivos valores de dicho contador. El contador se deberá inicializar a 2 para que se saquen números pares. El bucle se repetirá hasta que conta sea mayor que 100.

Diseño del algoritmo

TABLA DE VARIABLES:

entero : conta, suma



3.8. Sumar 10 números introducidos por teclado.

Análisis del problema

DATOS DE SALIDA: suma (suma de los números)

DATOS DE ENTRADA: núm (los números que introducimos por teclado)

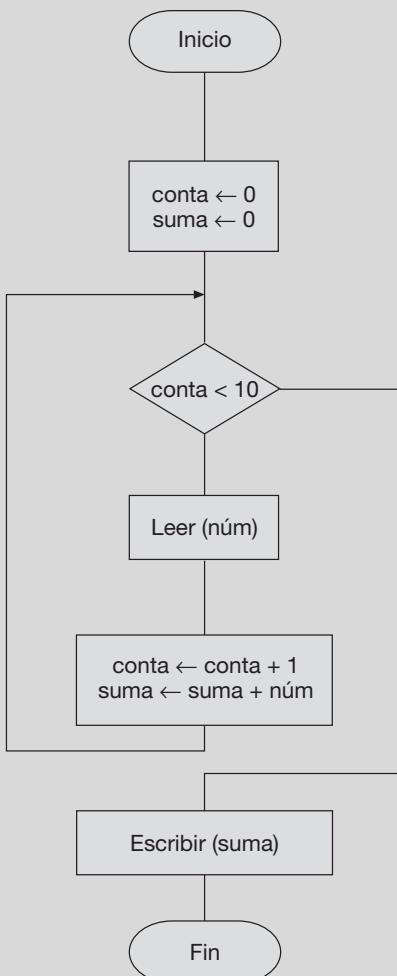
DATOS AUXILIARES: conta (contador que controla la cantidad de números introducidos)

Después de inicializar conta y suma a 0, se debe implementar un bucle que se ejecute 10 veces. En dicho bucle se incrementará el contador conta en una unidad, se introducirá por teclado núm y se acumulará su valor en suma. El bucle se ejecutará mientras que conta sea menor que 10.

Diseño del algoritmo

TABLA DE VARIABLES

entero : suma, núm, conta



3.9. Calcular la media de 50 números introducidos por teclado y visualizar su resultado.

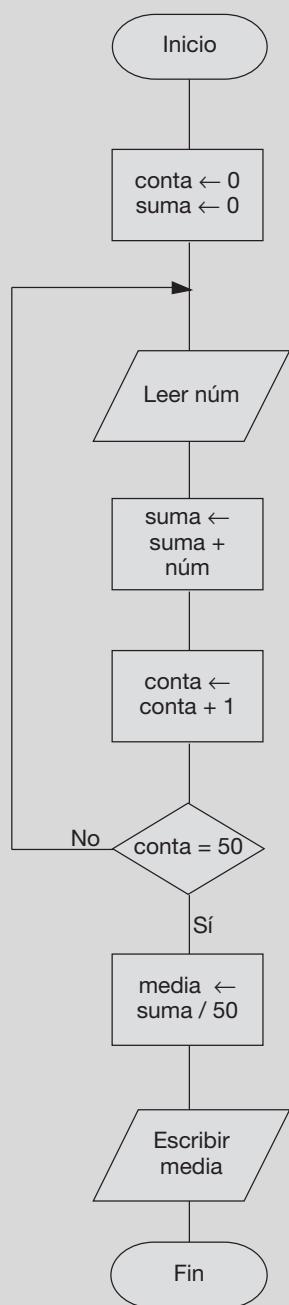
Análisis del problema

DATOS DE SALIDA:	media (contiene la media de los cincuenta números)
DATOS DE ENTRADA:	númer (cada uno de los cincuenta números introducidos por teclado)
DATOS AUXILIARES:	conta (contador que controla la cantidad de números introducidos), suma (acumula el valor de los números)

Después de inicializar conta y suma, se realiza un bucle que se repetirá 50 veces. En dicho bucle se lee un número (númer), se acumula su valor en suma y se incrementa el contador conta. El bucle se repetirá hasta que conta sea igual a 50. Una vez fuera del bucle se calcula la media (suma/50) y se escribe el resultado.

Diseño del algoritmo

TABLA DE VARIABLES
entero : númer, conta, suma
real : media



- 3.10.** Visualizar los múltiplos de 4 comprendidos entre 4 y N, donde N es un número introducido por teclado.

Análisis del problema

DATOS DE SALIDA: múltiplo (cada uno de los múltiplos de 4)
 DATOS DE ENTRADA: N (número que indica hasta qué múltiplo vamos a visualizar)
 DATOS AUXILIARES: conta (contador que servirá para calcular los múltiplos de 4)

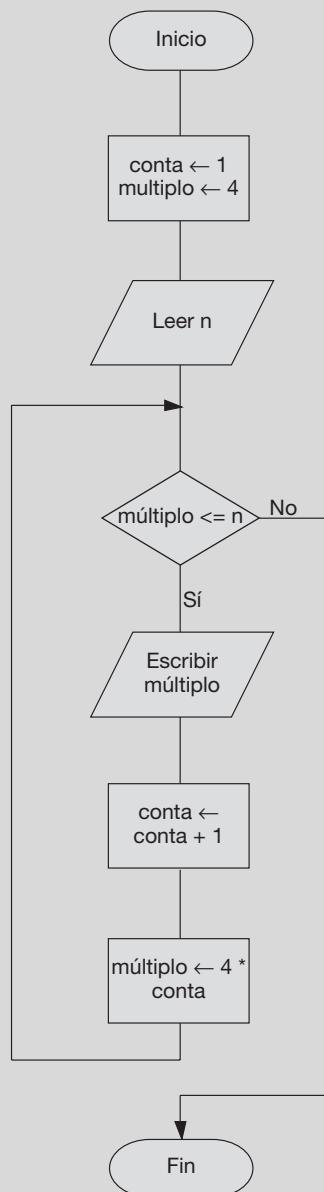
Para obtener los múltiplos de cuatro se pueden utilizar varios métodos. Un método consiste en sacar números correlativos entre 4 y N y para cada uno comprobar si es múltiplo de 4 mediante el operador **mod**. Otro método sería utilizar un contador que arrancando en 4 se fuera incrementado de 4 en 4. Aquí simplemente se irá multiplicando la constante 4 por el contador, que tomará valores a partir de 1.

Para realizar el algoritmo, después de inicializar **conta** a 1 y **múltiplo** a 4 y leer el número de múltiplos que se desean visualizar, se debe ejecutar un bucle mientras que **múltiplo** sea menor que N. Dentro del bucle hay que visualizar **múltiplo**, incrementar el contador en 1 y calcular el nuevo múltiplo ($4 * \text{conta}$).

Diseño del algoritmo

TABLA DE VARIABLES

entero : **múltiplo**, **N**, **conta**



3.11. Realizar un diagrama que permita realizar un contador e imprimir los 100 primeros números enteros.

Análisis del problema

DATOS DE SALIDA: Los números enteros entre 1 y 100

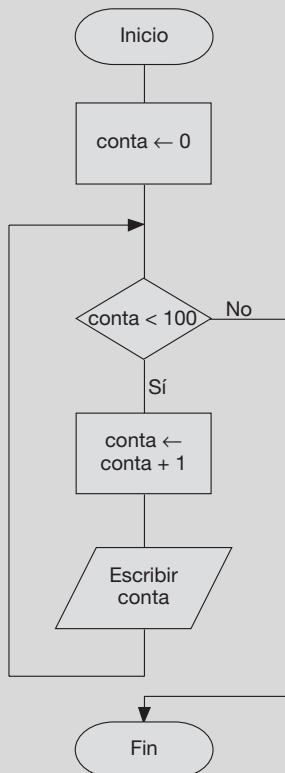
DATOS AUXILIARES: conta (controla el número de veces que se ejecuta el bucle)

Se debe ejecutar un bucle 100 veces. Dentro del bucle se incrementa en 1 el contador y se visualiza éste. El bucle se ejecutará mientras conta sea menor que 100. Previamente a la realización del bucle, conta se inicializará a 0.

Diseño del algoritmo

TABLA DE VARIABLES

entero : conta



3.12. Dados 10 números enteros que introduciremos por teclado, visualizar la suma de los números pares de la lista, cuántos números pares existen y cuál es la media aritmética de los números impares.

Diseño del algoritmo

DATOS DE SALIDA: spar (suma de pares), npar (cantidad de números pares), media (media de números impares)

DATOS DE ENTRADA: núm (cada uno de los números introducidos por teclado)

DATOS AUXILIARES: conta (contador que controla la cantidad de números introducidos), simpar (suma de los números impares), nimpar (cantidad de números impares)

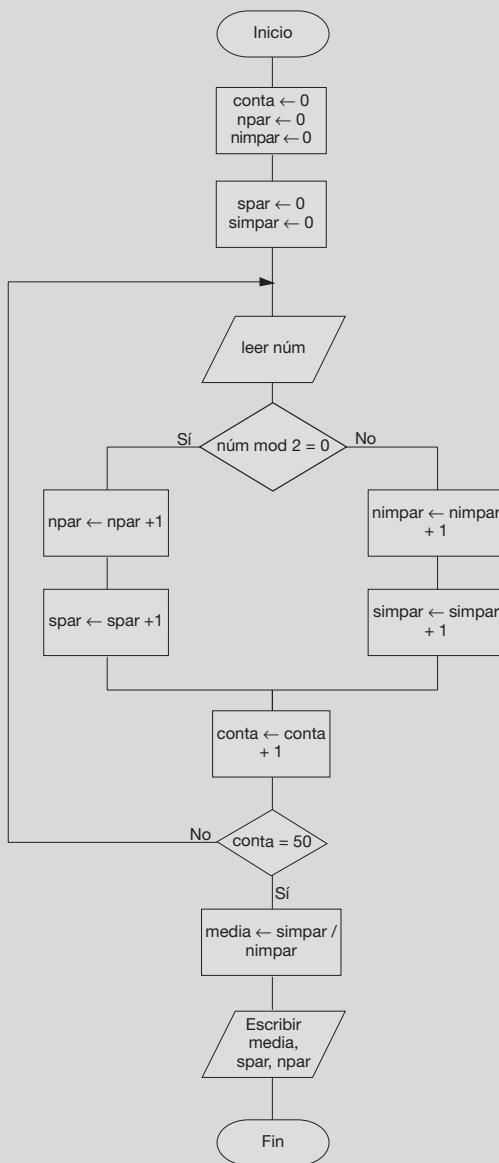
Se ha de realizar un bucle 10 veces. En ese bucle se introduce un número y se comprueba si es par mediante el operador **mod**. Si es par se incrementa el contador de pares y se acumula su valor en el acumulador de pares. En caso contrario se realizan las mismas acciones con el contador y el acumulador de impares. Dentro del bucle también se ha de incrementar el contador **conta** en una unidad. El bucle se realizará hasta que **conta** sea igual a 10.

Ya fuera del bucle se calcula la media de impares (**simpars/nimpar**) y se escribe **spar**, **npar** y **media**.

Diseño del algoritmo

TABLA DE VARIABLES

```
entero : conta, num, spar, npar, simpar, nimpar
real : media
```



- 3.13.** Calcular la nota media por alumno de una clase de a alumnos. Cada alumno podrá tener un número n de notas distintas.

Análisis del problema

DATOS DE SALIDA: media (media de cada alumno que también utilizaremos para acumular las notas)

DATOS DE ENTRADA: a (número de alumnos), n (número de notas de cada alumno), nota (nota de cada alumno en cada una de las asignaturas)

DATOS AUXILIARES: contaa (contador de alumnos), contan (contador de notas)

Para realizar este algoritmo se han de realizar dos bucles anidados. El primero se repetirá tantas veces como alumnos. El bucle interno se ejecutará por cada alumno tantas veces como notas tenga éste.

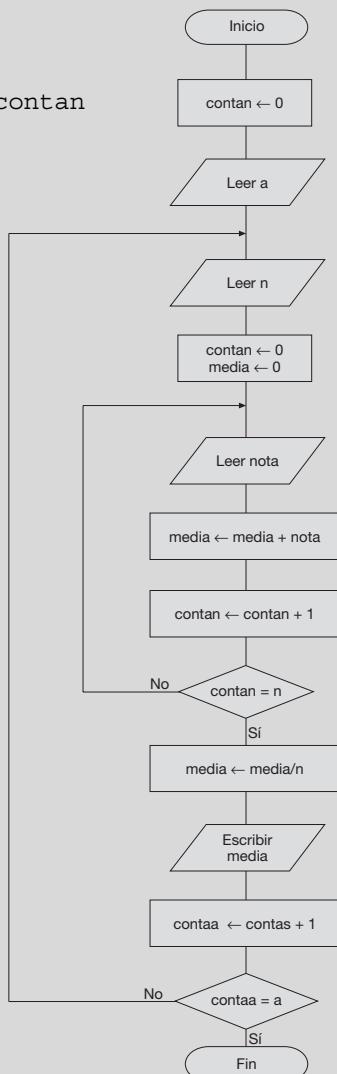
En el bucle de los alumnos se lee el número de notas y se inicializan las variables media y contan a 0. Aquí comenzará el bucle de las notas en el que leeremos una nota, se acumula en la variable media y se incrementa el contador de notas. Este bucle se ejecutará hasta que el contador de notas sea igual a n (número de notas).

Finalizado el bucle interno, pero todavía en el bucle de los alumnos, se calcula la media (media/n), se escribe y se incrementa el contador de alumnos. El programa se ejecutará hasta que contaa sea igual a a .

Diseño del algoritmo

TABLA DE VARIABLES

entero : $a, n, \text{contaa}, \text{contan}$
real : $\text{media}, \text{nota}$



4

INTRODUCCIÓN A LA PROGRAMACIÓN ESTRUCTURADA

9b`'U'U'Mi U]XUXzXUXc 'Y'WbgjXYfUVY'hLb U c 'XY`'Ug'a Ya cf]Ug'WbhfUYg'm'Ug'UhUg'j YcVAXUXg'XY
'cg' df c WgUXcfYgZ Y`'Ygh]c 'XY`'YgW]hi f U XY`'cg' df c[f Ua Uz XY` ei Y XYf l j U gi ' 'Y[JV]]XUX' mZzW
a cXZ]WV]]XUXzgY'Wbj JYf hY'Yb'i bU XY`'Ug'WfUWYf ggh]Wg'a zg'gcVfYgU]YbhYg'Yb`'Ug'hfWb]Wg'XY
df c[f Ua UW]C" 9ghY'Wdjh i `c 'Yl d'JWY Y'WbWdhc'XY' df c[f Ua UW]C'Yghfi Wi f UXUe ei Y'dYfa JhY'Ug!
W]hi f U XY' df c[f Ua Ug'ZzWYg'XY`'Yf' ma cXZ]Wfz m'Ug'Yghfi Wi f Ugi h]]nUXUg'df U Wbhfc'U'Y'Zi ^c
'C]W'Yb`'cg'a lga cg"9`'Yghi Xj c 'XY`'Ug'Yghfi Wi f Ug'XY'Wbhfc`'gYfYU]nUVUgUXc 'Yb`'Ug`'YffUa]YbhUg
XY' df c[f Ua UW]C" nU Yghi X]XUg' XjU f Ua UXY'Zi ^cZ XjU f Ua UB !G' mdgYi XcVWX][c"

4.1. PROGRAMACIÓN ESTRUCTURADA

La programación estructurada es un conjunto de técnicas para desarrollar algoritmos fáciles de escribir, verificar, leer y modificar. La programación estructurada utiliza:

- **Diseño descendente.** Consiste en diseñar los algoritmos en etapas, yendo de los conceptos generales a los de detalle. El diseño descendente se verá completado y ampliado con el modular.
- **Recursos abstractos.** En cada descomposición de una acción compleja se supone que todas las partes resultantes están ya resueltas, posponiendo su realización para el siguiente refinamiento.
- **Estructuras básicas.** Los algoritmos deberán ser escritos utilizando únicamente tres tipos de estructuras básicas.

4.2. TEOREMA DE BÖHM Y JACOPINI

Para que la programación sea estructurada, los programas han de ser propios. Un programa se define como propio si cumple las siguientes características:

- Tiene un solo punto de entrada y uno de salida.
- Toda acción del algoritmo es accesible, es decir, existe al menos un camino que va desde el inicio hasta el fin del algoritmo, se puede seguir y pasa a través de dicha acción.
- No posee lazos o bucles infinitos.

El teorema de Böhm y Jacopini dice que: «*un programa propio puede ser escrito utilizando únicamente tres tipos de estructuras: secuencial, selectiva y repetitiva*».

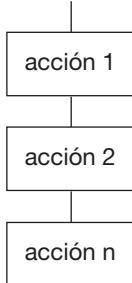
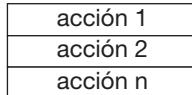
De este teorema se deduce que se han de diseñar los algoritmos empleando exclusivamente dichas estructuras, la cuales, como tienen un único punto de entrada y un único punto de salida, harán que nuestros programas sean propios.

4.3. CONTROL DEL FLUJO DE UN PROGRAMA

El flujo (orden en que se ejecutan las sentencias de un programa) es **secuencial** si no se especifica otra cosa. Este tipo de flujo significa que las sentencias se ejecutan en secuencia, una después de otra, en el orden en que se sitúan dentro del programa. Para cambiar esta situación se utilizan las estructuras de *selección, repetición y salto* que permiten modificar el flujo secuencial del programa. Así, las estructuras de selección se utilizan para seleccionar las sentencias que se han de ejecutar a continuación y las estructuras de repetición (repetitivas o iterativas) se utilizan para repetir un conjunto de sentencias. Las sentencias de *selección* son: *si* («**if**») y *según-sea* («**switch**»); las *sentencias de repetición o iterativas* son: *desde* («**for**»), *mientras* («**while**»), *hacer-mientras* («**do-while**») o *repetir-hasta que* («**repeat-until**»); las *sentencias de salto* incluyen *interrumpir* (**break**), *continuar* (**continue**), *ir-a* (**goto**), *volver* (**return**) y *lanzar* (**throw**).

4.3.1. Estructura secuencial

Una estructura secuencial es aquella en la cual una acción se ejecuta detrás de otra. El flujo del programa coincide con el orden físico en el que se sitúan las instrucciones.

Diagrama de flujo	Diagrama N-S	Pseudocódigo
 <pre> graph TD A[acción 1] --> B[acción 2] B --> C[acción n] </pre>	 <pre> graph TD A[acción 1] A --- B[acción 2] B --- C[acción n] </pre>	<pre> acción 1 acción 2 acción n </pre>

4.3.2. Estructura selectiva

Una *estructura selectiva* es aquella en que se ejecutan unas acciones u otras según se cumpla o no una determinada condición. La selección puede ser *simple, doble o múltiple*.

Simple

Se evalúa la condición y si ésta da como resultado verdad se ejecuta una determinada acción o grupo de acciones; en caso contrario se saltan dicho grupo de acciones.

Diagrama de flujo	Diagrama N-S	Pseudocódigo
<pre> graph TD A{condición} -- sí --> B[acción] A -- no --> C </pre>	<pre> graph TD A[condición] --- B[sí] A --- C[no] B --- D[acción] </pre>	<pre> si <condición> entonces acción fin_si </pre>

Doble

Cuando el resultado de evaluar la condición es verdad se ejecutará una determinada acción o grupo de acciones y si el resultado es falso otra acción o grupo de acciones diferentes.

Diagrama de flujo	Diagrama N-S	Pseudocódigo
<pre> graph TD A{condición} -- sí --> B[acción 1] A -- no --> C[acción 2] B --- D C --- D </pre>	<pre> graph TD A[condición] --- B[sí] A --- C[no] B --- D[acción 1] C --- E[acción 2] </pre>	<pre> si <condición> entonces acción 1 si_no acción 2 fin_si </pre>

Múltiple

Se ejecutarán unas acciones u otras según el resultado que se obtenga al evaluar una expresión. Aunque la flexibilidad de esta estructura está muy condicionada por el lenguaje, en nuestro pseudocódigo se considera que dicho resultado ha de ser de un tipo ordinal, es decir de un tipo de datos en el que cada uno de los elementos que constituyen el tipo, excepto el primero y el último, tiene un único predecesor y un único sucesor.

Cada grupo de acciones se encontrará ligado con: un valor, varios valores separados por comas, un rango, expresado como `valor_inicial..valor_final` o una mezcla de valores y rangos.

Se ejecutarán únicamente las acciones del primer grupo que, entre los valores a los que está ligado (su lista de valores), cuente con el obtenido al evaluar la expresión. Cuando el valor obtenido al evaluar la expresión no está presente en ninguna lista de valores se ejecutarán las acciones establecidas en la cláusula `si_no`, si existe dicha cláusula.

Diagrama de flujo	Diagrama N-S
<pre> graph TD A{expresión} -- "valor 1" --> B[acción 1] A -- "valor 2" --> C[acción 2] A -- "valor n" --> D[acción n] B --- E C --- E D --- E </pre>	<pre> graph TD A[expresión] --- B[v1] A --- C[v2] A --- D[v3] B --- E[acción 1] C --- F[acción 2] D --- G[acción 3] E --- H[acción n] I[si_no] </pre>

Pseudocódigo

```

según_sea <expresión> hacer
    <listal> : acción 1
    <lista2> : acción 2
    ...
[si_no
    acción ]
fin_según

```

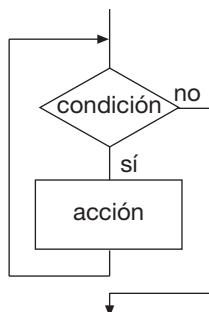
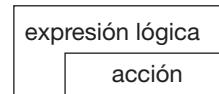
4.3.3. Estructura repetitiva

Las acciones del cuerpo del bucle se repiten mientras o hasta que se cumpla una determinada condición. Es frecuente el uso de contadores o banderas para controlar un bucle. También se utilizan con esta finalidad los centinelas.

Un *centinela* es un valor específico predefinido dado a una variable que permite detectar cuándo se desea terminar de repetir las acciones que constituyen el cuerpo del bucle. Por ejemplo, se puede diseñar un bucle que pida el nombre y la nota de una serie de alumnos y establecer que termine cuando se le introduzca un «*» como nombre. Podemos considerar tres tipos básicos de estructuras repetitivas: mientras, hacer-mientras, hasta, desde.

Mientras

Lo que caracteriza este tipo de estructura es que las acciones del cuerpo del bucle se realizan cuando la condición es cierta. Además, se pregunta por la condición al principio, de donde se deduce que dichas acciones se podrán ejecutar de 0 a n veces.

Diagrama de flujo**Diagrama N-S**

Pseudocódigo

```

mientras <expresión_lógica> hacer
    acción
fin_mientras

```

Hacer-mientras

El bucle **hacer-mientras** es análogo al bucle **mientras** desde el punto de vista de que el cuerpo del bucle se ejecuta una y otra vez mientras la condición (expresión *booleana*) es verdadera. La diferencia entre ellos consiste en que en el bucle **hacer-mientras** las sentencias del cuerpo se ejecutan, al menos una vez, antes de que se evalúe la expresión booleana. En otras palabras, el cuerpo del bucle siempre se ejecuta, al menos una vez, incluso aunque la expresión *booleana* sea falsa. Este tipo de bucle es típico de C/C++, Java o C#, pero no está presente en todos los lenguajes de programación por lo que será poco usado en nuestros algoritmos.

```
hacer
  <cuerpo_del_bucle>
mientras (<expresión_lógica>)
```

Hasta

Las acciones del interior del bucle se ejecutan una vez y continúan repitiéndose mientras que la condición sea falsa. Se interroga por la condición al final del bucle.

Diagrama de flujo	Diagrama N-S	Pseudocódigo
<pre> graph TD A[acción] --> B{condición} B -- no --> C[acción] B -- sí --> D[acción] </pre>	<pre> graph TD A[acción] --- B[expresión lógica] B --- C[repetir acción hasta que <expresión_lógica>] </pre>	<pre>repetir acción hasta que <expresión_lógica></pre>

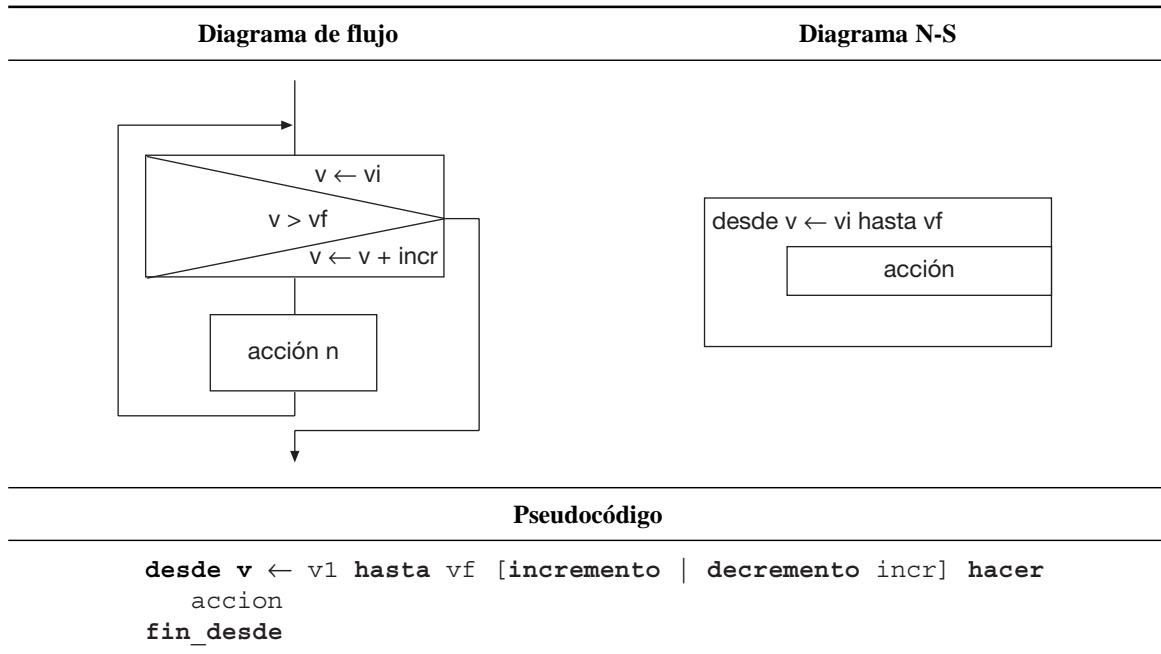
Desde

Se utiliza cuando se conoce, con anterioridad a que empiece a ejecutarse el bucle, el número de veces que se va a iterar.

La estructura **desde** comienza con un valor inicial de la variable índice y las acciones especificadas se ejecutan a menos que el valor inicial sea mayor que el valor final. La variable índice se incrementa en 1, o en el valor que especifiquemos, y si este nuevo valor no excede al final se ejecutan de nuevo las acciones.

Si establecemos que la variable índice se decremente en cada iteración el valor inicial deberá ser superior al final. Consideramos siempre la variable índice de tipo entero.

Es posible sustituir una estructura **desde** por otra de tipo **mientras** controlada por un contador.



4.3.4. Estructura anidada

Tanto las estructuras selectivas como las repetitivas pueden ser anidadas, e introducidas unas en el interior de otras.

La estructura selectiva múltiple es un caso especial de varias estructuras selectivas dobles anidadas en la rama **si_no**.

```

si <condición1> entonces
  <acciones1>
si_no
  si <condición2> entonces
    <acciones2>
  si_no
    si <condición3> entonces
      <acciones3>
    si_no
      <accionesX>
    fin_si
  fin_si
fin_si
  
```

Cuando se inserta un bucle dentro de otro la estructura interna ha de estar totalmente incluida dentro de la externa. Es posible anidar cualquier tipo de estructura repetitiva. Si se anidan dos estructuras **desde**, para cada valor de la variable índice del ciclo externo se debe ejecutar totalmente el bucle interno.

```

desde v1 ← vi1 hasta vf1 hacer
  desde v2 ← vi2 hasta vf2 hacer
    <acciones>
  fin_desde
fin_desde
  
```

Las acciones que componen el cuerpo del bucle más interno se ejecutarán el siguiente número de veces:

$$(vf1 - vi1 + 1) * (vf2 - vi2 + 1)$$

4.3.5. Sentencias de salto

Las sentencias de salto hacen que el flujo de control salte a otra parte del programa. Las sentencias de salto o bifurcación que se encuentran en los lenguajes de programación, tanto tradicionales como nuevos (Pascal, C, C++, C#, Java, etc.) son: ir-a (**goto**), interrumpir (**break**), continuar (**continue**), devolver (**return**) y lanzar (**throw**). Las cuatro primeras se suelen utilizar con sentencias de control y como retorno de ejecución de funciones o métodos. La sentencia **throw** se suele utilizar en los lenguajes de programación que poseen mecanismos de manipulación de excepciones, como suelen ser los casos de los lenguajes orientados a objetos tales como C++, Java y C#.

Con respecto a **interrumpir (break)**, uno de sus usos es *interrumpir un bucle* en un lugar determinado del cuerpo del mismo, en vez de esperar que termine de modo natural. A diferencia de la sentencia **continuar (continue)** que hace que el flujo de ejecución salte el resto de un cuerpo del bucle para continuar con la siguiente iteración, pero no interrumpe el bucle. Por su parte, la sentencia **ir-a (goto)** transfiere la ejecución del programa a una posición especificada por el programador. Las sentencias de salto **interrumpir (break)** e **ir-a (goto)** y **continuar (continue)** deben ser evitadas habitualmente. No obstante, algún lenguaje obliga a utilizar **break** y **goto** en sentencias **switch (según_sea)**. Así, en C# las acciones asociadas a un determinado valor de la expresión selectora deben terminar con una de estas instrucciones de salto, que habitualmente será **break**, o se producirá error. La sentencia **break** hace que **switch** se abandone tras la ejecución de las acciones asociadas a un determinado valor (situación por defecto en nuestro pseudocódigo en las sentencias **según_sea** sin necesidad de especificar **interrumpir**). La instrucción **goto** se utiliza en **switch** con la finalidad de transferir el control a otro punto, de forma que, después de ejecutadas las acciones asociadas a un valor, se ejecuten las asociadas a otro o las especificadas en la cláusula **default (si_no)**.

4.4. EJERCICIOS RESUELTOS

4.1. Dados tres números, deducir cuál es el central.

Análisis del problema

DATOS DE SALIDA: **central** (el número central)

DATOS DE ENTRADA: **a, b y c** (los números que vamos a comparar)

Se trata de ir comparando los tres números entre sí, utilizando selecciones de una sola comparación anidadas entre sí, ya que así se ahoran comparaciones (sólo utilizamos 5). Si se utilizan comparaciones con operadores lógicos del tipo **A < B y B > C**, se necesitarían seis estructuras selectivas por lo que se estarían haciendo dos comparaciones por selección.

Diseño del algoritmo (con comparaciones dobles)

```
algoritmo Ejercicio_4_1
var
    entero : a, b, c, central
inicio
    leer (a, b, c)
```

```
    si (a < b) y (b < c) entonces
        central ← b
    fin_si
    si (a < c) y (c < b) entonces
        central ← c
    fin_si
    si (b < a) y (a < c) entonces
        central ← a
    fin_si
    si (b < c) y (c < a) entonces
        central ← c
    fin_si
    si (c < a) y (a < b) entonces
        central ← a
    fin_si
    si (c < b) y (b < a) entonces
        central ← b
    fin_si
    escribir (central)
fin
```

Diseño del algoritmo (con comparaciones simples)

```
algoritmo ejercicio_4_1
var
    entero : a, b, c, central
inicio
    leer (a, b, c)
    si a > b entonces
        si b > c entonces
            central ← b
        si_no
            si a > c entonces
                central ← c
            si_no
                central ← a
            fin_si
        si_no
            si a > c entonces
                central ← a
            si_no
                si c > b entonces
                    central ← b
                si_no
                    central ← a
                fin_si
            fin_si
        fin_si
    escribir (central)
fin
```

4.2. Calcular la raíz cuadrada de un número y escribir su resultado.

Análisis del problema

Como dato de salida se tendrá la raíz y como entrada el número. Lo único que hay que hacer es asignar a **raíz** la raíz cuadrada del número, siempre que éste no sea negativo, ya que en ese caso no tendría una solución real. Se utilizará la función **raíz2** si se considera implementada; en caso contrario, deberá utilizarse la exponentiación.

Diseño del algoritmo

```
algoritmo ejercicio_4_2
var
    entero : n
    real : raíz
inicio
    leer (n)
    si n < 0 entonces
        escribir ('no hay solución real')
    si_no
        raíz ← n ** (1/2)
        escribir (raíz)
    fin_si
fin
```

4.3. Escribir los diferentes métodos para deducir si una variable o expresión numérica es par.

La forma de deducción se hace generalmente comprobando de alguna forma que el número o expresión numérica es divisible por dos. La forma más común sería utilizando el operador **mod**, el resto de la división entera. La variable a comprobar, **var**, será par si se cumple la condición:

var mod 2 = 0

y en caso de no disponer del operador **mod** y sabiendo que el resto es

resto = dividendo - divisor * ent(dividendo/divisor)

se trata de ver si **var** es par si se cumple la expresión siguiente

var - 2 * ent(var/2) = 0

Otra variante podría ser comprobar si la división real de **var** entre 2 es igual a la división entera de **var** entre 2

var/2 = var div 2

o, siguiendo el mismo método si

var/2 = ent(var/2)

En algunos casos, estos ejemplos pueden llevar a error según sea la precisión que la computadora obtenga en el cálculo de la expresión.

- 4.4.** Determinar el precio de un billete de ida y vuelta en ferrocarril, conociendo la distancia a recorrer y sabiendo que si el número de días de estancia es superior a siete y la distancia superior a 800 kilómetros el billete tiene una reducción del 30%.
El precio por kilómetro es de 2,5 pesetas.

Análisis del problema

DATOS DE SALIDA: Precio del billete
 DATOS DE ENTRADA: Distancia a recorrer, días de estancia

Se lee la distancia y el número de días y se halla el precio del billete de ida y vuelta (`precio = distancia * 2 * 2.5`). Se comprueba si la distancia es superior a 800 Km. y los días de estancia a 7 y si es cierto se aplica una reducción del 30%.

Diseño del algoritmo

```
algoritmo ejercicio_4_4
var
  entero : distancia,días
  real : precio
inicio
  leer (distancia,días)
  precio ← distancia * 2 * 2.5
  si días > 7 y distancia > 800 entonces
    precio ← precio * 0.3
  fin_si
  escribir (precio)
fin
```

- 4.5.** Diseñar un algoritmo en el que a partir de una fecha introducida por teclado con el formato DÍA, MES, AÑO, se obtenga la fecha del día siguiente.

Análisis del problema

DATOS DE SALIDA: dds, mms, aas (día, mes y año del día siguiente)
 DATOS DE ENTRADA: dd, mm, aa (día mes y año del día actual)

En principio lo único que habría que hacer es sumar una unidad al día. Si el día actual es menor que 28 (número de días del mes que menos días tiene) no sucede nada, pero se debe comprobar si al sumar un día ha habido cambio de mes o de año, para lo que se comprueba los días que tiene el mes, teniendo también en cuenta los años bisiestos. También se debe comprobar si es el último día del año en cuyo caso se incrementa también el año. Se supone que la fecha introducida es correcta.

Diseño del algoritmo

```
algoritmo ejercicio_4_5
var
  entero : dd, mm, aa, dds, mms, aas
inicio
  leer(dd,mm,aa)
  dds ← dd + 1
  mms ← mm
  aas ← aa
```

```

si dd >= 28 entonces
  según_sea mm hacer
    //si el mes tiene treinta días
    4,6,9,11 :   si dds > 30 entonces
      dds ← 1
      mms ← mm + 1
    fin_si
    //si el mes es febrero
    2 :   si (dds > 29) o (aa mod 4 <> 0) entonces
      dds ← 1
      mms ← 3
    fin_si
    //si el mes tiene 31 días y no es diciembre
    1,3,5,7,8,10 :   si dds > 31 entonces
      dds ← 1
      mms ← mm + 1
    fin_si
  si_no
    //si el mes es diciembre
    si dds > 31 entonces
      dds ← 1
      mms ← 1
      aas ← aa + 1
    fin_si
  fin_según
fin_si
escribir(dds,mms,aas)
fin

```

- 4.6.** Se desea realizar una estadística de los pesos de los alumnos de un colegio de acuerdo a la siguiente tabla:

Alumnos de menos de 40 kg.
 Alumnos entre 40 y 50 kg.
 Alumnos de más de 50 y menos de 60 kg.
 Alumnos de más o igual a 60 kg.

La entrada de los pesos de los alumnos se terminará cuando se introduzca el valor centinela -99.
 Al final se desea obtener cuántos alumnos hay en cada uno de los baremos.

Análisis del problema

DATOS DE SALIDA: conta1, conta2, conta3, conta4 (contadores de cada uno de los baremos)

DATOS DE ENTRADA: peso (peso de cada uno de los alumnos)

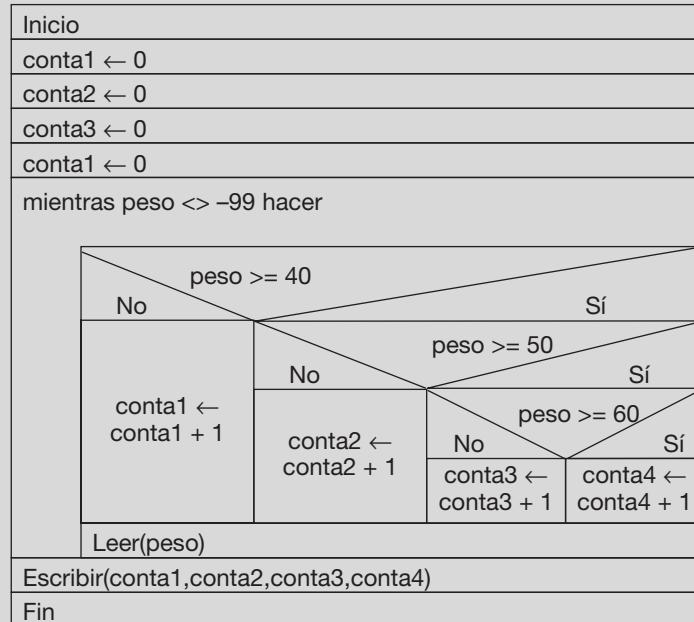
Se construye un bucle que se ejecute hasta que el peso introducido sea igual a -99. Dentro del bucle se comprueba con una serie de estructuras **si** anidadas a qué lugar de la tabla corresponde el peso, incrementándose los contadores correspondientes. Cuando se haya salido del bucle se escribirán los contadores.

Nótese que ha de leer el peso una vez antes de entrar en el bucle y otra vez como última instrucción de éste, pues de lo contrario se incluirá el centinela (-99) en la estadística la última vez que se ejecute el bucle.

Diseño del algoritmo

TABLA DE VARIABLES:

```
entero : conta1, conta2, conta3, conta4
real : peso
```



- 4.7.** Realizar un algoritmo que averigüe si dados dos números introducidos por teclado, uno es divisor del otro.

Análisis del problema

DATOS DE SALIDA: El mensaje que nos dice si es o no divisor

DATOS DE ENTRADA: núm1, núm2 (los números que introducimos por teclado)

DATOS AUXILIARES: divisor (variable lógica que será cierta si el segundo número es divisor del primero)

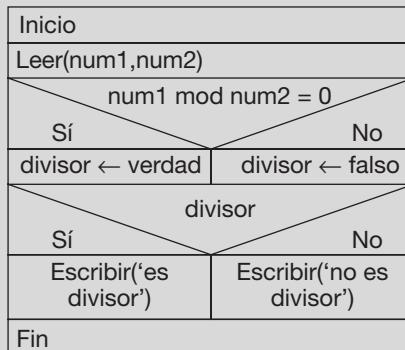
En este problema se utiliza el operador **mod** para comprobar si núm2 es divisor de núm1. Si el resto de dividir los dos números es 0 se establece la variable divisor como cierta, y en caso contrario como falsa.

Preguntando por el valor del divisor se obtiene un mensaje que nos indicará si núm2 es o no divisor de núm2

Diseño del algoritmo

TABLA DE VARIABLES:

```
entero : núm1, núm2
lógico : divisor
```



4.8. Analizar los distintos métodos de realizar la suma de T números introducidos por teclado.

Para realizar la suma de T números será imprescindible realizar un bucle que se ejecute T veces y que incluya una operación de lectura y un acumulador. Al conocer de antemano el número de veces que se ejecuta el bucle (T), lo más adecuado será utilizar un bucle de tipo **desde**:

```

.
.
suma ← 0
desde i ← 1 hasta T hacer
  leer(número)
  suma ← suma + número
fin_desde
.
.
```

El mismo bucle se puede realizar con una estructura **mientras**, **hacer-mientras** o **repetir**, en cuyo caso dentro del bucle se incluirá también un contador.

- Con una estructura **mientras**:

```

.
.
suma ← 0
conta ← 0
mientras conta < T hacer
  leer(número)
  suma ← suma + número
  conta ← conta + 1
fin_mientras
.
```

- Con una estructura **hacer-mientras**:

```

.
.
suma ← 0
conta ← 0
```

```

hacer
    leer(número)
    suma ← suma + número
    conta ← conta + 1
    mientras conta < T
    .
    .

```

- Con una estructura repetir:

```

    .
    .
    suma ← 0
    conta ← 0
    repetir
        leer(número)
        suma ← suma + número
        conta ← conta + 1
    hasta_que conta = T
    .
    .

```

- 4.9.** Se desea un algoritmo que realice la operación de suma o resta de dos números leídos del teclado en función de la respuesta S o R (suma o resta) que se dé a un mensaje de petición de datos.

Análisis del problema

DATOS DE SALIDA: resultado (resultado de la operación suma o resta de los dos números)
 DATOS DE ENTRADA: a, b (los números a operar), operación (tipo de la operación a efectuar)

Después de leer los números a y b, un mensaje ha de solicitar que pulsemos una tecla S o una tecla R para realizar una u otra operación. Aquí se incluye un bucle que se repita hasta que la entrada sea S o R, para evitar errores de entrada. En función de la respuesta se calcula el resultado que será a + b, en caso que la respuesta sea S, o a - b en caso que la respuesta sea R.

Diseño del algoritmo

```

algoritmo ejercicio_4_9
var
    entero : a,b,resultado
    carácter : operación
inicio
    leer(a,b)
    escribir('Pulse S para sumar, R para restar')
    repetir
        leer(operación)
        hasta_que (operación = 'S') o (operación = 'R')
        si operación = 'S' entonces
            resultado ← a + b
        si_no
            resultado ← a - b
        fin_si
        escribir(resultado)
fin

```

- 4.10.** Escribir un algoritmo que lea un número y deduzca si está entre 10 y 100, ambos inclusive.

Análisis del problema

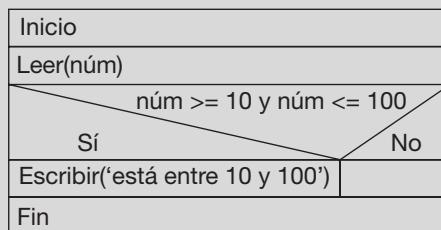
DATOS DE SALIDA: El mensaje que indica si el número está entre esos límites
 DATOS DE ENTRADA: num (el número que se desea comprobar)

Mediante una comparación múltiple se comprueba si el número está entre esos límites, en cuyo caso aparecerá un mensaje.

Diseño del algoritmo

TABLA DE VARIABLES

entero : num



- 4.11.** Se dispone de las calificaciones de los alumnos de un curso de informática correspondientes a las asignaturas de BASIC, FORTRAN y PASCAL. Diseñar un algoritmo que calcule la media de cada alumno.

Análisis del problema

DATOS DE SALIDA: media (una por alumno)
 DATOS DE ENTRADA: nota (tres por alumno) y n (número de alumnos)
 DATOS AUXILIARES: i, j (variables que controlan los bucles)

Hay que leer el número de alumnos para conocer las iteraciones que debe ejecutarse el bucle (si se utilizara un bucle **repetir** o **mientras** se debería establecer alguna otra condición de salida). Dentro del bucle principal se anida otro bucle que se repite tres veces por alumno (cada alumno tiene tres notas) en el que leemos la nota y la acumulamos en la variable media. Al finalizar el bucle interno se halla y se escribe la media.

Diseño del algoritmo

```

algoritmo ejercicio_4_11
var
  entero : n,i,j
  real : media, nota
inicio
  leer(n) //número de alumnos
  desde i = 1 hasta n hacer
    media ← 0
    desde j = 1 hasta 3 hacer
      leer(nota)
      media ← media + nota
  
```

```

fin_desde
media ← media/3
escribir(media)
fin_desde
fin

```

4.12. Escribir el ordinograma y el pseudocódigo que calcule la suma de los 50 primeros números enteros.

Análisis del problema

DATOS DE SALIDA: suma (suma de los primeros 50 números enteros)

DATOS AUXILIARES: conta (contador que irá sacando los números enteros)

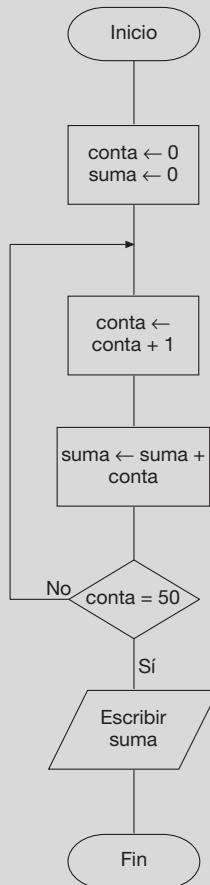
Se trata de hacer un bucle en el que se incremente un contador y que se acumule el valor de éste en un acumulador (suma). El bucle se ejecutará hasta que el contador sea igual a 50.

Diseño del algoritmo

TABLA DE VARIABLES:

entero : suma, conta

Ordinograma



Pseudocódigo

```

algoritmo ejercicio_4_12
var
    entero : suma,conta
inicio
    suma ← 0
    conta ← 0
    repetir
        conta ← conta + 1
        suma ← suma + conta
    hasta que conta = 50
    escribir(suma)
fin

```

4.13. Calcular y escribir los cuadrados de una serie de números distintos de 0 leídos desde el teclado.

Análisis del problema

DATOS DE SALIDA: cuadrado
 DATOS DE ENTRADA: núm

Dado que se desconoce la cantidad de números leídos, se debe utilizar un bucle **repetir** o **mientras**. Se construye un bucle hasta que el número sea 0. Dentro del bucle se lee núm y se calcula su cuadrado. Para no incluir el último número -el cero-, se ha de leer antes de entrar al bucle y otra vez al final de éste.

Diseño del algoritmo

```

algoritmo ejercicio_4_13
var
    entero : núm, cuadrado
inicio
    leer(núm)
    mientras núm <> 0 hacer
        cuadrado ← núm * núm
        escribir(cuadrado)
        leer(núm)
    fin_mientras
fin

```

4.14. Un capital C está situado a un tipo de interés R ¿Se doblará el capital al término de dos años?

Análisis del problema

DATOS DE SALIDA: El mensaje que nos dice si el capital se dobla o no
 DATOS DE ENTRADA: C (capital), R (interés)
 DATOS AUXILIARES: CF (capital al final del período)

Después de introducir el capital y el interés por teclado, se calcula el capital que se producirá en dos años por la fórmula del interés compuesto:

$$CF = C(1+R)^2$$

Si CF es igual a C*2 aparecerá el mensaje diciendo que el capital se doblará.

Diseño del algoritmo

```

algoritmo Ejercicio_4_14
var
    entero : C
    real : R,CF
inicio
    leer(C,R)
    CF ← C*(1+R/100)**2
    si C*2 <= CF entonces
        escribir('El capital se doblará o superará el doble')
    si_no
        escribir('El capital no se doblará')
    fin_si
fin

```

- 4.15.** Leer una serie de números desde el terminal y calcular su media. La marca de fin de lectura será el número -999.

Análisis del problema

DATOS DE SALIDA: media (media de los números)
 DATOS DE ENTRADA: núm (cada uno de los números)
 DATOS AUXILIARES: conta (cuenta los números introducidos excepto el -999), suma (suma los números excepto el -999)

Se debe construir un bucle que se repita hasta que el número introducido sea -999. Como ya se vio más arriba (ejercicio 4.13.) se debe leer antes de comenzar el número y al final del bucle. Dentro del bucle se incrementará un contador, necesario para poder calcular la media, y un acumulador guardará la suma parcial de los números introducidos. Una vez fuera del bucle se calcula e imprime la media.

Diseño del algoritmo

```

algoritmo ejercicio_4_15
var
    entero : conta, suma, núm
    real : media
inicio
    conta ← 0
    suma ← 0
    leer(núm)
    mientras núm <> -999 hacer
        conta ← conta + 1
        suma ← suma + núm
        leer(núm)
    fin_mientras
    media ← suma/conta
    escribir(media)
fin

```

4.16. Se considera la serie definida por:

$$a_1=0, a_2=1, a_n=3*a_{n-1}+2*a_{n-2} \text{ (para } n >= 3\text{)}$$

Se desea obtener el valor y el rango del primer término que sea mayor o igual que 1000.

Análisis del problema

- DATOS DE SALIDA: i (rango del primer término de la serie mayor o igual que 1000), último (valor de dicho término)
 DATOS AUXILIARES: penúltimo (penúltimo valor de la serie), valor (valor actual)

Se calcula el término por tanteo, es decir a partir del tercer término, se ejecuta un bucle que se realice mientras que el valor del término sea menor que 1000.

Para ello se inicializa i a 2, último a 1 y penúltimo a 0. Dentro del bucle se calcula el valor ($3 * \text{último} + 2 * \text{penúltimo}$). Antes de finalizar el bucle se deben actualizar los valores de penúltimo ($\text{penúltimo} \leftarrow \text{último}$) y último ($\text{último} \leftarrow \text{valor}$). También se ha de incrementar el contador i en una unidad. Al finalizar el bucle, primer valor mayor o igual a 1000 será último , y el término i .

Diseño del algoritmo

```
algoritmo ejercicio_4_16
var
  entero : i,valor,último,penúltimo
inicio
  i ← 2
  último ← 1
  penúltimo ← 0
  mientras último < 1000   hacer
    valor ← 3 * último + 2 * penúltimo
    i ← i+1
    penúltimo ← último
    último ← valor
  fin_mientras
  escribir(último,i)
fin
```

4.17. Escribir un algoritmo que permita calcular X^n , donde:

$$\begin{aligned} X &\text{ puede ser cualquier número real distinto de 0} \\ n &\text{ puede ser cualquier entero positivo, negativo o nulo} \end{aligned}$$

Nota: suponemos que no está implementado el operador de exponentiación.

Análisis del problema

- DATOS DE SALIDA: potencia (resultado de elevar X a la n)
 DATOS DE ENTRADA: X (base de la exponentiación), n (exponente)
 DATOS AUXILIARES: conta (contador que controla el número de veces que se multiplica X por sí mismo), solución (será falso si no hay solución)

Después de introducir X y n , se comprueba n . Si es 0, potencia valdrá 1. Si es positivo se ha de multiplicar X n veces, y ello se hará mediante un bucle en el que se vayan guardando en la variable producto las sucesivas multiplicaciones. En el bucle un contador irá controlando las veces que se ejecuta el bucle, que finalizará cuando sea igual a n . Si n es negativo, primero se comprueba que X sea distinto de cero, pues en ese

caso no tendrá solución. En caso contrario se procederá de la misma forma que si es positivo, pero la potencia será $1/\text{potencia}$.

Diseño del algoritmo

```

algoritmo ejercicio_4_17
var
    entero : n,conta
    real : X,potencia
    lógico : solución
inicio
    leer(X,n)
    solución ← verdad
    si n = 0 entonces
        potencia ← 1
    si_no
        si n > 0 entonces
            potencia ← 1
            conta ← 0
            repetir
                potencia ← potencia * X
                conta ← conta + 1
                hasta_que conta = n
            si_no
                si X = 0 entonces
                    escribir('No hay solución')
                    solución ← falso
                si_no
                    potencia ← 1
                    conta ← 0
                    repetir
                        potencia _ potencia * X
                        conta ← conta -1
                        hasta_que conta = n
                        potencia ← 1/potencia
                    fin_si
                fin_si
            fin_si
            si solución entonces
                escribir(potencia)
            fin_si
    fin

```

- 4.18.** Se desea leer desde teclado una serie de números hasta que aparezca alguno menor que 1000.

Análisis del problema

En este algoritmo, sólo hay un dato de entrada, que será cada uno de los números que leemos, y no tenemos ningún dato de salida. No hay más que ejecutar un bucle hasta que el número leído sea menor de 1000.

Diseño del algoritmo

```

algoritmo ejercicio_4_18
var
    real : num
inicio
repetir
    leer(num)
hasta_que num < 1000 // sólo se admiten los menores de 1000
fin

```

- 4.19.** Se desea obtener los cuadrados de todos los números leídos desde un archivo hasta que se encuentre el número 0.

Análisis del problema

Este ejercicio es muy similar al anterior aunque se introduce la noción de archivo que será vista en profundidad más adelante. Por el momento no vamos a tener en cuenta las operaciones de abrir y cerrar el archivo, así como la marca de fin de archivo o la definición de su estructura. Para leer números desde el archivo se utiliza una operación de lectura que indica que se lee desde un archivo:

leer(nombre del archivo, lista de variables)

Diseño del algoritmo

```

algoritmo ejercicio_4_19
var
    real : num
inicio
    //Instrucciones previas de apertura de archivo
repetir
    leer(archivo,num)
    si num <> 0 entonces
        escribir(num ** 2)
        fin_si
    hasta_que num = 0
    //Instrucciones de cierre
fin

```

- 4.20.** Algoritmo que reciba una fecha por teclado - dd, mm, aaaa -, así como el día de la semana que fue el primero de enero de dicho año, y muestre por pantalla el día de la semana que corresponde a la fecha que le hemos dado. En su resolución deben considerarse los años bisiestos.

Análisis del problema

Se comenzará realizando una verificación de datos correctos en la introducción de datos. Una vez aceptada una fecha como correcta, se inicializará a 0 la variable `dtotal` y se acumulan en ella los días correspondientes a cada uno de los meses transcurridos, desde enero hasta el mes anterior al que indica la fecha, ambos inclusive. Por último le añadiremos los días del mes actual.

Si hacemos grupos de 7 con `dtotal`, se obtendrá un cierto número de semanas y un resto. Para averiguar el día de la semana en que cae la fecha especificada, bastará con avanzar, a partir del día de la semana que fue el primero de enero de dicho año, el número de días que indica el resto.

Se ha de tener en cuenta que, considerando el inicio de la semana en lunes, avanzar desde el domingo consiste en volver a situarse en el lunes.

Diseño del algoritmo

```
algoritmo ejercicio_4_20
var
    entero : dd, mm, aaaa, contm, dtotal, d1
    carácter : día
    lógico    : correcta, bisiesto
inicio
    repetir
        bisiesto ← falso
        correcta ← verdad
        escribir('Deme fecha (dd mm aaaa) ')
        leer(dd, mm, aaaa)
        si (aaaa mod 4 = 0) y (aaaa mod 100 <> 0) o
            (aaaa mod 400 = 0) entonces
            bisiesto ← verdad
        fin_si
        según_sea mm hacer
            1, 3, 5, 6, 8, 10, 12:
                si dd > 31 entonces
                    correcta ← falso
                fin_si
            4, 7, 9, 11:
                si dd > 30 entonces
                    correcta ← falso
                fin_si
            2:
                si bisiesto entonces
                    si dd > 29 entonces
                        correcta ← falso
                    fin_si
                si_no
                    si dd > 28 entonces
                        correcta ← falso
                    fin_si
                fin_si
            si_no
                correcta ← falso
        fin_según
    hasta_que correcta
    dtotal ← 0
desde contm _ 1 hasta mm - 1 hacer
    según_sea contm hacer
        1, 3, 5, 6, 8, 10, 12:
            dtotal ← dtotal + 31
        4, 7, 9, 11:
            dtotal ← dtotal + 30
    2:
```

```
    si bisiesto entonces
        dtotal ← dtotal + 29
    si_no
        dtotal ← dtotal + 28
    fin_si
    fin_según
fin_desde
dtotal ← dtotal + dd
repetir
    escribir('Deme día de la semana que fue el 1
            ° de ENERO (l/m/x/j/v/s/d) ')
    leer(día)
    según_sea día hacer
        'l','L':
            d1 ← 0
        'm','M':
            d1 ← 1
        'x','X':
            d1 ← 2
        'j','J':
            d1 ← 3
        'v','V':
            d1 ← 4
        's','S':
            d1 ← 5
        'd','D':
            d1 ← 6
    si_no
        d1 ← -10
    fin_según
hasta_que (d1 >= 0) y (d1 <= 6)
dtotal ← dtotal + d1
según_sea dtotal mod 7 hacer
1:
    escribir('Lunes')
2:
    escribir('Martes')
3:
    escribir('Miércoles')
4:
    escribir('Jueves')
5:
    escribir('Viernes')
6:
    escribir('Sábado')
0:
    escribir('Domingo')
fin_según
fin
```


5

SUBPROGRAMAS (SUBALGORITMOS), PROCEDIMIENTOS Y FUNCIONES

@UfYgc i WDE'XY'dfcVYa UgWa d'Y^cg'gY'ZUW]hUWbgjXYfUVYa YbhYg]gYXj]XYbYb'dfcVYa Ug'a zg dYei Y cg'fgi VdfcVYa Ugl" @Ugc i WDE'XY'Yghcg'gi VdfcVYa Ug'gY'fYU]nUWb'gi VU[cf]ha cgzei Y gcb i b]XUXYg'XY'dfc[fUa Uc'a CXi'cg'XlgY UXcg'dUfUY'YWhUF'U[i bUhUFYUYgdYVZ]W midi YXYb'gYf'XY Xcg'h]dcg 'Zi bWcbYg midfcWXja]Ybhcg" 9b'YghY'Wdjh'i 'c gY'XYgW]VYb' 'Ug'Zi bWcbYg'mdfcWXja]Yb! hcg' 'i bhc 'Wb' 'cg'WbWdhcg'XY'dUfza Yhf cg'miXYj Uf'UV'Yg' 'c WYg'm['c VUYg"GY'jblfcXl W'hUa V]fb Y'WbWdhc'XY'fYwfglj]XUXWa c' i bUbi Yj U\YffUa]YbhU XY'fYgc i WDE'XY'dfcVYa Ug"

5.1. PROGRAMACIÓN MODULAR

El diseño descendente resuelve un problema efectuando descomposiciones en otros problemas más sencillos a través de distintos niveles de refinamiento. La programación modular consiste en resolver de forma independiente los subproblemas resultantes de una descomposición.

La *programación modular* completa y amplía el diseño descendente como método de resolución de problemas y permite proteger la estructura de la información asociada a un subproblema.

Cuando se trabaja de este modo, existirá un algoritmo principal o conductor que transferirá el control a los distintos módulos o subalgoritmos, los cuales, cuando terminen su tarea, devolverán el control al algoritmo que los llamó. Los *módulos* o *subalgoritmos* deberán ser pequeños, seguirán todas las reglas de la programación estructurada y podrán ser representados con las herramientas de programación habituales.

El empleo de esta técnica facilita notoriamente el diseño de los programas. Algunas ventajas significativas son:

- Varios programadores podrán trabajar simultáneamente en la confección de un algoritmo, repartiéndose las distintas partes del mismo, ya que los módulos son independientes.
- Se podrá modificar un módulo sin afectar a los demás.
- Las tareas, subalgoritmos, sólo se escribirán una vez, aunque se necesiten en distintas ocasiones en el cuerpo del algoritmo.

Existen dos tipos de subalgoritmos: *funciones* y *procedimientos*.

5.2. FUNCIONES

Una *función* toma uno o más valores, denominados *argumentos* o *parámetros actuales* y, según el valor de éstos, devuelve un *resultado* en el nombre de la función. Para *invocar* a una función se utiliza su nombre seguido por los parámetros actuales o reales entre paréntesis en una expresión. Es decir que se podrá colocar la llamada a una función en cualquier instrucción donde se pueda usar una expresión. Por ejemplo:

```
escribir (raíz2 (16) )
```

y si la función se denomina *f* y sus parámetros son *p1*, *p2* y *p3*

```
escribir (f (p1 , p2 , p3) )
```

Cada lenguaje de programación tiene sus propias funciones incorporadas, que se denominan internas o intrínsecas. Se considerarán como internas únicamente las más básicas y comunes a casi todos los lenguajes y se irán comentando a lo largo del libro en los capítulos adecuados, es decir cuando para explicar el tema se necesite una referencia a alguna de ellas.

Si las funciones estándar no permiten realizar el tipo de cálculo deseado será necesario recurrir a las funciones externas, que definiremos mediante una declaración de función.

5.2.1. Declaración de funciones

Las funciones, como subalgoritmos que son, tienen una constitución similar a los algoritmos. Por consiguiente, una función constará de:

- *Cabecera*, con la definición de la función.
- *Cuerpo* de la función.

Dentro del cuerpo de la función estará el *bloque de declaraciones* y el *bloque de instrucciones*. En este bloque se debe incluir una instrucción *devolver* que recibe un valor para devolverlo al algoritmo llamador.

Para que las acciones descritas en una función sean ejecutadas se necesita que ésta sea invocada, y se le proporcionen los argumentos necesarios para realizar esas acciones. En la definición de la función deberán figurar una serie de parámetros, denominados *parámetros formales* y en la llamada a la función se establece una correspondencia uno a uno y de izquierda a derecha entre los parámetros actuales y los formales. En el cuerpo de la función se utilizarán los parámetros formales cuando se quiera trabajar con información procedente del programa llamador. El pseudocódigo correspondiente a una función es:

```
<tipo_de_dato> función <nombre_función>(<lista_de_parámetros_formales>
    [declaraciones locales]
    inicio
        .....
        .....
        devolver (<expresión>)
    fin_función
```

La *lista_de_parámetros_formales* estará formada por una o más sublistas de parámetros de la siguiente forma:

{**E|S|E/S**}<tipo_de_dato>:<nombre_de_parámetro_formal>...

Las llaves representan la selección de una entre las distintas opciones que aparecen separadas por barra. En las funciones esta opción habitualmente será **E**. Todo esto se detallará más adelante. Los corchetes indican no obligatoriedad. El tipo de dato debe ser estándar o definido por el programador previamente. Se pueden separar distintos tipos de parámetros utilizando el punto y coma (;) entre cada declaración.

5.3. PROCEDIMIENTOS

Un procedimiento es un subalgoritmo que realiza una tarea específica y que puede ser definido con 0, 1 o N parámetros. Tanto la entrada de información al procedimiento como la devolución de resultados desde el procedimiento al programa llamador se realizarán a través de los parámetros. El nombre de un procedimiento no está asociado a ninguno de los resultados que obtiene.

La *invocación a un procedimiento* se realiza con una instrucción **llamar_a** o bien directamente con el nombre del procedimiento. Es decir:

```
[llamar_a]<nombre_procedimiento>[(lista_de_parámetros_actuales)]
```

No existe obligatoriedad en los parámetros actuales.

5.3.1. Declaración de procedimientos

La *declaración de un procedimiento* es similar a la de una función. Las pequeñas diferencias son debidas a que el nombre del procedimiento no se encuentra asociado a ningún resultado. La declaración de un procedimiento expresada en pseudocódigo es:

```
procedimiento <nombre_procedimiento>[(lista_de_parámetros_formales)]
[declaraciones locales]
inicio
    .....
    .....
    .....
fin_procedimiento
```

La *lista_de_parámetros_formales* estará formada por una o más sublistas de parámetros con el siguiente formato:

```
{E|S|E/S}<tipo_de_dato>:<nombre_de_parámetro_formal>...
```

y seguiría la mismas reglas que la declaración de parámetros en las funciones.

En el algoritmo principal las declaraciones de procedimientos y funciones, se situarán al final, al objeto de agilizar la escritura de algoritmos.

5.4. ESTRUCTURA GENERAL DE UN ALGORITMO

Aunque algunos lenguajes de programación exigen que los procedimientos y funciones se escriban antes de realizar la llamada, por claridad en la escritura del pseudocódigo se escriben después del programa principal.

```
algoritmo <nombre_algoritmo>
const
```

```

<nombre_de_constante1>=valor1
.....
tipo
  <clase>:<nombre_del_tipo>      // Se verá en capítulos posteriores
var
  <nombre_del_tipo>:<nombre_de_variable1>[,<nombre_de_variable2>,...]
.....
//Los datos han de ser declarados antes de poder ser utilizados
inicio
  <acción1>
  <acción2>
  llamar_a<nombre_de_procedimiento>[(lista_de_parámetros_actuales)]
  // la llamada a la función ha de realizarse en una expresión
  escribir(<nombre_de_función> (lista_de_parámetros_actuales))
  //Se utilizará siempre la sangría en las estructuras
  //selectivas y repetitivas.
  .....
  <acciónN>
fin

procedimiento <nombre_procedimiento>[(lista_de_parámetros_formales)]
[ declaraciones locales ]
inicio
  ...
  ...
  ...
fin_procedimiento

<tipo_de_dato> función <nombre_función>(lista_de_parámetros_formales)
// el <tipo_de_dato> devuelto por la función es un tipo estándar
[ declaraciones locales ]
inicio
  .....
  .....
  devolver(<expresión>)
fin_función

```

5.5. PASO DE PARÁMETROS

Cuando un programa llama a un procedimiento o función se establece una correspondencia entre los parámetros actuales y los formales. Existen dos formas para establecer la correspondencia de parámetros:

- **Posicional.** Emparejando los parámetros reales y formales según su posición en las listas. Esto requiere que ambas listas tengan el mismo número de parámetros y que los que se van a emparejar coincidan en el tipo. En la definición del subprograma deberá reflejarse siempre de qué tipo es cada uno de los parámetros formales.

(E <tipo_de_dato>:<nombre1_de_parámetro_formal>...)

El tipo de dato debe ser estándar o haber sido definido de antemano. Si los parámetros formales se separan por comas es necesario, aunque no suficiente, que tengan el mismo tipo. Si su tipo fuera distinto habría que poner:

```
(E <tipo_de_dato1>: <nombre1_de_parámetro_formal>;
  E <tipo_de_dato2>: <nombre2_de_parámetro_formal>)
```

- **Correspondencia por el nombre explícito.** En las llamadas se indica explícitamente la correspondencia entre los parámetros reales y formales.

Dado que la mayor parte de los lenguajes usan exclusivamente la correspondencia posicional, éste será el método que se seguirá en la mayoría de los algoritmos.

Al hablar de los procedimientos se decía que devuelven resultados al programa principal a través de los parámetros, pero que también pueden recibir información, desde el programa principal, a través de ellos. Esto nos lleva a una clasificación de los parámetros en:

Parámetros de entrada	Permiten únicamente la transmisión de información desde el programa llamador al subprograma.
Parámetros de salida	Sólo devuelven resultados.
Parámetros de entrada/salida	Actúan en los dos sentidos, tanto mandando valores al subprograma, devolviendo resultados desde el subprograma al programa llamador.

En los algoritmos, se debe especificar en la definición del subprograma el comportamiento de cada uno de los parámetros. Para ello se empleará la siguiente terminología:

- **E** equivaldría a parámetro de entrada.
- **S** querrá decir parámetro de salida.
- **E/S** parámetro de entrada/salida.

En la lista de parámetros siguiente

```
(E <tipo_de_dato1>: <nombre1_de_parámetro_formal> ;
  S <tipo_de_dato1>: <nombre2_de_parámetro_formal>)
```

<nombre1_de_parámetro_formal> es parámetro de entrada y va a proporcionar datos al subprograma.

<nombre2_de_parámetro_formal> es parámetro de salida y devolverá resultados al programa llamador.

Aunque ambos son del mismo tipo, *<tipo_de_dato1>*, habrá que repetir el tipo para cada uno de ellos y no se escriben separados por coma, ya que uno es de *entrada* y otro de *salida*.

Estas características afectarán tanto a procedimientos como a funciones. Por estas circunstancias una función va a tener la posibilidad de devolver valores al programa principal de dos formas:

- *Como valor de la función.*
- *A través de los parámetros.*

Un procedimiento sólo podrá devolver resultados a través de los parámetros, de modo que al codificar el algoritmo se ha de tener mucho cuidado con el paso de parámetros, siendo preciso conocer los métodos de transmisión que posee el lenguaje, para poder conseguir el funcionamiento deseado. Los lenguajes suelen disponer de:

Paso por valor

Los parámetros formales correspondientes reciben una copia de los valores de los parámetros actuales; por tanto los cambios que se produzcan en ellos por efecto del subprograma no podrán afectar a los parámetros actuales y no se devolverá información al programa llamador. Los parámetros resultarían de entrada, **E**.

Paso por valor resultado	Al finalizar la ejecución del subprograma los valores de los parámetros formales se transfieren o copian a los parámetros actuales.
Paso por referencia	Lo que se pasa al procedimiento es la dirección de memoria del parámetro actual. De esta forma, una variable pasada como parámetro actual es compartida; es decir, se puede modificar directamente por el subprograma. Los parámetros serían de entrada/salida, E/S.

Es posible pasar como parámetros datos y subprogramas.

5.6. VARIABLES GLOBALES Y LOCALES

Una variable es *global* cuando el ámbito en el que dicha variable se conoce es el programa completo. Se consideran como variables globales aquellas que hayan sido declaradas en el programa principal y como locales las declaradas en el propio subprograma.

Toda variable que se utilice en un procedimiento debe haber sido declarada en el mismo. De esta forma todas las variables del procedimiento serán locales y la comunicación con el programa principal se realizará exclusivamente a través de los parámetros. Al declarar una variable en un procedimiento no importa que ya existiera otra con el mismo nombre en el programa principal; ambas serán distintas y, cuando nos encontramos en el procedimiento, sólo tendrá vigencia la declaración que hayamos efectuado en él. Trabajando de esta forma obtendremos la independencia de los módulos.

5.7. RECURSIVIDAD

Un objeto es recursivo si forma parte de sí mismo o interviene en su propia definición. El instrumento necesario para expresar los programas recursivamente es el subprograma. La mayoría de los lenguajes de programación admiten que un procedimiento o función haga referencia a sí mismo dentro de su definición, denominada recursividad directa.

```
//procedimiento recursivo
procedimiento proc1(lista_de_parámetros_formales)
...
inicio
...
// instrucción/es que establece/n una condición de salida
...
proc1(lista_de_parámetros_actuales)
...
fin_procedimiento

//función recursiva
<tipo_de_dato> función func1(lista_de_parámetros_formales)
...
inicio
...
// instrucción/es que establece/n una condición de salida
...
devolver(...func1(lista_de_parámetros_actuales)...)
fin_función
```

La recursión se puede considerar como una alternativa a la iteración y, aunque las soluciones iterativas están más cercanas a la estructura de la computadora, resulta muy útil cuando se trabaja con

problemas o estructuras, como los árboles, definidos en modo recursivo. Por ejemplo la definición matemática del factorial de un número n como

$$n! = n * (n-1) !$$

es una definición recursiva. El problema general se resuelve en términos recursivos, hasta llegar a un caso que se resuelve de forma no recursiva.

$$0! = 1$$

y existe un acercamiento paulatino al caso no recursivo.

Así una función recursiva de factorial es:

```
entero factorial(E entero: n)
  inicio
    si (n = 1) entonces
      devolver (1)
    si_no
      devolver (n * factorial(n - 1))
    fin_si
  fin_función
```

Para comprender la recursividad se deben tener en cuenta las premisas:

- Un método recursivo debe establecer la condición o condiciones de salida.
- Cada llamada recursiva debe aproximar hacia el cumplimiento de la/las condiciones de salida.
- Cuando se llama a un procedimiento o función los parámetros y las variables locales toman nuevos valores, y el procedimiento o función trabaja con estos nuevos valores y no con los de anteriores llamadas.
- Cada vez que se llama a un procedimiento o función los parámetros de entrada y variables locales son almacenados en las siguientes posiciones libres de memoria y cuando termina la ejecución del procedimiento o función son accedidos en orden inverso a como se introdujeron.
- El espacio requerido para almacenar los valores crece conforme a los niveles de anidamiento de las llamadas.
- La recursividad puede ser directa e indirecta. La recursividad indirecta se produce cuando un procedimiento o función hace referencia a otro el cual contiene, a su vez, una referencia directa o indirecta al primero.

Todo algoritmo recursivo puede ser convertido en iterativo, aunque, en ocasiones, para ello se requerirá la utilización de pilas donde almacenar ciertos datos.

5.8. EJERCICIOS RESUELTOS

5.1. Realizar un procedimiento que permita intercambiar el valor de dos variables.

Análisis del Problema

Para intercambiar el contenido de dos variables, es necesaria una variable auxiliar, del mismo tipo de datos que las otras variables. Se pasan como parámetros de entrada las dos variables cuyo valor se desea intercam-

biar. Ya que su valor será modificado durante la ejecución del subalgoritmo, serán parámetros de entrada/salida y el subalgoritmo será un procedimiento.

Diseño del algoritmo

```
procedimiento Intercambio(E/S entero : a,b)
var
    entero : aux
inicio
    aux ← a
    a ← b
    b ← aux
fin_procedimiento
```

Este procedimiento sólo serviría para intercambiar variables de tipo entero. Se podría hacer un procedimiento más genérico utilizando un tipo de datos abstracto. En la cabecera del programa principal podemos definir un tipo de datos **TipoDatos** de la forma

```
tipo
    TipoDatos = .... // tipo de datos de las variables
                      // a intercambiar
```

y modificar la cabecera del procedimiento,

```
procedimiento Intercambio(E/S TipoDatos : a,b)
var
    TipoDatos: aux
```

5.2. Realizar una función que permita obtener el término n de la serie de Fibonacci.

Análisis del problema

La serie de Fibonacci se define como:

$$\begin{aligned} \text{Fibonacci}_n &= \text{Fibonacci}_{n-1} + \text{Fibonacci}_{n-2} && \text{para todo } n > 2 \\ \text{Fibonacci}_n &= 1 && \text{para } n = 2 \\ \text{Fibonacci}_n &= 1. && \text{para } n = 1 \end{aligned}$$

Por lo tanto se deben sumar los elementos mediante un bucle que debe ejecutarse desde 3 hasta n. Cada iteración debe guardar el último y el penúltimo término, para lo que se utilizan dos variables **último** y **penúltimo**, a las que irán cambiando sus valores. El subalgoritmo aceptará como entrada una variable entera y devolverá un valor también entero, por lo que deberemos utilizar una función.

Diseño del algoritmo

```
entero función Fibonacci(E entero : n)
var
    entero : i, último, penúltimo, suma
inicio
    suma ← 1
    último ← 1
    penúltimo ← 1
    desde i ← 3 hasta n hacer
```

```

penúltimo ← último
último ← suma
suma ← último + penúltimo
fin_desde
devolver(suma)
fin_función

```

- 5.3.** Implementar una función que permita devolver un valor entero, leído desde teclado, comprendido entre dos límites que introduciremos como parámetro.

Análisis de problema

Esta función puede ser útil para validar una entrada de datos de tipo entero y se podrá incluir en otros algoritmos. Consistirá simplemente en un bucle **repetir** que ejecutará la lectura de un dato hasta que esté entre los valores que se han pasado como parámetros de entrada.

Diseño del algoritmo

```

entero función ValidarEntero(E entero : inferior, superior)
var
    entero : n
inicio
    repetir
        leer(n)
        hasta_que (n >= inferior) y (n <= superior)
        devolver(n)
fin_función

```

Los parámetros **inferior** y **superior** deberán pasarse en dicho orden; es decir, primero el menor y luego el mayor, aunque con una pequeña modificación podrían pasarse en cualquier orden. Para ello debería incluirse una condición antes de comenzar el bucle que, de ser necesario, haría una llamada al procedimiento **Intercambio**, ya desarrollado:

```

.
.
si inferior > superior entonces
    Intercambio(inferior, superior)
fin_si
.
.
```

- 5.4.** Diseñar una función que permita obtener el valor absoluto de un número.

Análisis del problema

El valor absoluto de un número positivo, sería el mismo número; de un número negativo sería el mismo número sin el signo y de 0 es 0. Por lo tanto, esta función, únicamente debería multiplicar por -1 el número pasado como parámetro de entrada en el caso que éste fuera menor que 0.

Diseño del algoritmo

```

entero función Abs(E entero : n)
inicio

```

```

    si n < 0 entonces
        devolver(n * -1)
    si_no
        devolver(n)
    fin_si
fin_función

```

Esta función sólo es válida para números enteros. Para que valiera con algún otro tipo de dato numérico, deberíamos utilizar un tipo de dato abstracto.

5.5. Realizar un procedimiento que obtenga la división entera y el resto de la misma utilizando únicamente los operadores suma y resta.

Análisis del problema

La división se puede considerar como una sucesión de restas. El algoritmo trata de contar cuántas veces se puede restar el divisor al dividendo y dicho contador sería el cociente. Cuando ya no se pueda restar más sin que salga un número positivo, se tendrá el resto.

Por lo tanto se tienen dos parámetros de entrada, dividendo y divisor, y dos de salida, cociente y resto. cociente será un contador que se incrementará en 1 cada vez que se pueda restar el divisor al dividendo. El bucle a utilizar será de tipo **mientras**, ya que puede darse el caso que no se ejecute ninguna vez, en cuyo caso el cociente será 0 y el resto será el dividendo.

Diseño del algoritmo

```

procedimiento DivisiónEntera( E entero : dividendo, divisor;
                                S entero : cociente, resto)
inicio
    cociente ← 0
    mientras dividendo => divisor hacer
        dividendo ← dividendo - divisor
        cociente ← cociente + 1
    fin_mientras
    resto ← dividendo
fin_procedimiento

```

5.6. Diseñar un procedimiento que permita convertir coordenadas polares (radio, ángulo) en cartesianas (x,y)

$$\begin{aligned} x &= \text{radio} * \cos(\text{ángulo}) \\ y &= \text{radio} * \sin(\text{ángulo}) \end{aligned}$$

Análisis del problema

La resolución requiere aplicar la fórmula indicada más arriba. Habrá que tener en cuenta el tipo de parámetros. radio y ángulo serán de entrada y x e y de salida.

Diseño del algoritmo

```

procedimiento Polares( E real : ángulo, radio; S real : x, y)
inicio
    x _ radio * cos(ángulo)
    y _ radio * sen(ángulo)
fin_procedimiento

```

- 5.7.** Diseñe una función que permita obtener el factorial de un número entero positivo.

Análisis del problema

El factorial de n se puede definir para cualquier entero positivo como

$$\text{Factorial}_n = n * n-1 * n-2 * \dots * 1 \\ \text{por definición, } \text{Factorial}_0 = 1$$

Por lo tanto para implementar un función Factorial, se deberá realizar un bucle que se ejecute entre 2 y n , acumulando en su cuerpo las sucesivas multiplicaciones.

Diseño del algoritmo

```
entero función Factorial(E entero : n)
var
    entero : i, f
inicio
    f ← 1
    desde i ← 2 hasta n hacer
        f ← f * i
    fin_desde
    devolver(f)
fin_función
```

- 5.8.** Diseñar una función que permita obtener el máximo común divisor de dos números mediante el algoritmo de Euclides.

Análisis del problema

Para obtener el máximo común divisor de dos números enteros positivos a y b según el algoritmo de Euclides, se debe ejecutar un bucle que divida a entre b . Si el resto es 0, b será el divisor; en caso contrario, a tomará el valor de b y b el del resto de la división anterior. El bucle finalizará cuando el resto sea 0, es decir, cuando a sea divisible entre b .

Diseño del algoritmo

```
entero función Mcd (E entero : a,b)
var
    entero : resto
inicio
    mientras a mod b <> 0 hacer
        resto ← a mod b
        a ← b
        b ← resto
    fin_mientras
    devolver(b)
fin_función
```

5.9. Realizar una función que permita saber si una fecha es válida.

Análisis del problema

Esta función es muy normal en cualquier aplicación informática. Se trata de ver si una fecha, introducida como *mes, día y año* es una fecha correcta; es decir, el día ha de estar comprendido entre 1 y 31, el mes entre 1 y 12, y, si esto es correcto, ver si el número de días para un mes concreto es válido.

El tipo de la función será un valor lógico, verdadero si la fecha es correcta o falso si es errónea. Como ayuda, se utilizará una función *Esbisiesto*, a la que se pasará el valor entero correspondiente a un año determinado y devolverá un valor lógico verdadero si el año es bisiesto y falso en caso contrario. Un año será bisiesto si es divisible por 4, excepto los que son divisibles por 100 pero no por 400, es decir, menos aquellos con los que comienza el siglo. Un diseño modular de la función podría ser por tanto:



Diseño del algoritmo

```

lógico función FechaVálida(E entero : dd,mm,aa)
inicio
    FechaVálida ← verdad
    si (mm < 1) o (mm > 12) entonces
        devolver(falso)
    si_no
        si dd < 1 entonces
            devolver(falso)
        si_no
            según_sea mm hacer
                4,6,9,11 : si dd > 30 entonces
                    devolver(falso)
                fin_si
                2 : si Esbisiesto(aa) y (dd > 29) entonces
                    devolver(falso)
                si_no
                    si_no Esbisiesto(aa) y (dd > 28) entonces
                        devolver(falso)
                    fin_si
                si_no

```

```

    si dd > 31 entonces
        devolver(falso)
    fin_si
    fin_según
    fin_si
    fin_si
fin_función

lógico función EsBisiesto(E entero : aa)
inicio
    devolver((aa mod 4 = 0) y (aa mod 100 <> 0) o (aa mod 400 = 0))
fin_función

```

5.10. Implementar una función que permita hallar el valor de X^y , siendo X un número real e y un entero.

Análisis del problema

Se trata de una función similar a la del factorial, pues se trata de acumular multiplicaciones, pues debemos multiplicar X por sí mismo y veces. Si y es negativo, X^y es igual a su inversa.

Diseño del algoritmo

```

entero función Potencia( E entero : x,y)
var
    entero : i, p
inicio
    p ← 1
    desde i ← 1 hasta abs(y) hacer
        p ← p * x
    fin_desde
    si y < 0 entonces
        devolver(p)
    si_no
        devolver(1/p)
    fin_si
fin_función

```

5.11. Realizar tres funciones que permitan hallar el valor de p mediante las series matemáticas siguientes:

$$a) \pi = 4 \sum_{i=1}^{\infty} \frac{(-1)^i}{2i+1} = 4 \left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \dots \right)$$

$$b) \pi = \frac{1}{2} \sum_{i=1}^{\infty} \sqrt{\frac{(24)}{(i)^2}} = \frac{1}{2} \sqrt{24 + \frac{24}{2^2} + \frac{24}{3^2} + \frac{24}{4^2} + \frac{24}{5^2} + \dots}$$

$$c) \pi = 4 \cdot \frac{2}{3} \cdot \frac{4}{3} \cdot \frac{4}{5} \cdot \frac{6}{5} \cdot \frac{6}{7} \cdot \frac{8}{7} \cdot \dots$$

La precisión del cálculo dependerá del número de elementos de la serie, n, que será un parámetro que se pase a la función.

Análisis del problema

En cualquiera de los tres casos se debe implementar un bucle que se ejecute n veces y en el que se irá acumulando la suma de los términos de la serie.

En el caso (a), los términos pares suman y los impares restan, por lo que será preciso hacer una distinción. El numerador siempre es 1, y el denominador son los n primeros números impares. En el caso (b), el numerador es 24 y el denominador será el cuadrado de la variable del bucle. El resultado final será la raíz cuadrada de la serie.

Diseño del algoritmo

(a)

```
real función Pi(E entero : n)
var
    real : serie
    entero : i, denom
    lógico : par
inicio
    denom ← 1
    serie ← 0
    par ← verdad
    desde i ← 1 hasta n hacer
        si par entonces
            serie ← serie - 1/denom
        si_no
            serie ← serie + 1/denom
        fin_si
        par ← no par
        denom ← denom + 2
    fin_desde
    devolver(serie * 4)
fin_función
```

(b)

```
real función Pi(E entero : n)
var
    real : serie
    entero : i
inicio
    serie ← 0
    desde i ← 1 hasta n hacer
        serie ← serie + 24/i**2
    fin_desde
    devolver(raíz2(serie/2))
fin_función
```

(c)

```
real función Pi(E entero : n)
var
    real : serie
    entero : i
inicio
```

```

serie ← 2
desde i ← 1 hasta n hacer
    si i mod 2 = 0 entonces
        serie ← serie * i/(i + 1)
    si_no
        serie ← serie * (i + 1)/i
    fin_si
fin_desde
devolver(serie)
fin_función

```

5.12. Realizar un subprograma que calcule la suma de los divisores de n distintos de n .

Análisis del problema

Para obtener los divisores del número entero n , será preciso hacer un bucle en el que un contador se irá decrementando desde $n-1$ hasta 1. Por cada iteración, se comprobará si el contador es divisor de n . Como el primer divisor que podemos encontrar será $n/2$, el valor inicial del contador podrá ser $n \text{ div } 2$.

El tipo de subprograma deberá ser una función, a la que pasariamos como parámetro de entrada el número n .

Diseño del algoritmo

```

entero función SumaDivisor(E entero : n)
var
    entero : suma,i
inicio
    suma ← 0
    desde i ← n div 2 hasta 1 incremento -1 hacer
        si n mod i = 0 entonces
            suma ← suma + i
        fin_si
    fin_desde
    devolver(suma)
fin_función

```

5.13. Dos números son amigos, si cada uno de ellos es igual a la suma de los divisores del otro.

Por ejemplo, 220 y 284 son amigos, ya que:

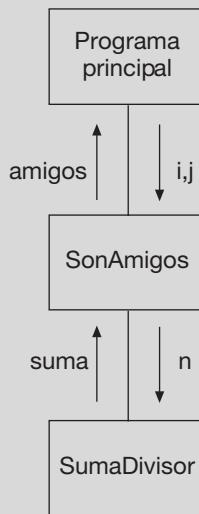
$$\text{Suma de divisores de } 284 : 1 + 2 + 4 + 71 + 142 = 220$$

$$\text{Suma de divisores de } 220 : 1 + 2 + 4 + 5 + 10 + 11 + 20 + 22 + 44 + 55 + 110 = 284$$

Diseñe un algoritmo que muestre todas las parejas de números amigos menores o iguales que m , siendo m un número introducido por teclado.

Análisis del problema

Este algoritmo se puede descomponer en tres partes diferenciadas. Por una parte un programa principal que debe ir sacando todas las parejas de números mayores o iguales a un número m que previamente habremos introducido por teclado. Desde ahí se llamará a una función lógica `SonAmigos` que dirá si la pareja de números son amigos. Para hacer esto, se calcula la suma de los divisores, para lo que llamaremos a la función `SumaDivisor`, desarrollada más arriba. Por lo tanto un diseño modular del programa podría ser como sigue:



El programa principal leerá el número m , e implementará dos bucles anidados para sacar las parejas menores o iguales que m . Para ello, un primer bucle realizará la iteraciones de i entre 1 y m , mientras que el bucle interno irá tomando valores entre $i+1$ y m . Dentro del bucle habrá una llamada a la función `SonAmigos`, y si ésta devuelve un valor verdadero escribiremos la pareja. La función `SonAmigos` se encarga de comprobar si una pareja de números son amigos. Recibe dos parámetros de entrada, y guarda en dos variables la suma de sus divisores mediante la función `SumaDivisor`. Si ambas sumas son iguales, devolverá un valor verdadero; en caso contrario, falso.

Diseño del algoritmo

```

algoritmo ejercicio_5_13
var
    entero : i,j,m
inicio
    leer(m)
    desde i ← 1 hasta m-1 hacer
        desde j ← i+1 hasta m hacer
            si SonAmigos(i,j) entonces
                escribir(i,j)
            fin_si
        fin_desde
    fin_desde
fin
lógico función SonAmigos(E entero : n,m)
inicio
    devolver((SumaDivisor(n) = m) y (SumaDivisor(m) = n))
fin_función
    
```

5.14. El número de combinaciones de m elementos tomados de n en n es:

$$\left(\frac{m}{n} \right) = \frac{m!}{n!(m-n)}$$

Diseñar una función que permita calcular el número combinatorio $\left(\frac{m}{n} \right)$

Análisis del problema

La función se reduce a una simple asignación, siempre que se tenga resuelto el problema de hallar el factorial, cuya función ya se ha diseñado más arriba. Por lo tanto se limitará a codificar la expresión del número combinatorio.

Diseño del algoritmo

```
entero función Combinatorio(E entero : m,n)
inicio
    devolver(Factorial(m) div Factorial(n) * Factorial(m - n))
fin_función
```

5.15. Implemente tres funciones que permitan averiguar los valores de e^x , $\cos(x)$ y $\operatorname{sen}(x)$ a partir de las series siguientes:

$$e^x = \sum_{i=1}^n \frac{x^i}{i!} = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots$$

$$\cos(x) = 1 + \sum_{i=1}^n (-1) \frac{x^{2i}}{(2i)!} = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \dots$$

$$\operatorname{sen}(x) = \sum_{i=1}^n (-1) \frac{x^{2i+1}}{(2i+1)!} = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots$$

El número de términos de la serie será el suficiente para que la diferencia absoluta entre dos valores sucesivos sea menor 10^{-3} .

Análisis del problema

En ambos casos se trata de hacer un bucle que se ejecute hasta que el valor absoluto de la diferencia entre el último y el penúltimo término sea menor que 0.001. Dentro del bucle habrá un acumulador que sume cada uno de los términos.

Para el caso de e^x , en cada término el numerador será x elevado al número de orden del término, mientras que el denominador será el factorial del número de orden del término. El contador del bucle por lo tanto irá tomando valores entre 1 y n.

Para $\cos(x)$, el numerador irá elevando x a las potencias pares, mientras que el denominador será el factorial de los números pares. Por lo tanto el contador del bucle irá sacando los números pares. Además, en él los términos de orden par restan, mientras que los de orden impar suman.

Por último, para $\operatorname{sen}(x)$, la diferencia estriba en que la potencia y el factorial serían de los números impares, por lo que el contador del bucle sacará los números impares.

En los tres casos, como se ha dicho más arriba, tendremos que utilizar una variable suma que acumulará el valor de la serie, un contador, con las características que se han indicado y dos variables reales para guardar el último y el penúltimo término, para ver si la diferencia entre ambos es menor que 0.001.

Diseño del algoritmo e^x

```
real función exponente(E entero : x)
var
    entero : i
    real : suma, último, término
inicio
    suma ← 1 + x
    i ← 1
    término ← x
repetir
    i ← i + 1
    último ← término
    término ← x**i/Factorial(i)
    suma ← suma + término
hasta que abs(término - último) < 0.001
devolver(suma)
fin función
```

 $\cos(x)$

```
real función cos(E entero : x )
var
    entero : i
    real : suma, último, término
    lógico :par
inicio
    suma ← 1
    término ← 1
    i ← 0
    par ← verdad
repetir
    i ← i + 2
    último ← término
    término ← x**i/Factorial(i)
    si par entonces
        suma ← suma - término
    si_no
        suma ← suma + término
    fin_si
    par ← no par
hasta que abs(término - último) < 0.001
devolver(suma)
fin función
```

 $\operatorname{sen}(x)$

```
real función sen(E entero : x)
var
    entero : i
    real : suma, último, término
```

```

lógico : par
inicio
    suma ← x
    término ← x
    i ← 1
    par ← verdad
    repetir
        i ← i + 2
        último ← término
        término ← x**i/Factorial(i)
        si par entonces
            suma ← suma - término
        si_no
            suma ← suma + término
        fin_si
        par ← no par
    hasta_que abs(término - último) < 0.001
    devolver(suma)
fin_función

```

5.16. Implementar una función Redondeo (*a*, *b*), que devuelva el número real *a* redondeado a *b* decimales.

Análisis del problema

Para redondear un número real *a* a *b* decimales —siendo *b* un número entero positivo—, hay que recurrir a la función estándar **ent** que elimina los decimales de un número real. Si se quiere dejar una serie de decimales, hay que obtener el entero de la multiplicación del número *a* por 10^b y dividir el resultado por 10^b . De esta forma, el resultado, siendo *a* = 3,546 y *b* = 2, sería:

$$\mathbf{ent}(3.546 \cdot 100) / 100 = \mathbf{ent}(354.6) / 100 = 354 / 100 = 3.54$$

Con esto se consigue truncar, no redondear. Para redondear, de forma que hasta 0.5 redondee por defecto y a partir de ahí por exceso, se debe sumar a la multiplicación 0.5:

$$\begin{aligned} \mathbf{ent}(3.546 \cdot 100 + 0.5) / 100 &= \mathbf{ent}(354.6 + 0.5) / 100 = \\ \mathbf{ent}(355.1) / 100 &= 355 / 100 = 3.55 \end{aligned}$$

Para realizar esta función, únicamente se ha de aplicar la expresión anterior.

Diseño del algoritmo

```

entero función Redondeo(E real : a; E entero : b)
inicio
    devolver(ent(a * 10**b + 0.5) / 10**b)
fin_función

```

5.17. Algoritmo que transforma un número introducido por teclado en notación decimal a romana.
El número será entero y positivo y no excederá de 3000.

Análisis del problema

Para pasar un número desde notación decimal a romana se transformará individualmente cada uno de sus dígitos. El procedimiento para transformar un dígito es siempre el mismo

Dígito	Escribir
de 1 a 3	de 1 a 3 veces p1
4	p1 seguido de p2
de 5 a 8	p2 seguido por, de 0 a 3 veces, p1
9	p1 y a continuación p3

Tenga en cuenta que, en las distintas llamadas que se realizan al procedimiento, se pasarán diferentes parámetros:

	Parámetro 1 (p1)	Parámetro 2 (p2)	Parámetro 3 (p3)
Unidades	'I'	'V'	'X'
Decenas	'X'	'L'	'C'
Centenas	'C'	'D'	'M'
Miles	'M'	' '	' '

Diseño del algoritmo

```

algoritmo Ejercicio_5_17
var
    entero : n, r, dígito
inicio
    repetir
        escribir('Deme número')
        leer(n)
        hasta que (n >= 0) y (n <= 3000)
        r ← n
        dígito ← r div 1000
        r ← r mod 1000
        calccifrarom(dígito, 'M', ' ', ' ')
        dígito ← r div 100
        r ← r mod 100
        calccifrarom(dígito, 'C', 'D', 'M')
        dígito ← r div 10
        r ← r mod 10
        calccifrarom(dígito, 'X', 'L', 'C')
        dígito ← r
        calccifrarom(dígito, 'I', 'V', 'X')
fin

procedimiento calccifrarom (E entero: dígito; E carácter: p1, p2, p3)
var
    entero: j
inicio
    si dígito = 9 entonces
        escribir(p1, p3)
    si_no

```

```

    si dígito > 4 entonces
        escribir(p2)
        desde j ← 1 hasta dígito - 5 hacer
            escribir( p1)
        fin_desde
    si_no
        si dígito = 4 entonces
            escribir( p1, p2)
        si_no
            desde j ← 1 hasta dígito hacer
                escribir(p1)
            fin_desde
        fin_si
    fin_si
fin_si
fin_procedimiento

```

- 5.18.** Escribir una función, INTEGRAL, que devuelva el área del recinto formado por el eje de las X, las rectas $x=a$ y $x=b$ y el arco de curva correspondiente a una función continua, recibida como parámetro, con valores positivos en el intervalo considerado.

Análisis de problema

La función integral dividirá el intervalo $[a,b]$ en n subintervalos y considerará n rectángulos con esas bases y cuyas alturas sean el valor de la función recibida como parámetro, $f(x)$, en el punto medio de los subintervalos. La suma de las áreas de todos estos rectángulos constituirá el valor devuelto por integral.

```

// tipo real función(E real : x) : func
real función integral (E func : f; E real : a, b; E entero : n)
var
    real    : baserectángulo,altura,x,s
    entero : i
inicio
    baserectángulo ← (b - a)/n
    x ← a + baserectángulo/2
    s ← 0
    desde i ← 1 hasta n hacer
        altura ← f(x)
        s ← s + baserectángulo * altura
        x ← x + baserectángulo
    fin_desde
    devolver(s)
fin_función

```

- 5.19.** Escribir una función recursiva que calcule el factorial de un número entero positivo.

Análisis del problema

El factorial de un número entero positivo n se puede definir como:

$$n! = n * n-1!$$

para cualquier número mayor que uno, ya que $1! = 1$ y $0! = 0$. Por lo tanto la condición de salida de la llamada recursiva será cuando n sea menor o igual que 1.

Diseño del algoritmo

```
entero función fact(E entero : n)
inicio
    si n < 0 entonces
        devolver (-1) //error
    si_no
        si n = 0 entonces
            devolver(1)
        si_no
            devolver(n * fact(n-1))
        fin_si
    fin_si
fin_función
```

5.20. Escriba una función recursiva que calcule la potencia de un número entero positivo.

Análisis del problema

Una definición recursiva de x^y para dos números enteros positivos es la siguiente:

$$x^y = x * x^{y-1}$$

(x^0 es 1. Esa será la condición de salida de las llamadas recursivas).

Diseño del algoritmo

```
entero función pot(E entero : x,y)
inicio
    si y = 0 entonces
        devolver(1)
    si_no
        devolver(x * pot(x,y-1))
    fin_si
fin_función
```

5.21. Escribir una función recursiva que calcule el término n de la serie de Fibonacci.

Análisis del problema

La serie de Fibonacci es la siguiente,

$$1 \ 1 \ 2 \ 3 \ 5 \ 8 \ 13 \ 21 \ 34\dots$$

es decir, cada número es la suma de los dos anteriores, con excepción de los dos primeros, que siempre son 1. Por lo tanto también podemos hacer una definición recursiva para averiguar el término n de la serie, puesto que:

$$\text{Fib}_n = \text{Fib}_{n-1} + \text{Fib}_{n-2}$$

para cualquier n mayor que 2.

Diseño del algoritmo

```

entero función Fib(E entero : n)
inicio
    si n = 1 o n = 2 entonces
        devolver(1)
    si_no
        devolver(Fib(n-1) + Fib(n-2))
    fin_si
fin_función

```

- 5.22.** Escribir un procedimiento recursivo que escriba un número en base 10 convertido a otra base entre 2 y 9.

Análisis del problema

Para convertir un número decimal a otra base se debe dividir el número entre la base, y repetir el proceso, haciendo que el dividendo sea el cociente de la siguiente división hasta que éste sea menor que la base. En ese momento, se recogen todos los restos y el último cociente en orden inverso a como han salido.

Este ejemplo es una muestra de cómo puede ser útil la recursividad cuando se trata de un proceso que debe recoger los resultados en orden inverso a como han salido. Para simplificar el ejercicio se ha limitado la base a 9, de forma que se evite tener que convertir los restos en letras.

Diseño del algoritmo

```

procedimiento convierte(E entero : n,b)
inicio
    si n >= b entonces
        convierte(n div b, b)
    fin_si
    escribir(n mod b)
fin_función

```

- 5.23.** Procedimiento recursivo que permita invertir una cadena

Análisis del problema

Tal y como se efectúa su implementación, este algoritmo es otro ejemplo de como se puede utilizar la recursividad para recoger datos en orden inverso a como fueron introducidos. Los caracteres de la cadena se toman de la misma con la función subcadena antes de efectuar la llamada recursiva y se colocan en la pila. A la salida de la recursividad, dichos caracteres se recuperan de la pila en orden inverso a como fueron depositados y se concatenan en el orden en que son recuperados¹.

Codificación

```

algoritmo Ejercicio_5_23;
var
    cadena: cad
inicio
    escribir('Deme cadena')
    leer(cad)

```

¹ Para la realización de este algoritmo se utilizarán las funciones de cadena `longitud()` y `subcadena()` que se verán más adelante.

```

invertir(cad)
escribir(cad)
fin

procedimiento invertir(E/S cadena: cad)
  inicio
    invertirdesde(cad,1)
  fin_procedimiento

procedimiento invertirdesde(E/S cadena: cad; E entero: i)
  var
    cadena: aux
  inicio
    si i<=longitud(cad) entonces
      aux ← subcadena(cad,i,1)
      invertirdesde(cad, i+1)
      cad ← cad + aux
    si_no
      cad ← ''
      aux ← ''
    fin_si
  fin_procedimiento

```

5.24. Deduzca la salida del siguiente algoritmo

```

algoritmo Ejercicio_5_24
var
  entero: n
inicio
  escribir('Deme un número')
  leer(n)
  b(n div 2, 1, n)
fin

procedimiento b(E entero: x, y; E/S entero: n)
  inicio
    a(x, y)
    si y < n entonces
      b(x-1, y+2, n)
      a(x, y)
    fin_si
  fin_procedimiento

procedimiento a(E entero: p, L)
  inicio
    si p >= 1 entonces
      escribir(' ') //escribir en la misma línea
      a(p-1, L)
    si_no

```

```
    si L > 1 entonces
        escribir('*') //escribir en la misma línea
        a(p, L-1)
    si_no
        escribir('*', AvLn); // AvLn representa salto de línea
    fin_si
    fin_si
fin_procedimiento
```

Análisis del problema

El resultado de la ejecución para n = 5 es:

```
*  
***  
*****  
***  
*
```

Para entender el funcionamiento del programa, en primer lugar, hay que fijarse en que, en el algoritmo, aparece un procedimiento denominado a que se encarga exclusivamente de dibujar líneas formadas por un cierto número de caracteres (x número de ' ' e y número de '*'). Una vez observado esto, conviene ver que el programa principal llama al procedimiento b y el procedimiento b primero llama al procedimiento a, luego se invoca a si mismo modificando el valor de sus parámetros x e y, y, por último, vuelve a llamar al mencionado procedimiento a. En esta última llamada a a los valores de x e y que se le pasan serán los que se recuperen de la pila (orden inverso), lo que explica en parte la forma del dibujo obtenido.

6

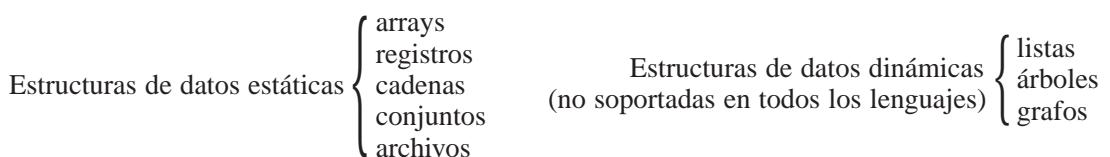
ESTRUCTURAS DE DATOS (ARRAYS Y REGISTROS)

9b`cg`Wdhi`cg`UbhYf]cfYg`gY`\U]bhfcXi W\Xc`Y`WbWdhc`XY`XUhcg`XY`hfdc`gla d`Y`ei Y`f`Ydf`Yg`Ybh`b`b`j`U`cf`Yg`XY`h`dc`gla d`Yz`Wa c`i`b`b`a`Yf`c`Ybh`Yf`c`z`f`YU`c`i`b`W`f`z`W`Yf`"9b`a`i`W`U`g`g`hi`W`cb`Yg`g`Y`b`Y!`W`g`j`h`U`g`j`b`Ya`V`U`f`[`c`z`d`c`W`g`U`f`i`b`U`W`Y`W`U`D`E`XY`j`U`cf`Yg`ei`Y`Y`gh`b`f`Y`U`W`cb`U`X`cg`Y`bh`f`Y`g`"9`d`c`W`g`l`a`Y`bh`c`XY`h`U`Yg`W`b`i`bh`cg`XY`X`Uh`cg`j`i`n`l`b`X`c`X`Uh`cg`gla d`Y`g`di`Y`XY`g`Yf`Y`l`hf`Ya`U`X`U`a`Y`bh`Y`X`J`Z`#`W`m`dc`f`Y`c`U`a`U`h`c`f`U`XY`cg`Y`b`[`i`U`Yg`XY`d`f`c`[`f`U`a`U`W`U`D`E`]`b`W`m`b`Y`gh`f`i`W`f`U`g`XY`X`Uh`cg`@`U`g`Y`gh`f`i`W`f`U`g`XY`X`Uh`cg`V`z`g`W`g`ei`Y`g`cd`f`h`U`b`U`a`U`h`c`f`U`XY`cg`Y`b`[`i`U`Yg`XY`d`f`c`[`f`U`a`U`W`U`D`E`g`c`b`cg`A`U`f`U`ng`"I`b`U`f`U`mc`U`ff`Y`[`c`Y`b`@`U`h`bc`U`f`f`J`W`Y`g`i`b`U`g`Y`W`Y`b`W`U`X`Y`d`cg`W`cb`Yg`XY`U`a`Ya`cf`J`U`W`b`h`f`U`U`U`g`ei`Y`g`Y`di`Y`XY`U`W`W`X`Y`X`f`Y`M`U`a`Y`bh`Y`m`ei`Y`W`bh`j`Y`b`Y`X`Uh`cg`XY`a`l`g`a`c`h`dc`ei`Y`di`Y`XY`b`g`Y`f`g`Y`Y`W`cb`U`X`cg`]`b`X`j`]`X`U`a`Y`bh`Y`a`Y`X`U`bh`Y`Y`i`gc`XY`gi`V`f`b`X`j`W`g`"I`b`f`Y`[`l`gh`f`c`Y`g`i`b`U`Y`gh`f`i`W`f`U`W`d`U`h`XY`U`a`U`W`b`U`f`Y`Y`a`Y`bh`cg`XY`X`l`gh`j`b`h`dc`"9`U`W`g`U`cg`Y`Y`a`Y`bh`cg`ei`Y`Z`c`f`a`U`b`d`U`f`h`Y`XY`i`b`f`Y`[`j`g`f`c`g`Y`f`Y`U`n`a`Y`X`U`bh`Y`Y`cd`Y`f`U`X`c`f`di`b`h`c"

6.1. DATOS ESTRUCTURADOS

Una estructura de datos es una colección de datos que se caracterizan por su organización y las operaciones que se definen en ella. Los datos de tipo estándar pueden ser organizados en diferentes estructuras de datos: *estáticas* y *dinámicas*.

Las **estructuras de datos estáticas** son aquellas en las que el espacio ocupado en memoria se define en tiempo de compilación y no puede ser modificado durante la ejecución del programa; por el contrario, las **estructuras de datos dinámicas** son aquellas en las cuales el espacio asignado en memoria puede ser modificado en tiempo de ejecución.



La elección de la estructura de datos idónea dependerá de la naturaleza del problema a resolver y, en menor medida, del lenguaje. Las estructuras de datos tienen en común que un identificador, nombre, puede representar a múltiples datos individuales.

6.2. ARRAYS (ARREGLOS)*

Un *array* es una colección de datos del mismo tipo, que se almacenan en posiciones consecutivas de memoria y reciben un nombre común. Para referirse a un determinado elemento de un *array* se deberá utilizar un índice, que especifique su posición relativa en el *array*. Los *arrays* podrán ser:

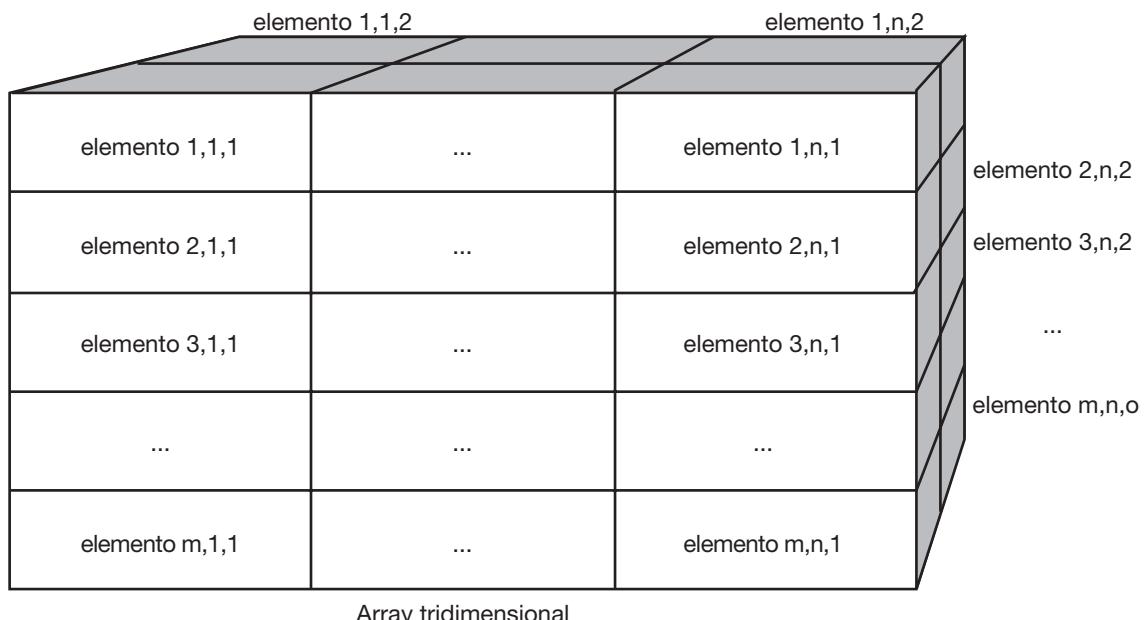
- Unidimensionales (una dimensión), también llamados vectores.
- Bidimensionales (dos dimensiones), denominados tablas o matrices.
- Multidimensionales, con tres o más dimensiones.

elemento 1
elemento 2
elemento 3
...
elemento m

Array unidimensional

elemento 1,1	...	elemento 1,n
elemento 2,1	...	elemento 2,n
elemento 3,1	...	elemento 3,n
...
elemento m,1	...	elemento m,n

Array bidimensional



Puesto que la memoria de la computadora es lineal, sea el *array* del tipo que sea, deberá estar linearizado para su disposición en el almacenamiento.

* En Latinoamérica el término inglés *array*, se suele traducir casi siempre por la palabra *arreglo*.

6.2.1. Arrays unidimensionales

Todo dato a utilizar en un algoritmo ha de haber sido previamente declarado. Los *arrays* no son una excepción y lo primero que se debe hacer es crear el tipo, para luego poder declarar datos de dicho tipo.

Al ser un tipo estructurado, la declaración se hará en función de otro tipo estándar o previamente definido, al que se denominará *tipo base*, por ser todos los elementos constituyentes de la estructura del mismo tipo.

```
tipo
  array[<liminf>..<limsup>] de <tipo_base> : <nombre_del_tipo>
var
  <nombre_del_tipo>: <nombre_del_vector>
```

El número de elementos del vector vendrá dado por la fórmula:

$$<\text{limsup}> - <\text{liminf}> + 1$$

Todos los elementos del vector podrán seleccionarse arbitrariamente y serán igualmente accesibles. Se les referenciará empleando un índice que señale su posición relativa dentro del vector. Si <nombre_del_vector> fuera vect al elemento *enésimo* del array se le referenciaría por vect[n], siendo n un valor situado entre el límite inferior (*liminf*) y el límite superior (*limsup*). Los elementos podrán ser procesados como cualquier otra variable que fuera de dicho *tipo_base*.

6.2.2. Arrays bidimensionales

Un *array bidimensional* es un vector de vectores; se denomina también *matriz* o *tabla*. Es por tanto un conjunto de elementos del mismo tipo en el que el orden de los componentes es significativo y en el que se necesitan especificar dos subíndices para poder identificar a cada elemento del array.

	1	2	...	j
1	m[1,1]	m[1,2]
2	m[2,1]	m[2,2]
...
i	m[i,j]

Su declaración es:

```
tipo
  array[liminf.. limsup, liminf.. limsup] de tipo_base: nombre_del_tipo
var
  <nombre_del_tipo> : <nombre_de_la_matriz>
```

Ejemplo: Un array de dos dimensiones F y C

```
tipo
  array[1..F, 1..C] de real : matriz
  //F y C habrán sido declaradas previamente como constantes
var
  matriz : m
```

La referencia a un determinado elemento de la matriz, requiere el empleo de un primer subíndice que indique la fila y un segundo subíndice que marque la columna.

Los elementos, $m[i, j]$, podrán ser procesados como cualquier otra variable de tipo real. El número de elementos de la matriz vendrá dado por la fórmula

$$(F - 1 + 1) * (C - 1 + 1)$$

6.2.3. Recorrido de todos los elementos del array

El recorrido de los elementos de un *array* se realizará utilizando estructuras repetitivas, con las que se manejan los subíndices del *array*. Si se trata de un vector, bastará con una estructura repetitiva. Para realizar el recorrido de una matriz o tabla se necesitarán dos estructuras repetitivas anidadas, una de las cuales controle las filas y otra las columnas. Por otra parte, en el caso de las matrices se consideran dos tipos de recorridos (por filas y por columnas)

Recorrido por filas

Supuestas las siguientes declaraciones:

```
const
    F = <valor1>
    C = <valor2>
tipo
    array[1..F, 1..C] de real : matriz
var
    matriz : m
```

y que todos los elementos de la matriz contienen información válida, escribir el pseudocódigo para que se visualice en primer lugar el contenido de los elementos de la primera fila, a continuación el contenido de los de la segunda, etcétera.

```
desde i ← 1 hasta F hacer
    desde j ← 1 hasta C hacer
        escribir(m[i,j])
    fin_desde
fin_desde
```

Recorrido por columnas

Tal y como aparece a continuación, se mostrará primero el contenido de los elementos de la primera columna, luego el de los elementos de la segunda columna, etcétera.

```
desde j ← 1 hasta C hacer
    desde i ← 1 hasta F hacer
        escribir(m[i,j])
    fin_desde
fin_desde
```

Para recorrer los elementos de una matriz de n dimensiones, utilizaremos n estructuras repetitivas anidadas.

6.2.4. Arrays como parámetros

Los *arrays* podrán ser pasados como parámetros, tanto a procedimientos como a funciones. Para ello se debe declarar algún parámetro formal del mismo tipo que el *array* que constituye el parámetro actual.

Ejemplo:

```

algoritmo pasodearrays
tipo
    array[1..F, 1..C] de real : arr
var
    arr : a
    entero : b
    ...
inicio
    ...
    b ← recibearray ( a )
    ...
fin

entero función recibearray( E arr : m)
//Los parámetros actuales y los formales no necesitan
//coincidir en el nombre}
...
inicio
    .....
    .....
fin_función

```

6.3. CONJUNTOS

Un conjunto es un dato estructurado y las variables declaradas de tipo conjunto han de poder almacenar una colección de elementos de un determinado tipo base, aunque nunca dos copias del mismo elemento, pues todos los miembros de un conjunto han de ser diferentes. Los conjuntos van a poder ser pues representados mediante *arrays* donde no se admitirá la inserción de elementos repetidos. Cuando el tipo base, es decir el tipo de los elementos del conjunto, sea ordinal será posible efectuar una representación bastante eficiente de un conjunto mediante un vector de N elementos de tipo lógico. En este caso los elementos del conjunto no se almacenan en el *array*, si no que vienen representados por los subíndices del mismo. Un elemento pertenecerá al conjunto cuando *vector[elemento]* sea verdadero y no pertenecerá a él si *vector[elemento]* es falso. La variable de tipo conjunto debe ser inicializada siempre a un conjunto vacío o al conjunto universal, que es el que consta de todos los valores del tipo base, antes de ser utilizada. La inicialización a conjunto vacío consistirá en recorrer el vector asignando a sus elementos el valor falso. En este caso, las declaraciones necesarias para el trabajo con conjuntos serán:

```

tipo
{Dado que el tipo base del conjunto podrá tener valores entre
<limite_inferior>..<limite_superior>}
array[<limite_inferior>..<limite_superior>] de lógico: <tipo_base_conj>
var
    <tipo_conjunto>: <variable_conjunto>, <variable_auxiliar>
    <tipo_base_conj>: <variable_elemento>

```

y los procedimientos y funciones que efectúan las operaciones básicas:

```

procedimiento inicializar( S <tipo_conjunto>: vector)
    // a conjunto vacío
var
    <tipo_base_conj>: elemento
inicio
    // <limite_inferior> y <limite_superior> son constantes
    desde elemento ← <limite_inferior> hasta <limite_superior> hacer
        vector[elemento] ← falso
    fin_desde
fin_procedimiento

procedimiento añadir( E <tipo_base_conj>: elemento;
                        E/S <tipo_conjunto>: vector)
inicio
    vector[elemento] ← verdadero
fin_procedimiento

procedimiento quitar( E <tipo_base_conj>: elemento;
                        E/S <tipo_conjunto>: vector)
inicio
    vector[elemento] ← falso
fin_procedimiento

logico funcion pertenece( E <tipo_base_conj>: elemento;
                            E <tipo_conjunto>: vector)
inicio
    devolver(vector[elemento])
fin_funcion

procedimiento union( E <tipo_conjunto>: vector1, vector2;
                        S <tipo_conjunto>: vector3)
var
    <tipo_base_conj>: elemento
inicio
    desde elemento ← <limite_inferior> hasta <limite_superior> hacer
        vector3[elemento] ← vector1[elemento] o vector2[elemento]
    fin_desde
fin_procedimiento

procedimiento intersección( E <tipo_conjunto>: vector1, vector2;
                            S <tipo_conjunto>: vector3)
var
    <tipo_base_conj>: elemento
inicio
    desde elemento ← <limite_inferior> hasta <limite_superior> hacer
        vector3[elemento] ← vector1[elemento] y vector2[elemento]
    fin_desde
fin_procedimiento
```

6.4. REGISTROS

Un **registro** es un dato estructurado, formado por elementos lógicamente relacionados, que pueden ser del mismo o de distinto tipo, a los que se denomina *campos*. Los campos de un registro podrán ser de un tipo estándar o de otro tipo registro previamente definido.

En los lenguajes orientados a objetos el almacenamiento de información de diferentes tipos con un único nombre suele efectuarse en clases, pero las clases no almacenan sólo campos con el estado de un objeto, si no también su comportamiento. Algunos de estos lenguajes, como C++ o C#, ofrecen además el tipo *estructura*. La estructura es similar a la *clase* en POO¹ e igual al registro en lenguajes estructurados como C o Pascal.

Ejemplo de un dato de tipo registro podría ser el que permitiera almacenar la situación de un punto en el plano, compuesta por dos números reales. De igual forma, si lo que se desea es describir a una persona, se podría utilizar un dato de tipo registro para almacenar, agrupada, la información más relevante.

Un tipo registro se declarará de la siguiente forma:

```
tipo
  registro: <nombre_del_tipo>
    <tipo_de_dato1>:<nombre_de_campo1> [,<nombre_de_campo2>] [...]
    <tipo_de_dato2>:<nombre_de_campoX> [,<nombre_de_campoY>] [...]
    .....
  fin_registro
```

Para declarar una variable de tipo registro:

```
var
  <nombre_del_tipo>: <nombre_de_variable>
```

Para acceder a un determinado campo de un registro se utilizará, como suelen emplear la mayoría de los lenguajes el nombre de la variable de tipo registro unido por un punto al nombre del campo.

```
<nombre_de_variable>.<nombre_de_campo1>
```

Si los campos del registro fueran a su vez otros registros habrá que indicar:

```
<nombre_de_variable>.<nombre_de_campo1>.<nombre_de_campo_de_campo1>
```

Los datos de tipo registro se pueden pasar como parámetros tanto a procedimientos como a funciones.

6.4.1. Arrays de registros y arrays paralelos

Los elementos de un *array* pueden ser de cualquier tipo, por tanto es posible la construcción de *arrays de registros*.

Otra forma de trabajar en *arrays* con información de distinto tipo y lógicamente relacionada es el uso de *arrays paralelos* que, al tener el mismo número de elementos, se podrán procesar simultáneamente.

¹ Programación orientada a objetos.

Array de registros		Arrays paralelos		
Nombre	Edad	Nombres	Edades	
'Ana'	28	a[1]	'Ana'	Nombres[1]
'Carlos'	36	a[2]	'Carlos'	Nombres[2]
...
'Juan'	34	a[n]	'Juan'	Nombres[n]
				28
				36
				...
				34
				Edades[1]
				Edades[2]
				...
				Edades[n]

Para acceder al campo nombre del 2.^º elemento del array A se escribiría a [2].nombre. Por ejemplo,

```
escribir(a[2].nombre)
```

presentaría en consola la cadena Carlos.

6.5. EJERCICIOS RESUELTOS

6.1. Determinar los valores de los vectores N y M después de la ejecución de las instrucciones siguientes:

```
var
    array [1..3] de entero : M, N
```

1. M[1] ← 1
2. M[2] ← 2
3. M[3] ← 3
4. N[1] ← M[1] + M[2]
5. N[2] ← M[1] - M[3]
6. N[3] ← M[2] + M[3]
7. N[1] ← M[3] - M[1]
8. M[2] ← 2 * N[1] + N[2]
9. M[1] ← N[2] + M[1]

	M[1]	M[2]	M[3]	N[1]	N[2]	N[3]
1	1	—	—	—	—	—
2	1	2	—	—	—	—
3	1	2	3	—	—	—
4	1	2	3	3	—	—
5	1	2	3	3	-2	—
6	1	2	3	3	-2	5
7	1	2	3	2	-2	6
8	1	6	3	2	-2	5
9	1	6	3	2	-2	5

6.2. Leer un vector V desde un terminal de entrada.

Análisis del problema

DATOS DE ENTRADA: v (el array que vamos a llenar)

DATOS AUXILIARES: n (número de elementos del array), i (contador que controla el número de lecturas y que proporciona además el índice de los elementos del array)

La lectura de un array es una operación repetitiva, por lo que se debe utilizar alguna estructura iterativa. Si es posible, lo mejor es una estructura **desde**, ya que se conoce de antemano el número de iteraciones que se han de realizar.

Diseño del algoritmo

```
algoritmo ejercicio_6_2
const
    //el número máximo de elementos de un array lo tomamos como
    //una constante
    MáxArray = ...
var
    array [1..MáxArray] de entero : v
    entero : i, n
inicio
    leer(n)
    desde i ← 1 hasta n hacer
        leer(V[i])
    fin_desde
fin
```

6.3. Escribir un algoritmo que permita calcular el cuadrado de los 100 primeros números enteros y a continuación escribir una tabla que contenga dichos cuadrados.

Análisis del problema

DATOS DE SALIDA: Cuadr (el vector que guarda los cuadrados de los 100 primeros enteros)

DATOS AUXILIARES: I (variable del bucle)

Si desea hacer la operación en dos fases, una de cálculo y otra de escritura, se ha de utilizar un array. Primero se hace un bucle en el que se vayan sacando los números del 1 al 100, se calcula su cuadrado y se asigna el resultado a un elemento del array. Un segundo bucle, que se repetirá también 100 veces nos servirá para escribir todos los elementos del array.

Diseño del algoritmo

```
algoritmo ejercicio_6_3
var
    array [1..100] de entero : cuadr
    entero : i
inicio
    //rellenar el array con los cuadrados
    desde i ← 1 hasta 100 hacer
        cuadr[i] ← i * i
    fin_desde
    //escritura del array
```

```

desde i ← 1 hasta 100 hacer
    escribir(cuadr[i])
fin_desde
fin

```

- 6.4.** Se tienen N temperaturas almacenadas en un array. Se desea calcular su media y obtener el número de temperaturas mayores o iguales que la media.

Análisis del problema

DATOS DE SALIDA: media (la media de las n temperaturas), mayores (contador que guardará el número de temperaturas mayores que la media)

DATOS DE ENTRADA: n (número de temperaturas), temp (array de n elementos donde se guardan las temperaturas)

DATOS AUXILIARES: i (variable de los bucles que indica también el índice de los elementos del array)

El algoritmo debe tener dos bucles. En el primero se leen por teclado todos los elementos del array. Mediante una función, se halla la media del array. Una vez leídos los elementos y hallada la media, se recorre otra vez el array para determinar cuántos elementos son superiores a la media. Cada vez que se encuentra un dato que cumpla esa condición se incrementa el contador mayores en una unidad.

Diseño del algoritmo

```

algoritmo ejercicio_6_4
const
    MáxArray = ...
tipos
    array[1..MáxArray] de real : vector
var
    Vector : temp
    real : mediatemp
    entero : mayores, n, i
inicio
    leer(n)
    desde i ← 1 hasta n hacer
        leer(temp[i])
    fin_desde
    mediatemp ← media(temp, n)
    mayores ← 0
    desde i ← 1 hasta n hacer
        si temp[i] >= mediatemp entonces
            mayores ← mayores + 1
        fin_si
    fin_desde
    escribir(mediatemp, mayores)
fin

entero función media( E Vector : v;  E entero n )
var
    real : m
    entero : i
inicio
    m ← 0
    para i = 1 hasta n hacer
        m ← m + v[i]
    fin_para
    m ← m / n
    devolver m
fin

```

```

desde i ← 1 hasta n hacer
    m ← m + temp[i]
fin_desde
devolver(m/n)
fin_función

```

6.5. Calcular la suma de todos los elementos de un vector de dimensión 100, así como su media aritmética.

Análisis del problema

DATOS DE SALIDA: suma (suma de los elementos del vector), media (media de los elementos)

DATOS DE ENTRADA: vector (el array que contiene los elementos a sumar)

DATOS AUXILIARES: i (variable del bucle)

Mediante un bucle **desde** se leen y suman los elementos del vector. Una vez hecho esto se calcula la media (suma/100) y se escriben la media y la suma.

Diseño del algoritmo

```

algoritmo ejercicio_6_5
var
    array[1..100] de entero : vector
    real : media
    entero : suma, i
inicio
    media ← 0
    suma ← 0
    desde i ← 1 hasta 100 hacer
        leer(vector[i])
        suma ← suma + vector[i]
    fin_desde
    media ← suma/100
    escribir(media, suma)
fin

```

6.6. Calcular el número de elementos negativos, cero y positivos de un vector dado de 60 elementos.

Análisis del problema

DATOS DE SALIDA: pos (contador de elementos positivos), cero (contador de ceros), neg (contador de elementos negativos)

DATOS DE ENTRADA: lista (vector a analizar)

DATOS AUXILIARES: i (variable del bucle)

Suponiendo que se tiene leído el *array* desde algún dispositivo de entrada, se ha de recorrer el *array* con un bucle **desde** y dentro de él comprobar si cada elemento *lista[i]* es igual, mayor o menor que 0, con lo que se incrementará el contador correspondiente.

Diseño del algoritmo

```

algoritmo ejercicio_6_6
var
    array [1..60] de entero : lista
    entero : pos, neg, cero, i

```

```

inicio
    pos ← 0
    neg ← 0
    cero ← 0
    //lectura del array
    desde i ← 1 hasta 60 hacer
        si lista[i] > 0 entonces
            pos ← pos + 1
        si_no
            si lista[i] < 0 entonces
                neg ← neg + 1
            si_no
                cero ← cero + 1
            fin_si
        fin_si
    fin_desde
    escribir(pos,neg,cero)
fin

```

6.7. Rellenar una matriz identidad de 4 por 4 elementos.

Análisis del problema

DATOS DE SALIDA: matriz (matriz identidad de 4 filas y cuatro columnas)

DATOS AUXILIARES: i (índice de las filas), j (índice de las columnas)

Una matriz identidad es aquella en la que la diagonal principal está llena de unos y el resto de los elementos son 0. Como siempre que se quiere trabajar con matrices de dos dimensiones, se han de utilizar dos bucles anidados, que en este caso serán de 1 hasta 4. Para llenar la matriz identidad, se ha de comprobar que los índices de los bucles -i y j. Si son iguales matriz[i,j] será igual a 1, en caso contrario se asigna 0 al elemento i,j.

Diseño del algoritmo

```

algoritmo ejercicio_6_7
var
    array [1..4,1..4] de enteros : matriz
    entero : i, j
inicio
    desde i ← 1 hasta 4 hacer
        desde j ← 1 hasta 4 hacer
            si i = j entonces
                matriz[i,j] ← 1
            si_no
                matriz[i,j] ← 0
            fin_si
        fin_desde
    fin_desde
fin

```

6.8. Diseñar un algoritmo que calcule el mayor valor de una lista L de N elementos.

Análisis del problema

DATOS DE SALIDA: N (número de elementos de la lista), máx (el elemento mayor de la lista)

DATOS DE ENTRADA: L (lista a analizar)

DATOS AUXILIARES: I (índice de la lista)

En primer lugar se ha de leer la dimensión de la lista (N) y rellenarla (en el ejercicio se rellena simplemente con la instrucción `LeerLista(L, N)`). Para hallar el elemento mayor de un *array*, se debe inicializar la variable `máx` con el primer elemento de la lista ($L[1]$) y hacer un bucle desde 2 hasta N en el que se comparará cada elemento $L[i]$ con `máx`. Si $L[i]$ es mayor que `máx`, $L[i]$ será el nuevo máximo.

Diseño del algoritmo

```

algoritmo ejercicio_6_8
const
    //Se considera que el array tiene un máximo de 100 elementos
    MáxArray = 100
tipo
    array [1..MáxArray] de enteros : lista
var
    lista : L
    entero : máx,N,i
inicio
    leer(N)
    LeerLista(L, N)
    máx ← L[1]
    desde i ← 2 hasta N hacer
        si L[i] > máx entonces
            máx ← L[i]
        fin_si
    fin_desde
    escribir(máx)
fin

```

- 6.9.** Dada una lista L de N elementos diseñar un algoritmo que calcule de forma independiente la suma de los números pares y la suma de los números impares.

Análisis del problema

DATOS DE SALIDA: `spar` (suma de números pares), `simpar` (suma de los números impares)

DATOS DE ENTRADA: `L` (lista a analizar)

DATOS AUXILIARES: `i` (variable del bucle)

Una vez leído el *array* se han de analizar cada uno de sus elementos comprobando si $L[i]$ es par o impar mediante el operador resto (es par si $L[i] \bmod 2 = 0$). Si esta condición se cumple se acumula el valor de $L[i]$ en `spar`, en caso contrario se acumulará en `simpar`.

Diseño del algoritmo

```

algoritmo ejercicio_6_9
const
    MáxArray = ...
var
    array [1..MáxArray] de entero : L
    entero : N,i,spar,simpar
inicio
    .
    .

```

```

// lectura de N elementos del array L
.
.
.
spar ← 0
simpar ← 0
desde i ← 1 hasta n hacer
    si L[i] mod 2 = 0 entonces
        spar ← spar + l[i]
    si_no
        simpar ← simpar + l[i]
    fin_si
fin_desde
escribir(spar,simpar)
fin

```

- 6.10.** Escribir el pseudocódigo que permita escribir el contenido de una tabla de dos dimensiones (5×4).

Análisis del problema

DATOS DE SALIDA: tabla (la tabla a escribir)

DATOS AUXILIARES: i, j (índices de la fila y columna de cada elemento del array)

Si se supone la tabla ya creada, para escribirla se han de recorrer todos sus elementos. Para recorrer un array de dos dimensiones, hay que implementar 2 bucles anidados, uno para recorrer las filas y otro para recorrer las columnas. Dentro del bucle interno, lo único que hay que hacer es escribir el elemento i,j del array.

Diseño del algoritmo

```

algoritmo ejercicio_6_10
var
    array [1..5,1..4] de entero : tabla
    entero : i, j
inicio
    desde i ← 1 hasta 5 hacer
        desde j ← 1 hasta 4 hacer
            escribir(tabla[i,j])
        fin_desde
    fin_desde
fin

```

- 6.11.** Se desea diseñar un algoritmo que permita obtener el mayor valor almacenado en una tabla VENTAS de dos dimensiones (4 filas y 5 columnas).

Análisis del problema

DATOS DE SALIDA: máx (el elemento mayor de la tabla)

DATOS DE ENTRADA: vVENTAS (la tabla a analizar)

DATOS AUXILIARES: i, j (índices de la fila y columna de cada elemento), fmáx, cmáx (fila y columna del elemento mayor de la lista)

En este ejercicio se empleará otro método para hallar el máximo. En vez de hacer la búsqueda tomando como referencia el contenido del elemento mayor, se usa como referencia la posición (fila y columna) que ocupa dicho elemento.

Después de leer la tabla, se inicializan los valores de fmáx y cmáx (fila y columna que ocupa el elemento mayor) a 1. Se debe recorrer el array con dos bucles anidados. En el bucle interno se compara VENTAS[i, j], con VENTAS[fmáx, cmáx]. Si es mayor, fmáx y cmáx pasarán a tener el valor que en esos momentos contengan i y j. Al final del bucle en fmáx y cmáx estará guardada la posición del elemento mayor por lo que asignamos a la variable máx el contenido de VENTAS[fmáx, cmáx] y lo escribimos.

Diseño del algoritmo

```

algoritmo ejercicio_6_11
var
    array [1..4,1..5] de entero : VENTAS
    entero : i,j,fmáx,cmáx,máx
inicio
    //lectura del array VENTAS
    fmáx ← 1
    cmáx ← 1
    desde i ← 1 hasta 4 hacer
        desde j ← 1 hasta 5 hacer
            si VENTAS[i,j] > VENTAS[fmáx,cmáx] entonces
                fmáx ← i
                cmáx ← j
            fin_si
        fin_desde
    fin_desde
    máx ← VENTAS[fmáx,cmáx]
    escribir(máx)
fin

```

6.12. Hacer diferentes listados de una lista de 10 números según el siguiente criterio:

si número ≥ 0 y número < 50 , ponerlo en LISTA1
 si número ≥ 50 y número < 100 , ponerlo en LISTA2
 si número ≥ 100 y número ≤ 150 , ponerlo en LISTA3

Análisis del problema

DATOS DE SALIDA: LISTA1, LISTA2, LISTA3

DATOS DE ENTRADA: LISTA (lista que contiene los números iniciales)

DATOS AUXILIARES: i (índice de la lista original), conta1, conta2, conta3 (índices de cada una de las listas de salida)

Se han de utilizar cuatro arrays que tendrán todos la dimensión del array original LISTA (10 elementos). Una vez leído el array LISTA, se debe recorrer para comprobar cada uno de sus elementos e introducirlo en la lista correspondiente incrementando también su índice.

Una vez llenas las tres listas de salida, se escriben mediante tres bucles que irán cada uno desde 1 hasta el último valor del índice correspondiente. Como la operación de escritura va a ser igual para las tres listas se implementará un procedimiento que se encargue de la operación de lectura. En dicho procedimiento se pasa la lista y el número de elementos de la misma (conta1, conta2, conta3). Al utilizar un procedimiento con un tipo de datos estructurado, es necesario crear un tipo de datos definido por el usuario (vector).

Diseño del algoritmo

```

algoritmo ejercicio_6_12
tipo

```

```

array [1..10] de entero : vector
var
    vector : LISTA,LISTA1,LISTA2,LISTA3
    entero : i,conta1,conta2,conta3
inicio
    conta1 ← 0
    conta2 ← 0
    conta3 ← 0
    //leemos el array LISTA
    desde i ← 1 hasta 10 hacer
        si LISTA[i] > 0 entonces
            si LISTA[i] < 50 entonces
                conta1 ← conta1 + 1
                LISTA1[conta1] ← LISTA[i]
            si_no
                si LISTA[i] < 100 entonces
                    conta2 ← conta2 + 1
                    LISTA2[conta2] ← LISTA[i]
                si_no
                    si LISTA[i] <= 150 entonces
                        conta3 ← conta3 + 1
                        LISTA3[conta3] ← LISTA[i]
                    fin_si
                fin_si
            fin_si
        fin_si
    fin_desde
    EscribirVector(LISTA1,conta1)
    EscribirVector(LISTA2,conta2)
    EscribirVector(LISTA3,conta3)
fin

procedimiento EscribirVector(ENTERO vector: v; ENTERO : n)
var
    entero : i
inicio
    desde i ← 1 hasta n hacer
        escribir(v[i])
    fin_desde
fin_procedimiento

```

- 6.13.** Rellenar un vector a de N elementos con enteros consecutivos de forma que para cada elemento $A_i \leftarrow i$.

Análisis del problema

DATOS DE ENTRADA: A (el array a llenar), N (número de elementos del array)

DATOS AUXILIARES: i (índice del array)

Se debe ejecutar un bucle **desde** N veces. Dentro del bucle hay que hacer la asignación $A[i] \leftarrow i$.

Diseño del algoritmo

```

algoritmo Ejercicio_6_13
const
    MáxArray = ...
var
    entero : i
    array [1..MáxArray] de entero : A
inicio
    desde i ← 1 hasta MáxArray hacer
        A[i] ← i
    fin_desde
fin

```

- 6.14.** Escribir un programa para introducir una serie de números desde el teclado. Utilizar un valor centinela $-1E5$ para terminar la serie. El programa calculará e imprimirá el número de valores leídos, la suma y la media de la tabla. Además, generará una tabla de dos dimensiones en la que la primera columna será el propio número y la segunda indicará cuánto se desvía de la media.

Análisis del problema

DATOS DE SALIDA: conta (número de valores leídos), suma (suma de todos los valores), media (media de dichos valores), desviación (tabla con las desviaciones de cada elemento con respecto a la media)

DATOS DE ENTRADA: vector (el vector que leemos), númer (cada uno de los números que leemos)

DATOS AUXILIARES: i (índice del vector)

Se debe dimensionar el *array* a un número lo suficientemente grande como para que quepan los valores a procesar. Como ese número no está determinado, se dimensiona al número máximo de valores previsto (representado por *MáxLista*).

El bucle utilizado para procesar el vector no puede ser un bucle *desde*, sino que se debe utilizar uno *repetir* o un *mientras*. Dentro de este bucle de lectura, se lee el elemento, se acumula su valor en suma y se incrementa el índice i. La variable leída será númer, y una vez verificado que es distinta de $-1E5$, se asigna el valor *vector[i]*.

El número de valores procesados (conta) será igual al último valor de i. Se halla la media (suma / conta) y se escribe conta, media y suma.

Para hallar la desviación de cada uno de los elementos se hará otro bucle (ahora sirve un bucle *desde*) en el que el índice vaya tomando valores entre 1 y conta. Dentro del bucle se calcula la desviación de elemento y escribimos ambos.

Diseño del algoritmo

```

algoritmo ejercicio_6_14
const
    MáxArray = ...
var
    array [1..MáxArray] de real : vector
    array [1..MáxArray, 1..2] de real : desviación
    entero : i, j, conta
    real : númer, suma, media, desviación
inicio
    i ← 0
    suma ← 0
    leer (númer)

```

```

mientras núm <> -1E5 hacer
    i ← i + 1
    vector[i] ← núm
    suma ← suma + vector [i]
    leer(núm)
fin_mientras
conta ← i
media ← suma/conta
escribir(conta,suma,media)
desde i ← 1 hasta conta hacer
    desviación[i, 1] ← vector[i]
    desviación[i, 2] ← vector[i] - media
fin_desde
fin

```

- 6.15.** Dado un vector X compuesto por N elementos, se desea diseñar un algoritmo que calcule la desviación estándar D .

$$D = \frac{\sqrt{\sum_{i=1}^n (x_i - \bar{x})^2}}{n-1}$$

$$\bar{x} = \frac{x_1 + x_2 + \dots + x_n}{n}$$

Análisis del problema

DATOS DE SALIDA: DesviaciónEst (función que calcula la desviación estandar)

DATOS DE ENTRADA: x (el vector), n (número de elementos)

DATOS AUXILIARES: i (índice del array), media (media de los elementos)

Se desarrolla una función que permita calcular la desviación estándar de N elementos de una lista de números reales. La función va a recibir como parámetros de entrada el vector x —al que damos un tipo de datos vector que debe estar definido previamente— y el número de elementos. Una función se encargará de hallar la media. Dentro de esta función, es necesario saber la suma de los elementos, para lo que utilizaremos otra función SumaReal. Una vez calculada la media, mediante un bucle se halla la suma de los cuadrados de cada elemento menos la media, necesaria para poder hallar la desviación. Por último se calcula la desviación estandar por la fórmula dada anteriormente.

Diseño del algoritmo

```

real función DesviaciónEst( E vector : x; E entero : n)
var
    real : media, suma
    entero : i
inicio
    media ← MediaReal(X,n)
    suma ← 0
    desde i ← 1 hasta n hacer
        suma ← suma + cuadrado(X[i] - media)

```

```

fin_desde
devolver(raíz2(suma) / (n-1))
fin_función

real función MediaReal(ENTERO vector : v; ENTERO entero : n)
inicio
    devolver(SumaReal(v,n) / n)
fin_función

real función SumaReal(ENTERO vector : v; ENTERO entero : n)
var
    real : suma
    entero : i
inicio
    suma ← 0
    desde i ← 1 hasta n hacer
        suma ← suma + v[i]
    fin_desde
    SumaReal ← suma
fin_función

```

6.16. Leer una matriz de 3×3 .

Análisis del problema

DATOS DE ENTRADA: matriz (la tabla o matriz de 3×3)

DATOS AUXILIARES: i, j (índices de las fila y columna de cada elemento del array)

Para leer una matriz es necesario utilizar dos bucles anidados. Dentro del bucle interior se incluye una operación de lectura con cada elemento de la matriz.

Diseño del algoritmo

```

algoritmo ejercicio_6_16
var
    array [1..3,1..3] enteros : matriz
    entero : i, j
inicio
    desde i ← 1 hasta 3 hacer
        desde j ← 1 hasta 3 hacer
            leer(matriz[i,j])
        fin_desde
    fin_desde
fin

```

6.17. Escribir el algoritmo que permita sumar el número de elementos positivos y el de negativos de una tabla T de n filas y m columnas.

Análisis del problema

DATOS DE SALIDA: pos (suma de los elementos positivos), neg (suma de los elementos negativos)

DATOS DE ENTRADA: T (tabla a procesar), n (número de filas), m (número de columnas)

DATOS AUXILIARES: i, j (índices de los elementos de la tabla)

Después de leer la tabla, se recorre mediante dos bucles anidados. En el bucle interno se ha de comprobar si $T[i,j]$ es positivo o negativo. Si es mayor que 0 se acumula el valor de $T[i,j]$ en pos. Si es menor que 0, se hace la misma operación en neg. En otro caso, $T[i,j]$ sería igual a 0, por lo que no se ha de hacer nada.

Diseño del algoritmo

```

algoritmo ejercicio_6_17
const
    MáxFila = ...
    MáxCol = ...
var
    array [1..MáxFila,1..MáxCol] de entero : T
    entero : T
inicio
    leer(n,m)
    // lectura de la tabla
    pos ← 0
    neg ← 0
    desde i ← 1 hasta n hacer
        desde j ← 1 hasta m hacer
            si T[i,j] > 0 entonces
                pos ← pos + T[i,j]
            si_no
                si T[i,j] < 0 entonces
                    neg ← neg + T[i,j]
                fin_si
            fin_si
        fin_desde
    fin_desde
    escribir(pos,neg)
fin

```

- 6.18.** Supongamos que existen N ciudades en la red ferroviaria de un país, y que sus nombres están almacenados en un vector llamado CIUDAD. Diseñar un algoritmo en el que se lea el nombre de cada una de las ciudades y los nombres con los que está enlazada.

Análisis del problema

DATOS DE SALIDA:	Listado de ciudades
DATOS DE ENTRADA:	CIUDAD (vector donde se guardan los nombres de las ciudades), ENLACE (array donde representamos los enlaces de unas ciudades con otras), N (número de ciudades)
DATOS AUXILIARES:	i, j (índices de los arrays)

En primer lugar, hay que ver las ciudades que están enlazadas entre sí; para ello se utiliza una tabla de $N \times N$ elementos que representan las ciudades. Por ejemplo, la indicación de que la ciudad 3 está enlazada con la 5 ($CIUDAD[3,5]$) se realiza mediante una marca, por ejemplo un 1. Si no están enlazadas en dicho elemento se pone un 0. La tabla de ciudades, será un array triangular: sólo se ha de llenar medio array, ya que si la ciudad 1 está enlazada con la ciudad 2, la ciudad 2 lo estará con la 1. Por tanto el bucle interno irá desde $i + 1$ hasta n .

Para sacar los nombres de las ciudades se necesita disponer de otro array en el que el elemento 1 contenga el nombre de la ciudad 1, el 2 el de la 2 y así hasta el número de ciudades, es decir, hasta N .

Después de llenar el array de enlaces y el de ciudades (para hacer el listado que pide el problema) es necesario recorrer toda la tabla de enlaces mediante 2 bucles anidados. En el bucle externo se debe escribir el

nombre de la ciudad a la que se sacan los enlaces; es decir si se está en la ciudad i , se debe escribir $\text{CIUDAD}[i]$. En el bucle interno se debe comprobar el contenido de $\text{ENLACE}[i, j]$. Si éste es distinto de 0, se escribe la ciudad con la que está enlazada ($\text{CIUDAD}[j]$).

Diseño del algoritmo

```

algoritmo ejercicio_6_18
const
    n = ...
var
    array[1..n] de cadena : ciudad
    array[1..n,1..n] de entero : n
    entero : i, j
inicio
    // lectura del array de ciudades y de enlaces
    desde i ← 1 hasta n hacer
        leer(ciudad[i])
        desde j ← i+1 hasta n hacer
            leer(enlace[i,j])
        fin_desde
    fin_desde
    // listado de las ciudades con sus enlaces
    desde i ← 1 hasta n hacer
        escribir(ciudad[i])
        desde j ← 1 hasta n hacer
            si enlace[i,j] = 1 entonces
                escribir(ciudad[j])
            fin_si
        fin_desde
    fin_desde
fin_desde

```

- 6.19.** Determinar si una matriz de tres filas y tres columnas es un cuadrado mágico. Se considera un cuadrado mágico aquel en el cual las filas, columnas, y las diagonales principal y secundaria suman lo mismo.

Análisis del problema

DATOS DE SALIDA:	Mensaje que indica si es un cuadrado mágico
DATOS DE ENTRADA:	cuad (la matriz que vamos a comprobar)
DATOS AUXILIARES:	i, j (índices de los arrays), $mágico$ (variable lógica que valdrá verdadero si la matriz es un cuadrado mágico), $suma$ (sumador que acumula la suma de los elementos de la primera fila), $diagonal1$ y $diagonal2$ (suma de los valores de la diagonal principal y secundaria respectivamente)

Después de sumar la primera fila mediante una función `SumaFila`, se utiliza un bucle `desde`, que comprueba las siguientes filas llamando a su vez a la función `SumaFila`. Para no tener que recorrer todo el `array`, el bucle externo será un bucle que se repita mientras que el índice sea menor o igual que 3 o la variable `mágico` sea cierta (previamente se inicializa la variable `mágico` a `falso`). Dentro de ese bucle se ha de pasar a la función la fila que se desea sumar (i), y se comprueba si el valor que retorna la función es igual al valor de la suma de la primera fila (`suma`).

De forma similar se procede con la suma de las columnas en la que utilizaremos la función `SumaCol`. Para las diagonales se utiliza un bucle `desde` en el que se acumulan todos los valores de `cuad[i,i]` en la varia-

ble diagonal1 y los valores de cuad[i, 4-i] en diagonal2. Si después de esto el valor de todas las sumas de filas, columnas y diagonales son iguales, mágico será verdadero, y aparecerá el mensaje diciendo que la matriz es un cuadrado mágico.

Diseño del algoritmo

```

algoritmo ejercicio_6_19
tipos
    array[1..3,1..3] de entero : tabla
var
    tabla : cuad
    entero : suma,diagonal1,diagonal2,i
    lógico : mágico
inicio
    mágico ← verdad
    // lectura de la matriz
    leer(cuad)
    //hallamos la suma de la primera fila
    suma ← SumaFila(cuad,3,1)
    //hallamos la suma de las restantes filas
    i ← 1
    mientras i < 3 y mágico hacer
        i ← i + 1
        mágico ← suma = SumaFila(cuad,3,i)
    fin_mientras
    //hallamos la suma de las columnas
    i ← 0
    mientras i < 3 y mágico hacer
        i ← i + 1
        mágico ← suma = SumaCol(cuad,3,i)
    fin_mientras
    //hallamos la suma de las diagonales
    si mágico entonces //si la variable mágico es cierta
        diagonal1 ← 0
        diagonal2 ← 0
        desde i ← 1 hasta 3 hacer
            diagonal1 ← diagonal1 + cuad[i,i]
            diagonal2 ← diagonal2 + cuad[i,4-i]
        fin_desde
    fin_si
    mágico ← mágico y diagonal1 = suma y diagonal2 = suma
    si mágico entonces
        escribir('La matriz es un cuadrado mágico')
    fin_si
fin

entero función SumaFila( E tabla : t ; E entero : n, i)
var
    entero : j, suma
inicio
    suma ← 0

```

```

desde j ← 1 hasta n hacer
    suma ← suma + t[i,j]
fin_desde
devolver(suma)
fin_función

entero función SumaCol( E tabla : t; E entero n, j)
var
    entero : i, suma
inicio
    suma ← 0
    desde i ← 1 hasta n hacer
        suma ← suma + t[i,j]
    fin_desde
    devolver(suma)
fin_función

```

6.20. Visualizar la matriz transpuesta de una matriz M de 6×7 elementos.

Análisis del problema

DATOS DE SALIDA: MT (matriz transpuesta de M)
 DATOS DE ENTRADA: M (matriz original)
 DATOS AUXILIARES: i, j (índice de las matrices)

Una matriz transpuesta (MT) a otra es aquella que tiene intercambiadas las filas y las columnas. Si una matriz M tiene 6 filas y 7 columnas, MT tendrá 7 filas y 6 columnas. Mientras se lee la matriz M, se puede obtener MT, ya que a cualquier elemento $M[i, j]$, le corresponderá $MT[j, i]$ en la matriz MT. Una vez leída, se escribe mediante otros dos bucles **desde** anidados.

Diseño del algoritmo

```

algoritmo ejercicio_6_20
var
    array[1..6,1..7] de entero : M
    array[1..7,1..6] de entero : MT
    entero : i,j
inicio
    desde i ← 1 hasta 6 hacer
        desde j ← 1 hasta 7 hacer
            leer(M[i,j])
            MT[j,i] ← M[i,j]
        fin_desde
    fin_desde
    desde i ← 1 hasta 7 hacer
        desde j ← 1 hasta 6 hacer
            escribir(MT[i,j])
        fin_desde
    fin_desde
fin

```

6.21. Diseñar una función con la que, se pueda comprobar si dos matrices pasadas como parámetros son idénticas.

Análisis del algoritmo

- DATOS DE SALIDA: Mensaje
 DATOS DE ENTRADA: A, B (las dos matrices), M, N (dimensiones de la matriz A), O, P (dimensiones de la matriz B)
 DATOS AUXILIARES: i, j (índices de las matrices), idénticas (variable lógica que valdrá verdadero si las dos matrices son iguales)

Para comprobar si dos matrices A y B son idénticas, lo primero que se ha de hacer es comprobar si sus dimensiones son iguales. Si esto es cierto se pasa a recorrer ambas matrices mediante dos bucles anidados. Como es posible que no sean idénticas, a veces no será necesario recorrer toda la matriz, sino sólo hasta que se encuentre un elemento diferente. Por esta razón no se utilizan bucles **desde** sino bucles **mientras** que se ejecutarán mientras i o j sean menores que los límites y sean idénticas las matrices (idéntica sea verdadero). Dentro del bucle interno se comprueba si $A[i, j]$ es distinto a $B[i, j]$, en cuyo caso se pone la variable idéntica a falso (antes de comenzar los bucles se ha puesto a cierto la variable). Si después de recorrer las tablas, idéntica es verdadero, la función devolverá un valor lógico verdadero.

Diseño del algoritmo

```
// suponemos creado el tipo vector
entero función MatrizIdéntica(ENTERO vector : A,B; ENTERO : M,N,O,P )
var
  entero : i,j
  lógico : idéntica
inicio
  idéntica ← M = O y N = P
  si idéntica entonces
    i ← 0
    mientras i < M y idéntica hacer
      j ← 0
      i ← i + 1
      mientras j < N y idéntica hacer
        j ← j + 1
        idéntica ← A[i,j] = B[i,j]
      fin_mientras
    fin_mientras
  fin_si
  devolver(idéntica)
fin_función
```

6.22. Un procedimiento que obtenga la matriz suma de dos matrices.

Análisis del problema

- DATOS DE SALIDA: S (matriz suma)
 DATOS DE ENTRADA: A, B (matrices a sumar), M, N (dimensiones de la matriz A), O, P (dimensiones de la matriz B)
 DATOS AUXILIARES: i, j (índices de las matrices)

Para realizar la suma de dos matrices, es necesario que ambas tengan las mismas dimensiones, por lo que lo primero que se ha de hacer con los parámetros M, N, O y P es comprobar que M es igual a O y N es igual a P.

Se leen las matrices A y B desde algún dispositivo de entrada y se realiza la suma. Si esto no ocurre, el parámetro de salida error devolverá un valor verdadero.

En la suma de matrices, cada elemento $S[i,j]$ es igual a $A[i,j] + B[i,j]$ y se debe, por tanto, recorrer las matrices A y B con dos bucles anidados para hallar la matriz suma.

Diseño del algoritmo

```

procedimiento MatrizSuma( E tabla : A,B; E entero : M,N,O,P;
                           S tabla : Suma; S lógico : error)

var
    entero : i, j
inicio
    si M <> O o N <> P entonces
        error ← V
    si_no
        error ← F
        desde i ← 1 hasta M hacer
            desde j ← 1 hasta O hacer
                Suma[i,j] ← A[i,j] + B[i,j]
            fin_desde
        fin_desde
    fin_si
fin_procedimiento
```

- 6.23.** Escribir el algoritmo de un subprograma que obtenga la matriz producto de dos matrices pasadas como parámetros.

Análisis del problema

DATOS DE SALIDA: Prod (matriz producto), error (da verdadero si hay un error en las dimensiones)

DATOS DE ENTRADA: A, B (matrices a multiplicar), M, N (dimensiones de la matriz A), O, P (dimensiones de la matriz B)

DATOS AUXILIARES: i, j, k (índices de las matrices)

Se recorren las matrices con dos bucles **desde** anidados siempre que N sea igual a O. La matriz producto tendrá una dimensión de $M \times P$. Cada elemento de la matriz producto $Prod[i,j]$ será:

$$A[i,1]*B[1,j]+A[i,2]*B[2,j]+\dots+A[i,N]*B[N,j]$$

por lo que dentro del bucle interno se necesita otro bucle **desde** para que vaya sacando números correlativos entre 1 y N.

El subprograma que se ha de utilizar será un procedimiento, ya que aunque devuelve un dato (la matriz producto), éste es estructurado, por lo que no puede relacionarse con el valor que devuelve una función. A dicho procedimiento se pasan como parámetros de entrada las dos matrices y sus dimensiones; como parámetro de salida la matriz producto **Prod** y **error**, que será un valor lógico verdadero si las matrices no se pueden multiplicar.

Diseño del algoritmo

```

procedimiento MatrizProducto(E tabla : A,B; E entero : M,N,O,P;
                           S tabla : Prod; S lógico error)

var
    entero : i,j,k
inicio
```

```

    si N <> 0 entonces
        error ← verdad
    si_no
        error ← falso
        desde i ← 1 hasta M hacer
            desde j ← 1 hasta P hacer
                Prod[i,j] ← 0
                desde k ← 1 hasta N hacer
                    Prod[i,j] ← Prod[i,j]+A[i,k]*B[k,j]
                fin_desde
            fin_desde
        fin_si
    fin_procedimiento

```

- 6.24.** Se tiene una matriz bidimensional de $m \times n$ elementos que se lee desde el dispositivo de entrada. Se desea calcular la suma de sus elementos mediante una función.

Análisis del problema

DATOS DE SALIDA: suma (suma de los elementos de la matriz)
 DATOS DE ENTRADA: tabla (matriz a procesar), m, n (dimensiones de tabla)
 DATOS AUXILIARES: i, j (índices de la matriz)

Para obtener la suma de los elementos de una matriz, se recorren mediante dos bucles `desde` anidados. Dentro del bucle interno se hace la asignación `suma ← suma + tabla[i,j]`. Antes de entrar en los bucles se ha de haber inicializado la variable suma a 0.

Diseño del algoritmo

```

entero función SumaTabla( E tabla : t; E entero : m,n)
var
    entero : i,j,suma
inicio
    suma ← 0
    desde i ← 1 hasta m hacer
        desde j ← 1 hasta m hacer
            suma ← suma + t[i,j]
        fin_desde
    fin_desde
    devolver(suma)
fin_función

```

- 6.25.** Una matriz A de m filas y n columnas es simétrica si m es igual a n y se cumple que

$$A_{ij} = A_{ji} \text{ para } 1 < i < m \text{ y } 1 < j < n$$

Se desea una función que tome como parámetro de entrada una matriz y sus dimensiones y devuelva un valor lógico que determine si se trata de una matriz simétrica o no.

Análisis del problema

DATOS DE SALIDA: Mensaje que nos indica si es o no simétrica
 DATOS DE ENTRADA: A (la matriz a comprobar), n, m (dimensiones de la matriz)
 DATOS AUXILIARES: i, j (índices de la matriz), simétrica (variable lógica que valdrá verdadero si A es simétrica)

Para comprobar si es simétrica primero se debe comprobar si m es igual a n y luego recorrer el *array* con dos bucles anidados que deberán terminar cuando se acaben de comprobar los elementos o cuando encuentren un elemento $A[i,j]$ distinto del $A[j,i]$. Por tanto se utilizan bucles **mientras** en vez de bucles **desde**.

Además, no es preciso recorrer todo el *array*. Si se comprueba que $A[3,4]$ es igual que $A[4,3]$, no hace falta ver si $A[4,3]$ es igual que $A[3,4]$. Lógicamente, cuando i es igual a j tampoco hace falta comprobar si $A[3,3]$ es igual a $A[3,3]$. Por tanto en el bucle externo, la i deberá ir tomando valores entre 1 y $M-1$. En el interno la j deberá ir tomando valores entre $i+1$ y N . Dentro del bucle interno, si $A[i,j] <> A[j,i]$ se pone la variable *simétrica* a falso con lo que se sale de los dos bucles.

Diseño del algoritmo

```

lógico función EsMatrizSimétrica( E tabla : A; E entero m,n )
var
    entero : i,j
    lógico : simétrica
inicio
    simétrica ← m = n
    si simétrica entonces
        i ← 0
        mientras i < M-1 y simétrica hacer
            i ← i + 1
            j ← i
            mientras j < N y simétrica hacer
                j ← j + 1
                simétrica A[i,j] = A[j,i]
            fin_mientras
        fin_mientras
    fin_si
    devolver(simétrica)
fin_función

```

- 6.26.** Una empresa de venta de productos por correo desea realizar una estadística de las ventas realizadas de cada uno de los productos a lo largo del año. Distribuye un total de 100 productos, por lo que las ventas se pueden almacenar en una tabla de 100 filas y 12 columnas. Se desea conocer:

El total de ventas de cada uno de los productos.

El total de ventas de cada mes.

El producto más vendido en cada mes.

El nombre, el mes y la cantidad del producto más vendido.

Como resultado final, se desea realizar un listado con el siguiente formato:

	Enero	Febrero	...	Diciembre	Total producto
Producto 1					
Producto 2					
...					
Producto 100					
Total mes					
Producto más vendido					

Nombre del producto y mes del producto más vendido en cualquier mes del año.

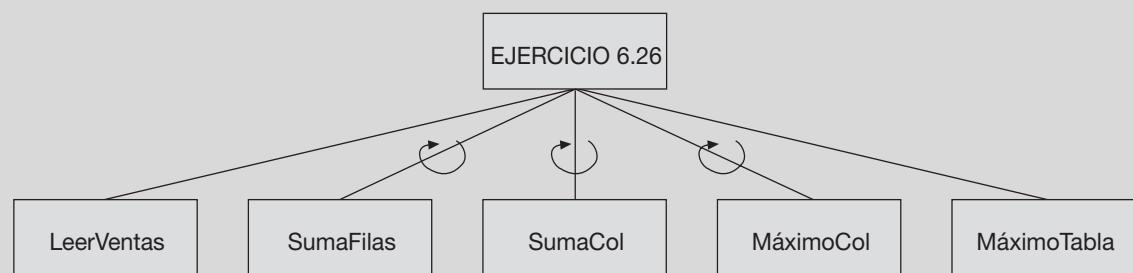
Análisis del problema

Para realizar este problema se utilizarán cinco *arrays*:



El *array* PRODUCTOS será un *array* de cadena, de 100 elementos en el que se guardarán los nombres de los productos. VENTAS es un *array* de dos dimensiones de tipo entero de 100 filas y 12 columnas que almacena las ventas de cada producto en cada mes del año. TOTALPROD será un *array* de 100 elementos en el que guarda el total de ventas anual de cada producto. TOTALMES tiene 12 posiciones y guarda el total de ventas de cada uno de los meses y MÁXIMOMES el número del producto más vendido cada uno de los meses del año.

El diseño modular del programa quedaría de la siguiente forma:



El programa principal llamará a un procedimiento que leerá los datos, es decir los nombres de los productos y las ventas de cada producto en cada uno de los meses. A partir de ahí, un bucle se encargará de ir sumando las filas, utilizando la función SumaFilas que ya desarrollamos anteriormente, y otro las columnas con la función SumaCol. En ambos bucles se almacenan los resultados en los *arrays* TOTALPROD y TOTALMES respectivamente. Una cuarta función se encargará de buscar la posición del máximo elemento de cada columna y una última buscará la fila y la columna (fmáx y cmáx) de elemento mayor del *array* para poder obtener el nombre, el mes y la cantidad del producto más vendido.

Para terminar, el listado se realizará dentro del propio programa principal que utilizará los *arrays* anteriores para la presentación de los resultados.

Diseño del algoritmo

```

algoritmo ejercicio_6_26
  tipo
    array [1..100,1..12] de entero : tabla
    array [1..12] de entero : vector
    array [1..100] de cadena : productos
    array [1..100] de entero : totales
  var
    tabla : tabla
    vector : TotalMes, Máximos
  
```

```
producto : prod
totales : tot
entero : i,fmáx,cmáx

inicio
    LeerDatos(ventas,prod,100,12)
    // cálculo del array con los totales por producto
    // (suma de filas)
    desde i ← 1 hasta 100 hacer
        tot[i] ← SumaFila(ventas,12,i)
    fin_desde
    // cálculo del array con los totales por mes
    // (suma de columnas)
    desde j ← 1 hasta 12 hacer
        TotalMes[j] ← SumaCol(ventas,100,j)
    fin_desde
    // cálculo del array con los máximos por mes
    // (máximo de columna)
    desde j ← 1 hasta 12 hacer
        Máximos[j] ← MáximoCol(ventas,100,j)
    fin_desde
    // obtención de la fila y columna de la tabla donde
    // se encuentra el máximo
    MáximoTabla(ventas,100,12,fmáx,cmáx)
    // listado de resultados
    desde i ← 1 hasta 100 hacer
        escribir(prod[i])
        desde j ← 1 hasta 12 hacer
            escribir(ventas[i,j]) // escribir en la
            // misma línea
        fin_desde
        // salto de línea
    fin_desde
    // escribir la fila con los totales por mes
    desde j ← 1 hasta 12 hacer
        escribir(TotalMes[j]) // escribir en la
        // misma línea
    fin_desde
    // escribir la fila con los máximos por mes
    // Como deseamos escribir el nombre del producto, y el
    // array está lleno con
    // la fila que ocupa la posición del máximo,
    // debemos escribir prod[Máximos[j]]
    desde j ← 1 hasta 12 hacer
        escribir(Prod[Máximos[j]]) // escribir en la
        // misma línea
    fin_desde
    // escribir el producto,el número del mes y la cantidad
    // que más se ha vendido
    escribir(prod[fmáx],cmáx,ventas[fmáx,cmáx])
fin
```

```

procedimiento LeerDatos(S tabla : v : tabla; S producto : p;
                        E entero : m,n)
var
    entero : i, j
inicio
    desde i ← 1 hasta m hacer
        leer(prod[i])
        desde j ← 1 hasta n hacer
            leer(ventas[i,j])
        fin_desde
    fin_desde
fin_procedimiento

// los procedimientos SumaFila y SumaCol ya se han implementado
// anteriormente

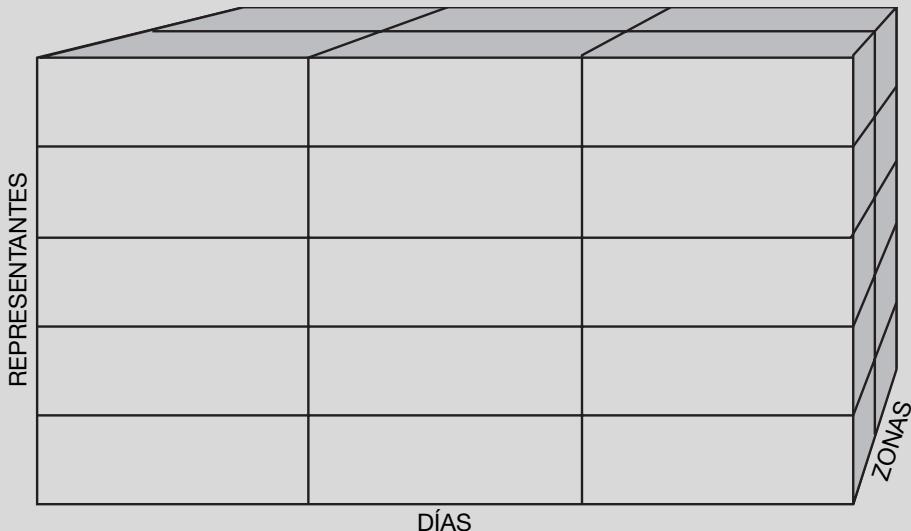
entero función MáximoCol(E tabla : t; E entero : n,j)
var
    entero : máx, i
inicio
    máx ← 1
    desde i ← 2 hasta n hacer
        si t[i,j] > t[máx,j] entonces
            máx ← i
        fin_si
    fin_desde
    devolver(máx)
fin_función
procedimiento MáximoTabla(E tabla : v; E entero : m,n; S entero : f,c)
var
    entero : i, j
inicio
    f ← 1
    c ← 1
    desde i ← 1 hasta m hacer
        desde j ← 1 hasta n hacer
            si t[i,j] > t[f,c] entonces
                f ← i
                c ← j
            fin_si
        fin_desde
    fin_desde
fin_procedimiento

```

- 6.27.** Una fábrica de muebles tiene 16 representantes que viajan por toda España ofreciendo sus productos. Para tareas administrativas el país está dividido en cinco zonas: Norte, Sur, Este, Oeste y Centro. Mensualmente almacena sus datos y obtiene distintas estadísticas sobre el comportamiento de sus representantes en cada zona. Se desea hacer un programa que lea los datos de todos los representantes con sus ventas en cada zona y calcule el total de ventas de una zona introducida por teclado, el total de ventas de un vendedor introducido por teclado en cada una de las zonas y el total de ventas de un día y para cada uno de los representantes.

Análisis del Problema

Una de las formas posibles de almacenar estos datos sería un *array* de 3 dimensiones. Se puede considerar que las filas son los representantes —de 1 a 16—, las columnas los días del mes —si se consideran meses de 31 días, de 1 a 31— y que esta estructura se repite cinco veces, una vez por cada zona. De esta forma se podría representar el *array* de la siguiente forma:



Al trabajar con un *array* de tres dimensiones es preciso utilizar tres índices, *r* para los representantes, *d* para los días y *z* para las zonas.

Una vez leído el *array*, para el primer proceso, se ha de introducir el número de la zona (*númzona*) y con esa zona fija, recorrer las filas y las columnas, acumulando el total en un acumulador. Para obtener el total de ventas de un vendedor, se introduce el número de representante (*númrep*) y con dicho índice fijo recorrer la tabla por zonas y días. Por fin, para el último punto se introduce el número del día (*númdía*) y recorreremos la tabla por representantes y zonas.

Diseño del algoritmo

```

algoritmo ejercicio_6_27
var
    array[1..16,1..31,1..5] de entero : ventas
    entero : r,d,z,númzona,númrep,númdía,suma
inicio
    // lectura de la tabla
    desde r ← 1 hasta 16 hacer
        desde d ← 1 hasta 31 hacer
            desde z ← 1 hasta 5 hacer
                leer(ventas[r,d,z])
            fin_desde
        fin_desde
    fin_desde
    // cálculo del total de una zona
    leer(númzona)
    suma ← 0
    desde d ← 1 hasta 31 hacer
        desde r ← 1 hasta 16 hacer
            
```

```

        suma ← suma + ventas[r,d,númzona]
    fin_desde
fin_desde
escribir(suma)
// cálculo de las ventas de un representante para cada una de las zonas
leer(númrep)
suma ← 0
desde z ← 1 hasta 5 hacer
    suma ← 0
    desde d ← 1 hasta 31 hacer
        suma ← suma + ventas[númrep,d,z]
    fin_desde
    escribir(suma)
fin_desde
// cálculo de las ventas de todos los representantes
// para un día determinado
leer(númdía)
suma ← 0
desde r ← 1 hasta 16 hacer
    suma ← 0
    desde z ← 1 hasta 5 hacer
        suma ← suma + ventas[r,númdía,z]
    fin_desde
    escribir(suma)
fin_desde
fin

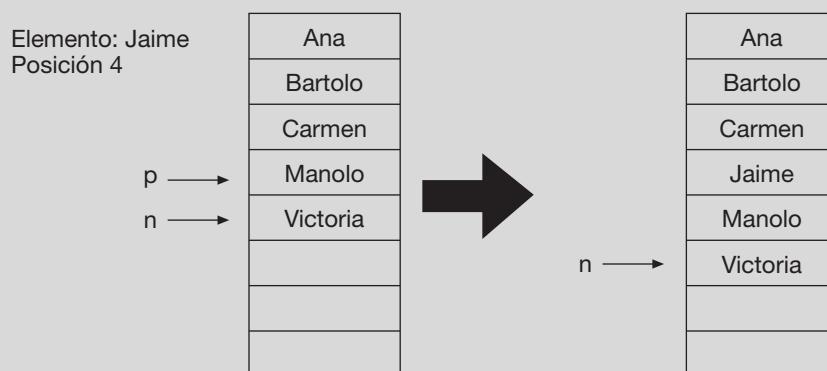
```

- 6.28.** Disponemos de un array unidimensional de MáxLista elementos de tipo cadena. Se desea hacer un procedimiento que inserte un elemento dentro del array en una posición determinada respetando el orden que tenía, siempre y cuando haya sitio para hacer la operación.

Análisis del problema

Para insertar un elemento en un *array* es necesario desplazar los elementos anteriores una posición hacia abajo, siempre y cuando haya sitio para insertarlo. Para controlar el número de elementos ocupados se utiliza una variable n. Por lo tanto sólo podremos hacer la operación si $n < \text{MáxLista}$.

Para dejar espacio para insertar el elemento se hará un bucle desde n hasta la posición donde se desea insertar menos uno, y hacer que cada elemento pase a la posición siguiente, es decir, a $\text{lista}[i+1]$ se le asigna $\text{lista}[i]$.



Una vez hecho esto en la posición deseada se introduce el nuevo elemento y se incrementa en 1 el número de posiciones ocupadas.

Al procedimiento se le pasan como parámetros el *array* (lista), el número de elementos ocupados (n), una variable lógica que informa si la operación se ha realizado con éxito (error), la posición donde se desea insertar el elemento (p), y el elemento a insertar (e).

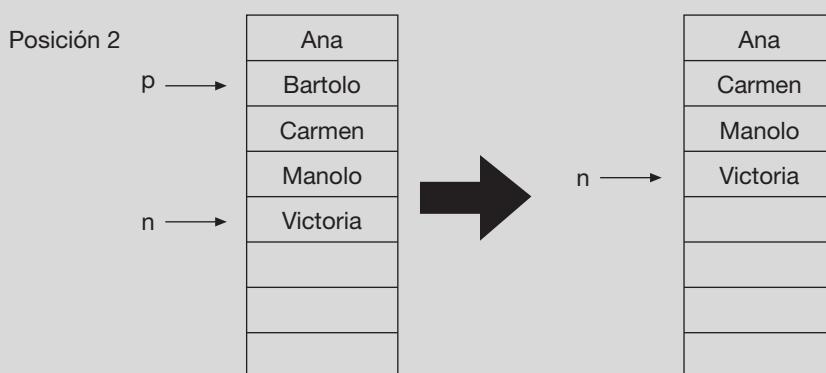
Diseño del algoritmo

```
procedimiento InsertarEnLista( E/S vector : lista; E/S entero: n;
                                S lógico : error;
                                E entero : p; E cadena : e)
var
    entero : i
inicio
    // consideraremos como error el que la lista esté llena o
    // o que la posición a insertar mas allá del número
    // de elementos + 1
    error ← n = MáxLista o p > n+1
    si no error entonces
        desde i ← n hasta p-1 incremento -1 hacer
            lista[i+1] ← lista[i]
        fin_desde
        lista[p] ← e
        n ← n + 1
    fin_si
fin_procedimiento
```

- 6.29.** Disponemos de un array unidimensional de MáxLista elementos de tipo entero. Se desea diseñar un procedimiento que elimine un elemento del array situado en una posición determinada que pasamos como parámetro, conservando el array en el mismo orden.

Análisis del problema

Para borrar un elemento es necesario subir todos los componentes de la lista, desde el lugar del elemento a borrar hasta la posición n-1. El número de elementos ocupados del *array* disminuirá en una unidad. El procedimiento detectará dos condiciones de error: que el número de elementos sea 0, o que la posición a borrar sea mayor que el número de elementos.



Diseño del algoritmo

```

procedimiento BorrarDeLista(E/S vector : lista; E/S entero : n;
                           S lógico : error; E entero : p)
var
    entero : i
inicio
    // consideramos como error el que la lista esté vacía
    // o que la posición a insertar mas allá del número
    // de elementos + 1 }
    error ← (n = 0) o (p > n)
    si no error entonces
        desde i ← p hasta n hacer
            lista[i] ← lista[i+1]
        fin_desde
        n ← n - 1
    fin_si
fin_procedimiento

```

- 6.30.** Algoritmo que triangule una matriz cuadrada y halle su determinante. En las matrices cuadradas el valor del determinante coincide con el producto de los elementos de la diagonal de la matriz triangulada, multiplicado por -1 tantas veces como hayamos intercambiado filas al triangular la matriz.

Análisis del problema

El proceso de triangulación y cálculo del determinante tiene las siguientes operaciones:

- Inicializar signo a 1.
- Desde i igual a 1 hasta $m-1$ hacer:
 - a) Si el elemento de lugar (i, i) es nulo, intercambiar filas hasta que dicho elemento sea no nulo o agotar los posibles intercambios. Cada vez que se intercambia se multiplica por -1 el valor de la variable signo.
 - b) A continuación se busca el primer elemento no nulo de la fila i -ésima y, en el caso de existir, se usa para hacer ceros en la columna de abajo.

Sea dicho elemento $\text{matriz}[i, r]$
 Multiplicar fila i por $\text{matriz}[i+1, r]/\text{matriz}[i, r]$ y restarlo a la $i+1$
 Multiplicar fila i por $\text{matriz}[i+2, r]/\text{matriz}[i, r]$ y restarlo a la $i+2$

 Multiplicar fila i por $\text{matriz}[m, r]/\text{matriz}[i, r]$ y restarlo a la m

Se deberá almacenar en una variable auxiliar, cs , el contenido de $\text{matriz}[i+x, r]$, para que, en las siguientes operaciones con la fila, resultado de

```

matriz[i+x, r+y] ← matriz[i+x, r+y] - matriz[i, r+y] *
                           (cs / matriz[i, r])

```

no afecte a

```

matriz[i+x, r] ← matriz[i+x, r] - matriz[i, r] *
                           (matriz[i+x, r] / matriz[i, r])

```

- Asignar al determinante el valor de signo por el producto de los elementos de la diagonal de la matriz triangulada.

Diseño del algoritmo

```
algoritmo ejercicio_6_30
const
    m = <expresión>           // en este caso m y n serán iguales
    n = <expresión>
tipo
    array[1..m, 1..n] de real : arr
var
    arr : matriz
    real : dt
inicio
    llamar_a leer_matriz(matriz)
    llamar_a triangula(matriz, dt)
    escribir('Determinante = ', dt)
fin

procedimiento leer_matriz (S arr : matriz)
var
    entero: i,j
inicio
    escribir('Deme los valores para la matriz')
    desde i ← 1 hasta m hacer
        desde j ← 1 hasta n hacer
            leer( matriz[i, j])
        fin_desde
    fin_desde
fin_procedimiento

procedimiento escribir_matriz (E arr : matriz)
var
    entero : i,j
    carácter : c
inicio
    escribir('Matriz triangulada')
    desde i ← 1 hasta m hacer
        desde j ← 1 hasta n hacer
            escribir( matriz[i, j])      // Escribir en la misma línea
        fin_desde
        // Saltar a la siguiente línea
    fin_desde
    escribir('Pulse tecla para continuar')
    leer(c)
fin_procedimiento

procedimiento interc(E/S real: a,b)
var
    real : auxi
inicio
    auxi ← a
    a ← b
    b ← auxi
```

```

fin_procedimiento

procedimiento triangula (E arr : matriz;  S real d: t)
var
    entero: signo
    entero: t,r,i,j
    real   : cs
inicio
    signo ← 1
    desde i ← 1 hasta m - 1 hacer
        t ← 1
        si matriz[i, i] = 0 entonces
            repetir
                si matriz[i + t, i] <> 0 entonces
                    signo ← signo * (-1)
                    desde j ← 1 hasta n hacer
                        llamar_a interc(matriz[i,j],  matriz[i + t,j])
                fin_desde
                llamar_a escribir_matriz(matriz)
            fin_si
            t ← t + 1
            hasta_que (matriz[i, i] <> 0) o (t = m - i + 1)
        fin_si
        r ← i - 1
        repetir
            r ← r + 1
        hasta_que (matriz[i, r] <> 0) o (r = n)
        si matriz[i, r] <> 0 entonces
            desde t ← i + 1 hasta m hacer
                si matriz[t, r] <> 0 entonces
                    cs ← matriz[t, r]
                    desde j ← r hasta n hacer
                        matriz[t,j]_ matriz[t, j]-matriz[i, j] * (cs/matriz[i, r])
                fin_desde
                llamar_a escribir_matriz(matriz)
            fin_si
            fin_desde
        fin_si
        fin_desde
    dt ← signo
    desde i ← 1 hasta m hacer
        dt ← dt * matriz[i, i]
    fin_desde
fin_procedimiento

```

- 6.31.** Se desean almacenar los datos de un producto en un registro. Cada producto debe guardar información concerniente a su Código de Producto, Nombre, y Precio. Diseñar la estructura de datos correspondiente y un procedimiento que permita cargar los datos de un registro.

Análisis del problema

La estructura de datos del registro utilizado va a tener tres campos simples: Código, que será una cadena de caracteres, Nombre, que será otra cadena, y Precio, que será un número entero. Como se utiliza un procedimiento de lectura, es necesario crear un tipo de dato Producto, que será el tipo de dato utilizado como parámetro al subprograma.

Diseño del algoritmo

```

tipo
    registro : producto
        cadena : Código , Nombre
        entero : Precio
    fin_registro
var
    producto : p
    ...

procedimiento LeerProducto(s producto : p)
inicio
    leer(p.Código)
    leer(p.Nombre)
    leer(p.Precio)
fin_procedimiento

```

- 6.32.** Una farmacia desea almacenar sus productos en una estructura de registros. Cada registro tiene los campos Código, Nombre, Descripción del Medicamento (antibiótico, analgésico, etc.), Laboratorio, Proveedor, Precio, Porcentaje de IVA, Stock y Fecha de Caducidad. La fecha deberá guardar por separado el día, mes y año. Diseñe la estructura de datos y un procedimiento que permita escribir los datos de un medicamento.

Análisis del problema

Es necesario un tipo de dato con el registro, ya que será utilizado como parámetro de un procedimiento. La estructura del tipo de registro tendrá los siguientes campos: Código de tipo cadena, Nombre de tipo cadena, Descripción también de tipo cadena, Precio de tipo entero, IVA que indica el porcentaje del IVA a aplicar en formato real (por ejemplo, 0.10 para un 10%), Stock de tipo entero y Caducidad. Caducidad será a su vez otro registro que tendrá los campos dd, mm y aa, que serán los tres de tipo entero. La definición de este último campo puede hacerse de dos formas; por una parte se puede definir un tipo de registro independiente, por ejemplo Fecha y luego hacer referencia a él en la definición del dato Caducidad; también se podría definir el registro Caducidad dentro del registro principal, utilizando lo que se llaman registros anidados.

Diseño del algoritmo

```

tipo
    registro : Fecha
        entero : dd, mm, aa
    fin_registro
    registro : Medicamento registro

```

```

cadena : Código, Nombre, Descripción
entero : Precio
real : IVA
entero : Stock
Fecha : Caducidad
fin_registro
var
    Medicamento : m

```

La declaración del tipo `Medicamento` también se podría hacer de la siguiente forma sin necesidad de crear el tipo `Fecha`:

```

registro : Medicamento
    cadena : Código, Nombre, Descripción
    entero : Precio
    real : IVA
    entero : Stock
    registro : Caducidad
        entero : dd, mm, aa
    fin_registro
fin_registro

```

La implementación del procedimiento de escritura quedaría como sigue:

```

procedimiento EscribirMedicamento(s Medicamento : m)
inicio
    escribir(m.Código)
    escribir(m.Nombre)
    escribir(m.Descripción)
    escribir(m.Precio)
    escribir(m.IVA)
    escribir(m.Stock)
    escribir(m.Caducidad.dd)
    escribir(m.Caducidad.mm)
    escribir(m.Caducidad.aa)
fin_procedimiento

```

6.33. Diseñar la estructura de datos necesaria para definir un cuadrilátero utilizando coordenadas polares.

Análisis del problema

Para definir un Punto en el plano utilizando coordenadas polares es preciso utilizar dos datos: el ángulo y el radio. Como cada inicio de línea es el final del siguiente, se puede guardar sólo un punto por línea. Un cuadrilátero está formado por cuatro líneas, por lo que se puede definir un registro `Cuadrilátero` que tendrá cuatro campos de tipo `Punto`, es decir, a su vez registros que guardan el ángulo y el radio del punto final de cada lado. Sin embargo, es preciso guardar también la posición de inicio de cada línea, por lo que además se guardará la posición de inicio de la primera línea en otro campo del registro `Cuadrilátero`. La dirección del último lado será la misma que la de inicio.

Diseño del algoritmo

```

tipo
    registro : Punto
        real : radio, ángulo
    fin_registro
    registro : Cuadrilátero
        Punto: Inicio, Lado1, Lado2, Lado3, Lado4
    fin_registro

```

- 6.34.** Una empresa tiene almacenados a sus vendedores en un registro. Por cada vendedor se guarda su DNI, Apellidos, Nombre, Zona, Sueldo Base, Ventas Mensuales, Total Anual y Comisión. Las ventas mensuales será un vector de 12 elementos que guardará las ventas realizadas en cada uno de los meses. Total Anual será la suma de las ventas mensuales del vendedor. La Comisión se calculará aplicando un porcentaje variable al Total Anual del vendedor. Dicho porcentaje variará según las ventas anuales del vendedor, según la siguiente tabla:

Hasta de 1.500.000	0,00%
Más de 1.500.000 y hasta 2.150.000	13,75%
Más de 2.150.000 y hasta 2.900.000	16,50%
Más de 2.900.000 y hasta 3.350.000	17,60%
Más de 3.350.000	18,85%

Dicha tabla se habrá cargado de un archivo secuencial que contiene tanto el límite superior como el porcentaje.

Diseñar las estructuras de datos necesarias y realizar un algoritmo que permita leer los datos del empleado, calcule el Total Anual y obtenga la Comisión que se lleva el empleado mediante la tabla descrita anteriormente, que previamente se habrá tenido que cargar del archivo.

Análisis del problema

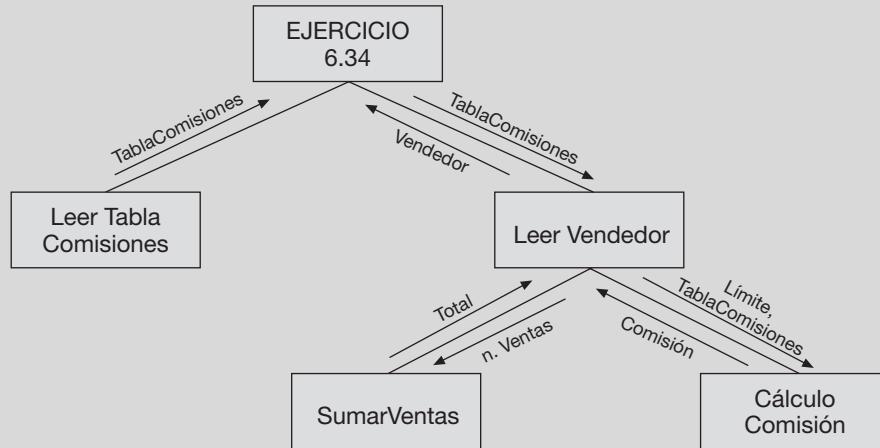
En este caso los campos del registro vendedor serán: DNI de tipo cadena, Nombre de tipo cadena, Apellidos de tipo cadena, Zona también de tipo cadena, Sueldo de tipo entero, Ventas que será un array de enteros, Total de tipo entero y Comisión que será real. Se define también un registro para la tabla de comisiones. Aunque el enunciado habla de un límite inferior y otro superior, la tabla sólo va a tener dos campos: Límite de tipo entero y Porcentaje que será real.

Para realizar el cálculo de la comisión se han de cargar previamente los datos de la tabla en un array. Aunque lo más corriente sería mantener los datos de la tabla en un archivo, vamos a realizar la lectura de los datos desde teclado antes de la llamada al procedimiento que carga los datos del vendedor. Hacer la lectura antes del procedimiento y no desde dentro de él, agilizará la operación de lectura del registro si hay que leer más de uno. En dicho procedimiento, para optimizar la búsqueda de los datos se obliga a que cada límite introducido sea mayor que el anterior. También se podrían haber introducido todos los datos y ordenar el vector con algunos de los métodos que se abordarán en el capítulo siguiente.

El procedimiento de lectura leerá los campos DNI, Apellidos, Nombre, Zona y Sueldo. Para leer las ventas se debe utilizar un bucle que lea cada uno de los elementos de la tabla. Para calcular el total también es necesario recorrer el campo Ventas acumulando el valor de cada uno de los elementos. Para ello se utiliza una función SumarVentas que suma los elementos de un vector de enteros de n posiciones.

La comisión se calculará utilizando la función CálculoComisión. Irá analizando secuencialmente los elementos de la tabla. Como los límites se han introducido de menor a mayor, cuando se encuentre un límite mayor que el total de las ventas, se habrá encontrado la posición donde está el porcentaje de comisión a aplicar.

Por tanto, la arquitectura de nuestro algoritmo podría quedar de la forma siguiente:



Diseño del algoritmo

```

algoritmo ejercicio_6_34
  tipo
    array[1..12] de entero : vector
    registro : RegistroComisiones
      entero : Límite, Porcentaje
    fin_registro
    registro : Vendedores
      cadena : DNI, Apellidos, Nombre, Zona
      vector : Ventas
      entero : Sueldo, Total, Comisión
    fin_registro
    array[1..5] de RegistroComisiones : Comisiones
  var
    Comisiones : TablaComisiones
    Vendedores : Vendedor
  inicio
    LeerTablaComisiones(TablaComisiones)
    LeerVendedor(TablaComisiones, Vendedor)
  fin

  procedimiento LeerTablaComisiones(s Comisiones : T)
  var
    entero : i
  inicio
    desde i ← 1 hasta 5 hacer
      // Por razones de facilidad de búsqueda, el límite que deberemos
      // introducir será el inmediato superior al fijado en la tabla
      // descrita más arriba. Los valores a introducir serían:
  
```

```
//          Límite          Porcentaje
//          _____
//          1500001          0.00
//          2150001          13.75
//          2900001          16.50
//          3350001          17.60
//          9999999          18.85
// Suponiendo que la venta máxima sea de 9.999.999
leer(T.Límite)
leer(T.Porcentaje)
fin_desde
fin_procedimiento

procedimiento LeerVendedor(E Comisiones : T; S Vendedores : v)
var
    entero : i
inicio
    leer(v.DNI)
    leer(v.Apellidos)
    leer(v.Nombre)
    leer(v.Zona)
    leer(v.Sueldo)
    desde i ← 1 hasta 12 hacer
        leer(v.Ventas[i])
    fin_desde
    v.Total ← SumarVentas(v.Ventas,12)
    v.Comisión ← CálculoComisión(T,v.Total)
fin_procedimiento

entero función SumarVentas(E vector : v; E entero : n)
var
    entero : i, total
inicio
    total ← 0
    desde i ← 1 hasta 12 hacer
        total ← total + v[i]
    fin_desde
    devolver(total)
fin_función

real función CálculoComisión(E Comisiones : T; E entero : total)
var
    entero : i
inicio
    i ← 1
    mientras T[i].Límite < total y i <= 5 hacer
        i ← i + 1
    fin_mientras
    //Si las ventas son mayores que el límite máximo (9.999.999)
    // saldríamos del bucle por la segunda condición, y i valdría 5
    devolver(total * T[i].Porcentaje)
fin_función
```

6.35. Podemos definir un polígono definiendo las coordenadas de cada uno de sus lados. Diseñar la estructura de datos que permita definir un polígono de lado n —con un máximo de 30 lados— y crear un algoritmo que permita introducir las coordenadas cartesianas de cada uno de sus lados.

Análisis del problema

Para definir un polígono de n lados, hay que definir cada una de las líneas que lo forman. Para definir una línea utilizando coordenadas cartesianas, son necesarias dos parejas de valores x , y que marcarán el inicio y el fin de cada línea en el plano. Por tanto se han de definir las siguientes estructuras de datos:

- Punto para almacenar la situación de un punto en el plano.
- Línea, formado una pareja de puntos que conforman el inicio y fin de la línea.
- Polígono que será un array de líneas. El número de elementos del array será de 30, ya que este es el número máximo que indica el enunciado. De estos 30 elementos, sólo se llenarán n , que será un parámetro pasado al procedimiento.

Para introducir los datos del polígono el procedimiento recibirá como parámetro de entrada el número de lados (n), que debe ser siempre mayor que 2, y devolverá el array de líneas como parámetro de salida. El procedimiento irá preguntando los valores de cada uno de los puntos que configuran el polígono, teniendo en cuenta que el fin de una línea será el comienzo de otra. El final de la última línea del polígono será el primer punto introducido.

Diseño del algoritmo

```

const
    MáxLados = 30 // Número máximo de lados especificado en el enunciado
tipo
    registro : Punto
        entero : x, y
    fin_registro
    registro : Línea
        Punto: Inicio, fin
    fin_registro
    array[1..30] de Línea : Polígono
var
    Polígono : P
    entero : n
    ...
    ...
procedimiento LeerPolígono(s Polígono : P; E entero : n)
var
    entero : i
inicio
    i ← 1
    LeerPunto(P[i].Inicio)
repetir
    LeerPunto(P[i].Fin)
    i ← i + 1
    P[i].Inicio ← P[i-1].Fin
hasta_que i = n
    P[n].Fin ← P[1].Inicio
fin_procedimiento

```

```
procedimiento LeerPunto(S Punto : p)
inicio
    leer(p.x)
    leer(p.y)
fin_procedimiento
```


LAS CADENAS DE CARACTERES

@Ug'Wa di hUXcfUg'bcfa Ua YbhY'gi []YfYb'cdYfUWcbYg'Uf]ha fhjWg'Y^YWhlXUg'geVfYXlhcg'bi a ff]Wg'glb Ya VUf[c 'WXXUj Yn 'Yg' a zg'ZfYWYbhY'Y'i gc 'XY' Ug'Wa di hUXcfUg'dfUdfcWgUF'XUhcg'XY'hdc'UZUbi ! a ff]Wz' WXYbUg' XY' WfUWYfYg'" 9' cV^Yhj c' XY' YghY' Wdjh'i 'c' Yg' Y' Yghi X|c' XY' dfcWgUa JYbhcm'cg'a fhcXcg'XY'a Ub]di 'WJDE'XY'WXYbUg'df cdccfWcbUXcg'dcf ``cg'Yb[i UYg'XY'Wa di hUXcfU'

7.1. CADENAS

Una cadena es un conjunto de caracteres, incluido el espacio en blanco, que se almacenan en un área contigua de la memoria central. La *longitud* de una cadena es el número de caracteres que contiene. La cadena que no contiene ningún carácter se denomina cadena vacía o nula y su longitud es cero; no se debe confundir con la cadena compuesta sólo de espacios en blanco, ya que ésta tendrá como longitud el número de blancos de la misma.

Constantes

Una constante de tipo cadena es un conjunto de caracteres válidos encerrados entre comillas, aquí se considerarán comillas simples, aunque muchos lenguajes utilizan las comillas dobles.

Variables

Una variable de cadena es aquella cuyo contenido es una cadena de caracteres. Atendiendo a la declaración de la longitud se dividen en:

- **Estáticas.** Su longitud se define antes de ejecutar el programa y no puede cambiarse a lo largo de éste.
- **Semiestáticas.** Su longitud puede variar durante la ejecución del programa, pero sin sobrepasar un límite máximo declarado al principio.
- **Dinámicas.** Su longitud puede variar sin limitación dentro del programa.

La representación de las diferentes variables de cadena en memoria utiliza un método de almacenamiento diferente. Las **cadenas de longitud fija** se consideran vectores de la longitud declarada con

blancos a la izquierda o derecha si la cadena no alcanza la longitud declarada. Las **cadenas de longitud variable** se consideran similares a registros con un campo de tipo vector y otros dos que permiten almacenar la longitud máxima y la longitud actual de la cadena. Las **cadenas de longitud indefinida** se representan mediante listas enlazadas, listas que se unen mediante punteros.

7.2. OPERACIONES CON CADENAS

En general, las instrucciones básicas, *asignación* y *entrada/salida*, se ejecutan de modo similar a como se ejecutan dichas instrucciones con datos numéricos. Por otra parte, según el tipo de lenguaje de programación elegido se tendrá mayor o menor facilidad para la realización de operaciones y hay que destacar que los nuevos lenguajes orientados a objetos C# y Java, merced a la clase *String* soportan una gran variedad de funciones para la manipulación de cadenas. En cualquier caso, las operaciones más usuales son:

- Cálculo de la longitud.
- Comparación.
- Concatenación.
- Extracción de subcadenas.
- Búsqueda de información.

La *longitud de una cadena*, como ya se ha comentado, es el número de caracteres de la cadena y la operación de determinación de la longitud de una cadena se representará por la función *longitud*, que recibe un argumento de tipo cadena y devuelve un resultado numérico entero.

La *comparación de cadenas* es una operación muy importante y los criterios de comparación se basan en el orden numérico del código o juego de caracteres que admite la computadora o el propio lenguaje de programación. En nuestro lenguaje algorítmico utilizaremos el código ASCII como código numérico de referencia. Para que dos cadenas sean iguales han de tener el mismo número de caracteres y cada carácter de una ser igual al correspondiente carácter de la otra. Al comparar cadenas la presencia de un carácter, aunque sea el blanco, se considera mayor que su ausencia. Además, la comparación de cadenas distingue entre mayúsculas y minúsculas puesto que su código no es el mismo, las letras mayúsculas tienen un número de código menor que las minúsculas. Para comprobar la igualdad o desigualdad entre cadenas generalmente basta con utilizar los operadores de relación, aunque en lenguajes como Java y C# resulta conveniente utilizar los diversos métodos que, en este sentido, proporcionan y analizar las diferencias que, en dichos lenguajes, presentan estos métodos con respecto al empleo de los operadores de relación.

La *concatenación* es la operación de reunir varias cadenas de caracteres en una sola, pero conservando el orden de los caracteres de cada una de ellas. El símbolo que representa la concatenación varía de unos lenguajes a otros: + & // o. En el pseudocódigo se utilizará + y en ocasiones &.

Otra operación —función— importante de las cadenas es aquella que permite la extracción de una parte específica de una cadena: *subcadena*. La operación *subcadena* se representa como:

```
subcadena (una_cadena, inicio, longitud_subcadena)
subcadena (una_cadena, inicio)
```

donde

una_cadena es la cadena de la que debe extraerse una subcadena,

inicio es un número o expresión numérica entera que corresponde a la posición, inicial de la subcadena,

longitud_subcadena es la longitud de la subcadena. Si este parámetro no se especifica se devuelve una nueva cadena que comienza donde indica inicio y se extiende hasta el final de la cadena original.

Una operación frecuente a realizar con cadenas es *localizar* si una determinada cadena forma parte de otra cadena más grande o *buscar* la posición en que aparece un determinado carácter o secuencia de caracteres de un texto. Estos problemas pueden resolverse con las funciones de cadena estudiadas hasta ahora, pero será necesario diseñar los algoritmos correspondientes. Esta función suele ser interna en algunos lenguajes y se define por índice o posición, y su formato es

```
posición (una_cadena, una_subcadena)
```

7.3. FUNCIONES ÚTILES PARA LA MANIPULACIÓN DE CADENAS

Además de las funciones ya comentadas al hablar de las operaciones básicas existen otras de gran utilidad, que habitualmente suelen encontrarse predefinidas (aunque a veces como procedimientos) en la mayor parte de los lenguajes de programación y se denominan **valor** y **aCadena** o **cad**

valor (una_cadena)	Convierte la cadena en un número; siempre que la cadena fuese de dígitos numéricos.
aCadena (valor)	Convierte un valor numérico en una cadena.

Otras funciones de tipo carácter muy utilizadas para transformar números en cadenas, cadenas en números o una cadena en mayúsculas a minúsculas y viceversa son: **código** (a la que también se denominará **aCódigo**) y **car** (llamada también **aCarácter**)

aCódigo (un_carácter)	Devuelve el código ASCII de un carácter.
aCarácter (un_código)	Devuelve el carácter asociado a un código ASCII.

7.4. EJERCICIOS RESUELTOS

7.1. Suponiendo que en su lenguaje algorítmico sólo están implementadas las funciones de cadena **subcadena**, **posición**, y **longitud**, diseñar funciones que permitan:

- Extraer los n primeros caracteres de una cadena.
- Extraer los n últimos caracteres de una cadena.
- Eliminar los espacios en blanco que haya al final de la cadena.
- Eliminar los espacios en blanco que haya al comienzo de la cadena.
- Eliminar de una cadena los n caracteres que aparecen a partir de la posición p .
- Eliminar la primera aparición de una cadena dentro de otra.
- Insertar una cadena dentro de otra a partir de la posición p .
- Sustituir una cadena por otra.
- Contar el número de veces que aparece una cadena dentro de otra.
- Borrar todas la apariciones de una cadena dentro de otra.
- Sustituir todas la apariciones de una cadena dentro de otra, por una tercera.

Análisis del problema

- Esta función equivaldría a la función **LEFT** que implementan algunos lenguajes. Debería devolver una subcadena de la cadena principal, a partir del carácter 1 los n primeros caracteres. Si n es 0, el resultado sería una cadena nula. Si es mayor que la longitud de la cadena el resultado sería la cadena original.

- b) Equivaldría a la función **RIGHT**. Tendrá que devolver una subcadena, formada por los caracteres situados a partir de la posición *longitud_de_la_cadena* - n + 1 hasta el final. Si n es 0, el resultado sería una cadena nula. Si es mayor que la longitud de la cadena el resultado sería la cadena original.
- c) Se recorre la cadena de derecha a izquierda hasta encontrar un carácter distinto de blanco. Una vez localizada esa posición se utiliza la función del apartado a) para extraer los caracteres situados a la izquierda. Si no tiene espacios en blanco al final de la cadena el resultado sería la cadena original. Si todos los caracteres son blancos el resultado sería una cadena vacía.
- d) Similar a la anterior, es necesario recorrer la cadena de izquierda a derecha hasta encontrar un carácter distinto de blanco. Una vez localizado dicho carácter se extraen mediante la función **subcadena** los caracteres restantes. Si no tiene espacios en blanco al inicio de la cadena, el resultado sería igual a la cadena original. Si todos los caracteres son blancos el resultado sería una cadena vacía.
- e) El resultado será la concatenación de una subcadena formada desde el carácter 1 hasta el carácter p-1, con otra extraída a partir del carácter p+n. Si la posición a partir de la que se va a borrar es 0 o es mayor que la longitud de la cadena, el resultado sería la cadena original. Si el número de caracteres a borrar es 0, el resultado también sería la cadena original. Si el número de caracteres a borrar es mayor que los caracteres restantes a la posición p, borraría toda la cadena a partir de la posición p.
- f) Mediante la función **posición** se busca la primera aparición de la cadena. Una vez localizada, se llama a la función del apartado e), pasando como parámetros la posición y la longitud de la cadena. Si la cadena a borrar no existe el resultado será la cadena original.
- g) Se deben concatenar tres cadenas. Primero la cadena formada por los caracteres entre el 1 y el p. Luego vendría la cadena a insertar, y por último la cadena formada por los caracteres entre el p y el final de la cadena. Si la posición a insertar es menor que 0, el resultado sería la cadena original. Si la posición es 0, se insertará como primer carácter de la cadena. Si la posición es mayor que la longitud de la cadena, la insertaría al final de la misma.
- h) Para sustituir una cadena por otra, es necesario utilizar las funciones de borrado e inserción. Primero se borra la cadena buscada y posteriormente se sustituye por la otra. Si la cadena a borrar no se encuentra en la cadena principal, el resultado sería la cadena original.
- i) Debemos implementar un bucle que se ejecute mientras la cadena buscada esté incluida en la principal. Dentro del bucle se incrementa un contador y se vuelve a buscar a partir de la posición siguiente a donde se encontrara.
- j) Similar a la anterior, pero dentro del bucle se debe borrar la cadena.
- k) Similar a la anterior, pero dentro del bucle se llama a la función **sustituir**.

Diseño del algoritmo

- a)

```
cadena función Izquierda( E cadena : c; E entero : n)
    inicio
        devolver(subcadena(c,1,n))
    fin_función
```
- b)

```
cadena función Derecha( E cadena : c; E entero n)
    inicio
        si n > longitud(c) entonces
            devolver(c)
        si_no
            devolver(subcadena(c,longitud(c) - n + 1))
        fin_si
    fin_función
```
- c)

```
cadena función SinBlancosDer( E cadena : c)
    var
```

```
    entero : i,p
inicio
    i ← longitud(c)
    mientras subcadena(c,i,1) = ' ' y i <> 0 hacer
        i ← i - 1
    fin_mientras
    devolver(Izquierda(c,i))
fin_función

d) cadena función SinBlancosIzq(E cadena : c)
var
    entero : i
inicio
    i ← 1
    mientras subcadena(c,i,1) = ' ' y i <= longitud(c) hacer
        i ← i + 1
    fin_mientras
    devolver(subcadena(c,i))
fin_función

e) cadena función Borrar(E cadena : c; E entero : p,n)
inicio
    si p <= 0 o n <= 0 entonces
        devolver(c)
    si_no
        devolver(subcadena(c,1,p-1) & subcadena(c, p+n))
    fin_si
fin_función

f) cadena función BorrarCadena(E cadena : c,borrada)
var
    p : entero
inicio
    p ← pos(c,borrada)
    si p <= 0 entonces
        devolver(c)
    si_no
        devolver(Borrar(c,p,longitud(borrada)))
    fin_si
fin_función

g) cadena función Insertar(E cadena : c,insertada; E entero : p)
inicio
    si p <= 0 entonces
        devolver(c)
    si_no
```

```

        devolver(subcadena(c,1,p-1) & insertada & subcadena(c,p))
    fin_si
fin_función

h) cadena función Sustituir(E cadena:c,borrada,sustituida)
var
    p : entero
inicio
    p ← posición(c,borrada)
    si p = 0 entonces
        devolver(c)
    si_no
        devolver(Insertar(Borrar(c,p,longitude(borrada)),sustituida,p)
    fin_si
fin_función

i) entero función Ocurrencias(E cadena : c,buscada)
var
    i,p : entero
inicio
    i ← 0
    p ← posición(c,buscada)
    mientras p <> 0 hacer
        i ← i + 1
        c ← subcadena(c,p+1)
        p ← posición(c,buscada)
    fin_mientras
    devolver(i)
fin_función

j) cadena función BorrarTodas(E cadena : c,buscada)
inicio
    mientras posición(c,buscada) <> 0 hacer
        c ← BorrarCadena(c,buscada)
    fin_mientras
    devolver(c)
fin_función

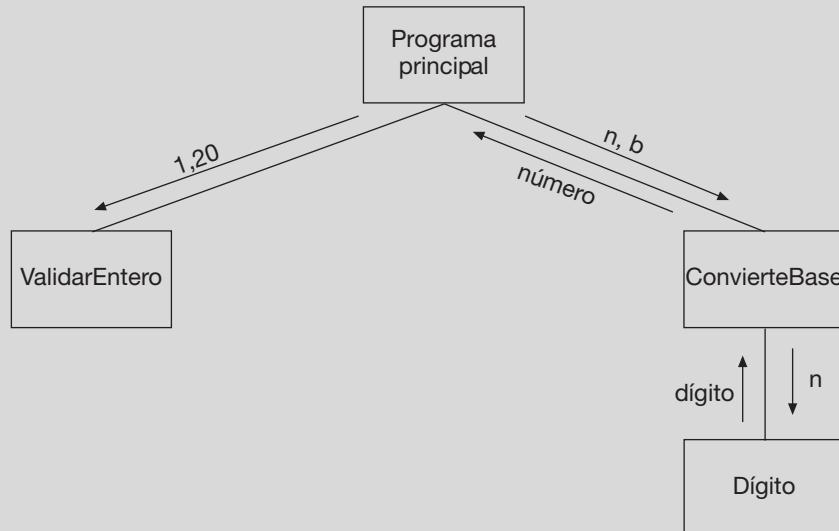
k) cadena función SustituirTodas(E cadena:c, buscada,sustituida)
inicio
    mientras posición(c,buscada) <> 0 hacer
        c ← Sustituir(c,buscada,sustituida)
    fin_mientras
    devolver(c)
fin_función

```

- 7.2.** Diseñar un algoritmo que mediante una función permita cambiar un número n en base 10 a la base b , siendo b un número entre 2 y 20.

Análisis del problema

La descomposición modular del problema sería la siguiente:



El programa principal se encargará de hacer las llamadas a `ValidarEntero` y a `ConvierteBase`, además de leer el número y escribir el resultado final. `ValidarEntero` es la función desarrollada más arriba. El núcleo del algoritmo está en la función `ConvierteBase`. Para convertir un número en base 10 a otra base, es preciso dividir el número entre la base y repetir la operación hasta que el cociente sea menor que la base. En ese momento, se debe tomar el último cociente y los restos de forma inversa a como han ido saliendo. Para obtener el resultado, `ConvierteBase` será una función de cadena, que devolverá el número convertido en cadena cuyo resultado irá concatenando los restos y el cociente en orden inverso a como han ido saliendo.

Si la base es mayor que 10, se debe convertir el número en una letra, A será 10, B será 11, etc. De esta conversión se encarga el módulo `Dígito`, que convierte los restos en un carácter. Si los restos son menores que 9, devuelve el propio número convertido a cadena —mediante la función estándar `cadena`, si no, devolverá A,B,C según el caso—.

Diseño del algoritmo

```

algoritmo ejercicio_7_2
var
    entero : n,b
inicio
    leer(n)
    b ← ValidarEntero(2,20)
    escribir(ConvierteBase(n,b))
fin
carácter función Dígito(E entero : n)
inicio
    si n < 10 entonces
        devolver(cadena(aux))
    fin
  
```

```

    si_no
        según_sea n hacer
            10 : devolver('A')
            11 : devolver('B')
            12 : devolver('C')
            13 : devolver('D')
            14 : devolver('E')
            15 : devolver('F')
            16 : devolver('G')
            17 : devolver('H')
            18 : devolver('I')
            19 : devolver('J')
            20 : devolver('K')
    fin_según
fin_función

cadena función ConvierteBase( E entero : n,b)
var
    cadena : aux
    entero : resto
inicio
    aux ← ''
    mientras n >= b hacer
        resto ← n mod b
        aux ← Dígito(resto) & aux
        n ← n div b
    fin_mientras
    devolver(Dígito(n) & aux)
fin_función

```

- 7.3.** Escribir el algoritmo de una función que convierta una cadena en mayúsculas y otra que la convierta en minúsculas.

Análisis del problema

Suponiendo que se esté utilizando el juego de caracteres ASCII, la diferencia entre el código ASCII de una letra en minúscula y otra en mayúscula es de 32. Por lo tanto para convertir una cadena en mayúsculas, habrá que ir recorriendo la cadena carácter a carácter; si el carácter se trata de una letra minúscula, es decir es mayor o igual a «a» y menor o igual a «z», restaremos 32 al código ASCII de la letra (utilizando la función `aCódigo()`) y se obtendrá el carácter correspondiente a dicho código utilizando la función `aCarácter()`). El proceso para convertir a minúsculas es igual, pero sumando 32 al código ASCII, en vez de restando.

Esta situación es válida para los caracteres ASCII estándar (hasta el 127). Para el resto de los caracteres como la eñe, o las vocales acentuadas utilizaremos una estructura `según`, ya que la diferencia entre mayúsculas o minúsculas no sigue ninguna norma.

Diseño del algoritmo

```

cadena función Mayúsculas(E cadena : c)
var
    entero : i

inicio

```

```

desde i ← 1 hasta longitud(c) hacer
    si c[i] >= 'a' y c[i] <= 'z' entonces
        c[i] ← acarácter(acódigo(c[i]) - 32)
    si_no
        según_sea c[i] hacer
            // incluir todas las excepciones a la norma
            'ñ' : c[i] ← 'Ñ'
            'á', 'à', 'â', 'ä' : c[i] ← 'Á'
            'é', 'è', 'ê', 'ë' : c[i] ← 'É'
            'í', 'ì', 'î', 'ï' : c[i] ← 'Í'
            'ó', 'ò', 'ô', 'ö' : c[i] ← 'Ó'
            'ú', 'ù', 'û' : c[i] ← 'Ú'
            'ü' : c[i] ← 'Ü'
        fin_según
    fin_si
fin_desde
devolver(c)
fin_función

cadena función Minúsculas(E cadena : c)
var
    entero : i
inicio
    desde i ← 1 hasta longitud(c) hacer
        si c[i] >= 'A' y c[i] <= 'Z' entonces
            c[i] ← acarácter(acódigo(c[i]) + 32)
        si_no
            según_sea c[i] hacer
                // incluir todas las excepciones a la norma
                'Ñ' : c[i] ← 'ñ'
                'Ü' : c[i] ← 'ü'
            fin_según
        fin_si
    fin_desde
    devolver(c)
fin_función

```

- 7.4.** Diseñar una función que informe si una cadena es un palíndromo (una cadena es un palíndromo si se lee igual de izquierda a derecha que de derecha a izquierda).

Análisis del problema

Una forma de ver si una cadena es un palíndromo sería dando la vuelta a esté y comparando la cadena original con la cadena al revés: si son iguales se trata de un palíndromo. Para ello podemos hacer otra función que dé la vuelta a la cadena (`CadenaRevés()`).

Sin embargo este método no es el más eficiente. Por ejemplo si el primer y el último carácter de la cadena ya son distintos, no es necesario seguir comparando. Por lo tanto también se podrá saber si una cadena es un palíndromo si mediante un bucle se compara el primer carácter con el último, el segundo con el penúltimo, y así sucesivamente. Para ello se utilizan dos índices y mientras uno se va incrementando, el otro se decrementará. El proceso debe acabar cuando el índice que va aumentando es mayor o igual al que va decrementando. Si al salir del bucle los dos caracteres apuntados por los índices son iguales, se tratará de un palíndromo.

Diseño del algoritmo

Vamos a desarrollar las dos versiones.

Versión 1

```
cadena función CadenaRevés(E cadena : c)
var
    entero : i
    cadena : revés
inicio
    revés ← ''
    desde i ← longitud(c) hasta 1 incremento -1 hacer
        revés ← revés & c[i]
    fin_desde
    devolver(revés)
fin_función

lógico función EsPalíndromo(E cadena : c)
inicio
    devolver(c = CadenaRevés(c))
fin_función
```

Versión 2

```
lógico función EsPalíndromo(E cadena : c)
var
    entero : i, j
inicio
    i ← 1
    j ← longitud(c)
    mientras (i < j) y (c[i] = c[j]) hacer
        i ← i + 1
        j ← j - 1
    fin_mientras
    devolver(c[i] = c[j])
fin_función
```

8

ARCHIVOS (FICHEROS). ARCHIVOS SECUENCIALES

ccg'Xlhcg'UdfcWgUf'dcf'i b'dfc[fUaUdiYXYbfYg]Xf'gla i'hzbYUaYbhY'Yb'UaYa cf]Udf]bWdU'XY'UWa di hUXcfUzgjb'YaVUF[czWUbXc'gYhfUVUVb'[fUbXYg'Wbh]XUXYg'XY'Xlhcg'fghcg'gY'UaU!WbUb'bcfaUaYbhY'Yb'Xlgdcglhjcg'XY'aYa cf]UU1]JUf'"9ghUg'W'YWWcbYg'XY'Xlhcg'gY'WbcWbWa c'UfWljcgfZ]WYfcgL"9b'Y'dfYgYbhY'Wdjh'i'c'gY'fYU]nUi bU]bfhcXi WWD'E'U'Ucf[Ub]nUWDE'm[Ygh]DE'XY'Xlhcg'Ua UWbUXcg'gcvfY'Xlgdcglhjcg'XY'Ua UWbUa]Ybhcg'Y'WbXUf]czhUYg'Wa c'WbhlgmX]gWg'aU[bfh]Wg'migY'dfYghU'i bU'YgdYWWU'UhYbWWD'E'U'Ucf[Ub]nUWDE'gYWYbWU"

8.1. CONCEPTOS GENERALES SOBRE ARCHIVOS

Los archivos (ficheros) podrán ser de programas o de datos. Un *archivo de datos* es una colección de datos estructurados, que se trata como una unidad y se encuentra almacenado sobre un dispositivo de almacenamiento externo. Un *archivo de programa* es un conjunto de sentencias que realizan una tarea específica y que está almacenada en un dispositivo del almacenamiento externo.

Como estructura de datos los archivos permiten el almacenamiento permanente y la manipulación de gran número de datos. Están formados por una colección de registros y éstos, a su vez, por campos, caracterizados por su tamaño o longitud y su tipo de datos. Una clave o indicativo es un campo que identifica a un registro diferenciándolo de los demás. En un archivo hay que distinguir entre los conceptos de registro físico y registro lógico.

- El *registro físico* es la cantidad de datos más pequeña que puede transferirse en una operación de entrada/salida a través del *buffer*. Su tamaño viene impuesto por el equipo material.
- El *registro lógico* se define por el programador y lo normal es que sea menor o igual que el registro físico.

Se denomina *factor de bloqueo* al número de registros lógicos que puede contener un registro físico. Un factor de bloqueo superior a uno es positivo, ya que puede consultar varios registros directamente en el *buffer*, sin tener que leer de nuevo en el disco.

8.1.1. Jerarquización

Las estructuras de datos se organizan de forma jerárquica. Las **bases de datos** ocupan el nivel más alto de la jerarquía y están formadas por un conjunto de archivos que contienen datos relacionados. Los **archivos** son un conjunto de registros relacionados entre sí. Un **registro** es un conjunto de campos. Un **campo** puede estar dividido en **subcampos**. Si no se encuentra dividido, constituye el nivel más bajo de la jerarquía desde el punto de vista lógico.

8.1.2. Clasificación de los archivos según su función

Los archivos se pueden clasificar según la función que realizan en:

Maestros	Contienen datos permanentes o históricos.
De movimientos	Son archivos auxiliares que contienen los registros necesarios para poder realizar las modificaciones de los ficheros permanentes en un periodo de tiempo predeterminado. Una vez realizado el proceso de actualización del archivo maestro, el de movimientos pierde su utilidad y se hace desaparecer para comenzar la creación de uno nuevo.
De maniobra	Tienen una vida limitada, normalmente menor que la duración de la ejecución de un programa. Se utilizan como auxiliares de los anteriores y sus registros contienen resultados semielaborados, como consecuencia de un determinado proceso, que sirven como datos de entrada para otro tratamiento.
De informes	Contiene datos que están organizados para presentaciones a los usuarios.

8.1.3. Operaciones básicas

El primer paso para poder gestionar un archivo mediante un programa es declarar un identificador lógico que se asocie al nombre externo del archivo para permitir su manipulación. Las operaciones básicas que se realizan al trabajar con archivos son:

Creación	Es la operación mediante la cual se introduce la información correspondiente al archivo en un soporte de almacenamiento de datos y donde el archivo se define mediante un nombre y unos atributos, nombre de archivo y de dispositivo, tamaño, tamaño de bloque y organización.
Apertura	Crea un canal que permite la comunicación de la CPU con el dispositivo de soporte físico del archivo; de esta manera los registros se vuelven accesibles.
Clausura	La operación de abrir archivos se puede aplicar para operaciones de entrada, salida o entrada/salida.
Lectura de datos	Cierra la conexión entre el identificador y el dispositivo de almacenamiento externo.
Escritura de datos	Copia los registros del archivo sobre variables en memoria central.
Eliminación del archivo	Copia la información contenida en variables sobre un registro del fichero.
	Elimina o borra el archivo del soporte físico, liberando el espacio.

Se deben utilizar instrucciones que permitan ejecutar estas operaciones básicas.

8.1.4. Otras operaciones usuales

Consulta	Operación que permite al usuario acceder al archivo de datos para conocer el contenido de uno, varios o todos los registros.
Altas	Permite la adición de un nuevo registro al archivo.
Modificación	Altera la información contenida en un registro.
Bajas	Eliminación o borrado lógico de un registro del archivo.
Clasificación	Ordena los registros del archivo con respecto al contenido de un determinado campo. Puede ser ascendente o descendente.
Reorganización	Optimiza la estructura de un archivo que ha degenerado.
Fusión (mezcla)	Reúne varios archivos en uno solo intercalándose unos en otros y siguiendo unos criterios determinados.
Rotura	Permite obtener varios archivos a partir de un mismo archivo inicial.

8.1.5. Soportes

El soporte es el medio físico donde se almacenan los datos. Los tipos de soporte utilizados en la gestión de archivos son:

Secuenciales	Los registros han de estar escritos uno a continuación de otro y para acceder al registro N se necesita recorrer los $N - 1$ anteriores. Por ejemplo, las cintas.
Direccionables	Se puede acceder directamente a la información. Son soportes direccionables los discos magnéticos.

Un soporte secuencial obliga a establecer una organización secuencial. En un soporte direccionable se pueden establecer diversos tipos de organización, secuencial, secuencial indexada y aleatoria o directa.

8.2. FLUJOS

Los lenguajes modernos (de *C* a *C#* pasando por *C++* y *Java*) realizan las operaciones en archivos a través de clases que manipulan los flujos, es decir, la conexión con el medio de almacenamiento. De esta forma, para crear y abrir un archivo, se requiere utilizar una clase que defina la funcionalidad del flujo. Los flujos determinan el sentido de la comunicación (lectura, escritura, o lectura/escritura), la posibilidad de posicionamiento directo o no en un determinado registro y la forma de leer y/o escribir en el archivo. Cerrar el archivo implica cerrar el flujo. La personalización de flujos se consigue por asociación o encadenamiento de otros flujos con los flujos base de apertura de archivos.

8.3. ORGANIZACIÓN SECUENCIAL

Con este tipo de organización los registros se almacenan consecutivamente sobre el soporte externo y se pueden designar por números enteros consecutivos, pero estos números de orden no pueden ser utilizados como funciones de acceso. Para realizar el acceso a un registro resulta obligatorio pasar por los que le preceden.

Esta organización se puede utilizar tanto en soportes secuenciales como en direccionables. Los archivos organizados secuencialmente tienen un registro particular, el último, que contiene una marca de fin de archivo y no pueden abrirse simultáneamente para lectura y escritura. La longitud de sus registros puede ser variable.

Declaración

```
tipo
    archivo_s de <Tipo_de_dato> [...] : <nombre_de_tipo>
var
    <nombre_de_tipo> : <id_archivo>
```

Instrucciones

crear(<id_archivo>,<nombre_físico>)

nombre_físico será una expresión de cadena para indicar el nombre de dispositivo y nombre de archivo. La instrucción **crear** colocará en el fichero la marca de fin archivo.

abrir(<id_archivo>,<modo>,<nombre_físico>).

Los archivos secuenciales se podrán abrir únicamente de uno de estos dos modos:

- para operaciones de escritura, lo representaremos colocando como *<modo>* la letra «e». Este modo de apertura colocaría un imaginario puntero de datos al final del archivo, antes de la marca de fin de archivo, permitiendo la adición, escritura, de nuevos registros.
- para operaciones de lectura, sustituiremos *<modo>* por la letra «l». El imaginario puntero de datos del que antes hablábamos, se colocará al principio del archivo permitiendo la lectura de la información almacenada en él.

escribir(<id_archivo>,<lista_de_variables>)

<lista_de_variables>, serán variables del tipo base del archivo.

leer(<id_archivo>,<lista_de_variables>)

<lista_de_variables>, serán variables del tipo base del archivo.

cerrar(<id_archivo>)

borra el identificador.

borrar(<nombre_físico>)

para borrar el archivo, debe encontrarse cerrado.

renombrar(<nombre_físico1>,<nombre_físico2>)

renombra el primero con el nombre que hayamos escrito en segundo lugar. También se requiere que el archivo esté cerrado.

Funciones

fda(<id_archivo>)

detecta la marca de fin de archivo, devolviendo un resultado lógico o booleano; **verdad** cuando se ha alcanzado el fin de archivo y **falso** en caso contrario.

8.3.1. Archivos de texto

Los archivos pueden ser binarios y de texto. Los *archivos binarios* almacenan cualquier tipo de información tal y como se encuentra en memoria. Los *archivos de texto* son archivos en los que cada

registro es del tipo cadena de caracteres y los registros se separan unos de otros por el carácter *fin de línea*, detectable mediante la función `fdl(<id_archivo>)`. Para el trabajo con este tipo de archivos se dispone también del procedimiento `leercar(<id_archivo>, <var_de_tipo_carácter>)` que lee carácter a carácter. Los archivos de texto son un caso particular de archivos con organización secuencial.

8.3.2. Mantenimiento de archivos secuenciales

El mantenimiento de un archivo incluye todas las operaciones que pueden sufrir los registros en un archivo durante su vida. Estas operaciones requieren que el archivo esté creado y la apertura del archivo en el modo adecuado. Las operaciones de mantenimiento básicas serán la **actualización** —altas, bajas y modificaciones— y la **consulta** —parcial, de la información de un determinado registro o grupo de registros, o total—, de todos los registros del archivo. Las operaciones que se permiten en un archivo secuencial son:

Creación	Esta operación sólo se realizará la primera vez que se trabaje con un archivo. Bastará con ejecutar la instrucción <code>crear</code> para crear el archivo.
Altas	Es la operación de añadir nuevos registros al archivo. Los registros se van almacenando consecutivamente, en el mismo orden en el que se introducen. Obligatoriamente al final del fichero.
Consulta, total o parcial	Obligatoriamente en modo secuencial.

El resto de las operaciones necesitan una programación especial. Una **baja** es la acción de eliminar un registro de un archivo. La baja de un registro puede ser lógica o física. La **baja lógica** supone el borrado del registro en el archivo. Se efectúa colocando en un determinado campo del registro una bandera, indicador o *flag* que lo marque como que ha sido borrado o rellenando de espacios en blanco algún campo del registro. La **baja física** implica el borrado y desaparición del registro y se necesitará crear un nuevo archivo que no incluya al registro dado de baja. Como no es posible abrir un archivo secuencial para lectura y escritura y, además, cuando se abre para escritura, el puntero de datos se coloca al final del archivo, la baja lógica también necesitará efectuar la creación de un nuevo archivo auxiliar. Los pasos, en ambos casos, podrían ser los siguientes:

- Hasta que se termine el archivo inicial se van leyendo los registros que contiene y, en función de su lectura, se decide si van a ser dados de baja o no. Cuando se trate del registro al que se desea dar la baja se marca con una señal en alguno de sus campos y se escribe (*baja lógica*), o bien no se marca pero se omite su escritura en el archivo auxiliar (*baja física*). Si no es un registro a dar de baja se escribe en el archivo auxiliar.
- El proceso de bajas concluye borrando el archivo inicial y cambiando el nombre del archivo auxiliar por el del inicial.

El proceso de **modificar** la información almacenada en un determinado registro de un archivo secuencial es similar a la baja lógica.

Si se desea incorporar nuevos registros en una determinada posición, que no sea al final del archivo, se necesitará también la creación de un archivo auxiliar y el renombrado final del archivo auxiliar como el inicial.

8.4. EJERCICIOS RESUELTOS

- 8.1.** Escribir un algoritmo que permita la creación e introducción de los primeros datos en un archivo secuencial, PERSONAL, que deseamos almacene la información mediante registros del siguiente tipo.

```

tipo
    registro: datos_personales
        <tipo_dato1> : nombre_campo1
        <tipo_dato2> : nombre_campo2
        ..... : .....
    fin_registro

```

Análisis del problema

Tras la creación y apertura en modo conveniente del archivo, el algoritmo solicitará la introducción de datos por teclado y los almacenará de forma consecutiva en el archivo. Se utilizará una función, `último_dato(persona)`, para determinar el fin en la introducción de datos.

Diseño del algoritmo

```

algoritmo ejercicio_8_1
tipo
    registro: datos_personales
        <tipo_dato1> : nombre_campo1
        <tipo_dato2> : nombre_campo2
        ..... : .....
    fin_registro
    archivo_s de datos_personales: arch
var
    arch : f
    datos_personales : persona
inicio
    crear(f,'Personal')
    abrir(f,'e','Personal')
    llamar_a leer_reg(persona)
    // Procedimiento para la lectura de un
    // registro campo a campo
    mientras no último_dato(persona) hacer
        llamar_a escribir_f_reg(f, persona)
        // Procedimiento auxiliar, no desarrollado, para la
        // escritura en el archivo del registro campo a campo
        llamar_a leer_reg(persona)
    fin_mientras
    cerrar(f)
fin

```

- 8.2.** Supuesto que deseamos añadir nueva información al archivo PERSONAL, anteriormente creado, diseñar el algoritmo correspondiente.

Análisis del problema

Al abrir el archivo para escritura se coloca el puntero de datos al final del archivo, permitiendo, con un algoritmo similar al anterior, la adición de nueva información al final del mismo.

Diseño del algoritmo

```
algoritmo ejercicio_8_2
tipo
    registro: datos_personales
        <tipo_dato1>: nombre_campo1
        <tipo_dato2>: nombre_campo2
        .... : .....
fin_registro
archivo_s de datos_personales: arch
var
    arch          : f
    datos_personales : persona
inicio
    abrir(f,'e','PERSONAL')
    llamar_a leer_reg(persona)
    mientras no ultimo_dato(persona) hacer
        llamar_a escribir_f_reg(f, persona)
        llamar_a leer_reg(persona)
    fin_mientras
    cerrar(f)
fin
```

- 8.3. Diseñar un algoritmo que muestre por pantalla el contenido de todos los registros del archivo PERSONAL.

Análisis del problema

Es necesario abrir el archivo para lectura y, repetitivamente, leer los registros y mostrarlos por pantalla hasta detectar el fin de fichero. Se considerará que la función **fda(id_arch)** detecta el final de archivo con la lectura de su último registro.

Diseño del algoritmo

```
algoritmo ejercicio_8_3
tipo
    registro: datos_personales
        <tipo_dato1>: nombre_campo1
        <tipo_dato2>: nombre_campo2
        .... : .....
fin_registro
archivo_s de datos_personales: arch
var
    arch          : f
    datos_personales : persona
inicio
    abrir(f,'l','PERSONAL')
    mientras no fda(f) hacer
        llamar_a leer_f_reg(f, persona)
        llamar_a escribir_reg(persona)
    fin_mientras
```

```

    cerrar(f)
fin

```

Si se considerara la existencia de un registro especial que marca el fin de archivo, la función `fda(id_arch)` se activaría al leer este registro y obligando a modificar el algoritmo.

```

inicio
    abrir(f,'l','PERSONAL')
    llamar_a leer_f_reg(f, persona)
    mientras no fda(f) hacer
        llamar_a escribir_reg(persona)
        llamar_a leer_f_reg(f, persona)
    fin_mientras
    cerrar(f)
fin

```

- 8.4.** Implementar la consulta de un registro, por el campo `nombre_campo1`, en el archivo `PERSONAL`. Se debe tener en cuenta que no existen registros que almacenen la misma información en el campo por el cual se realiza la búsqueda.

Análisis del problema

Es necesario abrir el archivo para lectura y recorrerlo hasta encontrar el registro deseado o el fin de archivo. La función `fda(id_arch)` detectará el final de archivo con la lectura de su último registro y se utilizará la función definida por el usuario `igual(clavebus, persona)` en la comparación del campo `persona.nombre_campo1` con la clave introducida desde teclado.

Diseño del algoritmo

```

algoritmo ejercicio_8_4
tipo
    registro: datos_personales
        <tipo_dato1>: nombre_campo1
        <tipo_dato2>: nombre_campo2
        ..... : .....
    fin_registro
    archivo_s de datos_personales: arch
var
    arch           : f
    datos_personales : persona
    <tipo_dato1>      : clavebus
    lógico          : encontrado
inicio
    abrir(f,'l','PERSONAL')
    encontrado ← falso
    leer(clavebus)
    mientras no encontrado y no fda(f) hacer
        llamar_a leer_f_reg(f, persona)
        si igual(clavebus, persona) entonces
            encontrado ← verdad
        fin_si

```

```
fin_mientras
Si no encontrado entonces
    escribir ('No existe')
si_no
    llamar_a escribir_reg(persona)
fin_si
cerrar(f)
fin
```

- 8.5. Algoritmo que nos permita localizar, por el campo nombre_campo1, un determinado registro del fichero PERSONAL y eliminarlo del mismo.

Análisis del problema

Para realizar la baja de un registro es necesario un archivo auxiliar, también secuencial. Se lee del archivo inicial registro a registro, escribiendo todos ellos en el auxiliar, excepto el que se desea dar de baja, el cual no se copiará.

Diseño del algoritmo

```
algoritmo ejercicio_8_5
tipo
    registro: datos_personales
        <tipo_dato1>: nombre_campo1
        <tipo_dato2>: nombre_campo2
        ..... : .....
fin_registro
archivo_s de datos_personales: arch
var
    arch : f, faux
    datos_personales : persona, personaux
    lógico : encontrado
inicio
    abrir(f, 'l', 'PERSONAL')
    crear(faux, 'nuevo')
    abrir(faux, 'e', 'nuevo')
    leer(personaux.nombre_campo1)
    encontrado ← falso
    mientras no fda(f) hacer
        llamar_a leer_f_reg(f, persona)
        si personaux.nombre_campo1 = persona.nombre_campo1 entonces
            encontrado ← verdad
        si_no
            llamar_a escribir_f_reg(faux, persona)
        fin_si
    fin_mientras
    si no encontrado entonces
        escribir ('No está')
    fin_si
    cerrar(f, faux)
borrar('PERSONAL')
```

```

renombrar('nuevo', 'PERSONAL')
fin

```

- 8.6.** Algoritmo que nos permita localizar, por el campo nombre_campo1, un determinado registro del fichero PERSONAL y modificar el contenido del mismo.

Análisis del problema

Puesto que no es posible abrir un archivo secuencial simultáneamente para lectura y escritura, el procedimiento para modificar la información almacenada en un determinado registro del archivo resultará similar al desarrollado para las bajas. Se necesita también un archivo auxiliar y cuando se encuentre el registro a modificar, en lugar de no incluirlo, se escribirá en dicho archivo auxiliar después de modificarlo.

Diseño del algoritmo

```

algoritmo ejercicio_8_6
tipo
    registro: datos_personales
        <tipo_dato1>: nombre_campo1
        <tipo_dato2>: nombre_campo2
        .... : .....
    fin_registro
    archivo_s de datos_personales: arch
var
    arch           : f, faux
    datos_personales : persona, personaaux
    lógico         : encontrado
inicio
    abrir(f, 'l', 'PERSONAL')
    crear(faux, 'nuevo')
    abrir(faux, 'e', 'nuevo')
    leer(personaaux.nombre_campo1)
    encontrado ← falso
    mientras no fda(f) hacer
        llamar_a leer_f_reg(f, persona)
        si personaaux.nombre_campo1 = persona.nombre_campo1 entonces
            encontrado ← verdad
            llamar_a modificar(persona)
        fin_si
        llamar_a escribir_f_reg(faux, persona)
    fin_mientras
    si no encontrado entonces
        escribir ('No está')
    fin_si
    cerrar(f, faux)
    borrar('PERSONAL')
    renombrar('nuevo', 'PERSONAL')
fin

procedimiento modificar(E/S datos_personales: persona)
var

```

```

carácter: opción
entero : n
inicio
  escribir('R.- registro completo')
  escribir('C.- campos individuales')
  escribir('Elija opción: ')
  leer(opción)
  según_sea opción hacer
    'R':
      escribir_reg(persona)
      leer_reg(persona)
    'C':
      escribir_reg(persona)
      //Muestra por pantalla el registro campo a campo y numerados
      leer(n)
      //Número del campo a modificar
      introducir_campo(n, persona)
  fin_según
fin_procedimiento

```

- 8.7.** Una librería almacena en un archivo secuencial la siguiente información sobre cada uno de sus libros: CÓDIGO, TITULO, AUTOR y PRECIO. El fichero está ordenado ascendente por los códigos de los libros (de tipo cadena) que no pueden repetirse. Se precisa un algoritmo con las opciones: Insertar, que permitirá insertar nuevos registros en el fichero, que debe mantenerse ordenado en todo momento, y Consulta, que buscará registros por el campo CÓDIGO.

Análisis del problema

El algoritmo comienza presentando un menú de opciones a través del cual se haga posible la selección de un procedimiento u otro.

- | | |
|-----------------|--|
| Insertar | Para poder colocar el nuevo registro en el lugar adecuado, sin que se pierda la ordenación inicial, es necesario utilizar un archivo auxiliar. En dicho archivo auxiliar se copian los registros hasta llegar al punto donde debe colocarse el nuevo; en ese momento se escribirá y se continuará la copia de los restantes registros. |
| Consulta | Como el archivo está ordenado y los códigos no repetidos el proceso de consulta se puede acelerar. Se recorre el archivo de forma secuencial hasta encontrar el código buscado, o hasta que éste sea menor que el código del último registro leído o hasta el fin del fichero, en el caso que no se encuentre. |

Cuando el código buscado sea menor que el código del registro leído desde el archivo se podrá deducir que de ahí en adelante ese registro ya no podrá estar en el fichero; por tanto es posible abandonar la búsqueda.

Diseño del algoritmo

```

algoritmo ejercicio_8_7
  tipo
    registro : reg
    cadena : cód
    cadena : título
    cadena : autor
    entero : precio

```

```

fin_registro
archivo_s de reg : arch
var
    entero : op
inicio
    repetir
        escribir('MENÚ')
        escribir('1.- INSERTAR')
        escribir('2.- CONSULTA')
        escribir('3.- FIN')
        escribir('Elija opción')
        leer(op)
        según_sea op hacer
            1: llamar_a insertar
            2: llamar_a consulta
        fin_según
    hasta_que op = 3
fin

procedimiento insertar
var
    arch      : f,f2
    reg       : rf,r
    lógico   : escrito
    carácter : resp
inicio
    repetir
        abrir(f,'l','Libros.dat')
        crear(f2,'Nlibros.dat')
        abrir(f2,'e', 'Nlibros.dat')
        escribir('Deme el código')
        leer(r.cód)
        escrito ← falso
        mientras no FDA(f)
            llamar_a leer_arch_reg(f, rf)
            si rf.cód > r.cód y no escrito entonces
                // Si leemos del fichero un registro con código
                // mayor que el nuevo y éste aún no lo
                // hemos escrito, es el momento de insertarlo
                escribir('Deme otros campos')
            llamar_a completar(r)
            llamar_a escribir_arch_reg(f2, r)
            escrito ← verdad
            // Deberemos marcar que lo hemos escrito
            // para que no siga insertándose, desde aquí
            // en adelante, todo el rato
        si_no
            si rf.cód = r.cód entonces
                escrito ← verdad
        fin_si

```

```
fin_si
llamar_a escribir_arch_reg(f2, rf)
// De todas formas escribimos el que
// leemos del fichero
fin_mientras
si no escrito entonces
// Si el código del nuevo es mayor que todos los del
// archivo inicial, llegaremos al final sin haberlo escrito
escribir('Deme otros campos')
llamar_a completar(r)
llamar_a escribir_arch_reg(f2, r)
fin_si
cerrar(f,f2)
borrar('Libros.dat')
renombrar('Nlibros.dat', 'Libros.dat')
escribir('¿Seguir? (s/n)')
leer(resp)
hasta_que resp='n'
fin_procedimiento

procedimiento consulta
var
reg      : rf,r
arch     : f
carácter : resp
lógico   : encontrado,pasado
inicio
resp ← 's'
mientras resp <> 'n' hacer
abrir(f, 'l', 'Libros.dat')
escribir('Deme el código a buscar')
leer(r.cód)
encontrado ← falso
pasado ← falso
mientras no FDA(f) y no encontrado y no pasado hacer
    llamar_a leer_arch_reg(f, rf)
    si r.cód = rf.cód entonces
        encontrado ← verdad
        llamar_a escribir_reg(rf)
    si_no
        si r.cód < rf.cód entonces
            pasado ← verdad
        fin_si
    fin_si
fin_mientras
si no encontrado entonces
    escribir('Ese libro no está')
fin_si
cerrar(f)
escribir('¿Seguir? (s/n)')
```

```

leer (resp)
fin_mientras
fin_procedimiento

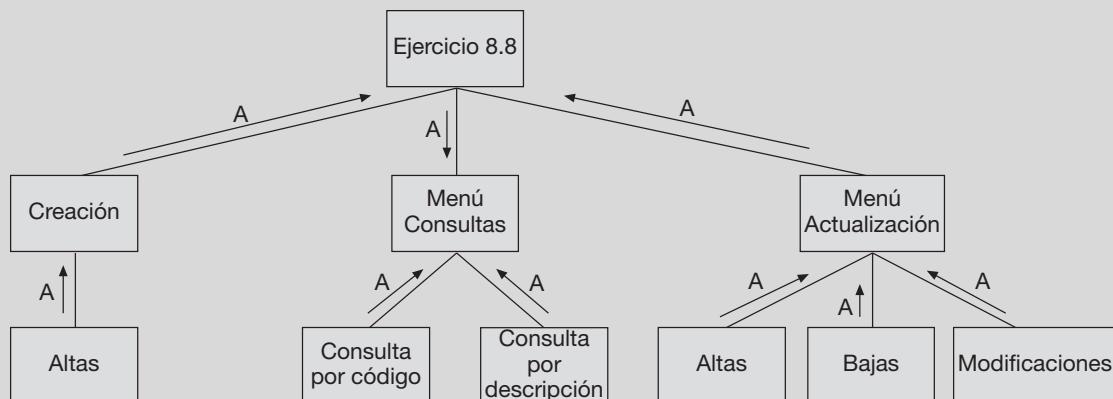
```

- 8.8.** Se desea realizar la gestión de un archivo secuencial de almacén. Los campos del archivo son: código del producto, descripción, precio, proveedor, stock actual, stock máximo y stock mínimo. La gestión del archivo debe incluir la creación, la consulta por campo clave o por descripción y la actualización del mismo (altas, bajas, modificaciones).

Análisis del problema

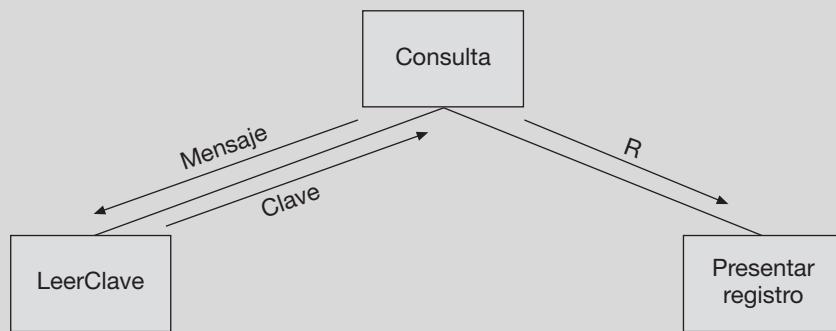
Se trata de un programa típico de mantenimiento de un archivo de forma interactiva, es decir mediante una serie de menús y con intervención del usuario para introducir los datos, aunque esta no es la forma lógica de procesar un archivo secuencial ya que el modo de proceso normal de éstos es mediante procesos por lotes.

El programa se va a dividir en una serie de procedimientos llamados desde el programa principal a base de menús. Las opciones del menú serán Creación, Consultas, Actualización y Fin del Programa. Las consultas llamarán a otro menú con las opciones de Consulta por Código y Consulta por Descripción. La opción de Actualización a su vez llamará a otro menú con las opciones de Altas, Bajas, Modificaciones y Fin. En un primer nivel, la estructura general del programa quedaría de la siguiente forma:

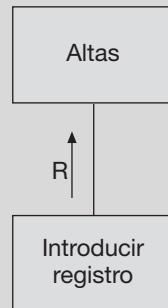


Para la opción de Creación, el archivo se pasará como parámetro de salida. Primero se comprueba si el archivo ya existe. Para ello los distintos lenguajes de programación pueden actuar de forma distinta. Se podría ver si existe el archivo abriéndolo para lectura, con lo que daría un error si no existe. También es posible comprobar la longitud del archivo. Si se abre el archivo para añadir y la longitud es 0, quiere decir que el archivo no existe, con lo que es posible crearlo sin ningún problema. La mayoría de los lenguajes permiten obtener la longitud del archivo mediante una función que devuelve la longitud del archivo (LOF en inglés). El lenguaje algorítmico utilizado dispone de una función **LDA** que permite saber la longitud del archivo. Si el archivo no existe, se abre para escritura, se cierra y se da la opción de introducir datos en ese momento mediante una llamada al procedimiento de **Altas**. Si el archivo ya existe, aparecerá un mensaje de advertencia. Si todavía se desea crear, se procederá de la misma forma que antes; si no, se volverá directamente al programa principal.

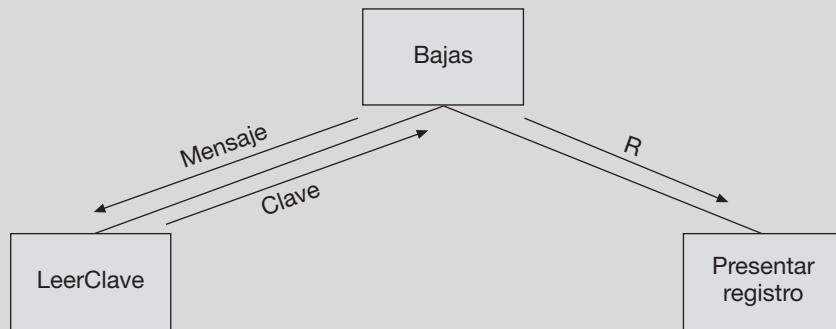
Para las consultas se utiliza un menú con 2 opciones, aunque ambas funcionarán de forma similar. Si el archivo existe, se abre para lectura y se introduce el criterio de búsqueda —el código del producto o la descripción del mismo— mediante el procedimiento **LeerClave** que se limitará a leer un dato de tipo cadena. En ambos procedimientos se utiliza un bucle hasta llegar al fin del archivo o encontrar el registro deseado. Al final, si no se ha encontrado el registro, saldrá un mensaje advirtiéndolo; en caso contrario, se visualizan los datos mediante el procedimiento **PresentarRegistro**. La descomposición modular de cualquiera de los dos modos de consulta podría quedar de la siguiente forma:



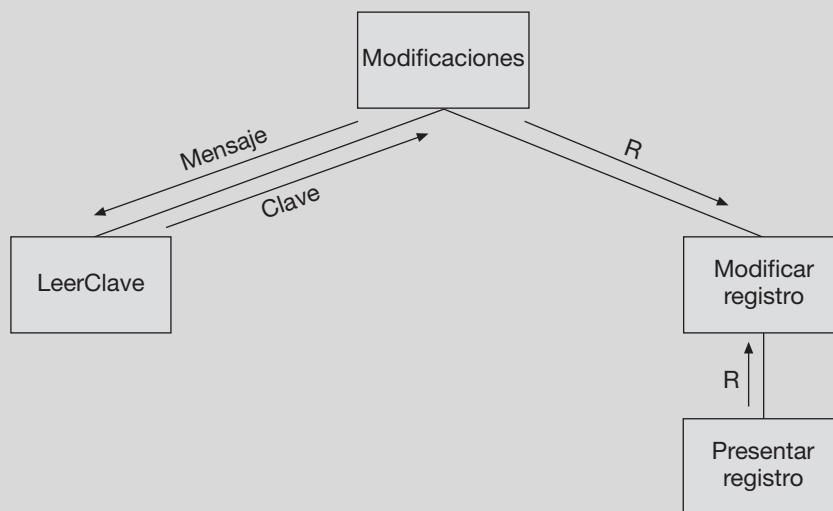
En las actualizaciones también se utiliza un menú que llamará a los procedimientos Altas, Bajas y Modificaciones. Para las altas, se abre el archivo para añadir y mediante un bucle controlado por centinela se introducen los datos de los registros mediante el procedimiento IntroducirRegistro. Por cada iteración, se pregunta si se desea grabar más registro. Contestando afirmativamente el bucle se repetirá; en caso contrario se cierra el archivo y finaliza la opción.



Para dar bajas en un archivo secuencial, es necesario utilizar un archivo auxiliar. Si el archivo original existe, se debe abrir éste para lectura y el auxiliar para escritura. Se introduce la clave con el procedimiento LeerClave y se ejecuta un bucle hasta llegar al final del archivo. Dentro del bucle, si el registro no es el buscado, se graba en el auxiliar; si se encuentra se dará de baja. Para ello simplemente no se escribe en el archivo original. Para mejorar un poco el algoritmo, cuando se encuentre el registro, éste saldrá por pantalla y pedirá la confirmación para borrar. Si la confirmamos no hace nada; si no, se escribe igualmente en el archivo auxiliar. Al final del bucle, si se ha encontrado el registro a dar de baja, se borra el archivo original —cuyos datos ya no están actualizados— y se cambia el nombre del archivo auxiliar —con los datos actualizados—, dándole el nombre del archivo original. Si el registro a borrar no se encuentra en el archivo, un mensaje advertirá del hecho.



Para las modificaciones también se utiliza un archivo auxiliar. Después de abrir los archivos e introducir la clave buscada, se recorre el archivo. Si el último registro leído no es el buscado se graba tal cual en el archivo auxiliar. Si el registro se encuentra, se llama a una rutina que modifique el registro (`ModificarRegistro`) y se graba el registro modificado en el archivo auxiliar. Para modificar el registro simplemente se pueden introducir los nuevos datos aprovechando el procedimiento `IntroducirRegistro`. Este procedimiento obliga a introducir todos los datos sean necesarios o no. Otra forma de modificarlo es mediante un menú en el que se elige el campo a modificar y sólo se lee ese campo. Este va a ser el procedimiento que vamos a utilizar.



Diseño del algoritmo

```

algoritmo Ejercicio_8_8
tipo
  registro : R-Almacén
    cadena : Código
    cadena : Descr
    entero : Precio
    cadena : Proveedor
    entero : Stock
    entero : Stock-Máx
    entero : Stock-Mín
  fin_registro
  archivo_s de R-Almacén : A-Almacén
var
  carácter : opción
inicio
  repetir
    escribir('1.-Creación/2.-Consultas/3.-Actualización/4.-Fin')
    leer(opción)
    según_sea opción
      '1' : Creación('ALMACEN.DAT')
      '2' : MenúConsultas('ALMACEN.DAT')
      '3' : MenúActualización('ALMACEN.DAT')
  fin_repetir
  
```

```
    fin_según
  hasta_que opción = '4'
fin

procedimiento Creación( E cadena : NombreArchivo)
var
  carácter : respuesta
  A-Almacén: A
inicio
  respuesta ← 'S'
  si LDA(NombreArchivo) <> 0 entonces
    escribir('El archivo existe, ¿Desea crearlo?')
    leer(respuesta)
  fin_si
  si respuesta = 'S' entonces
    crear(A,NombreArchivo)
    abrir(A,'e',NombreArchivo)
    cerrar(A)
    escribir('¿Desea introducir datos ahora?')
    leer(respuesta)
    si respuesta = 'S' entonces
      Altas(NombreArchivo)
    fin_si
  fin_si
fin_procedimiento

procedimiento MenúConsultas(E cadena : NombreArchivo)
var
  carácter : opción
inicio
  repetir
    escribir('1.-Consulta por Código/2.-Consulta por Descripción/3.-Fin')
    leer(opción)
    según_sea opción hacer
      '1' : ConsultaCódigo(NombreArchivo)
      '2' : ConsultaDescripción(NombreArchivo)
    fin_según
  hasta_que opción = '3'
fin_procedimiento

procedimiento ConsultaCódigo(E cadena : NombreArchivo)
var
  R-Almacén : R
  A-Almance : A
  cadena : Clave
inicio
  si LDA(NombreArchivo) = 0 entonces
    escribir('El archivo no existe')
  si_no
```

```

abrir(A,'l',NombreArchivo)
LeerClave(clave, 'Código:')
leer(A,R)
mientras no FDA(A) y R.Código <> Clave hacer
    leer(A,R)
fin_mientras
si R.Código = Clave entonces
    PresentarRegistro(R)
si_no
    escribir('El registro no está')
fin_si
cerrar(A)
fin_si
fin_procedimiento

procedimiento ConsultaDescripción(E cadena : NombreArchivo)
var
    A-Almacén : A
    R-Almacén : R
    cadena : Clave
inicio
    si LDA(NombreArchivo) = 0 entonces
        escribir('El archivo no existe')
    si_no
        abrir(A,'l',NombreArchivo)
        LeerClave(clave, 'Descripción:')
        leer(A,R)
        mientras no FDA(A) y R.Descripción <> Clave hacer
            leer(A,R)
        fin_mientras
        si R.Descripción = Clave entonces
            PresentarRegistro(R)
        si_no
            escribir('El registro no está')
        fin_si
        cerrar(A)
    fin_si
fin_procedimiento

procedimiento LeerClave(S cadena : Clave; E cadena : mensaje)
inicio
    escribir(mensaje)
    leer(clave)
fin_procedimiento

procedimiento PresentarRegistro(S R-Almacén : R)
inicio
    escribir(R.Código)
    escribir(R.Descripción)
    escribir(R.Precio)

```

```
escribir(R.Proveedor)
escribir(R.Stock)
escribir(R.Stock-Máx)
escribir(R.Stock-Mín)
fin_procedimiento

procedimiento MenúActualización(E cadena : NombreArchivo)
var
    carácter : opción
inicio
    repetir
        escribir('1.- Altas/2.-Bajas/3.-Modificaciones /4.-Fin')
        leer(opción)
        según_sea opción hacer
            '1' : Altas(NombreArchivo)
            '2' : Bajas(NombreArchivo)
            '3' : Modificaciones(NombreArchivo)
        fin_según
        hasta_que opción = '4'
fin_procedimiento

procedimiento Altas(E cadena : NombreArchivo)
var
    A-Almacén : A
    R-Almacén : R
    carácter : continuar
inicio
    abrir(A, 'e', NombreArchivo)
    repetir
        IntroducirRegistro(R)
        escribir(A,R)
        escribir('¿Desea continuar?')
        leer(continuar)
    hasta_que continuar = 'N'
    cerrar(A)
fin_procedimiento

procedimiento IntroducirRegistro(S R-Almacén : R)
inicio
    leer(R.Código)
    leer(R.Descri)
    leer(R.Precio)
    leer(R.Proveedor)
    leer(R.Stock)
    leer(R.Stock-Máx)
    leer(R.Stock-Mín)
fin_procedimiento

procedimiento Bajas(E cadena : NombreArchivo)
var
```

```

A-Almacén : A, A-Aux
R-Almacén : R
cadena : Clave
carácter : DarBaja
lógico : Encontrado
inicio
    si LDA(NombreArchivo) = 0 entonces
        escribir('El archivo no existe')
    si_no
        abrir(A, 'l', NombreArchivo)
        abrir(A-Aux, 'e', 'AUXILIAR')
        LeerClave(clave, 'Código: ')
        leer(A,R)
        encontrado ← falso
        mientras no FDA(A)hacer
            si R.Código = Clave entonces
                PresentarRegistro(R)
                escribir('¿Dar de baja al registro?')
                leer(DarBaja)
                si DarBaja <> 'S' entonces
                    escribir(A-Aux,R)
                fin_si
                encontrado ← verdad
            si_no
                escribir(A-Aux,R)
            fin_si
            leer(A,R)
        fin_mientras
        cerrar(A)
        cerrar(A-Aux)
        si encontrado entonces
            borrar(NombreArchivo)
            renombrar('AUXILIAR',NombreArchivo)
        si_no
            escribir('El registro no está')
        fin_si
    fin_si
fin_procedimiento

procedimiento Modificaciones(E cadena : NombreArchivo)
var
    A-Almacén : A-Aux, A
    R-Almacén : R
    cadena : Clave
    lógico : Encontrado
inicio
    si LDA(NombreArchivo) = 0 entonces
        escribir('El archivo no existe')
    si_no
        abrir(A, 'l', NombreArchivo)

```

```

abrir(A-Aux, 'e', 'AUXILIAR')
LeerClave(clave, 'Código: ')
leer(A,R)
encontrado ← falso
mientras no FDA(A) hacer
    si R.Código = Clave entonces
        ModificarRegistro(R)
        encontrado ← verdad
    fin_si
    escribir(A-Aux,R)
    leer(A,R)
fin_mientras
cerrar(A)
cerrar(A-Aux)
si encontrado entonces
    borrar(NombreArchivo)
    renombrar('AUXILIAR', NombreArchivo)
si_no
    escribir('El registro no está')
fin_si
fin_si
fin_procedimiento

procedimiento ModificarRegistro(s R-Almacén : R)
var
    carácter : opción
inicio
repetir
    PresentarRegistro(R)
    escribir('1.-Descripción/2.-Precio/3.-Proveedor/4.-Stock Actual')
    escribir('5.-Stock Máximo/6.-Stock Mínimo/7.-Fin')
    leer(opción)
    según_sea opción hacer
        '1' : leer(R.Descri)
        '2' : leer(R.Precio)
        '3' : leer(R.Proveedor)
        '4' : leer(R.Stock)
        '5' : leer(R.Stock-Máx)
        '6' : leer(R.Stock-Mín)
    fin_según
    hasta_que opción = '7'
fin_procedimiento

```

- 8.9.** Sobre el archivo del problema anterior se desea generar un archivo de pedidos. En dicho archivo aparecerán los productos cuyo stock actual sea menor que el stock mínimo. El archivo de pedidos tendrá los campos proveedor, código del producto y cantidad a pedir. La cantidad a pedir será la suficiente como para que el stock del producto sea igual al stock máximo.

Análisis del problema

Para realizar este algoritmo es preciso recorrer el archivo de almacén utilizado en el ejercicio anterior. Cada vez que se lee un registro hay que ver si su stock actual es menor que el stock mínimo. Si esto es cierto se

mueve el campo proveedor y el campo código del registro de almacén a los campos correspondientes del archivo de pedidos. Para calcular la cantidad de pedido utilizamos la fórmula:

$$\text{Cantidad} = \text{Stock Máximo} - \text{Stock Actual}$$

Una vez hechos los cálculos se graba el registro en el archivo de pedidos.

Diseño del algoritmo

```

algoritmo Ejercicio_8_9
tipo
    tipo
        registro : R-Almacén
            cadena : Código
            cadena : Descr
            entero : Precio
            cadena : Proveedor
            entero : Stock
            entero : Stock-Máx
            entero : Stock-Mín
    fin_registro
    archivo_s de R-Almacén : A-Almacén
    registro : R-Pedidos
        cadena : Proveedor
        cadena : Código
        entero : Cantidad
    fin_registro
    archivo_s de R-Pedidos : A-Pedidos
    fin_archivo
var
    A-Almacén : A
    A-Pedidos : P
    R-Almacén : RA
    R-Pedidos : RP
inicio
    abrir(A, 'l', 'ALMACEN.DAT')
    abrir(P, 'e', 'PEDIDOS.DAT')
    leer(A, RA)
    mientras no FDA(A) hacer
        si RA.Stock < RA.Stock-Mín entonces
            RP.Proveedor ← RA.Proveedor
            RP.Código ← RA.Código
            RP.Cantidad ← RA.Stock-Máx - RA.Stock
            escribir(P, RP)
        fin_si
        leer(A, RA)
    fin_mientras
    cerrar(A)
    cerrar(P)
fin

```

- 8.10.** Si el archivo de pedidos está ordenado por proveedor, se desea realizar un informe con los pedidos que hay de cada proveedor y el número de pedidos del mismo. La estructura del informe por cada proveedor será la siguiente:

Proveedor	Código Producto	Cantidad a Pedir	Nº de pedidos
.....	
.....	

Análisis del problema

Se trata de un caso típico de ruptura de control, en este caso por el campo proveedor. Hay que recorrer el archivo de pedidos; mientras no cambie el proveedor se escribe la línea y se incrementa el contador de pedidos. Cuando se cambia de proveedor se escribe el contador y se sigue con el siguiente proveedor.

Dependiendo de los datos escribir, puede que, tal y como se ve en el formato del informe, no sea necesario escribir todos los campos, en cuyo caso se mandan cadenas vacías.

Para hacer la ruptura de control es necesario utilizar dos bucles. Uno se ejecutará mientras no sea fin de archivo, el otro se realizará mientras no cambie el proveedor. Para saber si se ha cambiado de proveedor, es preciso utilizar una variable auxiliar que tome el valor del último proveedor. Comparándola con el proveedor del último registro leído es posible saber si el proveedor ha cambiado.

En los ejercicios anteriores se ha supuesto que la marca de fin de archivo se detecta si se intenta leer después del último registro, tal y como ocurre por ejemplo en COBOL. Ahora se supondrá que la marca de fin de archivo se detecta cuando se lee el último registro, tal y como ocurriría en PASCAL o BASIC. Para que no cambie la estructura de los bucles se simulará el modo de leer registros en COBOL, creando un procedimiento LeerRegistro que devuelve un centinela cuando se intenta leer después del último registro.

Diseño del algoritmo

```

algoritmo Ejercicio_8_10
tipo
    registro : R_Pedidos
    cadena : Proveedor
    cadena : Código
    entero : Cantidad
fin_registro
archivo_s de R_Pedidos : A_Pedidos
fin_archivo
var
    A_Pedidos : P
    R_Pedidos : RP
    entero : conta
    cadena : Proveedor_Aux
    lógico : FinDeArchivo
inicio
    abrir(P, 'l', 'PEDIDOS.DAT')
    LeerRegistro(P,RP,FinDeArchivo)
    mientras no FinDeArchivo hacer
        conta ← 0
        Proveedor_Aux ← RP.Proveedor
        escribir(RP.Proveedor)
        mientras Proveedor_Aux = RP.Proveedor y no FinDeArchivo hacer
            escribir(RP.Código,RP.Cantidad)

```

```

        conta ← conta + 1
        LeerRegistro(P,RP,FinDeArchivo)
    fin_mientras
    escribir(' ',',',conta)
fin_mientras
cerrar(P)
fin

procedimiento LeerRegistro(E/S A_Pedidos : P; S R_Pedidos : RP;
                           S lógico : Fin)
inicio
    si no FDA(P) entonces
        leer(P,RP)
        Fin ← falso
    si_no
        Fin ← verdad
    fin_si
fin

```

- 8.11.** El Servicio Meteorológico Nacional recibe de cada observatorio un archivo con información acerca de las temperaturas máximas y mínimas diarias registradas en ellos. Con ellos crea un archivo secuencial que tiene los campos: código del observatorio, fecha del registro (día, mes y año) temperatura máxima y temperatura mínima. El archivo no está ordenado. A partir de estos datos se desea realizar un informe en el que aparezcan las temperaturas medias mensuales.

Análisis del problema

Hay que averiguar la temperatura máxima y mínima de cada mes. Para ello se ha de comprobar que la temperatura máxima del último registro leído es mayor que la máxima provisional del mes y la mínima del registro menor que la mínima del mes. Como el archivo está desordenado, es necesario guardar las máximas y mínimas provisionales en un array de 12 elementos (uno por mes) que tenga dos campos: uno para la máxima y otro para la mínima. Antes de hacer esto, debemos dar a esos valores un valor inicial. Como no hay temperaturas menores que el frío absoluto (-273°C) ni en la naturaleza se dan temperaturas mayores de 100°C , se inicializan todas las máximas de la tabla a -273 y las mínimas a 100 .

Una vez leído todo el archivo, se debe recorrer el array y escribir el mes —que será el número del elemento—, la máxima, la mínima y la media. La media se calculará obteniendo el valor medio entre la máxima y la mínima del mes.

Diseño del algoritmo

```

algoritmo Ejercicio_8_11
tipo

    registro : Temperaturas
        real : Máxima, Mínima
    fin_registro
    array [1..12] de Temp : Tabla
    registro : R_Temp
    cadena : Observatorio
    registro : Fecha
        entero : dd, mm, aa
    fin_registro

```

```
    real : Máxima, Mínima
fin_registro
archivo_s de R-Temp : A_Temp
var
A_Temp : T
R_Temp : R
Tabla : Meses
entero : i
inicio
    InicializarTabla(Meses)
    abrir(T, 'l', 'TEMP.DAT')
    leer(T,R)
    mientras no FDA(T) hacer
        si R.Máxima > Meses[R.Fecha.mm].Máxima entonces
            Meses[R.Fecha.mm].Máxima ← R.Máxima
        fin_si
        si R.Mínima < Meses[R.Fecha.mm].Mínima entonces
            Meses[R.Fecha.mm].Mínima ← R.Mínima
        fin_si
        leer(T,R)
    fin_mientras
    cerrar(T)
    desde i ← 1 hasta 12 hacer
        escribir(i)
        escribir(Meses[i].Máxima)
        escribir(Meses[i].Mínima)
        escribir((Meses[i].Máxima + Meses[i].Mínima)/2)
    fin_desde
fin

// Inicializa los valores de la tabla. Máximo al mínimo posible
// y Mínimo al máximo posible
procedimiento InicializarTabla(s Tabla : T)
var
    entero : i
inicio
    desde i ← 1 hasta 12 hacer
        T[i].Máxima ← -273
        T[i].Mínima ← 100
    fin_desde
fin_procedimiento
```


9

ARCHIVOS DIRECTOS

9b·YghY·Wd

9.1. ORGANIZACIÓN DIRECTA

En un archivo organizado de modo directo el orden físico de los registros no tiene por qué corresponderse con aquel en el que han sido introducidos. Los registros son directamente accesibles mediante la especificación de un índice, que da la posición del registro respecto al origen del archivo aunque, generalmente, el acceso se realizará utilizando una clave.

La clave debe ser un campo del propio registro que lo identifique de modo único. Por ejemplo el NIF (número de identificación fiscal), en el caso de un archivo que almacenara datos personales sobre los clientes de una empresa. El programador creará una relación perfectamente definida entre la clave identificativa de cada registro y su posición (índice).

Cuando se aplica una función de conversión (*hash*) con la finalidad de transformar la clave en índice, puede ocurrir que dos registros con claves diferentes produzcan la misma dirección física en el soporte. Se dice, entonces, que se ha producido una *colisión* y habrá que situar este registro en una posición diferente a la indicada por el algoritmo de conversión. Al ocurrir esto, el acceso al registro se hace más lento. Es, pues, importante buscar una función de conversión que produzca pocas colisiones.

Cuando en los archivos con organización directa se realiza un acceso directo, la marca de fin de archivo no tiene sentido y se abrirán para lectura y escritura al mismo tiempo. Las instrucciones necesarias son:

```
tipo
  archivo_d de <tipo_de_dato>: <nombre_del_tipo>
var
  <nombre_del_tipo>: <id_archivo>
```

crear(<id_archivo>, <nombre_físico>), esta operación es destructiva; si el archivo ya existiera se perdería toda la información almacenada en él.

abrir(<id_archivo>, '1/e', <nombre_en_disco>), normalmente los archivos se abren siempre para lectura/escritura ('1/e'), aunque es posible también en determinadas circunstancias es posible abrirlos en modo de lectura ('1') o escritura ('e').

escribir(<id_archivo>, <var_del_tipo_base>, <p>), copia la información contenida en una variable del tipo base sobre el registro del archivo que ocupa la posición relativa especificada por p. La escritura secuencial a un archivo directo se realizará con el mismo formato que en los archivos secuenciales.

leer(<id_archivo>, <var_del_tipo_base>, <p>), copia el registro del fichero situado en la posición relativa p sobre una variable del tipo base. La lectura secuencial a un archivo directo se realizará con el mismo formato que en los archivos secuenciales.

cerrar(<id_archivo>), borra el identificador de archivo.

lda(<id_archivo>), función que nos devuelve la longitud del archivo en bytes.

tamaño_de(<nombre_de_var_o_tipo>), función que nos devuelve la longitud en bytes de una variable o tipo de dato. Para obtener el número de registros de un archivo directo se puede aplicar la siguiente fórmula:

$$\text{lدا}(<\text{id_archivo}>) / \text{tamaño_de}(<\text{var_tipo_base}>)$$

9.1.1. Funciones de conversión de clave

Normalmente cuando se trabaja con archivos directos se accede directamente a los registros empleando una clave. En ocasiones, la clave tendrá tales características que será posible utilizarla como índice, pero existirán otros muchos casos en los que esto no será así; por ejemplo, no es recomendable, cuando el porcentaje de claves que se piensan utilizar en la grabación de los registros es reducido en comparación con el rango en el que pueden oscilar los valores de las claves. Y no es posible cuando las claves son alfanuméricas. Una solución a esto es utilizar una función que transforme las claves a números en un determinado rango, que puedan servir como índices.

Existen funciones de transformación de clave que no dan origen a colisiones, como la que se puede aplicar si se desea transformar los números de las habitaciones de un hotel en un rango de direcciones que abarcara el total de habitaciones del mismo. No obstante, en la mayor parte de los casos, es necesario utilizar funciones que podrán originar colisiones, por ejemplo utilizando como clave el NIF y si se desea transformar en una dirección comprendida en un rango que abarque el número estimado de clientes de una empresa.

Las funciones de transformación de clave se estudiarán en el capítulo 10, y aquí se resalta que una función de este tipo debe reunir las siguientes características:

- Distribuir las claves uniformemente entre las direcciones para producir pocos sinónimos.
- No ser una función compleja que pueda ralentizar los cálculos.

Una de las más empleadas es la función módulo, que para la obtención de valores para el índice comprendidos en el rango deseado recurre a efectuar la división entera de la clave por el número de registros necesitados y tomar el resto, para después sumar 1 al resto.

Por ejemplo, si se tiene una clave numérica y se desea poder almacenar N registros en la zona de datos se puede definir de la forma siguiente:

```
entero función hash(E entero:clave)
inicio
    devolver(clave mod n + 1)
fin_función
```

En realidad, como la división por un número primo origina un menor número de colisiones, N será el número primo superior más próximo al número de registros estimado necesario para la zona de datos.

9.1.2. Tratamiento de sinónimos

El empleo de estas funciones puede producir que a dos registros con claves diferentes les corresponda la misma dirección relativa. Cuando a un registro le corresponde una dirección que ya está ocupada se dice que se ha producido una *colisión* o *sinónimo*. El tratamiento para las colisiones podrá ser de dos tipos:

- Buscar una nueva dirección libre en el mismo espacio donde se están introduciendo todos los registros, zona de datos.
- Crear una zona especial, denominada zona de excedentes, a donde llevar exclusivamente estos registros. La zona de desbordamiento o excedentes podría encontrarse a continuación de la zona de datos o ser, incluso, otro archivo.

9.1.3. Mantenimiento de archivos directos

Creación	La instrucción <i>crear</i> efectuará la creación del archivo.
Altas	La operación de altas consiste en introducir los sucesivos registros, en el soporte que los va a contener, en la posición que indique la clave o en la resultante de aplicar a la clave el algoritmo de conversión. Si, al introducir un nuevo registro y obtener su posición a través de un algoritmo de conversión, ésta se encuentra ya ocupada, el nuevo registro deberá ir a la zona de sinónimos.
Consulta	La consulta de un determinado registro en un archivo directo o aleatorio requiere la lectura del registro ubicado en la dirección que indica la clave o en la que se obtenga al aplicar a la clave el algoritmo de conversión. Cuando la función de conversión pueda dar origen a sinónimos se comparará la clave buscada con la que se acaba de leer y, en caso de no coincidir, se efectuará una búsqueda en la zona de sinónimos.
Bajas	Se efectuará la consulta del registro y, cuando se le encuentre, se volverá a escribir en la misma posición donde se le encontró, pero marcando alguno de sus campos con una señal que indique que dicho registro ha sido dado de baja. Este tipo de baja es una baja lógica.
Modificaciones	Análogo a las bajas, permitiéndose la modificación del contenido de cualquier campo que no sea el clave.

Excepto creación, todas las operaciones requieren la previa apertura del archivo de datos en el único modo posible. De lo anteriormente expuesto, se deduce que, cuando se emplee una función que transforme clave en dirección y pueda dar origen a colisiones, será necesario que los registros incluyan un campo donde almacenar la clave. Además, resultará siempre conveniente añadir otro campo auxiliar en el que marcar las bajas lógicas que efectuemos en el archivo.

9.2. ORGANIZACIÓN SECUENCIAL INDEXADA

Un archivo secuencial indexado consta de:

- El área de índices, que es un archivo secuencial que contiene las claves del último registro de cada bloque físico del archivo y la dirección de acceso al primer registro del bloque.

- El área principal, que contiene los registros de datos, clasificados en orden ascendente por el campo clave.
- El área de desbordamiento o excedentes, donde se almacenarán los nuevos registros que no se puedan situar en el área principal en las actualizaciones.

Los soportes que se utilizan para esta organización son los que permiten el acceso directo.

Área de índices		Área de datos	
Clave	Dirección	Clave	Datos
24	0010	0010	6
41	0020	0011	16
56	0030	0012	20
92	0040
...	...	0019	24
245	0100	0020	28
		0021	29
	
		0029	41
		0030	43
	
	
		0090	...
	
		0100	345

Se puede acceder a la información de forma secuencial o a través del índice. Secuencialmente, recorremos el archivo completo en orden de claves. A través del índice, se lee secuencialmente el archivo índice hasta encontrar una clave mayor o igual a la buscada; el otro campo nos dará la posición del primer registro del bloque donde se encuentra la información.

Las instrucciones para su manipulación serían

```

tipo
  archivo_i de <tipo_dato>:<nombre_de_tipo>
    // Clave primaria sin duplicados
  clave_p <lista_de_campos>
    // Clave secundaria con o sin duplicados
  [clave_s <lista_de_campos>[duplicada]]

var
  <nombre_de_tipo>:<id_archivo>

  crear(<id_archivo>, <nombre_físico>)
  abrir(<id_archivo>, 'l/e', <nombre_físico>)
  escribir(<id_archivo>, <var_tipo_base>), escribe el registro en la posición que tenga en ese momento la clave primaria.

  leer(<id_archivo>, <var_tipo_base> [, <campo_clave>]), lee el registro cuyo valor coincide con el que en ese momento tenga la clave primaria. Para leer por claves alternativas se incluiría el argumento <campo_clave>.

  leersec(<id_archivo>, <var_tipo_base>, [<campo_clave>]), lectura secuencial del siguiente registro de la clave especificada. Por omisión lee la siguiente clave secuencial.

  está(<id_archivo>), devuelve verdadero si la lectura ha sido correcta.

```

```
lida(<nombre_físico>)
fda(<id_archivo>)
borrar(<nombre_físico>)
renombrar(<nombre_físico1>, <nombre_físico2>)
```

Los archivos secuenciales indexados tienen como ventajas un rápido acceso y su sistema de gestión de archivos que se encarga de relacionar la posición de cada registro con su clave mediante la tabla de índices.

Como inconvenientes estarían el desaprovechamiento del espacio por quedar huecos intermedios en las actualizaciones del archivo y la necesidad de espacio adicional para el área de índices. Este tipo de organización no está disponible en todos los lenguajes.

9.3. MODOS DE ACCESO

El interés de distinguir entre las diferentes clases de ficheros obedece fundamentalmente a una razón de eficiencia. Sobre las organizaciones especificadas, pueden programarse formas de acceso que no son las propias de la organización y que, si no son adecuadas, podrían conducir a programas poco prácticos.

9.3.1. Archivos indexados

Se puede realizar el acceso a un archivo directo mediante indexación. Esto nos conduciría a la creación de dos archivos: índice y datos.

El *archivo índice* es conveniente sea de organización secuencial, o por lo menos con sus registros colocados de forma secuencial. Estos registros han de contener los campos clave y dirección.

El *archivo de datos* ha de tener organización directa y sus registros contendrán los datos.

Para su mantenimiento deberemos abrir cada uno de los ficheros del modo conveniente para que sus registros resulten accesibles. Interesaría comenzar cargando el archivo índice en un array de registros, al que desde ahora nos referiremos como índice, y realizar nuestras operaciones en él, pues el trabajo en memoria resulta más rápido. Las operaciones a realizar serán:

Altas	Consiste en introducir el nuevo registro en el archivo de datos, reflejando en el índice la clave y posición donde los datos han sido colocados. Antes de añadir el nuevo registro deberemos comprobar que no existe ningún otro con dicha clave. Tras efectuar un alta, el índice debe quedar ordenado por el campo clave.
Consultas	La consulta se efectuará por el campo clave y a través del índice, lo que posibilitará la realización de una rápida búsqueda binaria (que se verá en el capítulo siguiente) y un posterior posicionamiento directo sobre el registro que contiene los datos.
Bajas	Para efectuar una baja lógica podría bastar con retirar del índice el registro con dicha clave.
Modificaciones	Primero se buscará en el índice la clave; por ella se obtendrá la posición del registro en el archivo de datos y será posible situarse directamente sobre el registro correspondiente en el archivo de datos, para leerlo, modificarlo y volverlo a colocar en el mismo lugar después de modificado.

Al terminar se sobrescribe el archivo índice con el array de registros con el que se ha estado trabajando. Se pueden ampliar estos conceptos y establecer índices jerárquicos.

9.4. EJERCICIOS RESUELTOS

- 9.1.** Escribir un algoritmo que permita la creación e introducción de los primeros datos en un archivo directo, cuyos registros serán del siguiente tipo:

```

tipo
    registro: datos_personales
        <tipo_dato1> : cód           // Campo clave
        ..... : .....
        <tipo_datoN> : nombre_campoN
    fin_registro

```

Análisis del problema

Tras crear el archivo, la introducción de datos se hará especificando, mediante un índice, la posición con respecto al origen del fichero, donde deseamos almacenarlos.

El índice permitirá el posicionamiento directo sobre el byte del fichero que se encuentre en la posición (índice - 1) * tamaño_de(datos_personales).

Diseño del algoritmo

```

algoritmo Ejercicio_9_1
tipo
    registro: datos_personales
        <tipo_dato1> : cód // Podría no ser necesario
                           // su almacenamiento, en el caso
                           // de que coincidiera con el
                           // índice
        ..... : .....
        <tipo_daton> : nombre_campoN
    fin
    archivo_d de datos_personales: arch
var
    arch          : f
    datos_personales : persona
    entero        : n
inicio
    crear(f,<nombre_en_disco>
    abrir(f,'l/e',<nombre_en_disco>)
    escribir('Indique el número de registros que desea introducir ')
    leer(n)
    desde i ← 1 hasta n hacer
        llamar_a leer_reg(persona)
        escribir(f, persona, i)      // El índice es i
    fin_desde
    cerrar(f)
fin

```

- 9.2.** Diseñar un algoritmo que efectúe la creación de un archivo directo —PERSONAL—, cuyos registros serán del siguiente tipo:

```

tipo
    registro: datos_personales

```

```

<tipo_dato1> : cód          // Campo clave
..... : .....
<tipo_datoN> : nombre_campoN
fin_registro

```

y en el que, posteriormente, vamos a introducir la información empleando el método de transformación de clave.

Análisis del problema

El método de transformación de clave consiste en introducir los registros, en el soporte que los va a contener, en la dirección que proporciona el algoritmo de conversión. Su utilización obliga al almacenamiento del código en el propio registro y hace conveniente la inclusión en el registro de un campo auxiliar —ocupado— en el que se marque si el registro está o no ocupado.

Durante el proceso de creación se debe realizar un recorrido de todo el fichero inicializando el campo ocupado a vacío, por ejemplo a espacio.

Diseño del algoritmo

```

algoritmo Ejercicio_9_2
const
    Máx = <valor>
tipo
    registro: datos_personales
        <tipo_dato1> : cód // Podría no ser necesario
                            // su almacenamiento, en el caso
                            // de que coincidiera con el
                            // índice
        ..... : .....
        <tipo_datoN> : nombre_campoN
fin
archivo_d de datos_personales: arch
var
    arch : f
    datos_personales : persona
    entero : i
inicio
    crear(f,'PERSONAL')
    abrir(f,'l/e','PERSONAL')
    desde i ← 1 hasta Máx hacer
        persona.ocupado ← ''
        escribir(f, persona, i)
    fin_desde
    cerrar(f)
fin

```

- 9.3.** Diseñar un algoritmo que permita introducir información en el archivo PERSONAL mediante el método de transformación de clave.

Análisis del problema

Como anteriormente se explicó, el método de transformación de claves consiste en introducir los registros, en el soporte que los va a contener, en la dirección que proporciona el algoritmo de conversión. A veces, regis-

tros distintos, sometidos al algoritmo de conversión, proporcionan una misma dirección, por lo que se tendrá previsto un espacio en el disco para el almacenamiento de los registros que han colisionado. Aunque se puede hacer de diferentes maneras, en este caso se reservará espacio para las colisiones en el propio fichero a continuación de la zona de datos.

Se supondrá que la dirección más alta capaz de proporcionar el algoritmo de conversión es Findatos y se colocarán las colisiones que se produzcan a partir de allí en posiciones consecutivas del archivo.

La inicialización a espacio del campo ocupado se realizó hasta Máx, se da por supuesto que Máx es mayor que Findatos.

Diseño del algoritmo

```

algoritmo Ejercicio_9_3
const
    Findatos = <valor1>
    Máx      = <valor2>
tipo
    registro: datos_personales
        <tipo_dato1> : cód // Podría no ser necesario
                        // su almacenamiento, en el caso
                        // de que coincidiera con el
                        // índice
        ..... : .....
        <tipo_datoN> : nombre_campo
    fin
    archivo_d de datos_personales: arch
var
    arch          : f
    datos_personales : persona, personaaux
    lógico        : encontradohueco
    entero         : posi
inicio
    abrir(f,'l/e','PERSONAL')
    leer(personaaux.cód)
    posi ← HASH(personaaux.cód)
    // HASH es el nombre de la función de transformación de
    // claves. La cual devolverá valores
    // entre 1 y Findatos, ambos inclusive
    leer(f, persona, posi)
    si persona.ocupado = '*' entonces //El '*' indica que está
                                //ocupado
        encontradohueco ← falso
        posi ← Findatos
        mientras posi < Máx y no encontradohueco hacer
            posi ← posi+1
            leer(f, persona, posi)
            si persona.ocupado <> '*' entonces
                encontradohueco ← verdad
            fin_si
        fin_mientras
    si_no
        encontradohueco ← verdad

```

```

fin_si
si encontradohueco entonces
    llamar_a leer_otros_camplos(personaux)
    persona ← personaux
    persona.ocupado ← '*' // Al dar un alta marcaremos el campo ocupado
    escribir(f, persona, pos)
si_no
    escribir('No está')
fin_si
cerrar(f)
fin

```

9.4. Implementar la consulta de un registro en el archivo PERSONAL.

Análisis del problema

Para localizar el registro se aplicará al código la función de transformación de claves y, si no se encuentra en la dirección devuelta por la función, se examinará, registro a registro, la zona de colisiones.

Para decidir que se ha encontrado habrá que comprobar que no está de baja y que su código coincide con el que estamos buscando. En cuyo caso, se interrumpirá la búsqueda y mostraremos el registro por pantalla.

Diseño del algoritmo

```

algoritmo Ejercicio_9_4
const
    Findatos = <valor1>
    Máx      = <valor2>
tipo
    registro: datos_personales
        <tipo_dato1> : cód // Podría no ser necesario su almacenamiento, en
                           // el caso de que coincidiera con el índice
        ..... : .....
        <tipo_datoN> : nombre_campo
fin
archivo_d de datos_personales: arch
var
    arch          : f
    datos_personales : persona, personaux
    lógico        : encontrado
    entero        : pos
inicio
    abrir(f, 'l/e', 'PERSONAL')
    leer(personaux.cód)
    pos ← HASH(personaux.cód)
    leer(f, persona, pos)
    si persona.ocupado <> '*' o persona.cód <> personaux.cód entonces
        encontrado ← falso
        pos ← Findatos
        mientras pos < Máx y no encontrado hacer
            pos ← pos+1
            leer(f, persona, pos)

```

```

        si persona.ocupado = '*' y persona.cód = personaaux.cód entonces
            encontrado ← verdad
        fin_si
    fin_mientras
si_no
    encontrado ← verdad
fin_si
si encontrado entonces
    llamar_a escribir_reg(persona)
si_no
    escribir('No está')
fin_si
cerrar(f)
fin

```

9.5. Algoritmo que nos permita eliminar un determinado registro de PERSONAL.

Análisis del problema

Efectuaremos baja lógica, la cual consistirá en:

- Localizar el registro, buscándolo, por su código, primero en la zona directa y, si allí no se encuentra, en la zona de colisiones.
- Una vez localizado eliminar la marca existente en el campo ocupado, asignando a dicho campo el valor que se interprete como vacío, y escribir nuevamente el registro en el archivo en la posición donde se encontró.

Diseño del algoritmo

```

algoritmo Ejercicio_9_5
const
    Findatos = <valor1>
    Máx      = <valor2>
tipo
    registro: datos_personales
        <tipo_dato1> : cód // Podría no ser necesario
                        // su almacenamiento, en el caso
                        // de que coincidiera con el índice
        ..... : .....
        <tipo_datoN> : nombre_campo
fin_registro
archivo_d de datos_personales: arch
var
    arch          : f
    datos_personales : persona, personaaux
    lógico        : encontrado
    entero         : posi
inicio
    abrir(f,'l/e','PERSONAL')
    leer(personaaux.cód)
    posi ← HASH(personaaux.cód)
    leer(f, persona, posi)

```

```

si persona.ocupado <> '*' o persona.cód <> personaux.cód entonces
    encontrado ← falso
    posi ← Findatos
    mientras posi < Máx y no encontrado hacer
        posi ← posi+1
        leer(f, persona, posi)
        si persona.ocupado = '*' y persona.cód = personaux.cód entonces
            encontrado ← verdad
        fin_si
    fin_mientras
si_no
    encontrado ← verdad
fin_si
si encontrado entonces
    persona.ocupado ← ''
    escribir(f, persona, posi)
si_no
    escribir('No está')
fin_si
cerrar(f)
fin

```

9.6. Algoritmo para la modificación de un determinado registro del fichero PERSONAL.

Análisis del problema

El problema resulta totalmente similar a las bajas, pues en éstas se modificaba también un campo del registro, el campo ocupado.

Diseño del algoritmo

```

algoritmo Ejercicio_9_6
const
    Findatos = <valor1>
    Máx      = <valor2>
tipo
    registro: datos_personales
        <tipo_dato1> : cód // Podría no ser necesario
                        // su almacenamiento, en el caso
                        // de que coincidiera con el
                        // índice
        ..... : .....
        <tipo_datoN> : nombre_campo
    fin
    archivo_d de datos_personales: arch
var
    arch          : f
    datos_personales : persona, personaux
    lógico       : encontrado
    entero        : posi
inicio

```

```

abrir(f,'l/e','PERSONAL')
leer(personaux.cód)
posi ← HASH(personaux.cód)
leer(f, persona, posi)
si persona.ocupado >> '*' o persona.cód >> personaux.cód entonces
    encontrado ← falso
    posi ← Findatos
    mientras posi < Máx y no encontrado hacer
        posi ← posi+1
        leer(f, persona, posi)
        si persona.ocupado = '*' y persona.cód = personaux.cód entonces
            encontrado ← verdad
        fin_si
    fin_mientras
si_no
    encontrado ← verdad
fin_si
si encontrado entonces
    llamar_a leer_otros_campos(personaux)
    personaux.ocupado ← '*'
    escribir(f, personaux, posi)
si_no
    escribir('No está')
fin_si
cerrar(f)
fin

```

- 9.7.** Una empresa tiene la información sobre sus empleados dividida en dos archivos secuenciales. En uno de los ficheros almacena los siguientes datos: N.^o de contrato (3 dígitos), Nombre y apellidos, DNI. y N.^o de la Seguridad Social. En el otro se encuentran: N.^o de contrato, Salario, Comisiones y descuentos. Teniendo en cuenta que, en los ficheros originales, cada empleado aparece una sola vez y no existen erratas y que, entre ambos archivos, la información, con respecto a cada uno de los empleados, siempre se completa. Diseñar un algoritmo que cree un tercer fichero de acceso por transformación de clave, de manera que, cada uno de sus registros, contenga el N.^o de contrato y el resto de la información relacionada con dicho número. Reserve espacio para 97 registros en la zona de datos.

Análisis del problema

Aunque en los archivos iniciales haya el mismo número de registros y sepamos que cada registro de un fichero tiene su correspondiente en el otro, no podemos comenzar por emparejarlos, pues la información no tiene por qué encontrarse colocada en el mismo orden en ambos. Recurrirremos pues a introducir en el nuevo archivo la información de uno de los ficheros secuenciales completando, después, el contenido de los registros con la información proporcionada por el otro.

Diseño del algoritmo

```

algoritmo Ejercicio_9_7
tipo
    registro: reg1
    entero: ncontrato
    cadena: nombre

```

```

    cadena: dni
    cadena: nsegs
fin_registro
registro: reg2
    entero: ncontrato
    real : salario, comisiones, descuentos
fin_registro
registro: reg3
    entero: ncontrato
    cadena: nombre
    cadena: dni
    cadena: nsegs
    real : salario, comisiones, descuentos
fin_registro
archivo_d de reg3: arch3
archivo_s de reg1: arch1
archivo_s de reg2: arch2
var
    arch3: f3
    reg3 : r3
inicio
    crear(f3,'f3.dat')
    abrir(f3,'l/e','f3.dat')
    desde i ← 1 hasta 125 hacer
        r3.ncontrato ← 0
        escribir(f3, r3, i)
    fin_desde
    llamar_a introducir(f3)
    llamar_a modificar(f3)
    cerrar(f3)
fin

entero función hallarnreg (E entero : nct)
inicio
    devolver( nct mod 97 + 1 )
fin_función

procedimiento introducir(E/S arch3: f3)
var
    arch1 : f1
    reg1 : r1
    reg3 : r3
    entero : nreg
inicio
    abrir(f1,'l','f1.dat')
    mientras no FDA(f1) hacer
        llamar_a leer_arch_reg(f1, r1)
        nreg ← hallarnreg(r1.ncontrato)
        leer(f3, r3, nreg)
        si r3.ncontrato <> 0 entonces

```

```

encontradositio ← falso
nreg ← 97
mientras nreg < 125 y no encontradositio hacer
    nreg ← nreg + 1
    leer(f3,r3, nreg)
    si r3.ncontrato = 0 entonces
        encontradositio ← verdad
    fin_si
fin_mientras
si_no
    encontradositio ← verdad
fin_si
si encontradositio entonces
    r3.ncontrato ← r1.ncontrato
    r3.nombre ← r1.nombre
    r3.dni ← r1.dni
    r3.nsegs ← r1.nsegs
    escribir(f3,r3, nreg)
si_no
    escribir('No podemos almacenar el registro, no hay sitio')
fin_si
fin_mientras
cerrar(f1)
fin_procedimiento

procedimiento modificar(E/S arch3: f3)
var
    arch2 : f2
    reg2 : r2
    reg3 : r3
    entero : nreg
inicio
    abrir(f2,'l','f2.dat')
    mientras no FDA(f2) hacer
        llamar_a leer_arch_reg(f2, r2)
        nreg ← hallarnreg(r2.ncontrato)
        leer(f3, r3, nreg)
        si r2.ncontrato = r3.ncontrato entonces
            r3.salario ← r2.salario
            r3.comisiones ← r2.comisiones
            r3.descuentos ← r2.descuentos
            escribir(f3, r3, nreg)
        si_no
            nreg ← 97
            encontrado ← falso
        mientras nreg < 125 y no encontrado hacer
            nreg ← nreg + 1
            leer(f3, r3, nreg)
            si r2.ncontrato = r3.ncontrato entonces
                r3.salario ← r2.salario

```

```

        r3.comisiones ← r2.comisiones
        r3.descuentos ← r2.descuentos
        escribir(f3, r3, nreg)
        encontrado ← verdad
    fin_si
fin_mientras
fin_si
fin_mientras
cerrar(f2)
fin_procedimiento

```

- 9.8.** Implementar las rutinas necesarias para la gestión de un archivo directo con transformación de claves para control de personal que contenga los campos DNI, nombre del empleado, departamento y sueldo. El archivo está diseñado para contener 100 registros (más un 20% más para colisiones). Se supone que el programa principal tiene las siguientes declaraciones, y que el archivo está abierto.

```

const
MáxReg = 120
tipo
    registro : TipoR
        cadena : dni
        cadena : nombre
        cadena : dep
        entero : sueldo
        carácter : estado
        // El campo estado tendrá 'L' (libre), 'O'
        // (ocupado) o 'B' (baja)}
    fin_registro
    archivo_d de TipoR : TipoF
var
    TipoR : r
    TipoF : f
inicio
    abrir(f, '1/e', 'PERSONAL')
    .
    .
    .
    cerrar(f)
fin

```

Creación

Para crear el archivo se inicializan MáxReg posiciones, poniendo el campo estado como libre (L).

```

procedimiento Creación(E/S TipoF : f)
var
    TipoR : r
    entero : i
inicio
    r.estado ← 'L' // Al crear los registros están libres

```

```

para i ← 1 hasta MáxReg hacer
    escribir(f,r,i)
fin_desde
fin_procedimiento

```

Altas

Para dar un alta, primero se busca el registro; si no está se procede al alta. Primero calculamos la dirección mediante la función hash. Si está libre, se introducen los datos del registro y en dicha dirección se da un alta. Si no, se soluciona la colisión buscando el primer registro vacío.

Para solucionar la colisión utilizamos el direccionamiento a vacío. Desde la posición asignada según la función hash, se hace una búsqueda secuencial hasta encontrar el primer registro válido. Consideramos el archivo como circular, es decir, si en la búsqueda llegamos al registro MáxReg pasamos al registro número 1 para continuar el proceso. Siempre encontraremos sitio, ya que va a haber 100 registros y tenemos 120 posiciones.

```

procedimiento Altas(E/S TipoF : f)
var
    TipoR : r, RegAux
    entero : d // dirección hash donde almacenaremos el registro
    entero : p // posición donde está el registro , 0 si la clave no
                existe
inicio
    LeerClave(RegAux)
    p ← Buscar(f,RegAux)
    si p <> 0 entonces
        // error, el registro existe
    si_no
        d ← Hash(RegAux) //Cálculo de la dirección hash
                           // del registro
        leer(f,r,d)
        si r.estado = '0' entonces
            // si dicha posición está ocupada búsqueda de la primera
            // posición libre
            repetir
                d ← d + 1
                // si llegamos al final del archivo
                si d > MáxReg entonces
                    // vuelta al primer registro
                    d ← 1
                fin_si
                leer(f,r,d)
            hasta_que r.estado <> '0'// hasta encontrar un registro libre
        fin_si
        // tenga o no colisiones
        LeerDatos(RegAux) // rellenamos los campos del registro auxiliar
        RegAux.estado ← '0' // Marcamos el registro como ocupado
        escribir(f,RegAux,d)
    fin_si
fin_procedimiento

```

```
procedimiento LeerClave(E/S TipoR : r)
inicio
    leer(r.dni)
fin

entero función Hash(E/S TipoR : r)
inicio
    devolver(Valor(r.dni) mod 97 + 1)
fin_función
procedimiento LeerDatos(E/S TipoR : r)
inicio
    leer(r.nombre)
    leer(r.dep)
    leer(r.sueldo)
fin_procedimiento

entero función Buscar(E/S TipoF : f; E TipoR : r)
var
    entero : d
    TipoR : RegAux
inicio
    d← hash(r)
    leer(f,RegAux,d)
    si RegAux.estado = 'L' entonces // si la posición está
        // libre, el registro no está
        devolver(0)
    si_no
        // si la posición está ocupada y encontramos el
        // registro
        si RegAux.estado = 'O' y Clave(RegAux) = Clave(r) entonces
            devolver(d)
        si_no // puede ser una colisión
            repetir
                d← d + 1
                // si llegamos al final del registro
                si d > MáxReg entonces
                    d← 1 // vuelta al primer
                    // registro
                fin_si
                leer(f,RegAux,d)
            hasta_que RegAux.estado='L' o (RegAux.estado='O'
                y Clave(r)=Clave(RegAux))
            si Clave(r) = Clave(RegAux) entonces
                devolver(d)
            si_no
                devolver(0)
            fin_si
        fin_si
    fin_si
fin_función
```

Consultas

Para buscar un registro por la clave (DNI), después de leer la clave buscada, simplemente utilizaremos la función `buscar` para ver si existe en el archivo. Si la clave existe, utilizaremos el procedimiento `SalidaDatos()` para visualizar el contenido del registro encontrado.

```
procedimiento Consultas(E/S TipoF : f)
var
    TipoR : r, RegAux
    entero : p
inicio
    LeerClave(RegAux)
    p ← Buscar(f, RegAux)
    si p = 0 entonces
        // el registro no está
    si_no
        leer(f, r, p)
        SalidaDatos(r)
    fin_si
fin_procedimiento

procedimiento SalidaDatos(E/S TipoR : r)
inicio
    escribir(r.nombre)
    escribir(r.dep)
    escribir(r.sueldo)
fin_procedimiento
```

Bajas

Dar un registro de baja va a suponer únicamente poner el campo estado a «B». Por lo tanto, deberemos localizar el registro con la función `Buscar()`; si se encuentra confirmaremos la baja y grabaremos el registro con el campo estado modificado.

```
procedimiento Bajas( E/S TipoF : f)
var
    TipoR : r, RegAux
    entero : p
    carácter : baja
inicio
    LeerClave(RegAux)
    p ← Buscar(f, RegAux)
    si p = 0 entonces
        // el registro no está
    si_no
        leer(f, r, p)
        SalidaDatos(r)
        leer(baja) // confirmación de la baja
        si baja = 'S' entonces
            r.estado ← 'B'
            escribir(f, r, p)
    fin_si
```

```
fin_si
fin_procedimiento
```

Modificaciones

Modificar un registro va a consistir sólo en localizar el registro, modificar los datos y grabarlo en la misma posición.

```
procedimiento Modificaciones(E/S TipoF : f)
var
    TipoR : r, RegAux
    entero : p
inicio
    LeerClave(RegAux)
    p ← Buscar(f, RegAux)
    si p = 0 entonces
        // el registro no está}
    si_no
        leer(f, r, p)
        ModificarRegistro(r)
        escribir(f, r, p)
    fin_si
fin_procedimiento
```

- 9.9. Se desea crear un archivo directo por transformación de claves a partir de un archivo secuencial de productos. Los campos del archivo secuencial son Código de Producto, Descripción del Producto, Precio y Stock. Se supone que no hay códigos repetidos en el archivo secuencial. El archivo debe contener un máximo de 100 productos distintos, dejando un 20% para la zona de colisiones que estará situada al final del archivo. La función hash que se utilizará será: $\text{hash}(\text{clave}) = \text{clave mod } 97 + 1$.

```
algoritmo Ejercicio_9_9
const
    MáxReg = 120
    ZonaCol = 97
    //la dirección mayor que proporcional la función hash es 97
tipo
    registro : TipoR
        cadena : Cód-Prod
        cadena : Descripción
        entero : Precio
        entero : Stock
        carácter : Estado
    fin_registro
    archivo_d de TipoR : TipoA
    registro : TipoRS
        cadena : Cód-Prod
        cadena : Descripción
        entero : Precio
        entero : Stock
    fin_registro
```

```

archivo_s de TipoRS : TipoAS
var
    TipoA   : A
    TipoAS  : AS
    TipoR   : R
    TipoRS  : RS
    entero  : d
inicio
    abrir(AS, 'l', 'SECUENCIAL')
    crear(A, 'PRODUCTOS')
    abrir(A, 'l/e', 'PRODUCTOS')
    Creación (A,R)
    leer(AS,RS)
    mientras no FDA(AS) hacer
        d ← Hash(RS)
        leer(A,R,d)
        si R.estado = 'O' entonces
            d ← ZonaCol
            repetir
                d ← d + 1
                leer(A,R,d)
            hasta que R.estado <> 'O' o d = MáxReg
        fin_si
        si R.estado <> 'O' entonces
            R.Cód-Prod ← RS.Cód-Prod
            R.Descripción ← RS.Descripción
            R.Precio ← RS.Precio
            R.Stock ← RS.Stock
            R.Estado ← 'O'
        si_no
            // Zona de Colisiones llena
        fin_si
    fin_mientras
    cerrar(A)
    cerrar(AS)
fin

```

- 9.10.** Se desea actualizar el archivo creado en el ejercicio 9.9. Para ello se dispone de un archivo de movimientos con los campos Código del Producto, Descripción, Precio, Cantidad y Tipo. El tipo será A(alta), B(bajas), M(modificaciones), E (entrada de productos) y S (Salida de productos). Puede haber varios movimientos de tipo E y S por producto. Pueden presentarse incidencias —altas que ya existan y modificaciones, bajas, entradas o salidas que no existan—. Si esto ocurre, se dará un alta en un archivo secuencial de incidencias que tendrá los campos Código del Producto y Literal. El literal explicará el tipo de incidencia.

```

algoritmo Ejercicio_9_10
const
    MáxReg = 120
    ZonaCol = 97
    //la dirección mayor que proporcional la función hash es 97

```

```

tipo
  registro : TipoR
    cadena : Cód-Prod
    cadena : Descripción
    entero : Precio
    entero : Stock
    carácter : Estado
  fin_registro
  archivo_d de TipoR : TipoA
  registro : TipoRS
    cadena : Cód-Prod
    cadena : Descripción
    entero : Precio
    entero : Cantidad
    carácter : Tipo
  fin_registro
  archivo_s de TipoRM : TipoAM
  registro : TipoRI
    cadena : Cód-Prod
    cadena : Literal
  fin_registro
  archivo_s de TipoRI : AI
var
  TipoA : A
  TipoAS : AS
  TipoAI : AI
  TipoR : R
  TipoRM : RM
  TipoRI : RI
  entero : NúmReg
inicio
  abrir(AM, l, 'MOVIMIENTOS')
  abrir(AI, e, 'INCIDENCIAS')
  abrir(A, l/e, 'PRODUCTOS')
  leer(AM, RM)
  mientras no FDA(AM) hacer
    R.Cód-Prod ← RM.Cód-Prod
    NúmReg ← Buscar(F, R)
    si NúmReg = 0 entonces
      si RM.tipo <> 'A' entonces
        // Incidencia
        RI.Cód-Prod ← RM.Cód-Prod
        RI.Literal ← 'El registro no existe'
        escribir(AI, RI)
    si_no
      // Mover campos de RM a R
      Altas(A, R)
    fin_si
    leer(AM, RM)
  si_no

```

```

    si RM.tipo = 'A' entonces
        // Incidencia
        RI.Cód-Prod ← RM.Cód-Prod
        RI.Literal ← 'El registro ya existe'
        escribir(AI,RI)
        leer(AM,RM)
    si_no
        si RM.tipo = 'B' entonces
            R.Estado ← 'B'
            leer(AM,RM)
        si_no
            si RM.tipo = 'M' entonces
                // Mover campos de RM a R
                leer(AM,RM)
            si_no
                mientras R.Cód-Prod = RM.Cód-Prod y no FDA(AM) hacer
                    si RM.Tipo = 'E' entonces
                        R.Stock ← R.Stock + RM.Cantidad
                    si_no
                        R.Stock ← R.Stock - RM.Cantidad
                    fin_si
                    leer(AM,RM)
                fin_mientras
            fin_si
        fin_si
        escribir(A,R,d)
    fin_si
    fin_si
fin_mientras
cerrar(A)
cerrar(AM)
cerrar(AI)
fin

entero función Buscar(E/S TipoA : A; E TipoR : r)
var
    entero : d
    TipoR : RegAux
inicio
    d ← hash(r)
    leer(A,RegAux,d)
    si RegAux.estado = 'L' entonces // si la posición está
        // libre, el registro
        // no está
        devolver(0)
    si_no
        // si la posición está ocupada y
        // encontramos el registro
        si RegAux.estado = 'O' y Clave(RegAux) = Clave(r) entonces
            devolver(d)

```

```

si_no // puede ser una colisión}
    d ← ZonaCol
    repetir
        d ← d + 1
        leer(R, RegAux, d)
        hasta que RegAux.estado='L' o (RegAux.estado='O' y
            Clave(r)=Clave(RegAux)) o d = MáxReg
        si Clave(r) = Clave(RegAux) entonces
            devolver(d)
        si_no
            devolver(0)
        fin_si
    fin_si
fin_si
fin_función

```

9.11. Diseñar un algoritmo que permita efectuar ALTAS en un archivo indexado¹.

Análisis del problema

Supongamos la existencia de dos archivos: un archivo de datos o archivo principal, directo y otro, secuencial, cuyos registros estarán constituidos únicamente por dos campos: código o clave y posición, y con el que cargaremos la tabla de índices.

El archivo de índices se almacenará siempre con su información ordenada por código, por lo que al cargar con él la tabla no será necesario ordenarla.

Para efectuar el alta de un nuevo registro podremos consultar su existencia en la tabla que, como está en memoria y ordenada, permite consultas muy rápidas. Si no se encuentra, añadiremos el nuevo registro al final del fichero de datos y actualizaremos la tabla, insertando la información sobre el nuevo registro en el lugar adecuado para que permanezca ordenada.

Cuando termine el programa deberá ser destruido el antiguo archivo de índices y creado de nuevo a partir del contenido de la tabla.

Diseño del algoritmo

```

algoritmo Ejercicio_9_11
const
    Máx = <expresión>
tipo
    registro: reg
        <tipo_dato1> : cód
        ..... : ....
        ..... : ....
    fin_registro
    archivo_d de reg: archd
    registro: índ
        <tipo_dato1> : cód
        entero : posic
    fin_registro

```

¹ En los ejercicios propuestos con archivos indexados la implementación se ha realizado utilizando un área de datos utilizando formada por un archivo de organización directa y un área de índices formada por un array de registros. Por razones pedagógicas, los autores han preferido esta solución en vez de utilizar la instrucciones para el manejo de archivos indexados que aparecen en el apartado 9.2.

```

array[1..Máx] de índ: arr
archivo_s de índ: archi
var
    archd      : f
    archi      : t
    arr        : a
    entero     : i, n, primero, último, central
    lógico     : encontrado
    <tipo_dato> : cód
    reg        : r
inicio
    abrir(f,'l/e','Datos.dat')
    abrir(t,'l','Indice.dat')
    i ← 1
    mientras no fda(t) hacer
        leer(t, a[i].cód, a[i].posic)
        i ← i + 1
    fin_mientras
    n ← i - 1
    cerrar(t)
    leer(cód)
    mientras no último_dato(cód) hacer
        primero ← 1
        último ← n
        encontrado ← falso
        mientras primero <= último y no encontrado hacer
            central ← (primero + último) div 2
            si a[central].cód = cód entonces
                encontrado ← verdad
            si_no
                si a[central].cód > cód entonces
                    último ← central - 1
                si_no
                    primero ← central + 1
                fin_si
            fin_si
        fin_mientras
        si encontrado entonces
            escribir('Ya existe')
        si_no
            si n = Máx entonces
                escribir('Lleno')
            si_no
                desde i_ n hasta primero decremento 1 hacer
                    a[i+1] ← a[i]
                fin_desde
                a[primero].cód ← cód
                a[primero].posic ← lda(f)/tamaño_de(reg)+1
                n ← n + 1
            llamar_a leer_reg(r)

```

```

        escribir(f, ,r, a[primerol].posic)
    fin_si
fin_si
leer(cód)
fin_mientras
crear(t, 'Indice.dat')
abrir(t, 'e', 'Indice.dat')
desde i ← 1 hasta n hacer
    escribir(t, a[i].cód, a[i].posic)
fin_desde
cerrar(t)
cerrar(f)
fin

```

9.12. Implementar la operación de consulta de un registro en un archivo indexado.

Análisis del problema

El algoritmo consistirá en:

- Cargar la tabla de índices.
- Buscar el código en la tabla.
- Leer el registro del fichero de datos, de la posición donde indica la tabla que se encuentra.
- Presentar la información almacenada en el registro por pantalla.

Diseño del algoritmo

```

algoritmo Ejercicio_9_12
const
    Máx = <expresión>
tipo
    registro: reg
        <tipo_dato1> : cód
        <tipo_dato2> : nombre_campo2
        ..... : .....
        <tipo_datoN> : nombre_campoN
        ..... : .....
fin_registro
registro: índ
    <tipo_dato1> : cód
    entero       : posic
fin_registro
archivo_d de reg      : archd
archivo_s de índ      : archi
array[1..Máx] de índ : arr
var
    archd          : f
    archi          : t
    arr            : a
    reg            : r
    entero         : i, n, central
    <tipo_dato1>  : cód

```

```

lógico : encontrado
inicio
  abrir(f,'l/e',<nombre_en_disco1>)
  abrir(t,'l',<nombre_en_disco2>)
  i ← 1
  mientras no fda(t) hacer
    leer(t, a[i].cód, a[i].posic)
    i ← i + 1
  fin_mientras
  n ← i - 1
  cerrar(t)
  // El archivo índice almacena sus registros ordenados por
  // el campo cód
  leer(cód)
  mientras no último_dato(cód) hacer
    llamar_a búsqueda_binaria(a, n, cód, central,encontrado)
    si encontrado entonces
      leer(f, r, a[central].posic)
      llamar_a escribir_reg(r)
    si_no
      escribir('No está')
    fin_si
    leer(cód)
  fin_mientras
  cerrar(f)
fin

```

9.13. Algoritmo que efectúe bajas lógicas en un archivo indexado.

Análisis del problema

Las bajas lógicas pueden consistir únicamente en eliminar la información sobre dicho registro de la tabla.

Al final de la operación de bajas se creará de nuevo el archivo de índices con el contenido actual de la tabla.

Al efectuar los procesos de altas y bajas de la manera indicada se tendrá la posibilidad de recuperar la información borrada. Al cabo de un cierto tiempo, o cuando se decida que ya no es necesario, se reorganizará el archivo de datos, eliminando definitivamente los registros dados de baja.

Diseño del algoritmo

```

algoritmo Ejercicio_9_13
const
  Máx = <expresión>
tipo
  registro: reg
    <tipo_dato1> : cód
    <tipo_dato2> : nombre_campo2
    ..... : .....
    <tipo_datoN> : nombre_campón
    ..... : .....
fin_registro

```

```

registro: índ
    <tipo_dato1> : cód
    entero         : posic
fin_registro
archivo_d de reg      : archd
archivo_s de índ      : archi
array[1..Máx] de índ : arr

var
    archd          : f
    archi          : t
    reg            : r
    arr            : a
    entero         : i, n, primero, último, central
    <tipo_dato1>  : cód
    lógico         : encontrado
    carácter       : resp

inicio
    abrir(f,'l/e','Datos.dat')
    abrir(t,'l','Indice.dat')
    i ← 1
    mientras no fda(t) hacer
        leer(t, a[i].cód, a[i].posic)
        i ← i + 1
    fin_mientras
    n ← i - 1
    cerrar(t)
    leer(cód)
    mientras no último_dato(cód) hacer
        primero ← 1
        último ← n
        encontrado ← falso
        mientras primero <= último y no encontrado hacer
            central ← (primero + último) div 2
            si a[central].cód = cód entonces
                encontrado ← verdad
            si_no
                si a[central].cód > cód entonces
                    último ← central - 1
                si_no
                    primero ← central + 1
                fin_si
            fin_si
        fin_mientras
        si encontrado entonces
            leer(f, r, a[central].posic)
            llamar_a escribir_reg(r)
            escribir('¿Desea borrarlo? (s/n) ')
            leer(resp)
            si (resp='S') o (resp='s') entonces
                desde i ← central hasta n - 1 hacer

```

```

        a[i] ← a[i+1]
    fin_desde
    n ← n - 1
    escribir('El registro ha sido borrado')
fin_si
si_no
    escribir('No se encontró')
fin_si
leer(cód)
fin_mientras
crear(t, 'Indice.dat')
abrir(t, 'e', 'Indice.dat')
desde i ← 1 hasta n hacer
    escribir(t, a[i].cód, a[i].posic)
fin_desde
cerrar(t)
cerrar(f)
fin

```

- 9.14.** Partiendo del archivo indexado LIBROS.DAT, cuyos registros tienen la siguiente estructura: *CÓDIGO* (cadena), *TÍTULO*, *AUTOR* y *PRECIO*; escribir un algoritmo que permita modificar la información almacenada en cualquiera de los campos de un determinado registro, incluido el campo *código*, que es la clave y no puede repetirse.

Análisis del problema

Para modificar el campo clave se elimina el código antiguo de la tabla de índices y se inserta el nuevo en el lugar adecuado para que se mantenga ordenada. Si no se modifica el campo clave no se necesitará realizar cambios en la tabla.

Todo registro que haya sido modificado se almacenará, después de sufrir los cambios, en la misma posición en la que se encontraba antes en el fichero de datos.

Diseño del algoritmo

```

algoritmo Ejercicio_9_14
const
    Máx = 100
tipo
    registro : reg
    cadena: cód
    cadena: título
    cadena: autor
    entero: precio
fin_registro
registro : rínd
cadena: cód
entero: nreg
fin_registro
array[1..Máx] de rínd: arrínd
archivo_s de rínd      : archi
archivo_d de reg       : archd

```

```

var
    arrínd      : índice
    archd       : fich
    entero     : n, posí, nreg
    lógico    : encontrado
    carácter   : resp
    reg         : rdatos
    cadena     : códigoant, cód
inicio
    abrir(fich,'l/e','Libros.dat')
    cargar_índice(índice, n)
    si n <> 0 entonces
        repetir
            escribir('Deme código')
            leer(códigoant)
            búsqued_binaria(índice,n,códigoant,posí, encontrado)
            si encontrado entonces
                nreg ← índice[posí].nreg
                leer(fich,nreg, rdatos)
                escribir('Escriba los nuevos datos o pulse
                    <Return> para aceptar los actuales')
                escribir(rdatos.cód,'?')
                leer(cód)
                si cód <> '' entonces
                    rdatos.cód ← cód
                fin_si
                si códigoant <> rdatos.cód entonces
                    baja_índice(índice,n, posí)
                    búsqued_binaria(índice,n,rdatos.cód, posí,encontrado)
                    mientras encontrado hacer
                        escribir('Datos no válidos')
                        escribir('Introduzca nuevos datos o pulse <Return> para
                            aceptar los actuales')
                        escribir(rdatos.cód,'?')
                        leer(cód)
                        si cód <> '' entonces
                            rdatos.cód ← cód
                        fin_si
                        búsqued_binaria(índice,n,rdatos.cód, posí,encontrado)
                    fin_mientras
                    alta_índice(índice,n, posí,rdatos.cód,nreg)
                fin_si
                cambiar(rdatos)
                escribir(fich,rdatos,nreg)
            si_no
                escribir('No existe')
            fin_si
            escribir('¿Otra modificación? (s/n)')
            leer(resp)
hasta_que (resp = 'n') o (resp='N')

```

```

        guardar_índice(índice,n)
    si_no
        escribir('Vacío')
    fin_si
    cerrar(fich)
fin

procedimiento cambiar(E/S reg: rdatos)
var
    cadena: título, autor, precio
inicio
    escribir(rdatos.título,'?')
    leer(título)
    si título <> '' entonces
        rdatos.título← título
    fin_si
    escribir(rdatos.autor,'?')
    leer(autor)
    si autor <> '' entonces
        rdatos.autor← autor
    fin_si
    escribir(rdatos.precio,'?')
    leer(precio)
    si precio <> '' entonces
        rdatos.precio← valor(precio)
    fin_si
fin_procedimiento

procedimiento búsqueda_binaria(E arrínd: índice E entero: n,
                                E cadena: codbusc; S entero: posi; S lógico: encontrado)
var
    entero: último, central
inicio
    encontrado← falso
    posi← 1
    último← n
    mientras (posi <= último) y no encontrado hacer
        central← (posi + último) div 2
        si codbusc = índice[central].cód entonces
            encontrado← verdad
            posi← central
        si_no
            si codbusc < índice[central].cód entonces
                último← central - 1
            si_no
                posi← central + 1
            fin_si
        fin_si
    fin_mientras
fin_procedimiento

```

```

procedimiento alta_índice(E/S arrínd: índice E/S entero: n;
                           E entero: posí; E cadena: codbusc; E entero: núm)
var
    entero: i
inicio
    desde i ← n hasta posí decremento 1 hacer
        índice[i+1] ← índice[i]
    fin_desde
    índice[posí].cód ← codbusc
    índice[posí].nreg ← núm
    n ← n + 1
fin_procedimiento

procedimiento baja_índice(E/S arrínd: índice; E/S entero: n;
                           E entero: posí)
var
    entero: i
inicio
    desde i ← posí hasta n - 1 hacer
        índice[i] ← índice[i+1]
    fin_desde
    n ← n - 1
fin_procedimiento

procedimiento cargar_índice(S arrínd: índice; E/S entero: n)
var
    entero: i
    archi: ind
inicio
    abrir(ind,1,'Libind.dat')
    i ← 1
    mientras no fda(ind) hacer
        leer(ind, índice[i].cód, índice[i].nreg)
        i ← i + 1
    fin_mientras
    n ← i - 1
    cerrar(ind)
fin_procedimiento

procedimiento guardar_índice(E arrínd: índice; E entero: n)
var
    entero: i
inicio
    crear(ind,'Libind.dat')
    abrir(ind,'e','Libind.dat')
    desde i ← 1 hasta n hacer
        escribir(ind,índice[i].cód, índice[i].nreg)
    fin_desde
    cerrar(ind)
fin_procedimiento

```

- 9.15.** Una farmacia mantiene su stock de medicamentos en un archivo secuencial que tiene los campos código, nombre, precio, IVA, stock , stock-máx, stock-mín y proveedor. El archivo no presenta registros repetidos. Por un aumento del nivel de facturación desea convertir su archivo en un archivo indexado cuya clave principal será el código del producto.

Análisis del problema

Vamos a suponer que el número máximo de artículos es de 1.500. Al no haber productos repetidos en el archivo original, el proceso simplemente consistirá en recorrer éste e ir dando altas a cada uno de los registros. Primero inicializaremos la tabla de índices: en cada uno de los elementos de la tabla introduciremos en el campo número de registro la propia posición. Además, puesto que la tabla ahora está vacía, inicializaremos n (el número de elementos metidos en la tabla) a 0.

Para dar un alta a un registro primero veremos si hay sitio —si n es menor que el número máximo de productos—. De ser así incrementaremos n y obtendremos un número de registro válido y lo insertaremos en la tabla en la posición correspondiente a la clave del producto que deseamos indexar. Por último grabaremos el registro en el área de datos (archivo directo).

Al terminar grabaremos la tabla de índices en un archivo secuencial. En dicho archivo secuencial habrá un registro de cabecera que guarda el número de registros del índice.

Diseño del algoritmo

```

algoritmo ejercicio_9_15
const
    MáxReg = 1500
tipos
    registro : R_Farmacia
    cadena : código
    cadena : nombre
    entero : precio
    real : IVA
    entero : stock
    entero : stock_máx
    entero : stock_mín
    cadena : proveedor
fin_registro
archivo_d de R_Farmacia : A_Farmacia
registro : R_Índice registro
    cadena : clave
    entero : NúmReg
fin_registro
array [0..MáxReg] de R_Índice : Tabla_Índice
archivo_s de R_Farmacia : A_Secuencial
var
    A_Farmacia : AFar
    A_Secuencial : AS
    R_Farmacia : RFar
    R_Secuencial : RS
    Tabla_Índice : Tabla
    entero : n
    lógico : lleno
inicio
    crear(AFar, 'FARMACIA')

```

```

abrir(AFar, 1/e, 'FARMACIA')
abrir(AS, 1, 'SECUENCIAL')
CrearIndexado(Índice,n)
leer(AS,RS)
mientras no FDA(AS) hacer
    RFar ← RS
    AltaIndexado(AFar,RFar,Tabla,n,lleno)
    leer(AS,RS)
fin_mientras
GrabarTabla(Indice,n)
cerrar(AFar, AS)
fin

procedimiento CrearIndexado(s Tabla_Índice : Índice; s entero : n)
var
    entero : i
inicio
    n ← 0
    desde i ← 1 hasta MáxReg hacer
        Indice[i].NúmReg ← i
    fin_para
fin_procedimiento

procedimiento AltaIndexado(E/S A_Farmacia : A; E R_Farmacia : R;
                           E/S Tabla_Índices : t;
                           E/S entero : n; S lógico : lleno)
var
    entero : j
inicio
    si n = MáxReg entonces
        lleno ← verdad
    si_no
        lleno ← falso
        n ← n + 1
        t[0].código ← r.código
        t[0].NúmReg ← t[n].NúmReg
        j ← n - 1
        mientras t[j].código > t[0].código hacer
            t[j+1] ← t[j]
            j ← j - 1
        fin_mientras
        t[j+1] ← t[0]
        escribir(A,r,t[0].NúmReg)
    fin_si
fin_procedimiento

procedimiento GrabarTabla(E Índice : t; E entero : n)
var
    entero : i
    registro : R_Cabecera

```

```

    entero: Último
  fin_registro
  archivo_s de R_Índice, R_Cabecer : A_Índice
inicio
  crear(A_Índice, 'ÍNDICE')
  abrir(A_Índice, e, 'ÍNDICE')
  R_Cabecera.Último ← n
  escribir(A_Índice, R_Cabecera)
  desde i ← 1 hasta MáxReg hacer
    escribir(A_Índice, t[i])
  fin_registro
  cerrar(A_Índice)
fin_procedimiento

```

- 9.16.** Periódicamente, la farmacia del ejercicio anterior, recibe de los laboratorios, junto con los pedidos pendientes, un archivo secuencial en el que se incluyen el código del artículo, el nombre, el precio, el IVA, las unidades mandadas, el stock máximo y mínimo del artículo y el proveedor. Además un campo tipo se encarga de informar del tipo de movimiento que se ha realizado. Este campo puede ser A, B o M, según sea respectivamente un alta, una baja del producto o una modificación del mismo. La modificación se hará teniendo en cuenta las siguientes consideraciones:

- Si el precio es mayor que el precio anterior, se modificará el campo precio del archivo indexado.
- Si el stock máximo y el stock mínimo es <> de 0, se modificarán dichos campos y las unidades mandadas.
- Si ambos campos son 0, se entenderá que las unidades mandadas deben acumularse al campo stock del archivo indexado.

Además puede darse el caso de que el registro del archivo secuencial sea una incidencia, en cuyo caso se dará un alta en un archivo de incidencias cuya estructura será igual que la del archivo de movimientos. Se considerarán incidencias los registros de tipo A que existan en el indexado y los de tipo B o M que no existan. Así mismo será también una incidencia los registros que no quepan en el archivo indexado.

Diseño del algoritmo

```

algoritmo ejercicio_9_16
const
  MáxReg = 1500
tipos
  registro : R_Farmacia
  cadena : código
  cadena : nombre
  entero : precio
  real : IVA
  entero : stock
  entero : stock-máx
  entero : stock-mín
  cadena : proveedor
fin_registro
archivo_d de R_Farmacia : A_Farmacia
registro : R_Índice registro
  cadena : clave

```

```

        entero : NúmReg
fin_registro
array [0..MáxReg] de R_Índice : Tabla_Índice
registro : R-Mov
cadena : código
cadena : nombre
entero : precio
real : IVA
entero : unidades
entero : stock-máx
entero : stock-mín
cadena : proveedor
fin_registro
archivo_d de R_Farmacia : A_Farmacia
archivo_s de R_Mov : A_Mov, A_Inc
var
A_Farmacia : A_Far
A_Mov : AM
A_Mov : AInc
R_Farmacia : RFar
R_Mov : RM, RInc
Tabla_Índice : Índice
entero : n, p
lógico : lleno
inicio
abrir(AFar, 'l/e', 'FARMACIA')
abrir(AM, 'l', 'MOVIMIENTOS')
crear(AInc, 'INCIDENCIAS')
abrir(AInc, 'e', 'INCIDENCIAS')
CargarTabla(Índice,n)
leer(AM,RM)
mientras no FDA(AS) hacer
    RFar ← RM
    p ← BuscarIndexado(Índice,RFar,n)
    si p = 0 entonces
        si RM.tipo <> 'A' entonces
            RInc ← RM
            escribir(AInc,RInc)
        si_no
            AltaIndexado(AFar,RFar,Tabla,n, lleno)
            si lleno entonces
                RInc ← RM
                escribir(AInc,RInc)
            fin_si
        fin_si
    si_no
        si RM.tipo = 'A' entonces
            RInc ← RM
            escribir(AInc,RInc)
    si_no

```

```

    si RM.tipo = 'B' entonces
        BajaIndexado(Tabla,p,n)
    si_no
        leer(AFar,RFar, Indice[p].NúmReg)
        si RFar.precio < RM.precio entonces
            RFar.precio ← RM.precio
        fin_si
        si RM.stock_Mín <> 0 y RM.stock_Máx <> 0 entonces
            RFar.stock_mín ← RM.stock_mín
            RFar.stock_máx ← RM.stock_máx
        si_no
            si RM.unidades <> 0 entonces
                RFar.stock ← RM.unidades
            fin_si
        fin_si
        escribir(AFar,RFar,Indice[p].NúmReg)
    fin_si
fin_si
leer(AM,RM)
fin_mientras
GrabarTabla(Indice,n)
cerrar(AFar, AM, AInc)
fin

procedimiento CargarTabla(S : índice t; S entero : n)
var
    entero : i
    registro : R-Cabecera
    entero: Último
fin_registro
archivo_s de R_Índice, R_Cabecer : A_Índice
inicio
    abrir(A_Índice, 'l', 'ÍNDICE')
    leer(A_Índice, R_Cabecera)
    n ← R_Cabecera.Último
    leer(A_Índice, R_Índice)
    i ← 0
    mientras no FDA(A_Índice) hacer
        i ← i + 1
        t[i] ← R_Índice
        escribir(A_Índice, R_Índice)
    fin_registro
    cerrar(A_Índice)
fin_procedimiento

entero función BuscarIndexado(E Tabla_Índice : t;
                           E R_Índice : r; E entero : n)
// función de búsqueda binaria
fin_función

```

```
procedimiento BajaIndexado(E/S Tabla_Índice : t;
                           E/S R_Índice : r; E entero : p,n)
var
    entero : i
inicio
    t[0] ← t[p]
    desde i ← p hasta n - 1 hacer
        t[i] ← t[i + 1]
    fin_para
    t[n].NúmReg ← t[0].NúmReg
    n ← n - 1
fin_procedimiento
```


10

ORDENACIÓN, BÚSQUEDA E INTERCALACIÓN INTERNA

@Ug'Wa di hUXcfUg'Ya d'YUb'i bU[fUb'dUFhY'XY'gi'h]Ya dc'Yb'cdYfUWcbYg'XY'V•gei YXUmWb[Z]! WWDZii'cf[UbnW]XY'cg'XUhcg'Yb'U[•b'cfXYb'c'gYWYbWU'YgdYWZ]WU'Hyb]YbXe'Yghc'Yb'W'Yb! hU'miXUc'e iY'Yg'ZfYWYbhY'eiY'i b'dfc[fUaUhfUVU'Y'Wb'[fUbXYg'Wb]XUXYg'XY'XUhcg'Ua UW! bUXcg'Yb'UffUng'fUFfY['cg]ZfYgj'hU]a dfYgWbX]VY'WbcWf' 'cg'Xlj'Yfgcg'a fhcXcg'XY'cfXYbW]D miV•gei YXU'Yb'UffUng'eiY'gY'Yl'd]Wb'Yb'YghY'Wd]hi`c'"ChfUcdYfUW]D]bhYfYgUbhY'eiY'hUa V]fb gY'Wa YbhUfz'Yg'U]bhYfW]W]D'c'aYnWU'XY'UffUng'cfXYbUXcg'dUFUcVhYbYf'chfc'UffUnicfXY' bUXc"

10.1. BÚSQUEDA

La búsqueda es una de las operaciones más importantes en el procesamiento de la información, y permite la recuperación de datos previamente almacenados. Cuando el almacenamiento se encuentra en memoria principal la búsqueda se califica de interna.

En este capítulo se consideran los datos almacenados en arrays y se analiza el modo de realizar operaciones de ordenación, fusión o búsqueda en ellos.

10.1.1. Búsqueda secuencial

La búsqueda secuencial consiste en recorrer y examinar cada uno de los elementos hasta alcanzar el final de la lista de datos. Si en algún lugar de la lista se encontrara el elemento buscado, el algoritmo deberá informarnos sobre la o las posiciones donde ha sido localizado. Este algoritmo es posible optimizarlo cuando:

- Únicamente se desea obtener la posición donde se localiza por primera vez al elemento buscado, mediante el empleo de una variable lógica que evite seguir buscando una vez que el dato ha sido encontrado.
- El array se encuentra ordenado. Considerando que el dato no está y la búsqueda ha terminado cuando el elemento a encontrar sea menor que el elemento en curso en un array ordenado ascendente, o cuando sea mayor si se busca en un array ordenado descendente.

10.1.2. Búsqueda binaria

Este método de búsqueda requiere que la lista esté ordenada y para su comprensión es necesario definir el concepto de lista ordenada:

Dado un vector X y dos índices i y j que permiten recorrerlo, se dirá que está ordenado ascendente si para todo $i < j$ se cumple siempre que $X[i] \leq X[j]$. El vector estará ordenado descendente si cuando $i < j$ se cumple siempre, para todos sus elementos, que $X[i] \geq X[j]$

La búsqueda binaria consiste en comparar el elemento buscado con el que ocupa en la lista la posición central y, según sea mayor o menor que el central y la lista se encuentre clasificada en orden ascendente o descendente, repetir la operación considerando una sublista formada por los elementos situados entre el que ocupa la posición central + 1 y el último, ambos inclusive, o por los que se encuentran entre el primero y el colocado en central – 1, también ambos inclusive. El proceso finalizará cuando el dato sea encontrado o la sublista de búsqueda se quede sin elementos.

10.1.3. Búsqueda por transformación de claves

Este método utiliza una función para convertir la clave identificativa de un elemento en un subíndice que indique la posición en el array donde debiera encontrarse el elemento. El caso más sencillo sería aquel en que la clave pudiera ser utilizada directamente como índice.

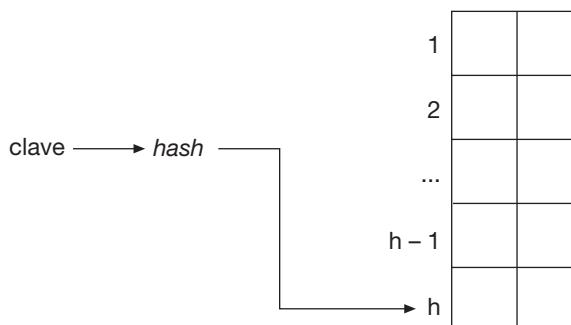
```
entero función hash(E entero: clave)
  inicio
    devolver(clave)
  fin_función
```

10.1.3.1. Funciones de conversión de clave

Cuando el rango en el que pueden oscilar los valores de las claves es muy grande en comparación con la cantidad de datos que deseamos almacenar, deberemos utilizar una función que transforme las claves a números en un determinado rango y permita la ubicación y posterior localización de los elementos a través de su clave sin que el array exceda las dimensiones apropiadas.

La conversión o correspondencia de clave a dirección puede ser realizada por cualquier función que transforme las claves en un rango de direcciones, pero es deseable que cumpla las siguientes condiciones:

- Distribuir las claves uniformemente entre las direcciones para producir pocos sinónimos.
- No ser una función compleja que pueda ralentizar los cálculos.



Algunas de las funciones que resultan más eficientes son las de restas sucesivas, aritmética modular, mitad del cuadrado, truncamiento o plegamiento.

Restas sucesivas

Esta función se emplea con claves numéricas entre las que existen huecos de tamaño conocido, obteniéndose direcciones consecutivas. Por ejemplo,

```
Clave = N°matrícula = año(aa) + curso + n°_de_alumno
```

donde cada curso tiene un máximo de 700 alumnos.

Curso	Clave	Dirección
1º	961001	961001 menos 961000
	961002	961002 menos 961000
	...	
2º	961700	961700 menos 961000
	962001	962001 menos 961000 más 700
	962002	962002 menos 961000 más 700

Aritmética modular

Consiste en efectuar la división entera de la clave por el tamaño del rango del índice y tomar el resto. Por ejemplo, si la clave es numérica y el tamaño del rango del índice es N se podría crear la siguiente función:

```
entero función hash(E entero:clave)
  inicio
    devolver(clave mod N + 1)
  fin_función
```

Para lograr una mayor uniformidad en la distribución es conveniente que N sea un número primo.

Mitad del cuadrado

Consiste en elevar al cuadrado la clave y tomar los dígitos centrales como dirección. El número de dígitos a tomar queda determinado por el rango del índice.

Truncamiento

Ignorar parte de la clave y utilizar la parte restante directamente como índice. Si las claves, por ejemplo, son enteros de ocho dígitos y la tabla tiene mil posiciones, entonces el primero, segundo y quinto dígitos desde la derecha pueden formar la función de conversión. Por ejemplo, 72588495 se convierte en 895.

Plegamiento

La técnica del plegamiento consiste en la división de la clave en diferentes partes y su combinación en un modo conveniente (a menudo utilizando suma o multiplicación) para obtener el índice. La clave x se divide en varias partes $x_1, x_2 \dots x_n$, donde cada una, con la única posible excepción de la última, tiene el mismo número de dígitos que la dirección especificada. A continuación, se suman todas

las partes. En esta operación se desprecian los dígitos más significativos que se obtengan de arrastre o acarreo. Por ejemplo, si la clave fuera 38162594 y el número de direcciones 100,

$$381 + 625 + 94 = 1100$$

la función devolverá 100 como la dirección correspondiente a dicho elemento.

Las claves no tienen por qué ser numéricas y en ellas podrán aparecer letras. En general cuando aparecen letras en las claves se suele asociar a cada letra un entero. Recuerde que existe un valor numérico entero asociado a cada carácter, su código ASCII.

10.1.3.2. Resolución de colisiones

Una función de conversión, *hash* (clave), puede devolver para dos claves diferentes la misma dirección. Esta situación se denomina colisión y se deben encontrar métodos para su correcta resolución.

Primera solución

Cuando la dirección que le corresponde a un nuevo elemento ya está ocupada se le asigna el primer hueco libre a partir de dicha posición.

```
...
leer(clave)
p ← hash(clave)
cont ← 1
mientras no libre(A[p].clave) y (cont < N) hacer
    p ← p mod N + 1
    cont ← cont + 1
fin_mientras
si libre(A[p].clave) entonces
    llamar_a leer_registro(A[p])
si_no
    escribir('Array lleno')
fin_si
```

Segunda solución

Reservar un espacio al final del array para el almacenamiento de las colisiones. Por ejemplo, si aplicáramos como función hash la aritmética modular, para reservar dicho espacio bastaría con efectuar la división por un número inferior al tamaño del rango del índice.

```
tipo
array[1..15] de tipo_registro: arr
var
arr: A
.....
entero función hash(E entero: clave)
inicio
devolver(clave mod 11 + 1)
fin_función
```

Cuando se produzcan colisiones podremos colocar los elementos colisionados secuencialmente en A[12], A[13], A[14] y A[15].

Tercera solución

Permitir que cada elemento del array sea un puntero que apunta al elemento del principio de la lista de elementos. Es decir crear una lista enlazada o encadenada a partir de cada elemento del array.

10.2. ORDENACIÓN

La clasificación de los datos consiste en organizar los mismos en un orden creciente o decreciente por el contenido de un determinado campo. Podrá ser:

- **Externa.** Los datos están en un dispositivo de almacenamiento externo.
- **Interna.** De arrays; los datos se encuentran en memoria y permite el acceso aleatorio o directo; por ambas razones resulta siempre más rápida que la externa. Es la que veremos en este capítulo, dejando la externa para otros posteriores.

10.2.1. Ordenación interna

Los métodos de ordenación interna se aplican a arrays unidimensionales, pero su uso puede extenderse a otro tipo de arrays, bidimensionales, tridimensionales, etc., considerando el proceso de ordenación con respecto a filas, columnas, páginas, etc. Los métodos de clasificación interna más usuales son: Selección, Burbuja, Inserción, Inserción binaria, Shell, Ordenación Rápida (*Quick Sort*).

10.2.1.1. Selección

Este método se basa en buscar el elemento menor del vector y colocarlo en primera posición. Luego se busca el segundo elemento más pequeño y se coloca en la segunda posición, y así sucesivamente.

Por ejemplo dada la siguiente lista 4 1 5 -3 8, se compara a [1] con los siguientes elementos:

<u>4</u>	1	5	-3	8
<u>1</u>	4	5	-3	8
<u>1</u>	4	5	-3	8
<u>-3</u>	4	5	1	8

con lo que el elemento menor queda seleccionado en a [1] y se compara a [2] con los siguientes elementos:

<u>4</u>	5	1	8
<u>4</u>	5	1	8
<u>1</u>	5	4	8

con los que el elemento menor queda seleccionado en a [2]. Se compara el elemento a [3] con los siguientes:

<u>5</u>	4	8
<u>4</u>	5	8

con lo que el elemento menor queda en a [3]. se compara el elemento a [4] con los siguientes:

<u>5</u>	8
<u>5</u>	8

con lo que la la lista quedaría -3 1 4 5 8.

También se podría buscar el elemento mayor y efectuar la clasificación en orden descendente.

10.2.1.2. Burbuja

El algoritmo de clasificación de intercambio o de la burbuja se basa en el principio de comparar pares de elementos adyacentes e intercambiarlos entre sí hasta que estén todos ordenados. Si partimos de la misma lista que en el método anterior, 4 1 5 -3 8:

4	1	5	-3	8
1	4	5	-3	8
1	4	5	-3	8
1	4	-3	5	8

quedando el 8 al final de la lista:

1	4	-3	5	[8]
1	4	-3	5	
1	-3	4	5	

quedando el 5 en penúltima posición:

1	-3	4	[5]
-3	1	4	

quedando el 4 en tercera posición:

-3	1	[4]
----	---	-----

quedando el uno en la segunda posición:

<u>-3</u>	[1]
-----------	-----

con lo que la lista resultante sería -3 1 4 5 8.

10.2.1.3. Inserción directa

Este método consiste en insertar un elemento en una parte ya ordenada del mismo, en el lugar adecuado para que no se pierda la ordenación y continuar así con los elementos restantes. El lugar de inserción se averigua mediante búsqueda secuencial en la sublista ordenada. Dada la lista 4 1 5 -3 8, inicialmente se considera una sublista con el 4, que como tiene un único elemento se encuentra ordenada. Se van tomando los restantes elementos, desde el que se encuentra en la posición 2 hasta el que está en la 5, y cuando se encuentra en la sublista la posición adecuada se inserta el elemento.

Los elementos de la lista se desplazan en caso necesario y la lista va tomando los siguientes valores:

4	1	5	-3	8
1	4	5	-3	8
1	4	5	-3	8
-3	1	4	5	8
-3	1	4	5	8

10.2.1.4. Inserción binaria

Este método es similar al de inserción directa, pero sustituyendo la búsqueda secuencial para encontrar el lugar de inserción por una búsqueda binaria, puesto que la sublista donde dicha búsqueda se efectúa se encuentra ordenada.

10.2.1.5. Shell

Este método se basa en realizar comparaciones entre elementos no consecutivos, separados por distancias o intervalos mayores que 1. Estas distancias sufrirán sucesivos decrementos. Es un método elaborado que resulta más eficiente cuando las listas son grandes.

Considere el siguiente array $a = [4 \quad 1 \quad 5 \quad -3 \quad 8]$ de 5 elementos. En una primera pasada las comparaciones e intercambios se realizarán entre elementos separados por un salto o intervalo de $5/2$. Se comparan pues $a[j]$ con $a[j+salto]$ (j toma valores de 1 a $N - N \text{ div } 2$), es decir se comparan $a[1]$ con $a[3]$, $a[2]$ con $a[4]$ (intercambio) y $a[3]$ con $a[5]$.

Vector a
$a[1] \quad a[2] \quad a[3] \quad a[4] \quad a[5]$
4 -3 5 1 8

A continuación el salto o distancia entre los elementos a comparar se convierte en la mitad y se efectúa un nuevo recorrido con las correspondientes comparaciones e intercambios en caso necesario. Este proceso se repite mientras el salto distancia o intervalo sea mayor o igual a la unidad.

$a[1] \quad a[2] \quad a[3] \quad a[4] \quad a[5]$
-3 4 5 1 8
-3 4 1 5 8

Hay que tener en cuenta que cuando se produce un intercambio debe comprobarse que dicho intercambio no ha estropeado la ordenación efectuada anteriormente. Por ejemplo, el intercambio entre $a[3]$ y $a[4]$ obliga a verificar si $a[2]$ y $a[3]$ se encuentran ordenados, esto produce un nuevo intercambio y hace necesario comparar nuevamente $a[1]$ y $a[2]$ para descubrir que la colocación es correcta en este caso.

$a[1] \quad a[2] \quad a[3] \quad a[4] \quad a[5]$
-3 1 4 5 8

10.2.1.6. Ordenación rápida

El método consiste en:

- Dividir el array en dos particiones, una con todos los elementos menores a un cierto valor específico y otra con todos los mayores que ese valor, uno cualquiera, tomado arbitrariamente del vector, al que se denomina elemento pivote.
- Tratar, análogamente a como se expuso en el primer apartado, una de estas particiones, la más pequeña, y guardar los límites de la que no se va a tratar de forma inmediata. Las particiones pendientes han de ser recuperadas, para ser tratadas, en orden inverso a como fueron generadas.

```
procedimiento ordenación_rápida_iterativo (E/S arr: a)
// observe que es un método iterativo
// en el capítulo de recursividad se verá la resolución recursiva
var
    entero: inf, sup, izq, der
    real: pivote, auxi
    // suponemos que el array a ordenar almacena números
    // reales
    entero: cima
    // cima.- variable que señala el último elemento
    // colocado en el array pila. El concepto de pila se verá más adelante
    array_de_registros: pila
    // pila.- array donde se irán guardando los límites
```

```

    //de las particiones a tratar posteriormente
inicio
    cima ← 1
    pila[cima].lim_izq ← 1
    pila[cima].lim_der ← 100      // límites del array inicial
repetir
    izq ← pila[cima].lim_izq // pila es un array de registros
    der ← pila[cima].lim_der
    cima ← cima - 1
repetir
    inf ← izq
    sup ← der
    pivote ← a[(izq + der)div 2]
repetir
    mientras a[inf] < pivote hacer
        inf ← inf + 1
    fin_mientras
    mientras a[sup] > pivote hacer
        sup ← sup - 1
    fin_mientras
    Si inf <= sup entonces
        auxi ← a[inf]
        a[inf] ← a[sup]
        a[sup] ← auxi
    fin_si
    hasta_que inf > sup
    si sup - izq < der - inf entonces // almacenamos los límites de
                                                // la partición más larga
        si inf < der entonces
            cima ← cima + 1
            pila[cima].lim_izq ← inf
            pila[cima].lim_der ← der
        fin_si
        der ← sup
    si_no
        si izq < sup entonces
            cima ← cima + 1
            pila[cima].lim_izq ← izq
            pila[cima].lim_der ← sup
        fin_si
        izq ← inf
    fin_si
    hasta_que izq >= der
    hasta_que cima = 0
fin_procedimiento

```

10.3. INTERCALACIÓN

Analizaremos la intercalación o mezcla de dos vectores ordenados para producir un nuevo vector también ordenado. Supongamos los vectores A y B con M y N elementos respectivamente. El nuevo vector, C, tendrá M + N elementos.

El algoritmo seleccionará el más pequeño entre los elementos en curso de los vectores A y B, situándolo en C. Para poder realizar las comparaciones sucesivas y avanzar en el nuevo vector C, necesitaremos: dos índices para los vectores A y B —por ejemplo, i y j— y un tercer índice para el vector C. Entonces nos referiremos al elemento i en la lista A, al elemento j en la lista B y al elemento k en la lista C. Se debe efectuar repetitivamente:

```

si elemento i de A es menor que elemento j de B entonces
    transferir elemento i de A a C
    avanzar i (incrementar en 1)
si_no
    transferir elemento j de B a C
    avanzar j
fin_si
```

Cuando uno de los vectores se termine de situar en C no se requerirá seguir efectuando comparaciones. La operación siguiente será copiar en C los elementos que restan del otro vector. El algoritmo total resultante de la intercalación de dos vectores A y B ordenados en uno C es:

```

algoritmo intercalación
.....
inicio
    llamar_a leerarrays(A, B)      // A, B vectores ordenados
                                    // de M y N elementos
    i ← 1
    j ← 1
    k ← 0
    mientras (i <= M) y (j <= N) hacer
        //seleccionar siguiente elemento de A o B y añadirlo a C
        k ← k+1
        si A[i] < B[j] entonces
            C[k] ← A[i]
            i ← i+1
        si_no
            C[k] ← B[j]
            j ← j+1
        fin_si
    fin_mientras
    //copiar el vector restante
    si i <= M entonces
        desde r ← i hasta M hacer
            k ← k+1
            C[k] ← A[r]
        fin_desde
    si_no
        desde r ← j hasta N hacer
            k ← k+1
            C[k] ← B[r]
        fin_desde
    fin_si
    escribir(C)    //vector clasificado
fin
```

10.4. EJERCICIOS RESUELTOS

- 10.1.** Algoritmo que permita la introducción de 10 números enteros en un array y, a través de un menú, la selección de uno de los siguientes métodos de ordenación: Selección, Burbuja, Inserción, Inserción binaria y Shell. Terminará con la presentación por pantalla del array clasificado.

Análisis del problema

Se trata únicamente de mostrar los procedimientos necesarios para la realización de los distintos métodos de ordenación.

Diseño del algoritmo

```

algoritmo ej_10_01
const
    N = 10
tipo
    array[1..N] de entero: arr
var
    arr    : a
    entero : i, op
inicio
    escribir('Dame diez números')
    desde i ← 1 hasta N hacer
        leer(a[i])
    fin_desde
    escribir('Elige método de ordenación:')
    escribir('1.-Selección  2.-Burbuja  3.-Inserción')
    escribir('4.-Inserción binaria  5.-Shell')
    leer(op)
    según_sea op hacer
        1 : ordenarselec(a)
        2 : ordenarburb(a)
        3 : ordenarinserc(a)
        4 : ordenarinsbin(a)
        5 : ordenarshell(a)
    fin_según
    desde i ← 1 hasta N hacer
        escribir(a[i], ' ')
    fin_desde
fin

procedimiento ordenarselec(E/S arr: a)
var
    entero: i, j
    entero: auxi
inicio
    desde i ← 1 hasta N - 1 hacer
        desde j ← i + 1 hasta N hacer
            si a[i] > a[j] entonces
                Auxi ← a[i]
                a[i] ← a[j]
                a[j] ← Auxi
            fin_si
        fin_desde
    fin_desde
fin

```

```
a[i] ← a[j]
a[j] ← Auxi
fin_si
fin_desde
fin_desde
fin_procedimiento

procedimiento ordenararburb(E/S arr: a)
var
    entero: i, j
    entero: auxi
inicio
    desde i ← 1 hasta N - 1 hacer
        desde j ← 1 hasta N - i hacer
            si a[j] > a[j+1] entonces
                Auxi ← a[j]
                a[j] ← a[j+1]
                a[j+1] ← Auxi
            fin_si
        fin_desde
    fin_desde
fin_procedimiento

procedimiento ordenarinserc(E/S arr:a)
var
    entero: k,i
    entero: auxi
    lógico: encsitio
inicio
    desde k ← 2 hasta N hacer
        auxi ← a[k]
        i ← k-1
        encsitio ← falso
        mientras (no encsitio) y(i >= 1) hacer
            si auxi < a[i] entonces
                a[i+1] ← a[i]
                i ← i - 1
            si_no
                encsitio ← verdad
            fin_si
        fin_mientras
        a[i+1] ← auxi
    fin_desde
fin_procedimiento

procedimientos ordenarinsbin(E/S arr: a)
var
    entero: k, j, prim, últ, centr
    entero: auxi
```

```

inicio
    desde k ← 2 hasta N hacer
        auxi ← a[k]
        prim ← 1
        últ ← k - 1
        mientras prim <= últ hacer
            centr ← (prim + últ) div 2
            si auxi < a[centr] entonces
                últ ← centr - 1
            si_no
                prim ← centr + 1
            fin_si
        fin_mientras
        desde j ← k - 1 hasta prim decremento 1 hacer
            a[j+1] ← a[j]
        fin_desde
        a[prim] ← auxi
    fin_desde
fin_procedimiento

procedimiento ordenarshell(E/S arr: a)
var
    entero: salto, j, i, k
    entero: auxi
inicio
    salto ← N div 2
    mientras salto > 0 hacer
        desde i ← (salto+1) hasta N hacer
            j ← i - salto
            mientras j > 0 hacer
                k ← j + salto
                si a[j] <= a[k] entonces
                    j ← 0
                si_no
                    auxi ← a[j]
                    a[j] ← a[k]
                    a[k] ← auxi
                    j ← j - salto
                fin_si
            fin_mientras
        fin_desde
        salto _ salto div 2
    fin_mientras
fin_procedimiento

```

- 10.2.** Realizar un procedimiento que nos permita ordenar por fechas y de mayor a menor un vector de N elementos ($N \leq 40$). Cada elemento del vector es un registro, con los campos, de tipo entero, día, mes, año y número de contrato. Damos por supuesto que la introducción de datos fue correcta, pudiendo existir diversos contratos con la misma fecha, pero no números de contrato repetidos.

Análisis del problema

Se creará una función de tipo lógico, EsMenor, la cual decidirá que una fecha es menor que otra cuando su año es menor al de la otra o, a igualdad de año, su mes es menor o, a igualdad de año y mes entre ambas, es menor su día.

Desde el procedimiento de ordenación, cuando se necesite comparar las fechas, se llama a la función y, si devuelve verdad, se intercambian los registros completos.

Diseño del algoritmo

```
algoritmo ej_10_02
tipo
    registro: elemento
        entero: día
        entero: mes
        entero: año
        entero: ncontrato
    fin
    array[1..40] de elemento: arr
var
    arr      : a
    entero   : n
inicio
    //Se necesitará algún procedimiento que realice la
    //introducción de datos en el vector
    .....
    ordenar_elementos(a,n)
    .....
    .....
fin

lógico función esmenor(E elemento: elemento1,elemento2)
inicio
    si (elemento1.año<elemento2.año) o
        (elemento1.año=elemento2.año) y (elemento1.mes<elemento2.mes) o
        (elemento1.año=elemento2.año) y (elemento1.mes=elemento2.mes) y
        (elemento1.día<elemento2.día) entonces
            devolver(verdad)
        si_no
            devolver(falso)
        fin_si
    fin_función

procedimiento ordenar_elementos(E/S arr: a;  E entero: n)
var
    entero:salto
    lógico:ordenada
    entero:j
    elemento:auxi
inicio
    salto ← n
```

```

mientras salto > 1 hacer
    salto ← salto div 2
    repetir
        ordenada ← verdad
        desde j ← 1 hasta n-salto hacer
            si esmenor(a[j], a[j+salto]) entonces
                auxi ← a[j]
                a[j] ← a[j+salto]
                a[j+salto] ← auxi
                ordenada ← falso
            fin_si
        fin_desde
    hasta_que ordenada
fin_mientras
fin_procedimiento

```

- 10.3.** Dada la lista ordenada en forma decreciente del ejercicio anterior, diseñar una función que devuelva el número de contratos realizados en una determinada fecha.

Análisis del problema

La función Contar, va a utilizar otras dos funciones auxiliares: EsMenor, anteriormente implementada, y Esigual, que compara las fechas de los dos registros pasados como parámetros e informa sobre si ambas son o no iguales.

Contar comenzará realizando una búsqueda binaria de la fecha, del registro que le pasamos como parámetro, en el vector. Si dicha fecha está en la lista, retrocede hasta encontrar la primera posición en donde aparece, después avanza contando los elementos hasta que éstos cambien de fecha y devuelve el valor del contador.

Diseño del algoritmo

```

lógico función esigual(ENTERO elemento: elemento1,elemento2)
inicio
    si (elemento1.año=elemento2.año) y (elemento1.mes=elemento2.mes) y
        (elemento1.día=elemento2.día) entonces
            devolver(verdad)
        si_no
            devolver(falso)
        fin_si
    fin_función

entero función contar(ENTERO arr: a; ENTERO entero: n; ENTERO elemento: elemento1)
var
    entero : primero,último,central,i
    lógico : encontrado, está
inicio
    primero ← 1
    último ← n
    está ← falso
    mientras (primero<=último) y (no está) hacer
        central ← (primero+último) div 2
        si esigual(a[central],elemento) entonces

```

```

está ← verdad
si_no
    si esmenor(a[central],elemento1) entonces
        último ← central-1
    si_no
        primero ← central+1
    fin_si
fin_si
fin_mientras
cont ← 0
si está entonces
    i ← central-1
    encontrado ← verdad
    mientras (i>=1) y (encontrado) hacer
        si esigual(a[i],a[central]) entonces
            i ← i-1
        si_no
            encontrado ← falso
        fin_si
    fin_mientras
    i ← i+1
    encontrado ← verdad
    mientras (i<=40) y encontrado hacer
        si esigual(a[i],a[central]) entonces
            cont ← cont+1
            i ← i+1
        si_no
            encontrado ← falso
        fin_si
    fin_mientras
fin_si
devolver(cont)
fin_función

```

10.4. Diseñar una función recursiva que realice una búsqueda binaria.

El método de búsqueda binaria ya ha sido explicado, por lo que simplemente se hará la función. Observar que ahora son necesarias dos salidas de las llamadas recursivas: cuando el elemento no se encuentra y cuando se encuentra.

```

entero función buscar(E entero : iz, de; E elemento : e; E arr :a)
var
    entero : ce
inicio
    ce ← (iz + de) div 2
    si Esigual(a[ce],e) entonces
        devolver(ce)
    si_no
        si iz > de entonces
            devolver(0)

```

```

    si_no
        si EsMenor(e,a[ce]) entonces
            devolver(buscar(iz,ce-1,e,a)
        si_no
            devolver(buscar(ce+1,de,e,a)
        fin_si
    fin_si
fin_si
fin_función

```

10.5. Diseñar un procedimiento recursivo que ordene una lista de elementos por el método Quick Sort.

```

procedimiento QuickSort(S arr : a; E entero : iz,de)
var
    entero : i,j
    elemento : pivote, aux
inicio
    i ← iz
    j ← de
    pivote ← a[(i + j) div 2]
repetir
    mientras EsMenor(a[i],pivote) hacer
        i ← i + 1
    fin_mientras
    mientras EsMenor(pivote,a[j]) hacer
        j ← j - 1
    fin_mientras
    si i <= j entonces
        aux ← a[i]
        a[i] ← a[j]
        a[j] ← a[i]
        i ← i + 1
        j ← j - 1
    fin_si
hasta_que i > j
si iz < j entonces
    QuickSort(iz,j)
fin_si
si i < de entonces
    QuickSort(i,de)
fin_si
fin_procedimiento

```

11

BÚSQUEDA, ORDENACIÓN Y FUSIÓN EXTERNAS (ARCHIVOS)

7i UbXc ``Ua U U XY Xhcg' U dfc WgU' dcf ' i bU Wa di hUXcf U Yg' [fUbXY mbc WVY Yb' gi ' a Ya cf]U WbhfU  fghcg' gY' cf [Ub]nUb 'Yb 'UfWlj cg' ei Yz' Ugi ' j Yn  gY' Ua UWbUb 'Yb 'Xlgdcgjhj cg' Yl hYfbcg' XY a Ya cf]U U 1 ']U' fxlgWg  WbhUg' a U[bf]Wg  YhW' '9b' Yghcg' Wgcz 'U' cdYf UWcbYg' Vzg|Wg' Yghi ! X]UXUg' Yb 'Y' '7Ud hi 'c' % cf XYbUW e  V'gei YXU Y]bhYf WVUW c' a YnWUgi ZfYb 'i b'Wa Vlc' ja dcf ! hUhY Yb' gi ' WbWdW e  XYf' j UXc' YgYbW Ua YbhY XY' \ YWc' Zlg|W' XY' ei Y' cg' Xhcg' U dfc WgU' bc WVYb' Yb' 'Ua Ya cf]U df]bW dU' XY' UWa di hUXcf U'

11.1. CONCEPTOS GENERALES

Para realizar este tipo de operaciones —búsqueda, ordenación o fusión— en archivos, se puede recurrir a cargar los registros en arrays (arreglos) y, a continuación, seguir cualquiera de los métodos, de búsqueda, clasificación o mezcla, internos vistos en capítulos anteriores.

En ocasiones, puede suceder que el volumen de datos a tratar sea demasiado grande y no quiepa en la memoria interna del ordenador. Será obligatorio entonces trabajar directamente con archivos, es decir con datos en la memoria externa o auxiliar. Esto origina un aumento del tiempo de ejecución, debido a las sucesivas operaciones de lectura y escritura en el soporte físico.

11.2. BÚSQUEDA EXTERNA

Búsqueda es el proceso de localizar un registro en un archivo con un valor específico en uno de sus campos. Los archivos de organización secuencial obligan a realizar búsquedas secuenciales. Los archivos de organización directa son estructuras de acceso aleatorio, por lo que permiten un mayor número de posibilidades a la hora de realizar las búsquedas, pudiendo implementarse búsquedas similares a las que se indicaban al hablar de arrays. El método a aplicar dependerá de la forma de colocación de los registros en el archivo.

11.3. FUSIÓN

La fusión o mezcla consiste en reunir varios archivos en uno sólo intercalándose los registros de unos en otros y siguiendo unos criterios determinados. El caso más corriente es la mezcla de dos archivos

que tienen sus registros colocados de forma secuencial y ordenados con respecto al valor de un determinado campo, el campo clave, para obtener un tercer archivo, también secuencial y en el que los registros sigan encontrándose ordenados con arreglo al contenido de dicho campo. El análisis del problema sería similar al ya estudiado, relativo a los vectores.

11.4. ORDENACIÓN EXTERNA

Por ordenación de archivos se entiende la clasificación de sus registros ascendente o descendenteamente con arreglo al valor de un determinado campo o a los valores de una jerarquía de campos. Con arreglo a los valores de una jerarquía de campos querría decir, clasificar primero por el contenido de un determinado campo, a igualdad de valor en ese campo, tener en cuenta y clasificar por el contenido de otro y así sucesivamente. Se pueden diferenciar dos casos: archivos de organización directa y archivos de organización secuencial.

Si el archivo tiene organización directa, aunque los registros se encuentran colocados en él de forma secuencial, servirá cualquiera de los métodos de clasificación vistos en el apartado de arrays, aunque con ligeras modificaciones debido a las operaciones de lectura y escritura de registros en el disco.

Si el archivo es de organización secuencial, se deben emplear otros métodos, que consiguen ordenar el archivo mediante la repetición de los siguientes procesos:

- **Partición.** Los registros se leen del archivo de entrada, no ordenado, y se dividen en distintos grupos de registros a los que se denomina particiones.
- **Mezcla.** Los registros de las distintas particiones se combinan de manera que obtengamos series de registros ordenadas.

El pseudocódigo y seguimiento de estos métodos de ordenación aparecen desarrollados en la parte práctica del presente capítulo.

11.4.1. Partición de archivos

Es posible efectuar distintos tipos de particiones.

11.4.1.1. Partición por contenido

El archivo inicial se divide en dos o más archivos dependiendo del valor de un determinado campo clave.

11.4.1.2. Partición en secuencias de longitud 1

El fichero inicial se dividirá en dos, colocando en un archivo los registros pares y en otro los impares. O bien el fichero inicial se dividirá en varios donde se irán colocando alternativamente secuencias de longitud 1.

11.4.1.3. Partición en secuencias de longitud N

Consiste en partir el fichero inicial en varios, copiando alternativamente en cada uno de ellos secuencias de longitud N. Se entiende por secuencia una sucesión de registros de la misma naturaleza.

11.4.1.4. Partición en secuencias de longitud N con clasificación interna de dichas secuencias

Se trata de leer del archivo inicial grupos de N registros que quepan en memoria. Cada grupo, al ser leído, se coloca en un array y se ordena. Los grupos ordenados se escriben alternativamente en dos o más archivos.

11.4.1.5. Partición según el método de selección por sustitución

Este método divide el archivo inicial en otros dos o más con largas secuencias ordenadas y consiste en:

1. Leer N registros del archivo desordenado, almacenándolos en un array y poniéndoles a todos una marca que denominaremos de no congelados.
2. Obtener el registro, $r \in a[m]$, con clave más pequeña de entre los no congelados y escribirlo en la partición.
3. Almacenar en $a[m]$ el siguiente registro del archivo de entrada y si su clave es más pequeña que la del último registro escrito, r , marcarlo como congelado.
4. Cuando todos los registros estén congelados o se haya alcanzado el fin de fichero, comenzará una nueva partición y deberemos descongelar los registros y cambiar el archivo destino.
5. Si no se alcanzó el fin del archivo inicial se repiten las acciones desde el 2.º paso. Si se llegó al fin del archivo de entrada, hasta que se hayan escrito todos los registros, se ejecutan también esas mismas acciones, pero sin leer y marcando todos los registros que se escriban como congelados.

11.4.1.6. Partición por el método de selección natural

Este método evita el tener que almacenar registros en un array y consiste en ir tomando secuencias ordenadas de la máxima longitud que copiaremos alternativamente sobre los ficheros de salida.

11.4.2. Ordenación por mezcla directa

Consiste en realizar sucesivas particiones y fusiones de un archivo de forma que se produzcan secuencias ordenadas de longitud cada vez mayor del siguiente modo:

1. Se establece como longitud inicial para la secuencia el valor uno, $N \leftarrow 1$.
2. Se efectúa la partición del archivo sobre otros, considerados de salida, copiando alternativamente en cada uno de ellos secuencias de longitud N .
3. Los archivos de salida resultantes mezclan los registros en dicha secuencia N , de tal forma que se obtenga una secuencia ordenada de longitud igual al número de archivos de salida multiplicado por N . Las sucesivas mezclas de las particiones sobrescriben el archivo original.
4. La longitud de la secuencia, N , se multiplica por el número de archivos y se vuelve al 2.º paso hasta que la longitud de la secuencia para la partición sea mayor o igual que el número de elementos del archivo original.

11.4.3. Ordenación por mezcla natural

Es uno de los mejores métodos de ordenación de ficheros secuenciales. Intenta aprovechar la posible ordenación interna de las secuencias del archivo, para lo cual:

1. Efectúa la partición del archivo desordenado sobre otros varios utilizando métodos de selección por sustitución o selección natural.
2. Realiza sucesivas fusiones de las secuencias ordenadas en estos archivos, sobrescribiendo con ellas el archivo inicial. La fusión ha de realizarse bajo el criterio de que en cada una de las secuencias resultantes exista ordenación.
3. Los pasos 1 y 2 se repetirán hasta que el archivo esté ordenado.

11.5. EJERCICIOS RESUELTOS

- 11.1.** Escribir un algoritmo que divida un archivo en dos, uno con los registros pares y otro con los impares.

Análisis del problema

Se utilizará una estructura repetitiva que nos permita la lectura de los registros del archivo inicial hasta la detección del fin del archivo. Cada registro leído se escribirá en un archivo u otro según el valor de una variable lógica o booleana, que conmutará a continuación de la escritura del registro.

Diseño del algoritmo

```

algoritmo Ejercicio_11_1
tipo
    registro: Datos_personales
        <tipo_dato>: <nombre_campo>
        ..... : ...
    fin_registro
    archivo_s de Datos_personales: arch
var
    Datos_personales: r
    Arch           : f1, f2, f
    lógico         : sw
inicio
    abrir(f, 'l', 'nombre')
    crear(f1,'nombre1')
    abrir(f1,'e','nombre1')
    crear(f2,'nombre2')
    abrir(f2,'e','nombre2')
    sw ← verdad
    mientras no fda(f) hacer
        llamar_a leer_reg_arch(f,r)
        // procedimiento auxiliar que realiza la lectura
        // del registro campo a campo
        si sw entonces
            llamar_a escribir_reg_arch(f1,r)
            //procedimiento auxiliar que escribe el registro campo a campo
        si_no
            llamar_a escribir_reg_arch(f2,r)
        fin_si
        sw ← no sw
    fin_mientras
    cerrar(f)
    cerrar(f1)
    cerrar(f2)
fin

```

- 11.2.** Algoritmo que realice la partición de un archivo copiando, alternativamente en otros dos, secuencias de una determinada longitud introducida por teclado.

Análisis del problema

Es similar al ejercicio anterior. La variable lógica commuta tras la escritura de N registros en el archivo.

```
algoritmo Ejercicio_11_2
tipo
    registro: Datos_personales
        <tipo_dato>: <nombre_campo>
        ..... : ...
    fin_registro
    archivo_s de Datos_personales: arch
var
    Datos_personales: r
    Arch           : f1, f2, f
    lógico         : sw
    entero         : i, n
inicio
    abrir(f, 'l', 'nombre')
    crear(f1, 'nombre1')
    abrir(f1, 'e', 'nombre1')
    crear(f2, 'nombre2')
    abrir(f2, 'e', 'nombre2')
    i ← 0
    escribir('Indique la longitud que desea para las secuencias')
    leer(n)
    sw ← verdad
    mientras no fda(f) hacer
        llamar_a leer_reg_arch(f,r)
        // procedimiento auxiliar que realiza la lectura
        // del registro campo a campo
        si sw entonces
            llamar_a escribir_reg_arch(f1,r)
            //procedimiento auxiliar que escribe el registro campo a campo
        si_no
            llamar_a escribir_reg_arch(f2,r)
        fin_si
        i ← i + 1
        si i = n entonces
            sw ← no sw
            i ← 0
        fin_si
    fin_mientras
    cerrar(f)
    cerrar(f1)
    cerrar(f2)
fin
```

- 11.3. Diseñar un algoritmo que realice una partición en secuencias, como la del ejercicio anterior, con clasificación interna de dichas secuencias.

Análisis del problema

Este algoritmo sería un primer paso en la ordenación de un archivo cuyo gran número de registros no permite una clasificación interna completa. Consistiría en leer un número de registros que quepa en memoria y colocarlo en una tabla. La tabla se ordena y se escribe alternativamente en dos archivos. Como los procedimientos para ordenar una tabla han sido vistos en capítulos anteriores, únicamente se hará a su llamada.

Diseño del algoritmo

```

algoritmo Ejercicio_11_3
const n = <expresión>
tipo
    registro: Datos_personales
        <tipo_dato>: nombre_campo_clave
        //Uno de los campos será el que contenga la clave por la que se
        //efectuará la clasificación
        ..... : ...
    fin_registro
    archivo_s de Datos_personales: arch
    array[1..n] de datos: arr
var
    Datos_personales: r
    Arr : a
    Arch : f1, f2, f
    lógico : cambio
    entero : i, n1
inicio
    abrir(f, 'l', 'nombre')
    crear(f1, 'nombre1')
    abrir(f1, 'e', 'nombre1')
    crear(f2, 'nombre2')
    abrir(f2, 'e', 'nombre2')
    cambio ← verdad
    n1 ← 1
    mientras no fda(f) hacer
        llamar_a leer_reg_arch(f,r)
        // procedimiento auxiliar que realiza la lectura
        // del registro campo a campo
        a[n1] ← r
        n1 ← n1 + 1
        si n1 = n + 1 entonces
            llamar_a ordenar(a,n)
            // procedimiento de clasificación de los registros en la tabla
            si cambio entonces
                desde i ← 1 hasta n hacer
                    llamar_a escribir_reg_arch(f1,a[i])
                    // procedimiento auxiliar que escribe el registro
                    // campo a campo
                fin_desde
            si_no

```

```

desde i ← 1 hasta n hacer
    llamar_a escribir_reg_arch(f2,a[i])
    fin_desde
fin_si
cambio ← no cambio
n1 ← 1
fin_si
fin_mientras
// En la tabla puede haber quedado un resto de n1-1 registros que
// todavía no hemos clasificado ni escrito
si n1 > 1 entonces
    n1 ← n1 -1
    llamar_a ordenar(a,n1)
    si cambio entonces
        desde i ← 1 hasta n1 hacer
            llamar_a escribir_reg_arch(f1,a[i])
            // procedimiento auxiliar que escribe el registro campo
            // a campo
        fin_desde
    si_no
        desde i ← 1 hasta n1 hacer
            llamar_a escribir_reg_arch(f2,a[i])
        fin_desde
    fin_si
fin_si
cerrar(f)
cerrar(f1)
cerrar(f2)
fin

```

11.4. Algoritmo que realice la partición de un archivo según el método de selección por sustitución.

Análisis del problema

Los pasos a seguir en la realización de este tipo de partición serían:

1. Leer N registros del archivo desordenado, poniéndolos todos a no congelados.
2. Obtener el registro R con clave más pequeña de entre los no congelados y escribirlo en partición.
3. Sustituir el registro por el siguiente del archivo de entrada. Este registro se congelará si su clave es más pequeña que la del registro R y no se congelará en otro caso. Si hay registros sin congelar, volver al paso 2.
4. Comenzar nueva partición. Si se ha llegado a fin de fichero se repite el proceso sin leer.

Al final de todas estas operaciones los ficheros con las particiones tienen secuencias ordenadas, lo que no quiere decir que ambos tengan que haber quedado completamente ordenados. Este sería el seguimiento del método a partir del archivo inicial F cuyas claves se muestran a continuación (el array auxiliar es de 4 elementos):

F:	3	31	14	42	10	15	8	13	63	18	50
----	---	----	----	----	----	----	---	----	----	----	----

Array

Se cargan los 4 primeros elementos en el array

3	31	14	42
---	----	----	----

F1:

F2:

10	31	14	42
----	----	----	----

F1: 3

F2:

15	31	14	42
----	----	----	----

F1: 3 10

F2:

15	31	8	42
----	----	---	----

F1: 3 10 14

F2:

13	31	8	42
----	----	---	----

F1: 3 10 14 15 13

F2:

13	63	8	42
----	----	---	----

F1: 3 10 14 15 31

F2:

13	63	8	18
----	----	---	----

F1: 3 10 14 15 31 42

F2:

13	50	8	18
----	----	---	----

F1: 3 10 14 15 31 42 63

F2:

13	50	8	18
----	----	---	----

F1: 3 10 14 15 31 42 63

F2:

13	50	8	18
----	----	---	----

F1: 3 10 14 15 31 42 63

F2: 8

13	50	8	18
----	----	---	----

F1: 3 10 14 15 31 42 63

F2: 8 13

13	50	8	18
----	----	---	----

F1: 3 10 14 15 31 42 63

F2: 8 13 18

13	50	8	18
----	----	---	----

F1: 3 10 14 15 31 42 63

F2: 8 13 18 50

Diseño del algoritmo

```
algoritmo Ejercicio_18_4
const n=<expresión>
tipo
    registro: Datos_personales
        <tipo_dato>: c //campo de claves
        .... : ...
fin_registro
registro: Datos
    Datos_personales: dp
    lógico : congela
fin_registro
array[1..n] de datos: arr
archivo_s de Datos_personales: arch
var
    Datos_personales: r
    Arr : a
    Arch : f1, f2, f
    lógico : sw
    entero : numcongelados, i, posiciónmenor
inicio
    abrir(f, 'l', 'nombre')
    crear(f1,'nombre1')
    abrir(f1,'e','nombre1')
    crear(f2,'nombre2')
    abrir(f2,'e','nombre2')
    numcongelados ← 0
    desde i ← 1 hasta n hacer
        si no fda(f) entonces
            llamar_a leer_reg_arch(f,r)
            a[i].dp ← r
            a[i].congela ← falso
        si_no
            a[i].congela ← verdad
            numcongelados ← numcongelados + 1
        fin_si
    fin_desde
    sw ← verdad
    mientras no fda(f) hacer
        mientras numcongelados < n y no fda(f) hacer
            llamar_a buscar_no_congelado_menor(a, posiciónmenor)
            si sw entonces
                llamar_a escribir_reg_arch(f1, a[posiciónmenor].dp)
            si_no
                llamar_a escribir_reg_arch(f2, a[posiciónmenor].dp)
            fin_si
            llamar_a leer_reg_arch(f,r)
            si r.c > a[posiciónmenor].dp.c entonces
                a[posiciónmenor].dp ← r
```

```

    si_no
        a [posiciónmenor].dp ← r
        a [posiciónmenor].congela ← verdad
        numcongelados ← numcongelados + 1
    fin_si
fin_mientras
sw ← no sw
llamar_a descongelar(a)
numcongelados ← 0
fin_mientras
mientras numcongelados < n hacer
    llamar_a buscar_no_congelado_menor(a, posiciónmenor)
    si sw entonces
        llamar_a escribir_reg_arch(f1, a [posiciónmenor].dp)
    si_no
        llamar_a escribir_reg_arch(f2, a [posiciónmenor].dp)
    fin_si
    a [posiciónmenor].congela ← verdad
    numcongelados ← numcongelados + 1
fin_mientras
cerrar(f, f1, f2)
fin

```

- 11.5.** Implementar un procedimiento para la ordenación de un archivo secuencial por el método de mezcla directa.

Análisis del problema

Este método de ordenación se basa en realizar partición en secuencias de los registros y mezclar luego los registros en esa secuencia. Por ejemplo, dado un archivo F cuyos registros almacenan la siguiente información en el campo clave:

F:	3 31 14 42 10 15 8 13 63 18 50
----	--

su ordenación por este método llevaría los siguientes pasos:

Partición en secuencias de longitud 1

F1	F2
3 14 10 8 63 50	31 42 15 13 18

Fusión de secuencias de longitud 1

F:	3 31 14 42 10 15 8 13 18 63 50
----	--

Partición en secuencias de longitud 2

F1	F2
3 31 10 15 18 63	14 42 8 13 50

Fusión de secuencias de longitud 2

F:	3 14 31 42 8 10 13 15 18 50 63
----	--

Partición en secuencias de longitud 4

F1

3	14	31	42	18	50	63
---	----	----	----	----	----	----

F2

8	10	13	15
---	----	----	----

Fusión de secuencias de longitud 4

F:

3	8	10	13	14	15	3	42	18	50	63
---	---	----	----	----	----	---	----	----	----	----

Partición en secuencias de longitud 8

F1

3	8	10	13	14	15	31	42
---	---	----	----	----	----	----	----

F2

18	50	63
----	----	----

Fusión de secuencias de longitud 8

F:

3	8	10	13	14	15	18	31	42	50	63
---	---	----	----	----	----	----	----	----	----	----

El proceso termina cuando la longitud de las secuencias es mayor o igual que el número de registros del archivo original. Como no se considera la existencia de un registro especial que indique el fin de archivo, la función **fda(id_arch)** retornará cierto con la lectura del último registro del archivo. Se hace necesario, en este caso, el uso de las variables lógicas **fin**, **fin1** y **fin2**, que controlan el que la salida de los bucles se produzca, no cuando se ha leído, sino cuando se ha escrito el último registro de un archivo.

Diseño del algoritmo

```

algoritmo ord_mezcla_directa
.....
procedimiento ordmezcladirecta
var
    datos_personales: r,r1,r2
    arch : f,f1,f2 // El tipo arch es archivo_s de datos_personales
    entero          : long,lgtud
    lógico          : sw,fin1,fin2
    entero          : i,j
inicio
    // calcularlong(f,r) es una función definida por el usuario que nos
    // devuelve el número de registros del archivo original}
    long ← calcularlong(f,r)
    lgtud ← 1
    mientras lgtud < long hacer
        // fd es el nombre que tiene en el disco el archivo que deseamos
        // ordenar
        abrir(f,'l','fd')
        crear(f1,'f1d')
        crear(f2,'f2d')
        abrir(f1,'e','f1d')
        abrir(f2,'e','f2d')
        i ← 0
        sw ← verdad
        mientras no fda(f) hacer

```

```
llamar_a leer_reg_arch(f,r)
si sw entonces
    llamar_a escribir_reg_arch(f1,r)
si_no
    llamar_a escribir_reg_arch(f2,r)
fin_si
i ← i+1
si i=lgtud entonces
    sw ← no sw
    i ← 0
fin_si
fin_mientras
cerrar(f,f1,f2)
abrir(f1,'l','f1d')
abrir(f2,'l','f2d')
crear(f,'fd')
abrir(f,'e','fd')
i ← 0
j ← 0
fin1 ← falso
fin2 ← falso
si fda(f1) entonces
    fin1 ← verdad
si_no
    llamar_a leer_reg_arch(f1,r1)
fin_si
si fda(f2) entonces
    fin2 ← verdad
si_no
    llamar_a leer_reg_arch(f2,r2)
fin_si
mientras no fin1 o no fin2 hacer
    mientras no fin1 y no fin2 y (i<lgtud) y (j<lgtud) hacer
        si menor(r1,r2) entonces
            llamar_a escribir_reg_arch(f,r1)
            si fda(f1) entonces
                fin1 ← verdad
            si_no
                llamar_a leer_reg_arch(f1,r1)
            fin_si
            i ← i+1
        si_no
            llamar_a escribir_reg_arch(f,r2)
            si fda(f2) entonces
                fin2 ← verdad
            si_no
                llamar_a leer_reg_arch(f2,r2)
            fin_si
            j ← j+1
        fin_si
```

```

fin_mientras
mientras no fin1 y (i<lgtud) hacer
    llamar_a escribir_reg_arch(f,r1)
    si fda(f1) entonces
        fin1 ← verdad
    si_no
        llamar_a leer_reg_arch(f1,r1)
    fin_si
    i ← i+1
fin_mientras
mientras no fin2 y (j<lgtud) hacer
    llamar_a escribir_reg_arch(f,r2)
    si fda(f2) entonces
        fin2 ← verdad
    si_no
        llamar_a leer_reg_arch(f2,r2)
    fin_si
    j ← j+1
fin_mientras
i ← 0
j ← 0
fin_mientras          // del mientras no fin1 o no fin2
cerrar(f,f1,f2)
lgtud ← lgtud*2
fin_mientras          // del mientras lgtud < long
borrar('f1d')
borrar('f2d')
fin_procedimiento

```

- 11.6.** Desarrollar un procedimiento que permita la ordenación de un archivo secuencial por el método de mezcla natural.

Análisis del problema

El método consiste en aprovechar secuencias ordenadas que pudieran existir inicialmente en el archivo, obteniendo con ellas particiones ordenadas de longitud variable sobre dos archivos auxiliares. A partir de estos ficheros auxiliares se construye un nuevo fichero, mezclando los segmentos crecientes máximos de cada uno de ellos.

El proceso se repetirá desde el principio hasta conseguir la ordenación completa del archivo. Por ejemplo, dado un archivo F cuyos registros almacenan la siguiente información en el campo clave:

F: 3 31 14 42 10 15 8 13 63 18 50

se observa que las secuencias ordenadas que existen inicialmente son:

3 31	14 42	10 15	8 13 63	18 50
-----------	------------	------------	-------------------	------------

y el proceso constaría de los siguientes pasos:

F1	F2
3 31	10 15 18 50
	14 42
	8 13 63

F:

3	14	31	42
---	----	----	----

F2

8	10	13	15	18	50	63
---	----	----	----	----	----	----

F1

F2

3	14	31	42
---	----	----	----

8	10	13	15	18	50	63
---	----	----	----	----	----	----

F:

3	8	10	13	14	15	18	31	42	50	63
---	---	----	----	----	----	----	----	----	----	----

Diseño del algoritmo

```

algoritmo ord_mezcla_natural
    .....
procedimiento ordmezclanatural
var
    datos_personales: r,r1,r2,ant,ant1,ant2
    arch : f,f1,f2      // El tipo arch es archivo_s de datos_personales
    lógico          : ordenado,crece,fin,fin1,fin2
    entero          : numsec
inicio
    ordenado _ falso
    mientras no ordenado hacer
        // Proceso de partición
        abrir(f,'l','fd')
        crear(f1,'f1d')
        crear(f2,'f2d');
        abrir(f1,'e','f1d')
        abrir(f2,'e','f2d')
        fin ← falso
        si fda(f) entonces
            fin ← verdad
        si_no
            llamar_a leer_reg_arch(f,r)
        fin_si
        mientras no fin hacer
            ant ← r
            crece ← verdad
        mientras crece y no fin hacer
            si menorigual(ant,r) entonces
                llamar_a escribir_reg_arch(f1,r)
                ant ← r
                si fda(f) entonces
                    fin ← verdad
                si_no
                    llamar_a leer_reg_arch(f,r)
                fin_si
                si_no
                    crece ← falso
                fin_si

```

```
fin_mientras
ant ← r
crece ← verdad
mientras crece y no fin hacer
    si menorigual(ant,r) entonces
        llamar_a escribir_reg_arch(f2,r)
        ant ← r
        si fda(f) entonces
            fin ← verdad
        si_no
            llamar_a leer_reg_arch(f,r)
        fin_si
        si_no
            crece ← falso
        fin_si
    fin_mientras
fin_mientras
cerrar(f,f1,f2)
// Proceso de mezcla
abrir(f1,'l','f1d')
abrir(f2,'l','f2d')
crear(f,'fd')
abrir(f,'e','fd')
fin1 ← falso
fin2 ← falso
si fda(f1) entonces
    fin1 ← verdad
si_no
    llamar_a leer_reg_arch(f1,r1)
fin_si
si fda(f2) entonces
    fin2 ← verdad
si_no
    llamar_a leer_reg_arch(f2,r2)
fin_si
numsec ← 0
mientras no fin1 y no fin2 hacer
    ant1 ← r1
    ant2 ← r2
    crece ← verdad
    mientras no fin1 y no fin2 y crece hacer
        si menorigual(ant1,r1) y menorigual(ant2,r2) entonces
            si menorigual(r1,r2) entonces
                llamar_a escribir_reg_arch(f,r1)
                ant1 ← r1
                si fda(f1) entonces
                    fin1 ← verdad
                si_no
                    llamar_a leer_reg_arch(f1,r1)
                fin_si
            fin_si
        fin_si
    fin_mientras
fin_mientras
```

```

    si_no
        llamar_a escribir_reg_arch(f,r2)
        ant2 ← r2
        si fda(f2) entonces
            fin2 ← verdad
        si_no
            llamar_a leer_reg_arch(f2,r2)
            fin_si
        fin_si
    si_no
        crece ← falso
    fin_si
fin_mientras
mientras no fin1 y menorigual(ant1,r1) hacer
    llamar_a escribir_reg_arch(f,r1)
    ant1 ← r1
    si fda(f1) entonces
        fin1 ← verdad
    si_no
        llamar_a leer_reg_arch(f1,r1)
    fin_si
fin_mientras
mientras no fin2 y menorigual(ant2,r2) hacer
    llamar_a escribir_reg_arch(f,r2)
    ant2 ← r2
    si fda(f2) entonces
        fin2 ← verdad
    si_no
        llamar_a leer_reg_arch(f2,r2)
    fin_si
fin_mientras
numsec ← numsec + 1
fin_mientras // del mientras no fin1 y no fin2
si no fin1 entonces
    numsec ← numsec+1
    mientras no fin1 hacer
        llamar_a escribir_reg_arch(f,r1)
        si fda(f1) entonces
            fin1 ← verdad
        si_no
            llamar_a leer_reg_arch(f1,r1)
        fin_si
    fin_mientras
fin_si
si no fin2 entonces
    numsec ← numsec+1
    mientras no fin2 hacer
        llamar_a escribir_reg_arch(f,r2)
        si fda(f2) entonces
            fin2 ← verdad

```

```

    si_no
        llamar_a leer_reg_arch(f2,r2)
    fin_si
    fin_mientras
fin_si
cerrar(f,f1,f2)
// Detectar si el archivo ya está ordenado
si numsec <= 1 entonces
    ordenado ← verdad
fin_si
fin_mientras                                // del mientras no ordenado
borrar('f1d')
borrar('f2d')
fin_procedimiento

```

- 11.7.** Realizar un algoritmo para la búsqueda por un determinado campo clave de la información almacenada en un registro de un archivo secuencial, ordenado crecientemente con arreglo al valor de dicho campo.

Análisis del problema

El planteamiento inicial será recorrer todo el archivo leyendo sus registros y comparando el campo clave de los registros leídos con el que estamos buscando.

Como el archivo está ordenado en orden creciente por el valor del campo por el cual se realiza la búsqueda, se puede acelerar el proceso, de forma que no sea necesario, en casi ningún caso, llegar al final del fichero, se encuentre o no en el archivo un registro con dicha clave. Se saldrá del recorrido del archivo en cuanto el registro sea encontrado o la clave buscada sea menor a la del registro que se acaba de leer.

Diseño del algoritmo

```

algoritmo Ejercicio_18_7
tipo
    registro: datos_personales
        <tipo_dato1>: nombre_campo1
        <tipo_dato2>: nombre_campo2
        ..... : .....
    fin_registro
    archivo_s de datos_personales: arch
var
    arch           : f
    datos_personales : persona
    <tipo_dato1>      : clavebus
    lógico          : encontrado, pasado
inicio
    abrir(f,'l',<nombre_en_disco>)
    encontrado ← falso
    pasado ← falso
    leer(clavebus)
    mientras no encontrado y no pasado y no fda(f) hacer
        llamar_a leer_f_reg(f, persona)
        si igual(clavebus, persona) entonces

```

```

        encontrado ← verdad
    si_no
        si menor(clavebus, persona) entonces
            pasado ← verdad
        fin_si
    fin_si
fin_mientras
si no encontrado entonces
    escribir ('No existe')
si_no
    llamar_a escribir_reg(persona)
    // Procedimiento auxiliar que presenta en pantalla
    // la información almacenada en el registro
fin_si
cerrar(f)
fin

```

- 11.8.** Dos sucursales A y B de una misma empresa tienen cada una un archivo (FICHA y FICHB) con los artículos que tienen en stock. Cada registro comprende el código de un artículo y la cantidad disponible en stock para la sucursal correspondiente. Las dos sucursales utilizan los mismos códigos para los mismos artículos, pero no tienen obligatoriamente los mismos artículos en stock. Los dos archivos son secuenciales y están dispuestos en orden creciente de códigos de artículos.

Escriba un algoritmo que permita sustituir FICHA y FICHB por un archivo único FUSIÓN que tenga las características siguientes:

- El archivo fusión está dispuesto en orden creciente de códigos de artículos.
- Cada registro comprenderá el código de un artículo y la cantidad disponible en stock en el conjunto de las dos sucursales.

Análisis del problema

Para resolver el problema se deberá:

- Tomar dos registros, uno de cada archivo.
- Compararlos y, si los códigos son iguales, escribir el código y la suma de los stocks en un nuevo archivo, seleccionando el registro siguiente en cada uno de los archivos iniciales.
- Si, al compararlos, los códigos son distintos se escribirá el de menor código en el nuevo archivo y se leerá el siguiente registro en el archivo al que pertenece el registro que acaba de ser escrito.
- Repetir el proceso de comparación hasta que toda la información, de al menos uno de los archivos, haya sido incluida en el destino.
- Si queda información del otro archivo aún sin incluir, copiar dicha información en el destino.

Diseño del algoritmo

```

algoritmo Ejercicio_11_8
tipo
    registro: reg
        ..... : cod
        entero : cantidad
    fin_registro
var
    reg: r1, r2
    archivo_s de reg: f, f1, f2

```

```
lógico: fin1, fin2
inicio
    abrir(f1,'l','FICHA')
    abrir(f2,'l','FICHB')
    crear(f,'FUSION')
    abrir(f,'e','FUSION')
    fin1 ← falso
    fin2 ← falso
    si fda(f1) entonces
        fin1 ← verdad
    si_no
        llamar_a leer_reg_arch(f1,r1)
    fin_si
    si fda(f2) entonces
        fin2 ← verdad
    si_no
        llamar_a leer_reg_arch(f2,r2)
    fin_si
mientras no fin1 y no fin2 hacer
    si igual(r1,r2) entonces
        r1.cantidad ← r1.cantidad + r2.cantidad
        llamar_a escribir_reg_arch(f,r1)
        si fda(f1) entonces
            fin1 ← verdad
        si_no
            llamar_a leer_reg_arch(f1,r1)
        fin_si
        si fda(f2) entonces
            fin2 ← verdad
        si_no
            llamar_a leer_reg_arch(f2,r2)
        fin_si
    si_no
        si menor(r1,r2) entonces
            llamar_a escribir_reg_arch(f,r1)
            si fda(f1) entonces
                fin1 ← verdad
            si_no
                llamar_a leer_reg_arch(f1,r1)
            fin_si
        si_no
            llamar_a escribir_reg_arch(f,r2)
            si fda(f2) entonces
                fin2 ← verdad
            si_no
                llamar_a leer_reg_arch(f2,r2)
            fin_si
        fin_si
    fin_mientras
```

```

//Tenga en cuenta que un bucle mientras puede ejecutarse 0 veces
mientras no fin1 hacer
    llamar_a escribir_reg_arch(f,r1)
    si fda(f1) entonces
        fin1 ← verdad
    si_no
        llamar_a leer_reg_arch(f1,r1)
    fin_si
fin_mientras
//Tenga en cuenta que un bucle mientras puede ejecutarse 0 veces
mientras no fin2 hacer
    llamar_a escribir_reg_arch(f,r2)
    si fda(f2) entonces
        fin2 ← verdad
    si_no
        llamar_a leer_reg_arch(f2,r2)
    fin_si
fin_mientras
cerrar(f,f1,f2)
fin

```

- 11.9.** Supuesto que el archivo es directo, pero la información fue inicialmente almacenada de forma secuencial y el archivo es tan grande que no cabe en memoria para su clasificación, escribir un algoritmo que ordene los registros de modo ascendente por el contenido de un determinado campo.

Análisis del problema

Las modificaciones con respecto a lo visto en la clasificación de arrays son debidas principalmente a las operaciones de lectura y escritura de registros.

Diseño del algoritmo

```

algoritmo Ejercicio_11_9
tipo
    registro: reg
        .... : c      //campo clave
        .... : ....
        .... : ....
    fin_registro
    archivo_d de reg: arch
var
    arch    : f
    entero : n, salto, j
    reg     : aux1, aux2
    lógico : ordenado
inicio
    abrir(f,'l/e','fdd.dat')
    //cálculo del número de registros del archivo
    n ← lda(f)/tamaño_de(reg)
    salto ← n
    mientras salto > 1 hacer
        salto ← salto div 2

```

```

repetir
    ordenado ← verdad
    desde j ← 1 hasta n - salto hacer
        leer(f,aux1,j)
        leer(f,aux2,j+salto)
        si aux1.c > aux2.c entonces
            escribir(f,aux2,j)
            escribir(f,aux1,j+salto)
            ordenado ← falso
        fin_si
    fin_desde
    hasta_que ordenado
fin_mientras
cerrar(f)
fin

```

- 11.10.** Escribir un algoritmo que permita la búsqueda por el campo clave en el archivo del ejercicio anterior, después de su ordenación.

Análisis del problema

Se puede aplicar el método de búsqueda binaria ya explicado en el capítulo relativo a la ordenación interna.

Diseño del algoritmo

```

algoritmo Ejercicio_11_10
tipo
    registro: reg
        .... : c                                //campo clave
        .... : ...
        .... : ...
fin_registro
    archivo_d de reg: arch
var
    reg      : r
    arch     : f
    ..... : clavebuscada           //del mismo tipo que el campo clave
    entero   : primero, último, central
    lógico   : encontrado
inicio
    abrir(f,1/e,'fdd.dat')
    escribir('Buscar: ')
    leer(clavebuscada)
    primero ← 1
    último ← lda(f)/tamaño_de(reg)
    encontrado ← falso
mientras (primero <= último) y no encontrado hacer
    central ← (primero + último) div 2
    leer(f,r,central)
    si clavebuscada = r.c entonces
        encontrado ← verdad

```

```
    si_no
        si clavebuscada > r.c entonces
            primero ← central+1
        si_no
            último ← central-1
        fin_si
    fin_si
fin_mientras
si encontrado entonces
    llamar_a escribir_reg(r)    //presenta el registro en pantalla
si_no
    escribir('No existe ningún registro con ese valor en el campo clave')
fin_si
cerrar(f)
fin
```

12

ESTRUCTURAS DINÁMICAS LINEALES DE DATOS (PILAS, COLAS Y LISTAS ENLAZADAS)

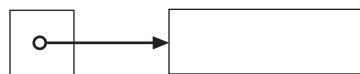
I bU Yghf i Wi f U XY XUhcg Xjbza]W Yg i bU W YW]E XY YYa Ybhcg Yb UhUXcg XYbca]bUXcg bcXcg XY
'U Yghf i Wi f U' 9ghcg YYa Ybhcg gY WYUb mYb UhUb c XYgYb UhUb mXYghf i mYb gY[•b gob bYWgjhU
Xcg Xi f UbhY U Y YW]E XY dfc f Ua U @g Yghf i Wi f Ug Xjbza]Wg XY XUhcg gY WUg Z]Wb Yb ']bYU
'Yg i 'bc ']bYUYg' 9 'Yghi Xjc XY 'Ug Yghf i Wi f Ug ']bYUYg' lghUg Yb UhUXUg d]Ug mW Ugz Yg Y cVYY!
hj c XY YghY Wdhi 'c"

12.1. ESTRUCTURAS DINÁMICAS

Hasta ahora se ha trabajado con estructuras que, durante la ejecución del programa, siempre ocupan la misma cantidad de espacio en memoria (estáticas), como arrays y registros, pero hay otro tipo de estructuras que pueden ampliar o limitar su tamaño mientras se ejecuta el programa (dinámicas).

Las estructuras dinámicas deben estar formadas por variables que se crean y destruyen durante la ejecución del programa. Así, Java y C# permiten la creación de listas enlazadas vinculando objetos que contienen al menos un miembro que es una referencia a otro objeto de su mismo tipo. En estos lenguajes (Java, C#) hay que tener en cuenta que al crear un objeto se reserva espacio en memoria para él y que este espacio se libera automáticamente a través de un proceso denominado recolección automática de basura cuando dicho objeto deja de estar referenciado. En otros lenguajes los nodos o elementos de la estructura son variables dinámicas generadas mediante un tipo de dato conocido con el nombre de puntero. Las variables dinámicas que constituyen los nodos o elementos de la estructura son registros con al menos un campo de tipo puntero. Una variable de tipo puntero almacena la dirección o posición de otra variable y la principal ventaja que representa manejar este tipo de datos es que se pueden adquirir posiciones de memoria a medida que se necesitan y liberarlas cuando ya no se requieran. De esta manera se pueden crear estructuras dinámicas que se expandan o contraigan según se les agreguen o eliminjen elementos.

Un **puntero** es una variable que almacena la dirección de memoria o posición de una variable dinámica de un tipo determinado. La representación gráfica de un puntero es una flecha que sale del puntero y llega a la variable dinámica apuntada.



Una variable de tipo puntero se declara de la siguiente forma:

```
tipo
    puntero_a <tipo_dato>: punt
var
    punt : p, q
```

El `<tipo_dato>` podrá ser simple o estructurado. Con los punteros se pueden hacer las siguientes operaciones:

- **Inicialización.** `p ← nulo`, a `nulo` para indicar que no apunta a ninguna variable.
- **Comparación.** `p = q`, con los operadores `=` ó `<>`.
- **Asignación.** `p ← q`, implica hacer que el puntero `p` apunte a donde apuntaba `q`.
- **Creación de variables dinámicas.** `Reservar(p)`, reserva espacio en memoria para la variable dinámica.
- **Eliminación de variables dinámicas.** `Liberar(p)`, libera el espacio en memoria ocupado por la variable dinámica.

Una **variable dinámica** es una variable simple o estructura de datos sin nombre y creada en tiempo de ejecución. Para acceder a una variable dinámica apuntada, como no tiene nombre emplearemos `nombre_variable_tipo_puntero1`. Las variables `nombre_variable_tipo_puntero1`, por ejemplo `p1`, podrán intervenir en toda operación o expresión de las permitidas para una variable estática de su mismo tipo. Las estructuras dinámicas de datos podrán ser lineales y no lineales; en ambos casos estarán formadas por **nodos**. Un nodo es una variable dinámica constituida por *al menos* dos campos:

- El campo dato o valor (elemento).
- El campo enlace, de tipo puntero (sig).

```
tipo
    puntero_a tnodo: punt
registro: tnodo
    tipo_elemento : elemento
    punt           : sig
    fin_registro
var
    punt           : inic, p
```

La variable dinámica no tiene nombre y nos referiremos a `p1.sig` y `p1.elemento`.

12.2. LISTAS

Una lista es una secuencia de 0 o más elementos de un tipo dado almacenados en memoria. Son estructuras lineales, donde cada elemento de la lista, excepto el primero, tiene un único predecesor y cada elemento de la lista, excepto el último, tiene un único sucesor.

El número de elementos de una lista se llama longitud. Si una lista tiene 0 elementos se denomina lista vacía. En una lista podremos añadir nuevos elementos o suprimirlos en cualquier posición. Se considerarán distintos tipos de listas:

Contiguas

Los elementos son adyacentes en la memoria del ordenador y tienen unos límites, izquierdo y derecho, que no pueden ser rebasados cuando se añade un nuevo elemento. Se implementan a través de

¹ Por razones de legibilidad del código, para hacer referencia a la variable dinámica se ha utilizado el operador `↑`, sin embargo, también sería posible utilizar el operador `→`.

arrays. La inserción o eliminación de un elemento, excepto en la cabecera o final de la lista, necesitará una traslación de parte de los elementos de la misma.

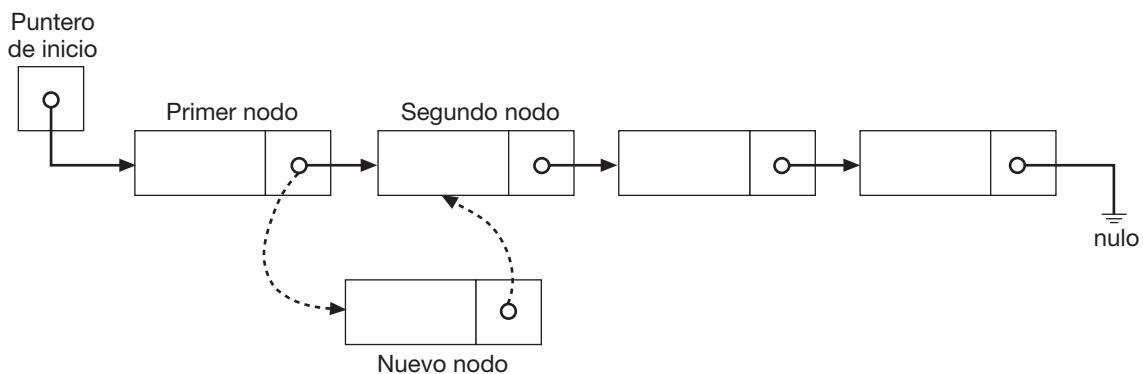


Enlazadas

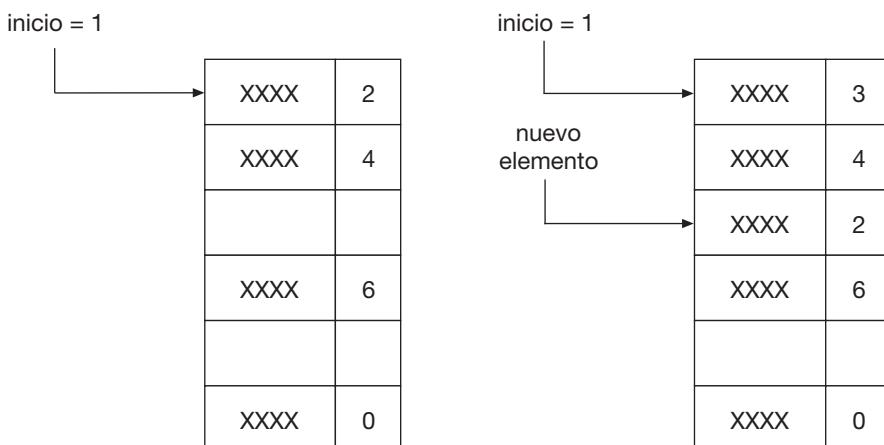
Los elementos se almacenan en posiciones de memoria que no son contiguas o adyacentes, por lo que cada elemento necesita almacenar la posición o dirección del siguiente elemento de la lista.

Son mucho más flexibles y potentes que las listas contiguas. La inserción o borrado del elemento n -ésimo no requiere el desplazamiento de los otros elementos de la lista. Se han de implementar de forma dinámica, pero si el lenguaje no lo permite se implementarán a través de arrays, con lo cual se imponen limitaciones en cuanto al número de elementos que podrá contener la lista y establece una ocupación en memoria constante.

Implementación con punteros



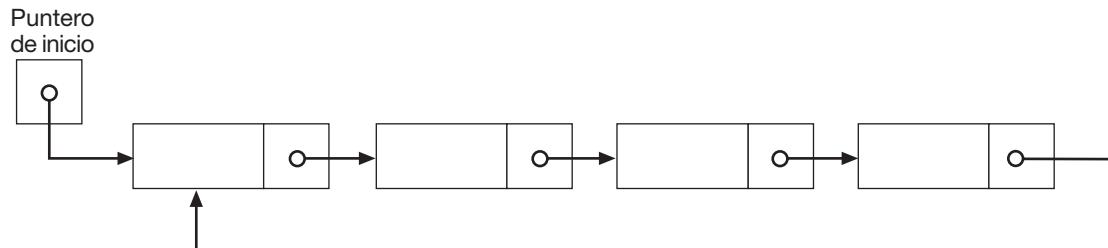
Implementación con arrays



Como se puede observar, tanto cuando se implementa con punteros como cuando se hace a través de arrays, la inserción de un nuevo elemento no requiere el desplazamiento de los que le siguen.

Circulares

Son una modificación de las enlazadas en las que el puntero del último elemento apunta al primero de la lista.



inicio = 1

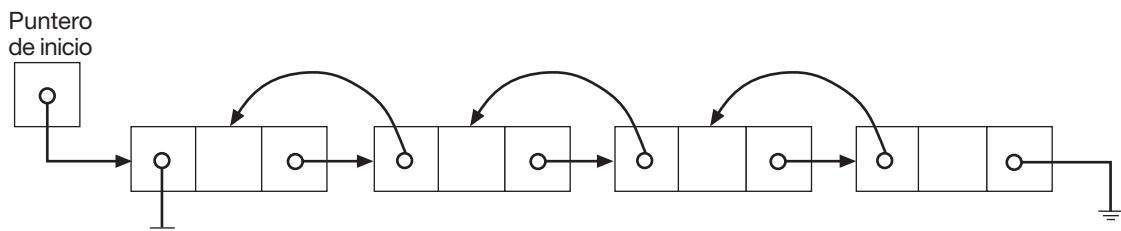
XXXX	2
XXXX	4
XXXX	6
XXXX	1

Con cabecera

Se debe diseñar un nodo especial, *cabecera*, permanentemente asociado a la existencia de la lista y cuyo campo para almacenar información no se utiliza. Al efectuar un recorrido de la lista, el nodo *cabecera* permitirá detectar cuando han sido visitados todos los demás nodos.

Dblemente encadenadas

Su recorrido puede realizarse tanto de frente a final como de final a frente. Cada nodo de dichas listas consta de un campo con información y otros dos de tipo puntero (Ant y Sig) y será referenciado por dos punteros, uno de su nodo sucesor y otro del anterior.



inicio = 1

0	XXXX	3
3	XXXX	4
1	XXXX	2
2	XXXX	6
4	XXXX	0

Listas doblemente encadenadas circulares

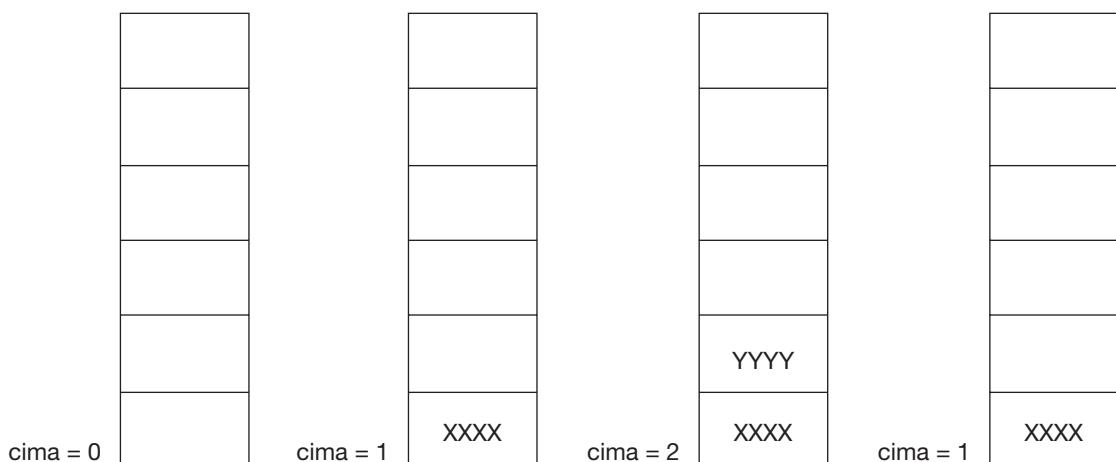
En este tipo de listas el campo `ant` del primer nodo de la lista apunta al último y el campo `sig` del último nodo al primero.

12.3. PILAS

Las pilas son una estructura de almacenamiento de la información bastante usual. El concepto de pila aparece en la vida diaria: pila de libros, de discos, de monedas, de cajas.

La pila se utiliza siempre que se quieren recuperar una serie de elementos en orden inverso a como se introdujeron. La extracción de un elemento de una pila se realiza por la parte superior, lo mismo que la inserción, lo que implica que el único elemento accesible de una pila es el último. Como el último elemento que se pone en la pila es el primero que se puede sacar, a estas estructuras se las conoce por el nombre de **LIFO (LAST INPUT FIRST OUTPUT)**.

Se deben implementar las pilas de forma dinámica, es decir con punteros, pero si el lenguaje no tiene punteros se pueden realizar mediante arrays y utilizando una variable auxiliar, `cima`, que apunte al último elemento de la pila. El uso de arrays limita el máximo número de elementos que la pila puede contener.



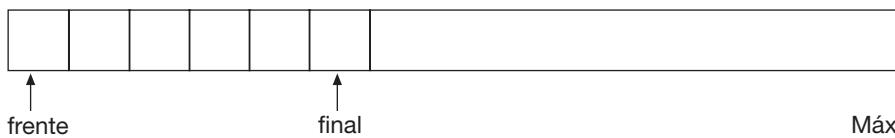
12.3.1. Aplicaciones de las pilas

Cuando un programa llama a un subprograma, internamente, se utilizan pilas para guardar el lugar desde donde se hizo la llamada, y el estado de las variables en ese momento. Además hay que recordar que todo algoritmo recursivo, puede ser implementado en forma iterativa utilizando una pila para almacenar los valores de las variables y parámetros.

12.4. COLAS

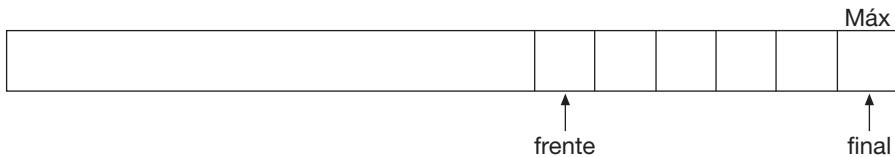
Una cola es una estructura de datos lineal en donde las eliminaciones se realizan por uno de sus extremos, denominado frente, y las inserciones por el otro, denominado final. Se las conoce como estructuras **FIFO** (*FIRST INPUT FIRST OUTPUT*). La cola de un autobús o de un cine son ejemplos de colas que aparecen en la vida diaria.

Una cola se deberá implementar de forma dinámica, es decir con punteros, pero si el lenguaje no tiene punteros se podrá realizar mediante un array y dos variables numéricicas (*frente*, *final*).



Esta implementación tiene el inconveniente de que puede ocurrir que la variable final llegue al valor máximo de la tabla, con lo cual no se puedan seguir añadiendo elementos a la cola, aún cuando queden posiciones libres a la izquierda de la posición frente.

Existen diversas soluciones a este problema:



Retroceso

Consiste en mantener fijo a 1 el valor de frente, realizando un desplazamiento de una posición para todas las componentes ocupadas cada vez que se efectúa una supresión.

Reestructuración

Cuando final llega al máximo de elementos se desplazan las componentes ocupadas hacia atrás las posiciones necesarias para que el principio coincida con el principio de la tabla.

Mediante un array circular

Un array circular es aquel en el que se considera que la componente primera sigue a la componente última. Esta implementación obliga a dejar siempre una posición libre para separar el principio y el final de la tabla. Evidentemente seguirá existiendo la limitación de que pueda llenarse completamente el array.

12.4.1. Doble cola

Existe una variante de la cola simple estudiada anteriormente y que es la *doble cola*. La *doble cola* o *bicola* es una cola bidimensional en la que las inserciones y eliminaciones se pueden realizar en cualquiera de los dos extremos de la lista.

Dentro de ella deberemos a su vez considerar dos tipos. La **doble cola de entrada restringida** acepta inserciones sólo al final de la cola. La **doble cola de salida restringida** acepta eliminaciones sólo al frente de la cola.

Los procedimientos de inserción y eliminación de nuevos elementos en las dobles colas son variantes de los que se emplean en las colas simples.

12.4.2. Aplicaciones de las colas

Una aplicación de las colas puede verse en las colas de impresión. En un sistema de tiempo compartido suele haber un procesador central y una serie de periféricos compartidos: discos, impresoras, etc. Los recursos se comparten por los diferentes usuarios y se utiliza una cola para almacenar los programas o peticiones de los diferentes usuarios que esperan su turno de ejecución. El procesador central atiende, normalmente, por riguroso orden de llamada del usuario; por tanto, todas las llamadas se almacenan en una cola. Existe otra aplicación muy utilizada que se denomina cola de prioridades; en ella el procesador central no atiende por riguroso orden de llamada, aquí el procesador atiende por prioridades asignadas por el sistema o bien por el usuario, y sólo dentro de las peticiones de igual prioridad se producirá una cola.

12.5. EJERCICIOS RESUELTOS

- 12.1.** *Diseñar las operaciones primitivas para el manejo de listas enlazadas utilizando estructuras de tipo array.*

Definición de las estructuras de datos utilizadas

```
const máx = <expresión> // Por ejemplo const = 6
tipo
    registro: tipo_elemento
    ... : ...
    ... : ...
fin_registro
registro: tipo_nodo
    tipo_elemento : elemento
    entero         : sig
    // actúa como puntero, almacenando la posición donde se
    // encuentra el siguiente elemento de una lista
fin_registro
array[1..Máx] de tipo_nodo: arr
var
    entero        : inic, posic, anterior, vacío
    arr           : m
    tipo_elemento : elemento
    lógico        : encontrado
```

Procedimientos y funciones

En una lista se pueden insertar elementos y borrarlos en cualquier posición. En las listas enlazadas la inserción o borrado del elemento *n*-ésimo no requerirá el desplazamiento de los otros elementos de la lista. Para

conseguirlo, se considerará el array como si fuera la memoria del ordenador y, por tanto, será utilizado para almacenar dos listas: la lista de elementos y la lista de vacíos, cuyo primer elemento está apuntado por vacío. Se inicializará de la siguiente forma: `vacío ← 1`, indicando así como primer registro libre el `m[1]`.

Todos los registros, excepto el último que recibirá un 0, almacenarán, en su campo `sig`, la posición siguiente en el array, `m[i].sig ← i+1`, informando que el registro situado a continuación también está vacío. Además, a la variable `inic` se le dará el valor 0. `inic` apunta al primer elemento de la lista, y el valor 0 indica que no hay elementos en ella.

<code>inicio = 0</code> <code>vacío = 1</code>	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr><td>1</td><td>2</td></tr> <tr><td>2</td><td>3</td></tr> <tr><td>3</td><td>4</td></tr> <tr><td>4</td><td>5</td></tr> <tr><td>5</td><td>6</td></tr> <tr><td>6</td><td>0</td></tr> </table>	1	2	2	3	3	4	4	5	5	6	6	0
1	2												
2	3												
3	4												
4	5												
5	6												
6	0												

```

procedimiento inicializar(S entero: inic) // lista de elementos
inicio
    inic ← 0
fin_procedimiento

procedimiento iniciar(S arr: m; S entero: vacío) // lista de vacíos
var
    entero: i
inicio
    vacío ← 1
    desde i ← 1 hasta Máx 1 hacer
        m[i].sig ← i+1
    fin_desde
    m[Máx].sig ← 0 // Como ya no hay más posiciones libres
                      // a las que apuntar, recibe un 0
fin_procedimiento

```

Al insertar un nuevo elemento en la lista se debe recurrir al procedimiento `reservar(...)` que proporciona, a través de `auxi`, la primera posición vacía para almacenar en ella el nuevo elemento. Además, eliminará dicha posición de la lista de vacíos.

Por ejemplo al insertar el primer elemento, `vacío` señala que la primera posición libre es la 1, el campo `sig` del registro `m[vacío]` proporciona la siguiente posición vacía y `reservar` hará que `vacío` apunte a esta nueva posición. Más adelante, cuando se desarrolle el procedimiento `insertar(...)`, se verá como:

```

m[1].elemento ← 'xxxx'
m[1].sig ← 0
inic ← 1

```

Siguiendo los punteros a partir de `inic` se recorre la lista de elementos. Conviene que, de momento, sólo nos fijemos en el valor de `auxi`, la lista de vacíos y las modificaciones sufridas por `vacío`.

Al insertar un segundo elemento, como `vacío` tiene el valor 2, `auxi ← 2` y `vacío` será `m[2].sig`, es decir, `vacío ← 3`. En `insertar(...)` el valor de `inic` no se modifica.

inicio = 1
vacío = 2

1	XXXX	0
2		3
3		4
4		5
5		6
6		0

inicio = 1
vacío = 3

1	XXXX	2
2	XXXX	0
3		4
4		5
5		6
6		0

```
procedimiento reservar(S entero: auxi; E arr: m; E/S entero: vacío)
inicio
    si vacío = 0 entonces
        // Memoria agotada
        auxi ← 0
    si_no
        auxi ← vacío
        vacío ← m[vacío].sig
    fin_si
fin_procedimiento
```

Para eliminar un elemento de la lista se ha de recurrir al procedimiento `Suprimir(...)`, que, a su vez, llamará al procedimiento `Liberar(...)` para que inserte el elemento eliminado en la lista de vacíos. Si se trata de eliminar el elemento marcado con `****` y, tras sucesivas inserciones de elementos, el aspecto de la lista es el siguiente:

inicio = 1
vacío = 5

1	XXXX	2
2	XXXX	3
3	****	4
4	XXXX	0
5		6
6		0

La posición del elemento a eliminar es la 3 (`posic ← 3`), el elemento anterior al 3 ocupa en el array la posición 2 (`anterior ← 2`), el primer vacío está en 5 (`vacío ← 5`). Al suprimir el elemento 3 la lista quedaría:

inicio = 1
vacío = 3

1	XXXX	2
2	XXXX	4
3	****	5
4	XXXX	0
5		6
6		0

es decir, `m[2].sig ← 4`. Mediante el procedimiento `Liberar(...)` se añade el nuevo elemento vacío en la lista de vacíos (`m[3].sig ← 5` y `vacío ← 3`). Como el que se suprime no es el primer elemento de la lista, el valor de `inic` no varía (`inic ← 1`).

```

procedimiento liberar(E entero: posic; E/S arr: m; E/S entero: vacío)
inicio
    m[posic].sig ← vacío
    vacío ← posic
fin_procedimiento

procedimiento suprimir(E/S entero: inic,anterior, posic;
                           E/S arr: m; E/S entero: vacío)
inicio
    si anterior = 0 entonces
        inic ← m[posic].sig
    si_no
        m[anterior].sig ← m[posic].sig
    fin_si
    llamar_a liberar(posic,m,vacío)
    anterior ← 0 // Opcional
    posic ← inic // Opcional
    // Preparar los punteros para que, si no especificamos otra cosa,
    // la próxima eliminación se realice por el principio de la lista
fin_procedimiento

```

El procedimiento insertar coloca un nuevo elemento a continuación de anterior; si anterior fuera 0 significa que ha de insertarse al comienzo de la lista. Si se desea insertar un nuevo elemento en la lista a continuación del primero y la situación actual, tras sucesivas inserciones y eliminaciones, es como se muestra a continuación:

inicio = 1 vacío = 3	<table border="1" style="width: 100%; border-collapse: collapse;"> <tbody> <tr><td style="text-align: center;">1</td><td style="text-align: center;">XXXX</td><td style="text-align: center;">2</td></tr> <tr><td style="text-align: center;">2</td><td style="text-align: center;">XXXX</td><td style="text-align: center;">4</td></tr> <tr><td style="text-align: center;">3</td><td></td><td style="text-align: center;">5</td></tr> <tr><td style="text-align: center;">4</td><td style="text-align: center;">XXXX</td><td style="text-align: center;">6</td></tr> <tr><td style="text-align: center;">5</td><td></td><td style="text-align: center;">0</td></tr> <tr><td style="text-align: center;">6</td><td style="text-align: center;">XXXX</td><td style="text-align: center;">0</td></tr> </tbody> </table>	1	XXXX	2	2	XXXX	4	3		5	4	XXXX	6	5		0	6	XXXX	0
1	XXXX	2																	
2	XXXX	4																	
3		5																	
4	XXXX	6																	
5		0																	
6	XXXX	0																	

el nuevo elemento se colocará en el array en la primera posición libre y lo único que se hará es modificar los punteros.

inicio = 1 vacío = 5	<table border="1" style="width: 100%; border-collapse: collapse;"> <tbody> <tr><td style="text-align: center;">1</td><td style="text-align: center;">XXXX</td><td style="text-align: center;">3</td></tr> <tr><td style="text-align: center;">2</td><td style="text-align: center;">XXXX</td><td style="text-align: center;">4</td></tr> <tr><td style="text-align: center;">3</td><td style="text-align: center;">nuevo el.</td><td style="text-align: center;">2</td></tr> <tr><td style="text-align: center;">4</td><td style="text-align: center;">XXXX</td><td style="text-align: center;">6</td></tr> <tr><td style="text-align: center;">5</td><td></td><td style="text-align: center;">0</td></tr> <tr><td style="text-align: center;">6</td><td style="text-align: center;">XXXX</td><td style="text-align: center;">0</td></tr> </tbody> </table>	1	XXXX	3	2	XXXX	4	3	nuevo el.	2	4	XXXX	6	5		0	6	XXXX	0
1	XXXX	3																	
2	XXXX	4																	
3	nuevo el.	2																	
4	XXXX	6																	
5		0																	
6	XXXX	0																	

reservar(...) nos proporciona la primera posición libre y hace que vacío pase a apuntar al siguiente elemento vacío (auxi ← 3, vacío ← 5 y m[3].elemento ← nuevo elemento). Como se desea insertar el nuevo elemento a continuación del primero de la lista, su anterior será el apuntado por inic(anterior ← 1, m[3].sig ← 2 y m[1].sig ← 3).

```

procedimiento insertar(E/S entero: inic,anterior;
                      E tipo_elemento: elemento;
                      E/S arr: m; E/S vacío: entero)

var
    entero: auxi
inicio
    llamar_a reservar(auxi,m,vacío)
    m[auxi].elemento ← elemento
    si anterior = 0 entonces
        m[auxi].sig ← inic
        inic ← auxi
    si_no
        m[auxi].sig ← m[anterior].sig
        m[anterior].sig ← auxi
    fin_si
    anterior ← auxi // Opcional
    // Prepara anterior para que, si no especificamos otra cosa,
    // la siguiente inserción se realice a continuación de la actual
fin_procedimiento

lógico función vacía(E entero: inic)
inicio
    si inic = 0 entonces
        devolver(verdad)
    si_no
        devolver(falso)
    fin_si
fin_función

lógico función llena(E entero: vacío)
inicio
    si vacío = 0 entonces
        devolver(verdad)
    si_no
        devolver(falso)
    fin_si
fin_función

```

El procedimiento consultar informa si un elemento se encuentra o no en la lista, la posición que ocupa dicho elemento en el array y la que ocupa el elemento anterior. En él se supone que la lista se encuentra ordenada ascendente por alguno de los campos de elemento.

```

procedimiento consultar(E entero: inic; S entero: posic,anterior;
                      E tipo_elemento: elemento;
                      S lógico: encontrado; E arr: m)

inicio
    anterior ← 0
    posic ← inic
    // Las funciones menor(...) e igual(...) comparan los registros
    // que le son pasados como parámetros
    mientras menor(m[posic].elemento,elemento) y (posic<>0) hacer

```

```

    anterior ← posic
    posic ← m[posic].sig
fin_mientras
si igual(m[posic].elemento,elemento) entonces
    encontrado ← verdad
si_no
    encontrado ← falso
fin_si
fin_procedimiento

```

El recorrido de la lista se realizará siguiendo los punteros a partir de su primer elemento, el señalado por `inic`. El procedimiento `recorrer(...)` irá mostrando por pantalla los diferentes elementos que la componen:

```

procedimiento recorrer(E entero: inic)
var
    entero: posic
inicio
    posic ← inic
    mientras posic <> 0 hacer
        // Recurrimos a un procedimiento, proc_escribir(...),
        // para presentar por pantalla los campos del registro pasado
        // como parámetro
        llamar_a proc_escribir(m[posic].elemento)
        posic ← m[posic].sig
    fin_mientras
fin_procedimiento

```

12.2. Diseñar las operaciones primitivas para el manejo de listas enlazadas utilizando estructuras dinámicas de datos.

Definición de las estructuras de datos utilizadas

```

tipo
    puntero_a nodo: punt
    registro: tipo_elemento
    ... : ...
    ... : ...
fin_registro
    registro: nodo
    tipo_elemento : elemento
    punt          : sig
fin_registro
var
    punt          : inic
    punt          : posic
    punt          : anterior
    tipo_elemento : elemento
    lógico        : encontrado
inicio
    Inicializar(inic)
    .....
    .....
fin

```

Procedimientos y funciones

```

procedimiento inicializar(S punt: inic)
inicio
    inic ← nulo
fin_procedimiento

lógico función vacía(E punt: inic)
inicio
    si inic = nulo entonces
        devolver(verdad)
    si_no
        devolver(falso)
    fin_si
fin_función

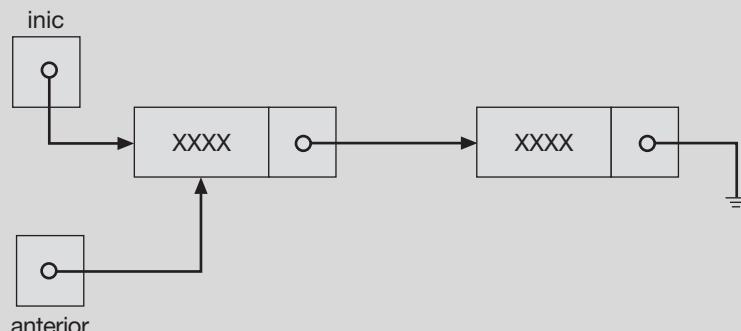
procedimiento consultar(E punt: inic; S punt: posic,anterior;
                        E tipo_elemento: elemento;
                        S lógico: encontrado)
inicio
    anterior ← nulo
    posic ← inic
    // la lista está ordenada ascendente con arreglo a alguno
    // de los campos de elemento
    mientras menor(posic↑.elemento,elemento) y (posic<>nulo) hacer
        anterior ← posic
        posic ← posic↑.sig
    fin_mientras
    si igual(posic↑.elemento,elemento) entonces
        encontrado ← verdad
    si_no
        encontrado ← falso
    fin_si
fin_procedimiento

```

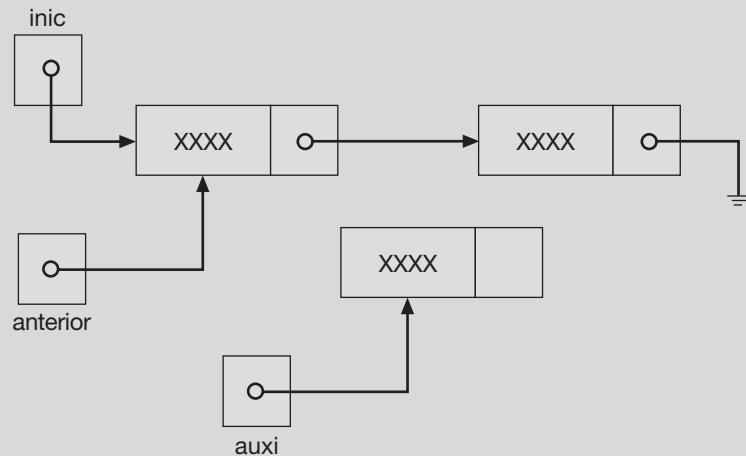
El procedimiento Insertar, inserta un nuevo elemento a continuación de anterior, si anterior fuera **nulo** significa que ha de insertarse al comienzo de la lista.

Insertar(inic,anterior,elemento)

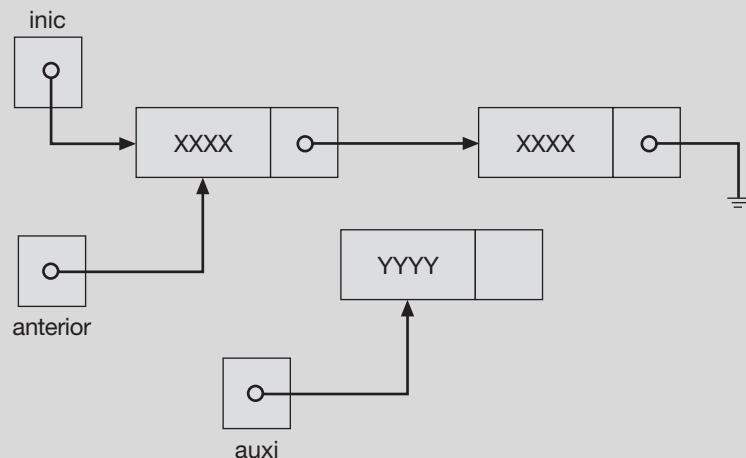
1. Situación de partida



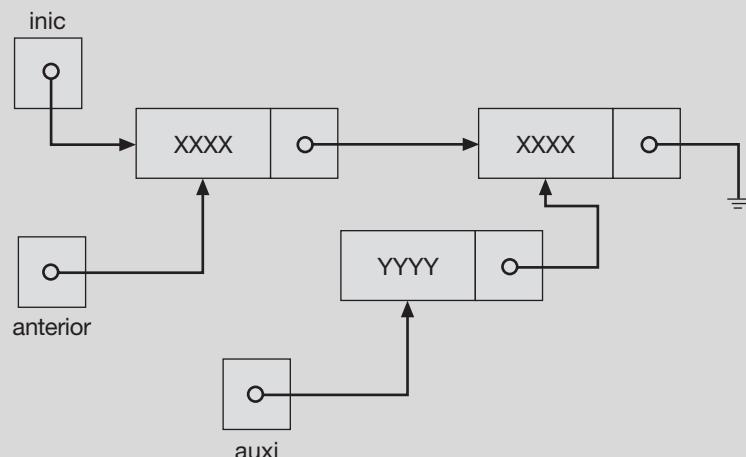
2. Reservar auxi



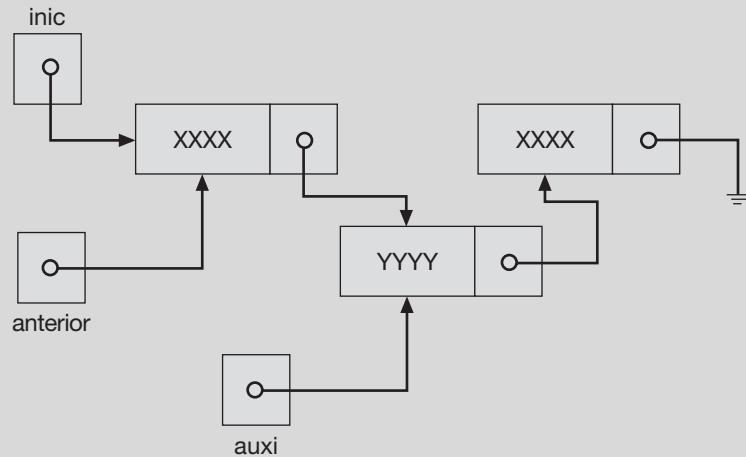
3. Introducir la nueva información en $\text{auxi}^\uparrow.\text{elemento}$



4. Hacer que $\text{auxi}^\uparrow.\text{sig}$ apunte a donde lo hacía $\text{anterior}^\uparrow.\text{sig}$



5. Conseguir que $\text{anterior}^\uparrow.\text{sig}$ apunte a donde lo hace auxi



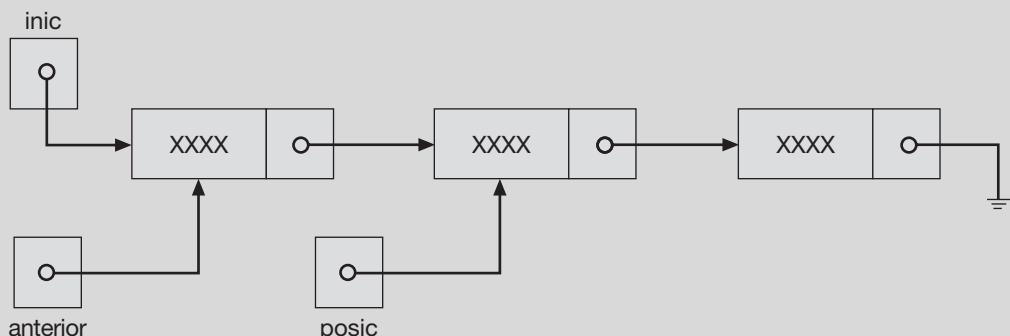
```

procedimiento insertar(E/S punt: inic,anterior;  E tipo_elemento: elemento)
var
    punt: auxi
inicio
    Reservar(auxi)
    auxi↑.elemento = elemento
    si anterior = nulo entonces
        auxi↑.sig ← inic
        inic ← auxi
    si_no
        auxi↑.sig ← anterior↑.sig
        anterior↑.sig ← auxi
    fin_si
    anterior ← auxi // Opcional
fin_procedimiento

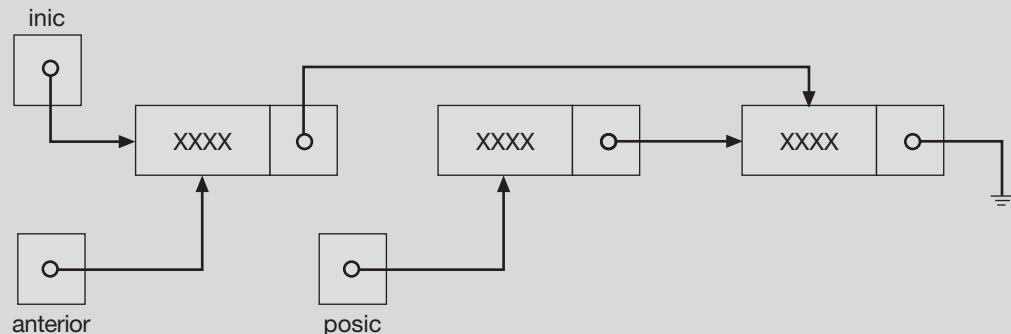
```

Suprimir(inic,anterior, posic)

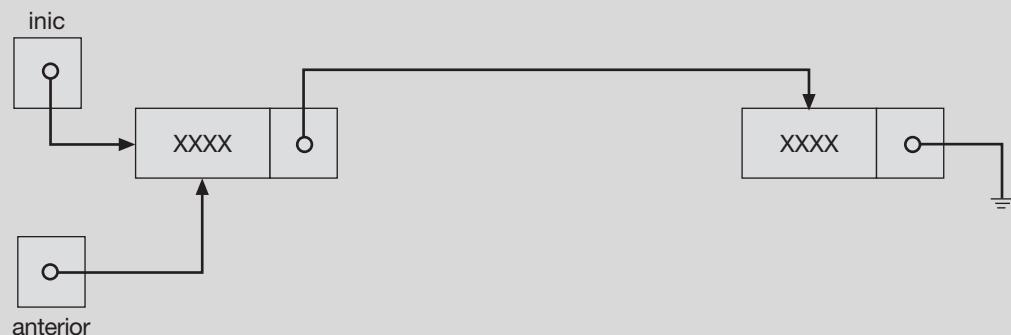
1. Situación de partida



2. anterior.sig apunta a donde posic \uparrow .si.



3. liberar(posic).



```

procedimiento suprimir(E/S punt: inic,anterior,posic)
  inicio
    si anterior = nulo entonces
      inic <- posic $\uparrow$ .sig
    si_no
      anterior $\uparrow$ .sig <- posic $\uparrow$ .sig
    fin_si
    Liberar(posic)
    anterior <- nulo // Opcional
    posic <- inic // Opcional
  fin_procedimiento

  procedimiento recorrer(E punt:inic)
    var
      punt: posic
  inicio
    posic <- inic
    mientras posic <> nulo hacer
      llamar_a proc_escribir(posic $\uparrow$ .elemento)
      posic <- posic $\uparrow$ .sig
    fin_mientras
  fin_procedimiento
  
```

- 12.3.** Realizar una implementación de cadenas utilizando listas enlazadas. Se deberán contemplar los procedimientos o funciones para: leer una cadena, escribir una cadena, calcular su longitud, calcular la posición de un carácter dentro de una cadena y obtener una subcadena a partir de una cadena principal.

Análisis del problema

Se ha de leer la cadena carácter a carácter, e insertar dichos caracteres en una lista en el mismo orden en que se han leído. Los restantes procedimientos y funciones pedidos son, en su mayor parte, diferentes versiones del recorrido de listas. Para obtener una subcadena a partir de la cadena inicial, hay que recorrerla hasta encontrar la posición de comienzo de la subcadena y, en ese momento, complementar el recorrido con la inserción en una lista auxiliar del número de caracteres indicado.

Se resolverá mediante arrays y mediante estructuras dinámicas. Las diferencias existentes entre un algoritmo y otro son debidas a las diferentes estructuras utilizadas.

Diseño del algoritmo

Mediante arrays

```
algoritmo Ejercicio_12_3a
const máx = .....
tipo
    registro: tipo_nodo
        carácter : elemento
        entero   : sig
    fin_registro
    array[1..Máx] de tipo_nodo:lista
var
    entero: inic, iniciaxi vacío, p, cuántos
    lista : m
    carácter: opción, c
inicio
    Iniciar(m, vacío)
    escribir('Deme cadena:')
    leer_cad(inic,vacío,m)
repetir
    escribir('1.  escribir cadena')
    escribir('2.  Longitud cadena')
    escribir('3.  Posición de carácter en cadena')
    escribir('4.  Obtener subcadena a partir de cadena principal')
    escribir('5.  Fin')
    escribir('Elija opción:')
    leer(opción)
    según_sea opción hacer
        '1':  escribir_cad(inic,m)
        '2':  escribir(longitud_cad(inic,m))
        '3':  escribir('Deme carácter a buscar:')
        leer(c)
        escribir(posición_car(inic,m,c))
        '4':  escribir('¿A partir de qué posición?')
        leer(p)
        escribir('¿Cuántos caracteres?')
        leer(cuántos)
```

```

        escribir_cad(subcad_cad(inic,vacío,m,p,cuántos),m)
    fin_según
    hasta_que opción='5'
fin

procedimiento inicializar(S entero: inic)
inicio
    inic ← 0
fin_procedimiento

procedimiento iniciar(S lista: m;  S entero: vacío)
var
    entero: i
inicio
    vacío ← 1
    desde i ← 1 hasta Máx    1 hacer
        m[i].sig ← i + 1
    fin_desde
    m[Máx].sig ← 0
fin_procedimiento

procedimiento reservar (S entero: auxi; E lista: m;  E/S entero: vacío)
inicio
    si vacío = 0 entonces
        auxi ← 0
    si_no
        auxi ← vacío
        vacío ← m[vacío].sig
    fin_si
fin_procedimiento

lógico función llena(E entero: vacío)
inicio
    si vacío = 0 entonces
        devolver(verdad)
    si_no
        devolver(falso)
    fin_si
fin_función

procedimiento insertar(E/S entero: inic, anterior; E carácter: elemento;
                      E/S lista: m;  E/S entero: vacío)
var
    entero: auxi
inicio
    reservar (auxi,m,vacío)
    m[auxi].elemento ← elemento
    si anterior = 0 entonces
        m[auxi].sig ← inic
        inic ← auxi
    si_no

```

```
m[auxi].sig ← m[anterior].sig
m[anterior].sig ← auxi
fin_si
anterior ← auxi
fin_procedimiento

procedimiento leer_cad(E/S entero: inic; E/S entero:vacio; E/S lista: m)
var
    carácter: c
    entero : anterior
inicio
    Inicializar(inic)
    leercar(c)
    anterior ← 0
    mientras (c<>car(13)) y no llena(vacio) hacer
        escribir(c)
        insertar( inic, anterior, c, m, vacío)
        si no llena(vacio) entonces
            leercar(c)
        si_no
            escribir()
            escribir('La cadena no puede sobrepasar esta longitud')
        fin_si
    fin_mientras
    escribir()
fin_procedimiento

procedimiento escribir_cad(E entero:inic; E lista: m)
var
    entero: posic
inicio
    posic ← inic
    mientras posic <> 0 hacer
        escribir(m[posic].elemento)
        posic ← m[posic].sig
    fin_mientras
    escribir()
fin_procedimiento

entero función longitud_cad(E entero: inic; E lista: m)
var
    entero: posic, contador
inicio
    posic ← inic
    contador ← 0
    mientras posic <> 0 hacer
        contador ← contador + 1
        posic ← m[posic].sig
    fin_mientras
    devolver(contador)
fin_función
```

```

entero función posición_car (E entero: inic; E lista:m; E carácter : c)
var
    entero : posic, contador
    lógico : encontrado
inicio
    encontrado ← falso
    posic ← inic
    contador ← 0
    mientras (posic <> 0) y no encontrado hacer
        contador ← contador + 1
        si m[posic].elemento = c entonces
            encontrado ← verdad
        si_no
            posic ← m[posic].sig
        fin_si
    fin_mientras
    si encontrado entonces
        devolver(contador)
    si_no
        devolver(0)
    fin_si
fin_función

entero función subcad_cad(E entero: inic; E/S entero: vacío;
                           E/S lista: m; E entero: p, cuántos)
var
    entero    : i, posic, inicioauxi, anteriorauxi
    carácter : c
inicio
    Inicializar(inicioauxi)
    si p < 1 entonces
        escribir('Error')
    si_no
        i ← 0
        posic ← inic
        mientras (posic <> 0) y (i < p 1) hacer
            posic ← m[posic].sig
            i ← i+1
        fin_mientras
        i ← 0
        anteriorauxi ← 0
        mientras (posic <> 0) y (i < cuántos) y no llena(vacío) hacer
            c ← m[posic].elemento
            insertar(inicioauxi, anteriorauxi, c, m, vacío)
            posic ← m[posic].sig
            i ← i+1
        fin_mientras
    fin_si
    devolver(inicioauxi)
fin_procedimiento

```

Mediante estructuras dinámicas

```
algoritmo Ejercicio_12_3b
tipo
    puntero_a nodo : cade
    registro : nodo
        carácter : elemento
        cade      : sig
    fin_registro
var
    cade      : inic, vacío
    entero   : p, cuántos
    carácter : opción, c
inicio
    escribir('Deme cadena:')
    leer_cad(inic)
    repetir
        escribir('1. Escribir cadena')
        escribir('2. Longitud cadena')
        escribir('3. Posición de carácter en cadena')
        escribir('4. Obtener subcadena a partir de cadena principal')
        escribir('5. Fin')
        escribir('Elija opción:')
        leer(opción)
        según_sea opción hacer
            '1': escribir_cad(inic)
            '2': escribir(longitud_cad(inic))
            '3': escribir('Deme carácter a buscar:')
            leer(c)
            escribir(posición_car(inic,c))
            '4': escribir('¿A partir de qué posición?')
            leer(p)
            escribir('¿Cuántos caracteres?')
            leer(cuántos)
            Escribir_cad(subcad_cad(inic,p,cuántos))
        fin_según
    hasta_que opción='5'
fin

procedimiento inicializar(S cade: inic)
inicio
    inic ← nulo
fin_procedimiento

procedimiento insertar(E/S cade: inic,anterior; E carácter: elemento)
var
    cade: auxi
inicio
    Reservar(auxi)
    auxi↑.elemento ← elemento
    si anterior = nulo entonces
```

```

auxi $\uparrow$ .sig  $\leftarrow$  inic
inic  $\leftarrow$  auxi
si_no
    auxi $\uparrow$ .sig  $\leftarrow$  anterior $\uparrow$ .sig
    anterior $\uparrow$ .sig  $\leftarrow$  auxi
fin_si
    anterior _ auxi
fin_procedimiento

procedimiento leer_cad(E/S cade: inic)
var
    carácter : c
    cade      : anterior
inicio
    Inicializar(inic)
    leercar(c)
    anterior  $\leftarrow$  nulo
    mientras c < $\neq$  car(13) hacer
        escribir(c)
        insertar(inic, anterior, c)
        leercar(c)
    fin_mientras
    escribir()
fin_procedimiento

procedimiento escribir_cad(E cade: inic)
var
    cade: posic
inicio
    posic  $\leftarrow$  inic
    mientras posic < $\neq$  nulo hacer
        escribir(posic $\uparrow$ .elemento)
        posic  $\leftarrow$  posic $\uparrow$ .sig
    fin_mientras
    escribir()
fin_procedimiento

entero función longitud_cad(E cade: inic)
var
    cade: posic
    entero: contador
inicio
    posic  $\leftarrow$  inic
    contador  $\leftarrow$  0
    mientras posic < $\neq$  nulo hacer
        contador  $\leftarrow$  contador + 1
        posic  $\leftarrow$  posic $\uparrow$ .sig
    fin_mientras
    devolver(contador)
fin_función

```

```
entero función posición_car (E cade: inic; E carácter: c)
var
    cade    : posic
    lógico : encontrado
    entero : contador
inicio
    encontrado ← falso
    posic ← inic
    contador ← 0
    mientras (posic <> nulo) y no encontrado hacer
        contador ← contador + 1
        si posic↑.elemento = c entonces
            encontrado ← verdad
        si_no
            posic ← posic↑.sig
        fin_si
    fin_mientras
    si encontrado entonces
        devolver(contador)
    si_no
        devolver(0)
    fin_si
fin_función

cade función subcad_cad(E cade: inic; E entero: p, cuántos)
var
    entero   : i
    cade      : posic,inicioauxi,anteriorauxi
    carácter : c
inicio
    Inicializar(inicioauxi)
    si p < 1 entonces
        escribir('Error')
    si_no
        i ← 0
        posic ← inic
        mientras (posic <> nulo) y (i < p - 1) hacer
            posic ← posic↑.sig
            i ← i+1
        fin_mientras
        i ← 0
        anteriorauxi ← nulo
        mientras (posic <> nulo) y (i < cuántos) hacer
            c ← posic↑.elemento
            insertar( inicioauxi, anteriorauxi, c)
            posic ← posic↑.sig
            i ← i+1
        fin_mientras
    fin_si
    devolver(inicioauxi)
fin_función
```

12.4. Escribir un algoritmo que lea dos polinomios con una única variable o letra y los sume. La implementación se realizará mediante listas enlazadas.

Análisis del problema

Para sumar dos polinomios es necesario sumar entre sí los monomios semejantes; los monomios que no sean semejantes se expondrán en el polinomio suma tal como los proponían. Como el enunciado nos indica que los polinomios tienen una única variable o letra, los monomios resultarán semejantes cuando su exponente sea el mismo, pudiéndose diferenciar en el signo y el coeficiente.

El algoritmo comenzará por almacenar en dos listas los polinomios iniciales, ordenando sus monomios de mayor a menor grado y no admitiendo en un mismo polinomio varios términos del mismo grado. La obtención del polinomio suma se basará en considerar repetitivamente dos términos, uno de cada polinomio, comparar sus exponentes e insertar en una nueva lista el de mayor grado; si ambos son del mismo grado se inserta la suma de ambos. Cuando se ha insertado un término de un polinomio se toma el siguiente de ese mismo polinomio, cuando se haya insertado una suma se pasa al término siguiente en ambos.

Como los coeficientes pueden tener signo positivo o negativo, si al obtener un monomio suma su coeficiente resultara 0, dicho monomio, no se insertaría en la nueva lista, pero pasaríamos al término siguiente en cada uno de los sumandos como hacemos cuando lo insertamos.

Diseño del algoritmo

```

algoritmo Ejercicio_12_4
const máx = ....
tipo
    registro: tipo_elemento
        entero: coef, expon
        // Al tener una única variable o letra, ésta es toda la
        // información sobre los monomios que es necesario almacenar
        // y el exponente coincide con el grado
    fin_registro
    registro: tipo_nodo
        elemento : Tipo_Elemento
        entero   : sig
    fin_registro
    array[1..Máx] de tipo_nodo : lista
var
    entero : inic1,inic2,inicsuma,vacío
    lista  : m
inicio
    Iniciar(m,vacío)
    escribir('Deme primer polinomio:')
    leer_polinom(inic1,vacío,m)
    escribir('Deme segundo polinomio:')
    leer_polinom(inic2,vacío,m)
    escribir_polinom(inic1,m)
    escribir_polinom(inic2,m)
    sumar_polinom(inic1,inic2,inicsuma,vacío,m)
fin

procedimiento inicializar(s entero: inic)
inicio
    inic ← 0
fin_procedimiento

```

```
lógico función vacía(E entero: inic)
inicio
    si inic = 0 entonces
        devolver(verdad)
    si_no
        devolver(falso)
    fin_si
fin_función

procedimiento iniciar(S lista: m; S entero: vacío)
var
    entero: i
inicio
    vacío ← 1
    desde i ← 1 hasta Máx 1 hacer
        m[i].sig ← i + 1
    fin_desde
    m[Máx].sig ← 0
fin_procedimiento

procedimiento reservar (S entero: auxi; E lista: m;
                        E/S entero: vacío)
inicio
    si vacío = 0 entonces
        auxi ← 0
    si_no
        auxi ← vacío
        vacío ← m[vacío].sig
    fin_si
fin_procedimiento

lógico función llena(E entero: vacío)
inicio
    si vacío = 0 entonces
        devolver(verdad)
    si_no
        devolver(falso)
    fin_si
fin_función

procedimiento insertar(E/S entero: inic, anterior;
                        E tipo_elemento: elemento; E/S lista: m;
                        E/S entero: vacío)
var
    entero: auxi
inicio
    reservar (auxi,m,vacío)
    m[auxi].elemento ← elemento
    si anterior = 0 entonces
        m[auxi].sig ← inic
        inic ← auxi
```

```

    si_no
        m[auxi].sig ← m[anterior].sig
        m[anterior].sig ← auxi
    fin_si
    anterior ← auxi
fin_procedimiento

procedimiento consultar(E entero: inic; S entero: anterior;
                        E tipo_elemento: elemento;
                        S lógico: puedo; E lista: m)
var
    lógico: parar
    entero: posic
inicio
    // Puedo nos indica si podemos insertar el nuevo término.
    // Si ya existe alguno con dicho exponente no se
    // permitirá la inserción
    anterior ← 0
    posic ← inic
    parar ← falso
    mientras (posic <> 0) y no parar hacer
        si (m[posic].elemento.expon > elemento.expon) entonces
            anterior ← posic
            posic ← m[posic].sig
        si_no
            parar ← verdad
        fin_si
    fin_mientras
    si (posic <> 0) y (m[posic].elemento.expon = elemento.expon) entonces
        puedo ← falso
    si_no
        puedo ← verdad
    fin_si
fin_procedimiento

procedimiento leer_polinom(E/S entero: inic, vacío; E/S lista: m)
var
    tipo_elemento: elemento
    entero: anterior
    lógico: puedo
inicio
    Inicializar(inic)
    escribir('Deme coeficiente:')
    leer(elemento.Coef)
    // Cuando el coeficiente sea 0 el término se anula,
    // por tanto lo utilizamos para terminar
    mientras (elemento.coef <> 0) Y no llena(vacío) hacer
        escribir('Deme exponente:')
        leer(elemento.expon)
        //El exponente podrá ser 0
        consultar(inic,anterior, elemento, puedo, m)

```

```
    si puedo entonces
        insertar(inic, anterior, elemento, m, vacío)
        si no llena(vacío) entonces
            escribir('Deme coeficiente:')
            leer(elemento.coef)
        si_no
            escribir('El polinomio no puede sobrepasar esta longitud')
        fin_si
    si_no
        escribir('Error')
    fin_si
fin_mientras
escribir()
fin_procedimiento

procedimiento escribir_polinom(E entero: inic; E lista: m)
var
    entero: posic
inicio
    posic ← inic
    mientras posic <> 0 hacer
        escribir(m[posic].elemento.coef)
        escribir('* x^')
        escribir(m[posic].elemento.expon)
        posic ← m[posic].sig
        si (posic <> 0) y (m[posic].elemento.coef > 0) entonces
            escribir(' + ')
        fin_si
    fin_mientras
    escribir()
fin_procedimiento

procedimiento sumar_polinom(E entero: inic1,inic2; S entero: iniciosuma;
                           E/S entero: vacío; E/S lista: m)
var
    entero: posic1,posic2,anteriorsuma
    tipo_elemento: e1,e2
inicio
    Inicializar(iniciosuma)
    posic1 ← inic1
    posic2 ← inic2
    si posic1 <> 0 entonces
        e1 ← m[posic1].elemento
    fin_si
    si posic2 <> 0 entonces
        e2 ← m[posic2].elemento
    fin_si
    anteriorsuma ← 0
    mientras (posic1 <> 0) y (posic2 <> 0) y no llena(vacío) hacer
        si e1.expon > e2.expon entonces
            insertar(iniciosuma, anteriorsuma, e1, m, vacío)
        fin_si
        anteriorsuma ← anteriorsuma + e1.coef * x^e1.expon
        posic1 ← posic1.sig
        si posic1 <> 0 then
            e1 ← m[posic1].elemento
        fin_si
        anteriorsuma ← anteriorsuma + e2.coef * x^e2.expon
        posic2 ← posic2.sig
        si posic2 <> 0 then
            e2 ← m[posic2].elemento
        fin_si
    fin_mientras
    anteriorsuma ← anteriorsuma + m[inic2].elemento.coef * x^m[inic2].elemento.expon
    escribir(anteriorsuma)
fin_procedimiento
```

```

        posic1 ← m[posic1].sig
        si posic1 <> 0 entonces
            e1 ← m[posic1].elemento
        fin_si
    si_no
        si e1.expon = e2.expon entonces
            e1.coef ← e1.coef + e2.coef
            si e1.coef <> 0 entonces
                insertar(iniciosuma, anteriorsuma, e1, m, vacío)
            fin_si
        posic1 ← m[posic1].sig
        posic2 ← m[posic2].sig
        si posic1 <> 0 entonces
            e1 ← m[posic1].elemento
        fin_si
        si posic2 <> 0 entonces
            e2 ← m[posic2].elemento
        fin_si
    si_no
        insertar(iniciosuma, anteriorsuma, e2, m, vacío)
        posic2 ← m[posic2].sig
        si posic2 <> 0 entonces
            e2 ← m[posic2].elemento
        fin_si
    fin_si
fin_mientras
mientras (posic1 <> 0) y no llena(vacío) hacer
    insertar(iniciosuma, anteriorsuma, e1, m, vacío)
    posic1 ← m[posic1].sig
    si posic1 <> 0 entonces
        e1 ← m[posic1].elemento
    fin_si
fin_mientras
mientras (posic2 <> 0) y no llena(vacío) hacer
    insertar(iniciosuma, anteriorsuma, e2, m, vacío)
    posic2 ← m[posic2].sig
    si posic2 <> 0 entonces
        e2 ← m[posic2].elemento
    fin_si
fin_mientras
si (posic1 <> 0) o (posic2 <> 0) entonces
    escribir('Error el polinomio suma no admite más términos')
fin_si
    escribir_polinom(iniciosuma, m)
fin_procedimiento

```

- 12.5.** Una tienda de artículos deportivos desea almacenar en una lista enlazada, con un único elemento por producto, la siguiente información sobre las ventas realizadas: Código del artículo, Cantidad y Precio. Usando estructuras de tipo array, desarrollar un algoritmo que permita tanto la creación de la lista como su actualización al realizarse nuevas ventas o devoluciones de un determinado producto.

Análisis del problema

El algoritmo contemplará la creación de la lista y colocará los elementos clasificados por código, para que las búsquedas puedan resultar algo más rápidas. Al producirse una venta se han de considerar las siguientes posibilidades:

- Es la primera vez que se vende ese artículo y esto nos lleva a la inserción de un nuevo elemento en la lista.
- Ya se ha vendido alguna otra vez dicho artículo; por tanto, es una modificación de un elemento de la lista, incrementándose la cantidad vendida.

Una devolución nos hará pensar en las siguientes situaciones:

- El comprador devuelve parte de lo que se había vendido de un determinado artículo, lo que representa una modificación de la cantidad vendida, decrementándose con la devolución.
- Se devuelve todo lo que se lleva vendido de un determinado artículo y, en consecuencia, el producto debe desaparecer de la lista de ventas.

Diseño del algoritmo

```

algoritmo Ejercicio_12_5
const máx = ....
tipo
    registro : tipo_elemento
    cadena : cód
    entero : cantidad
    real : precio
fin_registro
registro : tipo_nodo
    tipo_elemento : elemento
    entero : sig
fin_registro
array[1..Máx] de tipo_nodo : lista
var
    entero : inic, vacío
    lista : m
    carácter : opción
inicio
    Iniciar(m,vacío)
    Inicializar(inic)
repetir
    escribir('1.- Ventas')
    escribir('2.- Devoluciones')
    escribir('3.- Mostrar lista')
    escribir('4.- Fin')
    escribir('Elija opción:')
    leer(opción)
    según_sea opción hacer
        '1': nuevasventas(inic,vacío,m)
        '2': devoluciones(inic,vacío,m)
        '3': recorrer(inic,m)
    fin_síguin
hasta_que opción='4'
fin

```

```

procedimiento inicializar(S entero: inic)
inicio
    inic ← 0
fin_procedimiento

lógico función vacía(E entero: inic)
inicio
    si inic = 0 entonces
        devolver(verdad)
    si_no
        devolver(falso)
    fin_si
fin_función
procedimiento iniciar(E/S lista: m; E/S entero: vacío)
var
    entero : i
inicio
    vacío ← 1
    desde i ← 1 hasta Máx - 1 hacer
        m[i].sig ← i + 1
    fin_desde
    m[Máx].sig ← 0
fin_procedimiento

procedimiento reservar(S entero: auxi; E lista: m; E/S entero: vacío)
inicio
    si vacío = 0 entonces
        escribir('Memoria agotada')
        auxi ← 0
    si_no
        auxi ← vacío
        vacío ← m[vacío].sig
    fin_si
fin_procedimiento

lógico función llena(E entero: vacío)
inicio
    si vacío = 0 entonces
        devolver(verdad)
    si_no
        devolver(falso)
    fin_si
fin_función

procedimiento consultar(E entero: inic; S entero: posic,anterior;
                        E tipo_elemento: elemento;
                        S lógico: encontrado; E lista: m)
inicio
    anterior ← 0
    posic ← inic
    mientras (m[posic].elemento.cód < elemento.cód) y (posic <> 0) hacer

```

```
anterior ← posic
posic ← m[posic].sig
fin_mientras
si m[posic].elemento.cód = elemento.cód entonces
    encontrado ← verdad
si_no
    encontrado ← falso
fin_si
fin_procedimiento

procedimiento insertar(E/S entero: inic, anterior;
                           E tipo_elemento: elemento;
                           E/S lista: m; E/S entero: vacío)
var
    entero: auxi
inicio
    llamar_a reservar(auxi,m,vacío)
    m[auxi].elemento ← elemento
    si anterior = 0 entonces
        m[auxi].sig ← inic
        inic ← auxi
    si_no
        m[auxi].sig ← m[anterior].sig
        m[anterior].sig ← auxi
    fin_si
    anterior ← auxi
fin_procedimiento

procedimiento recorrer(E entero: inic; E lista: m)
var
    entero: posic
inicio
    posic ← inic
    mientras posic <> 0 hacer
        escribir(m[posic].elemento.cód, ' ', m[posic].elemento.cantidad, '',
                  m[posic].elemento.precio)
        posic ← m[posic].sig
    fin_mientras
fin_procedimiento

procedimiento liberar(E entero: posic; E/S lista: m; E/S entero: vacío)
inicio
    m[posic].sig ← vacío
    vacío ← posic
fin_procedimiento

procedimiento suprimir(E/S entero: inic,anterior,posic;
                           E/S lista: m; E/S entero: vacío)
inicio
    si anterior = 0 entonces
        inic ← m[posic].sig
```

```

si_no
    m[anterior].sig ← m[posic].sig
fin_si
liberar(posic,m,vacío)
anterior ← 0
posic ← inic
fin_procedimiento

procedimiento nuevasventas(E/S entero: inic,vacío; E/S lista: m)
var
    tipo_elemento : elemento
    lógico : encontrado
    entero : anterior, posic
inicio
    repetir
        escribir('Código:')
        leer(elemento.cód)
        si elemento.cód <> '*' entonces
            si vacía(inic) entonces
                anterior ← 0
                escribir('Cantidad:')
                leer(elemento.cantidad)
                escribir('Precio:')
                leer(elemento.precio)
                insertar(inic,anterior,elemento,m,vacío)
            si_no
                consultar(inic,posic,anterior,elemento,encontrado,m)
                si no encontrado entonces
                    si no llena(vacío) entonces
                        escribir('Cantidad:')
                        leer(elemento.cantidad)
                        escribir('Precio:')
                        leer(elemento.precio)
                        insertar(inic,anterior,elemento,m,vacío)
                    si_no
                        escribir('Llena')
                    fin_si
                si_no
                    escribir('Cantidad:')
                    leer(elemento.cantidad)
                    m[posic].elemento.cantidad ←
                        m[posic].elemento.cantidad + elemento.cantidad
                fin_si
            fin_si
        fin_si
        hasta_que (elemento.cód='*')
fin_procedimiento

procedimiento devoluciones(E/S entero: inic,vacío; E/S lista: m)
var
    tipo_elemento : elemento

```

```

entero      : posic,anterior
entero      : cantidad
lógico      : encontrado

inicio
    si no vacía(inic) entonces
        escribir('Deme elemento y * para fin')
        escribir('Código:')
        leer(elemento.cód)
    si_no
        escribir('No hay ventas, no puede haber devoluciones')
    fin_si
    mientras (elemento.cód <> '*') y no vacía(inic) hacer
        consultar(inic,posic,anterior,elemento,encontrado,m)
        si encontrado entonces
            repetir
                escribir('Deme cantidad devuelta')
                leer(cantidad)
                si cantidad > m[posic].elemento.cantidad entonces
                    escribir('No se puede devolver más de lo que se compró')
                fin_si
                hasta_que cantidad <= m[posic].elemento.cantidad
                m[posic].elemento.cantidad ← m[posic].elemento.cantidad-cantidad
                si m[posic].elemento.cantidad = 0 entonces
                    suprimir(inic,anterior,posic,m,vacío)
                fin_si
            si_no
                escribir('No existe')
            fin_si
        si_no vacía(inic) entonces
            escribir('Deme elemento y * para fin')
            escribir('Código:')
            leer(elemento.cód)
        si_no
            escribir('No hay ventas, no puede haber devoluciones')
        fin_si
    fin_mientras
fin_procedimiento

```

12.6. Diseñar las operaciones primitivas necesarias para trabajar con pilas utilizando estructuras de tipo array.

Definición de las estructuras de datos

Si los datos se van a almacenar en un array, la estructura pila será un registro compuesto por un array del tipo base de la pila (TipoElemento) y una variable entera, cima, que indica la posición del último elemento de la pila. El número máximo de elementos de la pila vendrá determinado por una constante (MáxPila).

```

const MáxPila = ...
tipo
    registro : TipoElemento
    ... : ...
    ... : ...

```

```

fin_registro
registro : Pila
    entero : cima
    array[1..MáxPila] de TipoElemento : el
fin_registro
var
    pila : p

```

Procedimientos y funciones

Se considerarán cinco primitivas para trabajar con pilas:

PilaVacía	Inicializa una pila, crea una pila como vacía.
EsPilaVacía	Es una función lógica que indica si una pila está vacía.
PInsertar	Inserta un elemento en la pila.
Tope	Devuelve el elemento situado en la cima de la pila.
PBorrar	Borra un elemento de la pila.

El procedimiento PilaVacía, cuando se trabaja con arrays, se limita a inicializar el entero **cima** a cero —podemos considerar el cero como un puntero nulo a arrays.

```

procedimiento PilaVacía(s pila : p)
inicio
    p.cima ← 0
fin_procedimiento

```

La función lógica EsPilaVacía, indica si una pila está vacía, es decir, si su puntero **cima** es nulo. Esta función va a ser útil cuando se desea recorrer una pila hasta el final.

```

lógico función EsPilaVacía(E pila : p)
inicio
    devolver(p.cima = 0)
fin_función

```

Para insertar un elemento en la pila, simplemente se deberá incrementar **cima** en una unidad y en la posición **cima** del array de elementos insertar el nuevo elemento que pasaremos como parámetro al procedimiento. Hay que advertir que se puede dar una condición de error si no hay sitio en el array, lo que ocurrirá cuando **cima** sea igual a MáxPila.

```

procedimiento PInsertar(s pila : p; E TipoElemento : e)
inicio
    si p.cima = MáxPila entonces
        // Error, la pila está llena
    si_no
        p.cima ← p.cima + 1
        p.el[p.cima] ← e
    fin_si
fin_procedimiento

```

El único elemento accesible de la pila es el que ocupa la posición **cima**. El procedimiento Tope se utiliza para recuperar dicho elemento. No se podrá recuperar si la pila está vacía.

```

procedimiento Tope(E pila : p; s TipoElemento : e)
inicio
    si EsPilaVacía(p) entonces
        // Error la pila está vacía

```

```

si_no
    e ← p.el[p.cima]
fin_si
fin_procedimiento

```

Por último el procedimiento `PBorrar` se limita a decrementar la `cima` en una unidad. Como el único elemento accesible de la pila es el elemento apuntado por `cima`, eso será suficiente para eliminarlo. No se puede borrar un elemento si la pila está vacía.

```

procedimiento PBorrar(S pila : p)
inicio
    si EsPilaVacía(p) entonces
        // Error la pila está vacía
    si_no
        p.cima ← p.cima - 1
    fin_si
fin_procedimiento

```

12.7. Diseñar las operaciones primitivas necesarias para trabajar con pilas utilizando estructuras dinámicas de datos.

Definición de las estructuras de datos

En este caso los elementos de la estructura, los nodos, se almacenan de forma dispersa por la memoria del ordenador, por lo que no es necesaria ninguna estructura auxiliar para almacenarlos. La pila será por tanto un puntero a nodo.

```

tipo
    registro : TipoElemento
    ... : ...
    ... : ...
fin_registro
puntero_a Nodo : pila
registr : Nodo
    TipoElemento : info
    pila : sig
fin_registro
var
    pila : p

```

Procedimientos y funciones

Las primitivas para trabajar con pilas utilizando estructuras dinámicas de datos son las mismas que las utilizadas para trabajar con arrays. El procedimiento `PilaVacía`, también inicializa una pila, pero en este caso haciendo que el puntero `pila` tome el valor predefinido `nulo`.

```

procedimiento PilaVacía(S pila : p)
inicio
    p ← nulo
fin_procedimiento

```

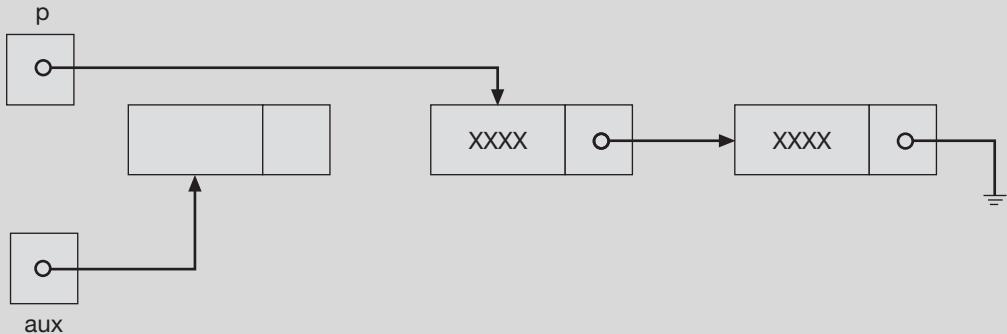
Para averiguar en este caso si la pila está vacía, lo único que se debe hacer es comprobar su valor con la constante `nulo`.

```

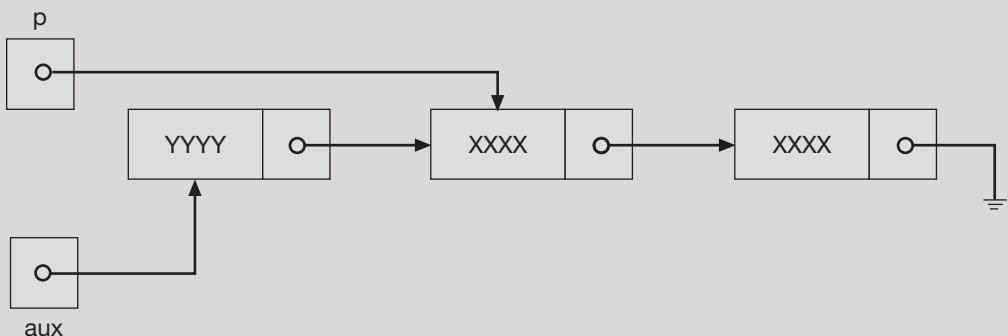
lógico función EsPilaVacia(E pila : p)
inicio
    devolver(p = nulo)
fin_función

```

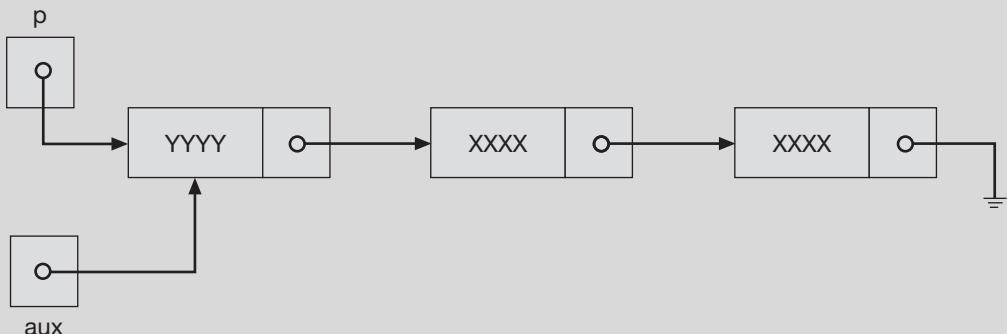
Para insertar un elemento, primero se ha de reservar espacio para un nuevo elemento mediante el procedimiento predefinido **reservar** y hacer que una variable de tipo pila apunte a dicha posición.



Después se introduce el nuevo elemento en el campo **info** del nodo, y haremos que su campo **sig** apunte a donde apunta **p**.



Por último la pila apuntará a la misma dirección de memoria a la que apunta aux.



```

procedimiento PIInsertar(E/S pila : p; E TipoElemento : e)
var
    pila : aux

```

```

inicio
    Reservar(aux)
    aux $\uparrow$ .sig  $\leftarrow$  p
    aux $\uparrow$ .info  $\leftarrow$  e
    p  $\leftarrow$  Aux
fin_procedimiento

```

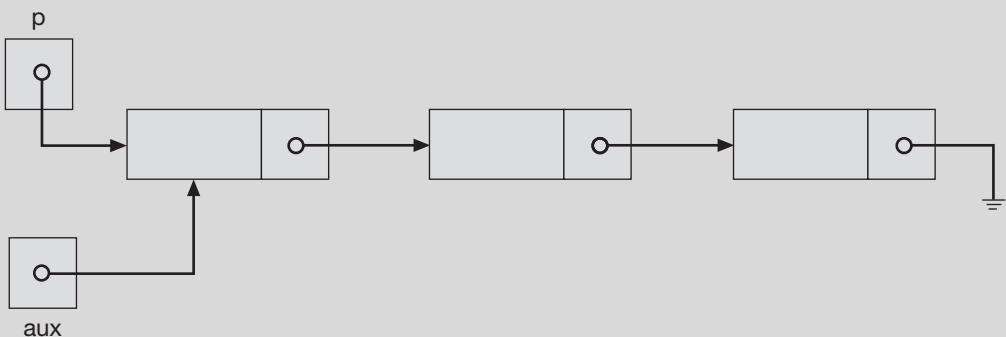
El procedimiento **Tope**, se limitará a tomar la información del campo **info** del elemento apuntado por la pila.

```

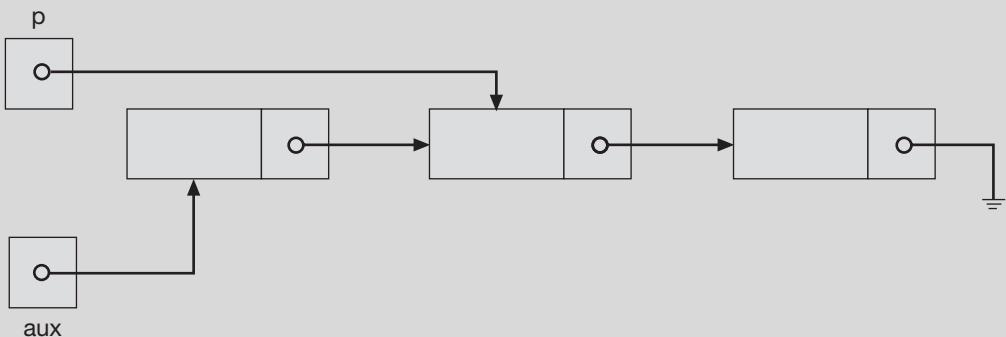
procedimiento Tope(E pila : p; S TipoElemento : e)
inicio
    si EsPilaVacía(p) entonces
        // Error, la pila está vacía
    si_no
        e  $\leftarrow$  p $\uparrow$ .info
    fin_si
fin_procedimiento

```

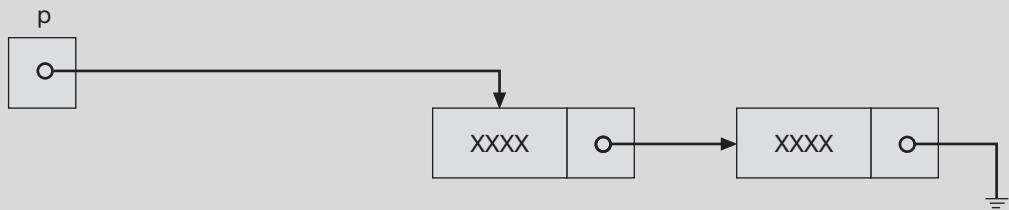
Para borrar un elemento de la pila, es necesario «puentejar» el primer elemento, es decir, hacer que **pila** apunte al nodo al que apunta el primer elemento. Para ello se utilizará una variable auxiliar (**aux**) que apunte al primer nodo.



A continuación se hace que la pila apunte al segundo elemento.



Y por ultimo, mediante el procedimiento estándar **liberar**, se libera la posición de memoria ocupada por el nodo apuntado por aux.



```

procedimiento PBorrar( E/S pila : p)
var
    pila : aux
inicio
    si EsPilaVacía(p) entonces
        // error, la pila está vacía
    si_no
        aux ← p
        p ← p↑.sig
        liberar(aux)
    fin_si
fin_procedimiento

```

- 12.8.** Diseñar las operaciones primitivas necesarias para trabajar con colas utilizando un array circular.

Definición de las estructuras de datos

Si los datos se almacenan en un array, la estructura cola será un registro compuesto por un array del tipo base de la cola (*TipoElemento*) y dos variables enteras, *principio* y *final*, la posición del primero y del último elemento de la cola. El número máximo de elementos vendrá determinado por una constante (*MáxCola*).

```

const MáxCola = ...
tipo
    registro : TipoElemento
    ... : ...
    ... : ...
fin_registro
    registro : cola
    entero : p, f
    array[1..MáxCola] de TipoElemento : el
fin_registro
var
    cola : c

```

Procedimientos y funciones

Se consideran cinco primitivas para trabajar con colas:

ColaVacia	Inicializa una cola, crea una cola como vacía.
EsColaVacia	Es una función lógica que indica si una cola está vacía.
CInsertar	Inserta un elemento en la cola.
CPrimero	Devuelve el elemento situado en la primera posición de la cola.
CBorrar	Borra el último elemento de la cola.

El procedimiento ColaVacía inicializa una cola. Como se utilizan en el ejercicio colas circulares, se considera que una cola está vacía cuando el siguiente elemento de final es igual al primer elemento. Por tanto inicializaremos principio a 1 y final a MáxCola.

```
procedimiento ColaVacía(S cola : c)
inicio
    c.p ← 1
    c.f ← MáxCola
fin_procedimiento
```

Como se ha dicho, la cola estará vacía cuando el siguiente elemento de final sea igual a principio. Al tratarse de una estructura circular, el siguiente elemento no será siempre el resultado de incrementar en 1 el puntero, por lo que se desarrolla una función siguiente que realiza dicha operación.

```
entero función Siguiente(E entero : n)
inicio
    devolver(n mod MáxCola + 1)
fin_función
```

De esta forma la función EsColaVacía quedaría de la siguiente forma:

```
lógico función EsColaVacía(E cola : c)
inicio
    devolver(Siguiente(c.f) = c.p)
fin_función
```

Cada vez que se inserta un elemento se hace por el final. Por tanto, sólo hay que modificar dicho puntero (en el caso de tratarse de la primera inserción, ya hemos hecho que el puntero *c.p* apunte al primer elemento). Se presenta el problema de cómo detectar que la cola está llena. Al ser una estructura circular no sirve comprobar si se ha llegado al elemento MáxCola. Tampoco se puede ver si está llena cuando *c.f* ha llegado a *c.p*, ya que dicha condición se ha reservado para comprobar si la cola está vacía. La solución elegida será reservar un espacio «colchón» siempre entre el principio y el final, de forma que cuando sólo quede un elemento entre ambos punteros la cola estará llena, es decir, estará llena cuando *Siguiente(Siguiente(c.f))* sea igual a *c.p*.

```
procedimiento CInsertar(E/S cola : c; E TipoElemento : e)
inicio
    si Siguiente(Siguiente(c.f)) = c.p entonces
        // Error, la cola está llena
    si_no
        c.f ← Siguiente(c.f)
        c.el[c.f] ← e
    fin_si
fin_procedimiento
```

Con el procedimiento Primero se accede al primer elemento de la cola. Para ello, después de comprobar si la cola está vacía, simplemente se extrae la información del elemento del array al que apunta el puntero *c.p*.

```
procedimiento Primero(E cola : c; S TipoElemento : e)
inicio
    si EsColaVacía(c) entonces
        // Error, la cola está vacía
```

```

si_no
    e ← c.el[c.p]
fin_si
fin_procedimiento

```

Por último, para borrar un elemento, sólo es necesario hacer que `c.p` —el puntero que apunta al único elemento que se puede borrar— señale a su elemento siguiente.

```

procedimiento CBorrar(E/S cola : c)
inicio
    si EsColaVacía(c) entonces
        // Error, la cola está vacía
    si_no
        c.p ← Siguiente(c.p)
    fin_si
fin_procedimiento

```

12.9. Diseñar las operaciones primitivas necesarias para trabajar con colas utilizando estructuras dinámicas de datos.

Definición de las estructuras de datos

La cola necesita dos punteros para mantener la posición del último y del primer elemento de la estructura. Por tanto será un registro compuesto por dos punteros a nodo que serán `p` y `f`.

```

tipo
    puntero_a nodo : ptr
    registro : nodo
        ... : ...
        ... : ...
fin_registro
    registro : cola
        ptr : p, f
fin_registro
var
    cola : c

```

Procedimientos y funciones

Para inicializar la cola (procedimiento `ColaVacía`) se debería poner a `nulo` los dos componentes de la cola.

```

procedimiento ColaVacía(S cola : c)
inicio
    c.p ← nulo
    c.f ← nulo
fin_procedimiento

```

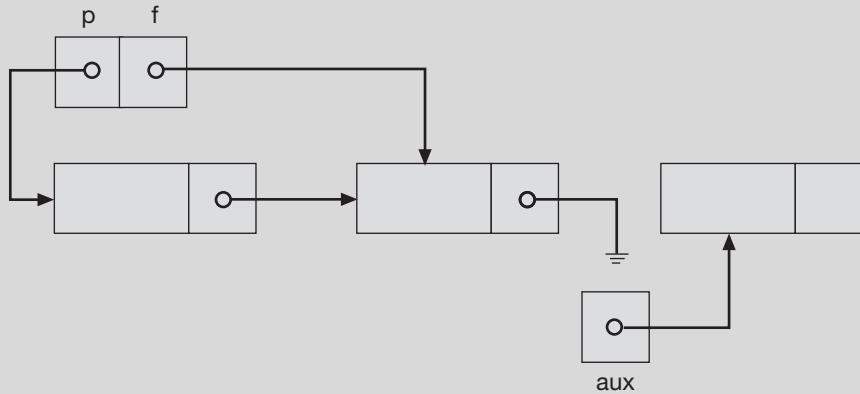
Se sabrá si la cola está vacía si el puntero `c.p` o `c.f` están a nulo.

```

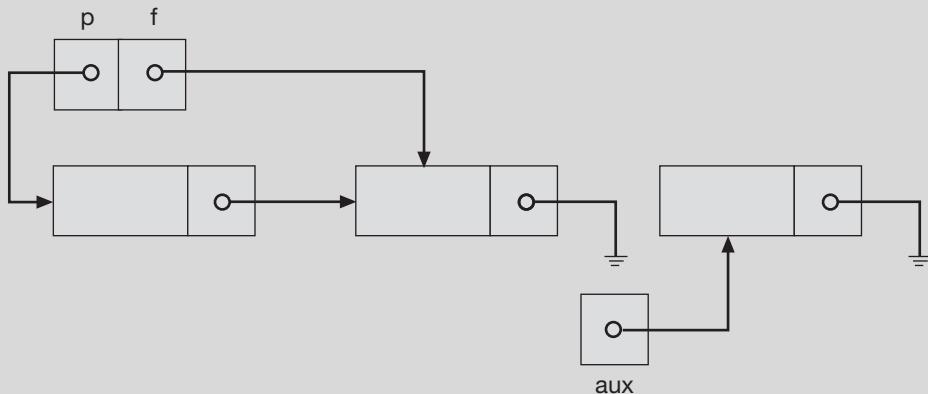
lógico función EsColaVacía(E cola : c)
inicio
    devolver(c.f = nulo)
fin_función

```

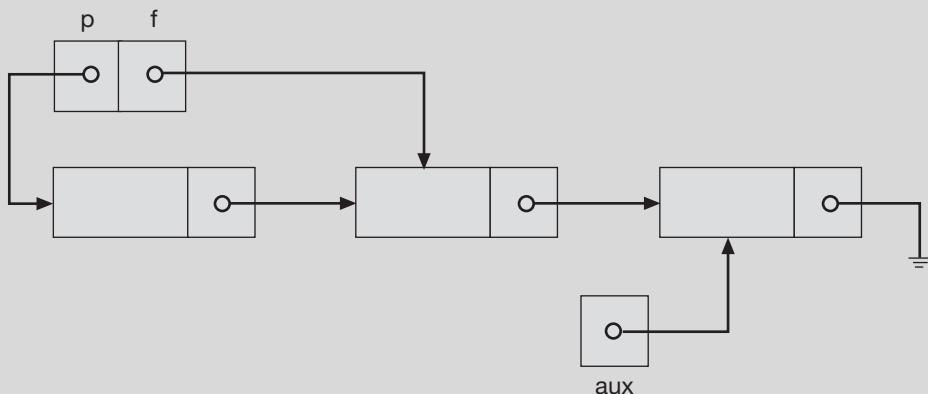
Para insertar un elemento, primero hay que reservar espacio para él.



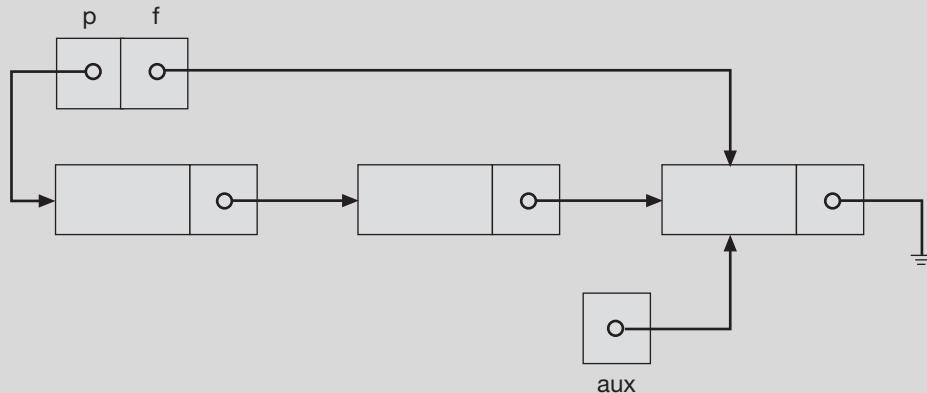
Después se introduce en el campo `info` el elemento a insertar y, como se trata del último elemento de la estructura, se pone su campo `sig` a **nulo**.



Si la cola está vacía, es decir, sin el primer elemento de la estructura, es necesario hacer que `c.p` apunte también a ese nuevo elemento. En caso contrario el campo `sig` del último elemento, es decir, el nodo apuntado por `c.f`, debe apuntar al nuevo elemento.



Por último, `c.f` siempre deberá apuntar al nuevo elemento.

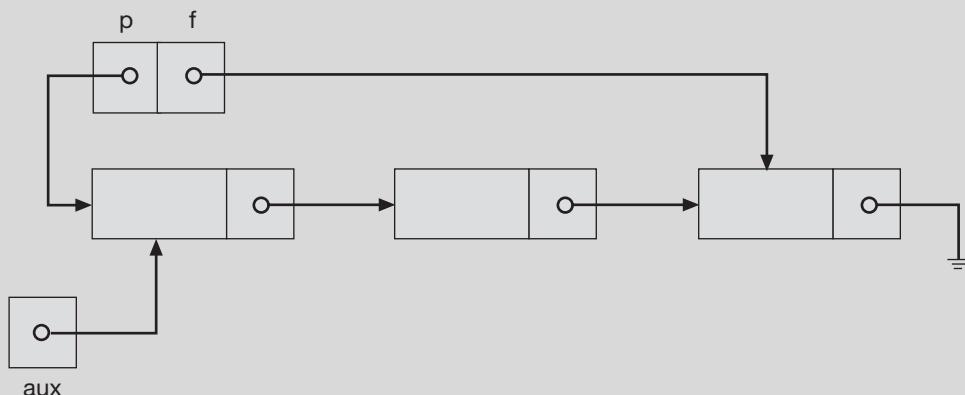


```
procedimiento CInsertar(E/S cola : c; E TipoElemento : e)
var
    ptr : aux
inicio
    Reservar(aux)
    aux↑.sig ← nulo
    aux↑.info ← e
    si EsColaVacia(c) entonces
        c.p ← aux
    si_no
        c.f↑.sig ← aux
    fin_si
    c.f ← aux
fin_procedimiento
```

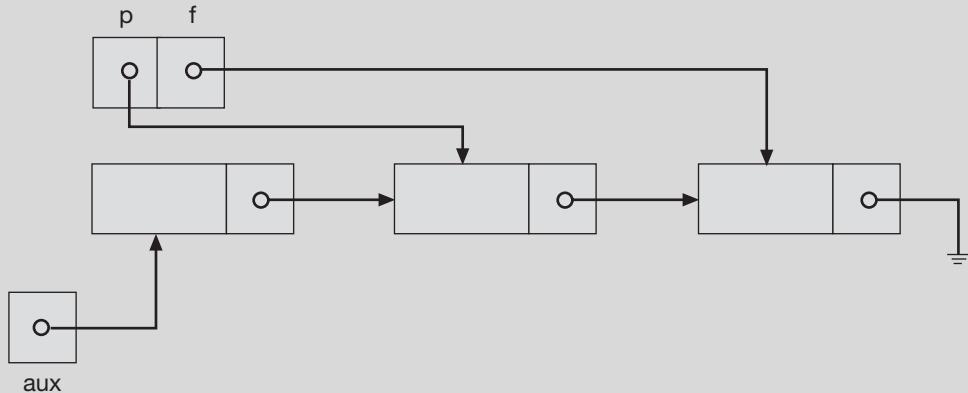
Al primer elemento se accede simplemente mediante el campo `info` del elemento apuntado por `c.p`.

```
procedimiento Primero(E cola : c; S TipoElemento : e)
inicio
    si EsColaVacia(p) entonces
        // Error, la cola está vacía
    si_no
        e ← c.p↑.info
    fin_si
fin_procedimiento
```

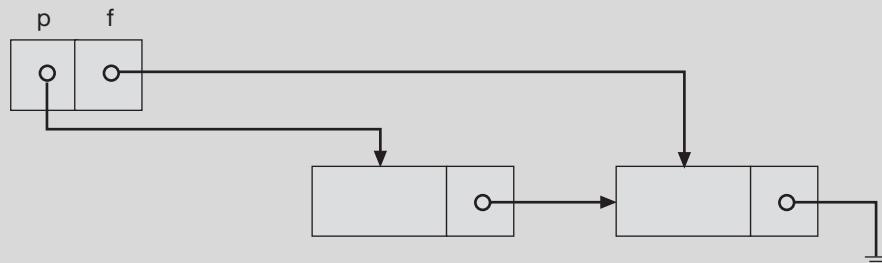
Por último, para borrar un elemento, se asigna a una variable auxiliar la dirección del nodo a borrar (el apuntado por `c.p`).



Después $c.p$ debe apuntar al nodo al que apunta el campo sig del primer elemento.



Por último hay que liberar la posición de memoria ocupada por el nodo apuntado por aux . Además, si la cola se ha quedado vacía (si $c.p$ es **nulo**), se debe hacer que también $c.f$ sea **nulo**.



```

procedimiento CBorrar(E/S cola : c)
var
    ptr : aux
inicio
    si EsColaVacia(c) entonces
        // Error, la cola está vacía
    si_no
        aux ← c.p
        c.p ← c.p.sig
        disponer(aux)
        si c.p = nulo entonces
            c.f ← nulo
        fin_si
    fin_procedimiento

```

12.10. Diseñar un procedimiento que realice una copia de una pila en otra.

Análisis del problema

Se entiende por copiar llenar otra pila con los mismos elementos y en el mismo orden. Por tanto, si simplemente se sacan los elementos de la pila y se meten en otra, tendrá los mismos elementos, pero en orden distinto. Hay dos soluciones, una sería utilizar una pila auxiliar: se sacan los elementos de la pila principal y se

meten en la auxiliar, para después volcarlos en la pila de salida. La otra solución sería recursiva: se sacan los elementos de la pila mediante llamadas recursivas; cuando la pila esté vacía se inicializa la pila copia y a la vuelta de la recursividad se van introduciendo los elementos en orden inverso a como han salido.

Observe que el procedimiento valdrá tanto para la implementación con arrays como con estructuras dinámicas de datos. Sin embargo en este segundo caso, al llamar al procedimiento PBorrar, se liberan los nodos de la memoria, con lo que se pierde la información de éstos. En este caso el procedimiento PBorrar, no debería incluir la llamada al procedimiento estándar **liberar** para no perder la información. En el caso de los arrays, al pasar la pila original como entrada no se pierde su información.

Diseño del algoritmo

Solución iterativa

```
procedimiento CopiarPila(E pila : p; S pila : copia)
var
    pila : aux
    TipoElemento : e
inicio
    PilaVacía(aux)
    mientras no EsPilaVacía(p) hacer
        Tope(p,e)
        PInsertar(aux,e)
        PBorrar(p)
    fin_mientras
    PilaVacía(copia)
    mientras no EsPilaVacía(aux) hacer
        Tope(aux,e)
        PInsertar(copia,e)
        PBorrar(aux)
    fin_mientras
fin_procedimiento
```

Solución recursiva

```
procedimiento CopiarPila(E pila : p; S pila : copia)
var
    TipoElemento : e
inicio
    si no EsPilaVacía(p) hacer
        Tope(p,e)
        PBorrar(p)
        CopiarPila(p,salida)
        PInsertar(salida,e)
    si_no
        PilaVacía(salida)
    fin_si
fin_procedimiento
```

12.11. Diseñar un procedimiento que elimine el elemento nsimo de una pila.

Análisis del problema

También en este caso se debe utilizar una pila auxiliar o recursividad para poder restaurar los elementos en el mismo orden. Es necesario borrar elementos e insertarlos en la pila auxiliar hasta llegar al elemento n. En ese

punto, se sacan todos los elementos de la pila auxiliar y se introducen en la pila original. Obsérvese que en este caso es totalmente indistinto utilizar estructuras de datos dinámicas o estáticas.

Diseño del algoritmo

Solución iterativa

```
procedimiento BorrarElementoN(s pila : p; E entero : n)
var
    pila : aux
    TipoElemento : e
    entero : i
inicio
    i ← 1
    PilaVacía(aux)
    mientras no EsPilaVacía(p) y i < n hacer
        i ← i + 1
        Tope(p, e)
        PInsertar(aux, e)
        PBorrar(p)
    fin_mientras
    PBorrar(p)
    mientras no EsPilaVacía(aux) hacer
        Tope(aux, e)
        PInsertar(p, e)
        PBorrar(aux)
    fin_mientras
fin_procedimiento
```

Solución recursiva

```
procedimiento BorrarElementoN(s pila : p; E entero : n)
var
    TipoElemento : e
inicio
    si i > 0 y no EsPilaVacía(p) entonces
        Tope(p, e)
        PBorrar(p)
        BorrarElementoN(p, n-1)
        PInsertar(p, e)
    si_no
        PBorrar(p)
    fin_si
fin_procedimiento
```

- 12.12.** Diseñar un algoritmo para que, utilizando pilas y colas, compruebe si una frase es un palíndromo (un palíndromo es una frase que se lee igual de izquierda a derecha que de derecha a izquierda).

Análisis del problema

Se puede aprovechar el distinto orden en que salen los elementos de una pila y una cola para averiguar si una frase es igual a ella misma invertida. Para ello, una vez introducida la frase, se insertan en una pila y una cola

todos los caracteres (se evitarán los signos de puntuación, y se podría mejorar si se convierten todos los caracteres a mayúsculas o minúsculas y se eliminan los acentos).

A continuación, se van sacando elementos de la pila y de la cola. Si aparece algún carácter distinto, es que la frase no es igual a ella misma invertida, y por tanto no será un palíndromo. Si al acabar de sacar los elementos, todos han sido iguales, se trata de un palíndromo.

Diseño del algoritmo

```

algoritmo Ejercicio_12_12
    // Aquí deberían incluirse las declaraciones, procedimientos y
    // funciones para trabajar con pilas y colas.
    // Es indistinto trabajar con estructuras estáticas o dinámicas.
    tipo
        carácter : TipoElemento
    var
        pila : p
        cola : c
        carácter : car1, car2
        cadena : frase
        entero : i
    inicio
        ColaVacía(c)
        PilaVacía(p)
        leer(frase)
        desde i ← 1 hasta longitud(frase) hacer
            car1 ← subcadena(frase, i , 1)
            // si no es un signo de puntuación
            si posición(car1, ' ,.;.' ) = 0 entonces
                CInsertar(c,car1)
                PInsertar(p,car1)
            fin_si
        fin_desde
        repetir
            Primero(c,car1)
            Tope(p,car2)
            PBorrar(p)
            CBorrar(c)
            hasta_que car1 <> car2 o EsColaVacía(c)
            si car1 = car2 entonces
                escribir('Es un palíndromo')
            si_no
                escribir('No es un palíndromo')
            fin_si
        fin
    
```

13

ESTRUCTURAS DE DATOS NO LINEALES (ÁRBOLES Y GRAFOS)

@Ug`Yghfi Wi fUg`Xjbza Jwlg`jbYUYg`XY`Xlhcg` `]ghlUg`Yb`UhUXUg`d]`Ug`mW`Ug` h]YbYb`[fUbXYg`j Yb! hu`Ug`XY`Z`Yi`Jv`]JUX`geVfY`Ug`fYdfYgYbhUWcbYg`Wbh`[i`Ug`/g`b`Ya`VUf`[c`h]YbYb`i`b`di`bhc`XfV`]. gcb` `]ghlUg`gYWYbWU`Yg`Yg`XYWf`z`Yghzb`Xlgdi`YghlUg`XY`a`cXc`ei`Y`Yg`bYWgUf`c`a`cj`Yfg`Y`Uhf`U`fg`XY`Y`Ug`i`bU`dcg`W`Q`b`W`XU`j`Yn`fW`XU`Y`Ya`Ybhc`h]YbY`i`b`g`[i`]YbhY`Y`Ya`Ybhcg`"9b`YghY`W`d`hi`c`g`Y`hf`U` hUfzb` `Ug`Yghfi`Wi`fUg`XY`Xlhcg`bc` `]bYUYg`ei`Y`f`Yg`Y`j`Yb` `cg`df`cV`Ya`Ug`ei`Y`d`UbhYUb` `Ug` `]ghlUg` `!` bYUYg`m`Yb` `Ug`ei`Y`W`XU`Y`Ya`Ybhc`di`YXY`h`YbYf`X`ZYf`YbhYg`Ag`[i`]YbhYg`A`Y`Ya`Ybhcg`" @Ug`a`Yb`W`c` bUXUg`Yghfi`Wi`fUg`gcb` `zf`Vc`Yg`W`UbXc`W`XU`Y`Ya`Ybhc`h]YbY`i`b`•`b`W`df`YXY`W`gcf`d`Yfc`di`YXY`h`YbYf` X`ZYf`YbhYg`gi`W`gcf`Yg`m`/`f`U`Z`cg`g`]`W`XU`Y`Ya`Ybhc`di`YXY`h`YbYf`j`Uf`]cg`df`YXY`W`gcf`Yg`m`j`Uf`]cg`gi`! W`gcf`Yg`

13.1. ÁRBOLES

Las estructuras de tipo árbol se usan para representar datos con una relación jerárquica entre sus elementos. La definición de árbol implica una naturaleza recursiva, ya que un árbol o es vacío o se considera formado por un nodo raíz y un conjunto disjunto de árboles que se llaman subárboles del raíz. Se puede representar un árbol de varias maneras, tal y como aparece en las Figuras 13.1 y 13.2:

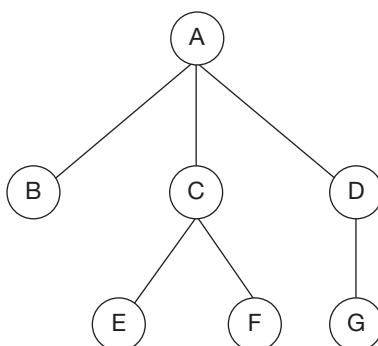


Figura 13.1. Representación gráfica de árboles mediante árboles.

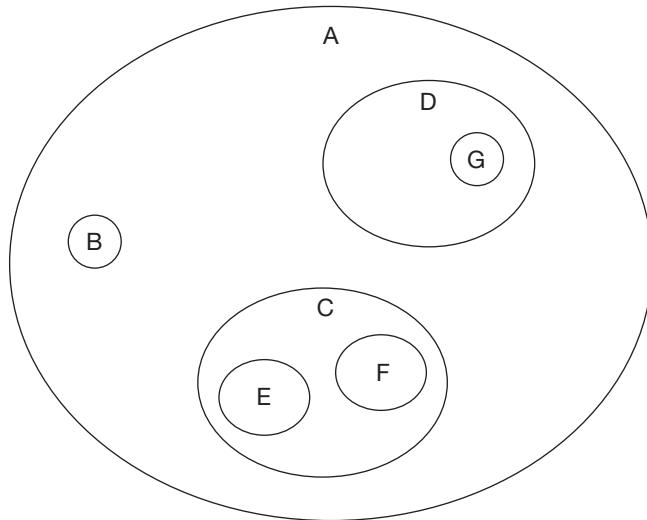


Figura 13.2. Representación gráfica de árboles mediante conjuntos.

13.1.1. Terminología

La estructura de datos *árbol* requiere una terminología específica, entre la que es preciso destacar la siguiente:

- **Nodos** son los elementos o vértices del árbol.
- Todo árbol que no está vacío tiene un **nodo raíz**, que es aquél del cual derivan o descienden todos los demás elementos del árbol.
- Cada nodo tiene un único antecesor o ascendiente denominado **padre**, excepto el nodo raíz.
- Cualquier nodo, incluido el raíz puede tener varios descendientes, denominados *hijos*, que salen de él.
- Se llama **grado de un nodo** al número de hijos que salen de él y a los nodos con grado 0 se les denomina *nodos terminales u hojas*.
- Dos nodos hijos del mismo padre reciben el nombre de **hermanos**.
- Cada nodo de un árbol tiene asociado un número de **nivel** que se determina por el número de antecesores que tiene desde la raíz, teniendo en cuenta que el nivel de la raíz es 1.
- **Profundidad o altura** de un árbol es el máximo de los niveles de los nodos del árbol.
- **Peso** de un árbol es el número de nodos terminales.
- Una colección de dos o más árboles se llama **bosque**.

13.1.2. Arboles binarios

Arbol binario es aquél en el que cada nodo tiene como máximo el grado 2. Un árbol binario es *equilibrado* si para todos sus nodos la altura de sus dos subárboles se diferencia en 1 como máximo y es *completo* si todos sus nodos, excepto las hojas, tienen exactamente dos hijos. Un árbol binario en el que todos los nodos tienen dos hijos excepto las hojas y éstas tienen todas el mismo nivel se dice que está lleno. El número de nodos de un árbol binario lleno será: $2^{\text{altura} - 1}$.

Dos árboles binarios son *distintos* cuando sus estructuras son diferentes. Dos árboles binarios son *similares* cuando sus estructuras son idénticas, pero la información que contienen los nodos es dife-

rente entre uno y otro. Dos árboles binarios son *iguales* o *equivalentes* cuando son similares y además los nodos contienen la misma información.

13.1.2.1. Conversión de un árbol general en árbol binario

Existe un método para convertir un árbol general en binario:

- La raíz del árbol binario será la raíz del árbol *n-ario*.
- Se deja enlazado el nodo raíz con el que tenga más a la izquierda y se enlaza éste con sus hermanos.
- Se repite este segundo apartado con los nodos de los sucesivos niveles hasta llegar al nivel más alto.
- Se gira el diagrama 45 grados hacia la izquierda.

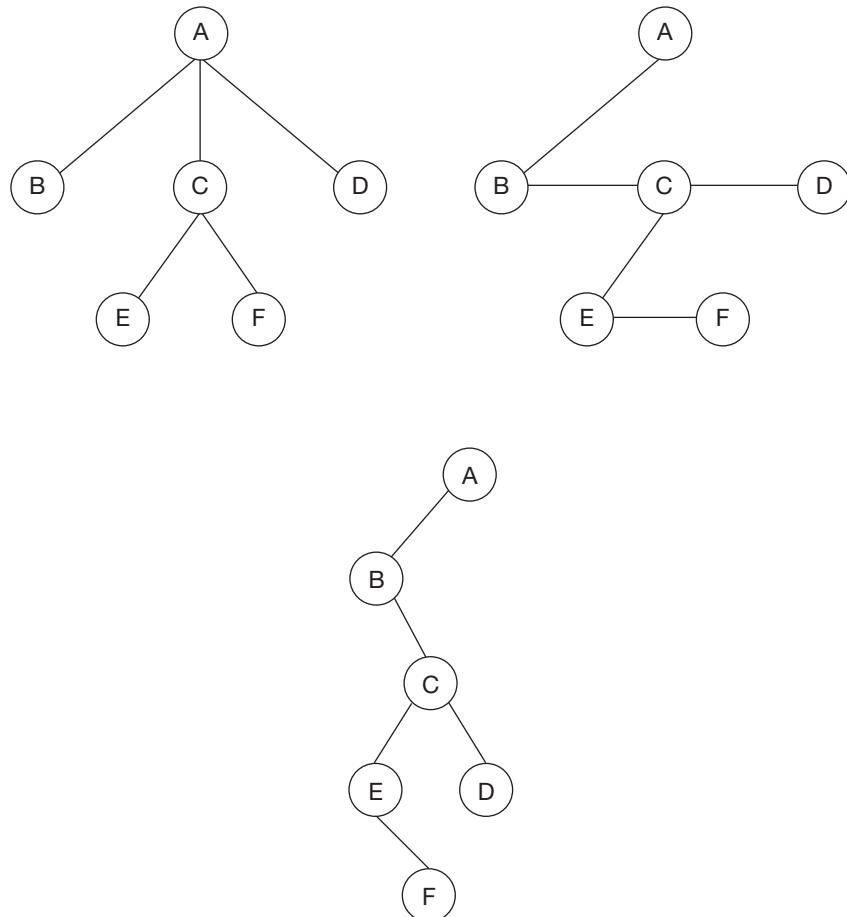


Figura 13.3. Conversión de árboles generales en árboles binarios.

Un algoritmo muy parecido es el que se aplica en la conversión de un bosque de árboles generales en un único árbol binario. La única modificación sería que deberíamos comenzar enlazando en forma horizontal las raíces de los distintos árboles generales.

13.1.2.2. Implementación

Los árboles se deben implementar como estructuras dinámicas; es decir, en Java y C#, vinculando objetos con miembros que referencian otros objetos y, en otros lenguajes, mediante punteros, que permiten adquirir posiciones de memoria cuando se necesitan durante la ejecución de un programa y liberarlas cuando ya no hacen falta. No obstante, también es posible implementar árboles mediante arrays.

Como un árbol binario es un árbol donde cada nodo tiene como máximo el grado 2, para su representación interna se puede imaginar cada nodo de un árbol binario como un registro constituido por tres campos: uno con información y otros dos que apuntan a los hijos izquierdo y derecho.

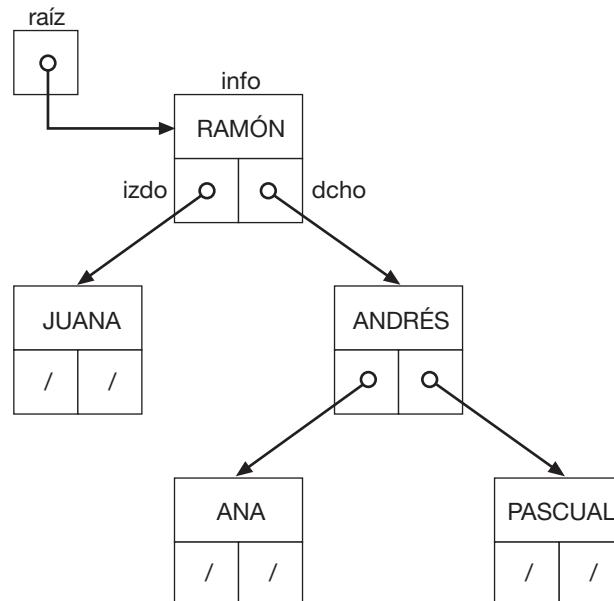


Figura 13.4. Implementación de un árbol binario con punteros.

La definición de las estructuras de datos necesarias para realizar un árbol binario utilizando punteros es la siguiente:

```

tipo
puntero_a nodo: punt
registro : tipo_elemento
... : ...
... : ...
fin_registro
registro : nodo
  tipo_elemento : elemento
  punt           : izdo,dcho
fin_registro
var
punt : raíz
tipo_elemento : elemento
  
```

Para su implementación mediante arrays podríamos utilizar un array de registros con la siguiente estructura:

	info	izdo	dcho
raíz = 3			
1			
2	PASCUAL	0	0
3	RAMÓN	5	9
4			
5	JUANA	0	0
6			
7	ANA	0	0
8			
9	ANDRÉS	7	2

Implementación de un árbol binario mediante arrays.

Las estructuras de datos necesarias serían las siguientes:

```

const
    Máx = <expresión>
tipo
    registro: TipoElemento
        .... : ....
        .... : ....
    fin_registro
    registro: TipoNodo
        TipoElemento : Elemento
        entero : Izdo,Dcho
        // Izdo y Dcho son campos numéricicos para indicar la posición
        // en que están los hijos izquierdo y derecho
    fin_registro
    array[1..Máx] de TipoNodo : Arr
var
    Arr : m
    TipoElemento : elemento
    entero : raíz
  
```

13.1.2.3. Recorridos de un árbol binario

Se denomina recorrido al proceso que permite acceder una sola vez a cada uno de los nodos del árbol. Existen tres formas de efectuar el recorrido, todas ellas de naturaleza recursiva:

- Preorden** Visitar la raíz, recorrer el subárbol izquierdo, recorrer el subárbol derecho.
- Inorden** Recorrer el subárbol izquierdo, visitar la raíz, recorrer el subárbol derecho.
- Postorden** Recorrer el subárbol izquierdo, recorrer el subárbol derecho, visitar la raíz.

Considérese el árbol binario de la Figura 13.5:

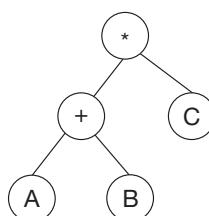


Figura 13.5. Árbol de expresión.

Este árbol representa una expresión algebraica. El orden de operación queda establecido por la ubicación de los operadores con respecto a los operandos. El recorrido preorden produce la *notación polaca prefija*, el recorrido inorden la *notación convencional* (sin los paréntesis que indican la precedencia de los distintos operadores) y el recorrido postorden la *notación polaca postfija*. Recordar que una expresión en notación postfija lleva el operador después que los operandos, mientras que en las expresiones prefijas el operador se escribe antes que los operandos.

Por ejemplo, la expresión en notación convencional $a + b * c$, en forma postfija sería

a b c * +

mientras que la expresión $(a + b) * c$ en forma postfija quedaría

a b + c *

13.1.2.4. Árbol binario de búsqueda

Un tipo especial de árbol binario es el denominado *árbol binario de búsqueda*, en el que para cualquier nodo su valor es superior a los valores de los nodos de su subárbol izquierdo e inferior a los de su subárbol derecho.

La utilidad de este árbol se refiere a la eficiencia en la búsqueda de un nodo, similar a la búsqueda binaria, y a la ventaja que presentan los árboles sobre los arrays en cuanto a que la inserción o borrado de un elemento no requiere el desplazamiento de todos los demás.

El recorrido inorden del árbol proporciona los valores clasificados en orden ascendente. Para la inserción de un nuevo elemento en un árbol de este tipo se seguirán los siguientes pasos:

- Comparar el elemento a insertar con el nodo raíz, si es mayor avanzar hacia el subárbol derecho, si es menor hacia el izquierdo.
- Repetir el paso anterior hasta encontrar un elemento igual o llegar al final del subárbol donde debiera estar el nuevo elemento.
- Cuando se llega al final es porque no se ha encontrado y se añadirá como hijo izquierdo o derecho según sea su valor comparado con el elemento anterior.

El borrado o eliminación de un elemento requerirá, una vez buscado, considerar las siguientes posibilidades:

- Que no tenga hijos, sea hoja. Se suprime y basta.
- Que tenga un único hijo. El elemento anterior se enlaza con el hijo del que queremos borrar.
- Que tenga dos hijos. Se sustituye por el elemento inmediato inferior, situado lo más a la derecha posible de su subárbol izquierdo.

Si en un árbol binario de búsqueda se insertan una serie de elementos ordenados en forma ascendente o descendente el árbol se desequilibra y disminuye el rendimiento en las futuras operaciones de búsqueda, por lo que debe de ser reestructurado.

13.2. GRAFOS

Se define un grafo como un conjunto de vértices unidos por un conjunto de arcos, en el que se elimina la restricción que presentaban los árboles, en los que cada nodo sólo podía estar apuntado por otro, su padre (Figura 13.6).

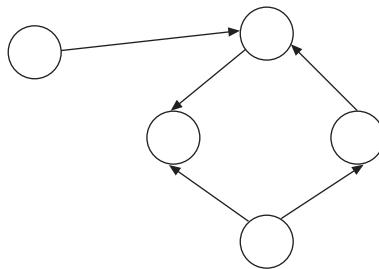


Figura 13.6. Grafo.

13.2.1. Terminología

La estructura de datos *grafo* requiere una terminología específica, entre la que es preciso destacar la siguiente:

- Al número de vértices que tiene un grafo se le llama **orden** del grafo.
- Un grafo **nulo** es un grafo de orden 0.
- Los **arcos** o **aristas** de un grafo se representan por los vértices que conectan.
- Dos vértices son **adyacentes** si hay un arco que los une. Deberemos considerar que existen **grafos dirigidos**, los arcos tienen dirección, y no **dirigidos**. Si el grafo es dirigido puede ocurrir que el vértice A sea adyacente al B, pero el B no sea adyacente al A.
- **Camino** es una secuencia de uno o más arcos que conectan dos nodos.
- Un grafo se denomina **conectado** cuando existe siempre un camino que une dos vértices cualquiera y **desconectado** en caso contrario.
- Un grafo es **completo** cuando cada vértice está conectado con todos y cada uno de los restantes nodos.

13.2.2. Representación de los grafos

Existen dos técnicas estándar para representar un grafo: la *matriz de adyacencia* y la *lista de adyacencia*. La matriz de adyacencia consiste en un array bidimensional en el que se representan las conexiones entre pares de vértices. Por ejemplo, dado este grafo dirigido, (Figura 13.7)

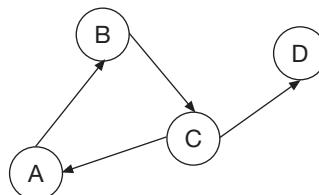


Figura 13.7. Grafo dirigido.

en su matriz de adyacencia, marcaríamos las conexiones existentes con verdad y las restantes con falso.

	A	B	C	D
A	F	V	F	F
B	F	F	V	V
C	V	F	F	V
D	F	F	F	F

Si el grafo no fuera dirigido su matriz de adyacencia resultaría simétrica, pudiéndose ocupar sólo media matriz en el caso de que no existieran lazos (un lazo es un vértice que se apunta a sí mismo). Dado el siguiente grafo no dirigido, (Figura 13.8)

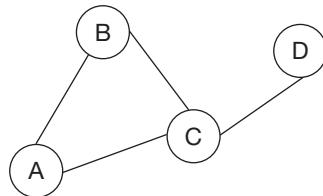


Figura 13.8. Grafo no dirigido.

su matriz de adyacencia resultante sería,

	A	B	C	D
A	F	V	V	F
B	V	F	V	F
C	V	V	F	V
D	F	F	V	F

Un grafo valorado es aquel cuyas aristas están ponderadas; en dicho caso, en las celdas de la matriz de adyacencia almacenaremos los pesos de las aristas en lugar del valor lógico que indicaba conexión. Dado el siguiente grafo valorado, (Figura 13.9)

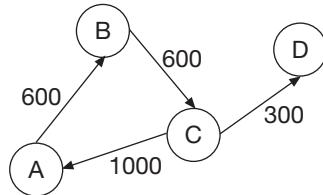


Figura 13.9. Grafo valorado.

su matriz de adyacencia sería,

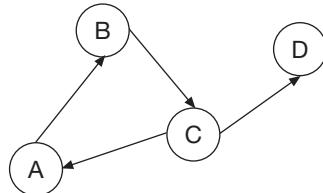
	A	B	C	D
A	0	600	1000	0
B	600	0	600	0
C	1000	600	0	300
D	0	0	300	0

Otra forma de representar grafos es mediante una lista de adyacencia. Este método es especialmente útil para representar grafos cuando éstos tienen muchos vértices y pocas aristas. Se utiliza una lista enlazada por cada vértice v del grafo que tenga otros adyacentes desde él.

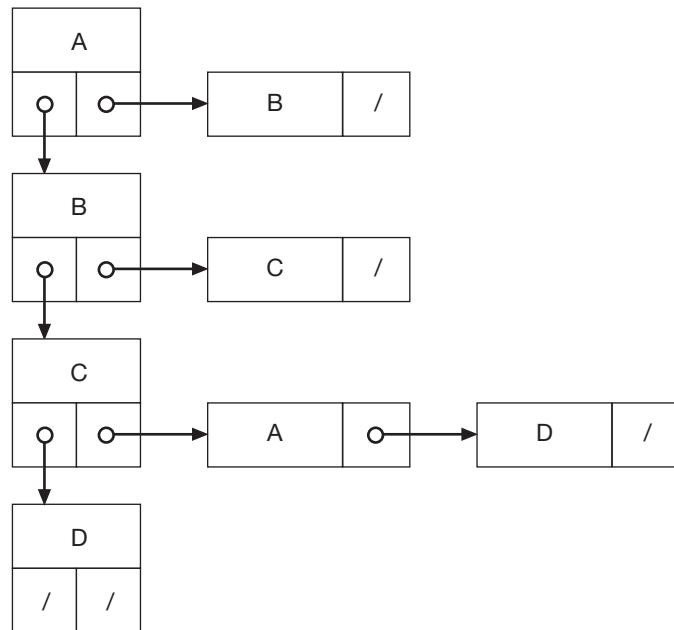
El grafo completo incluye dos partes: un directorio y un conjunto de listas enlazadas. Hay una entrada en el directorio por cada nodo del grafo. La entrada en el directorio del nodo i apunta a una lista enlazada que representa los nodos que son conectados al nodo i . Cada registro de la lista enlazada

tiene dos campos: uno es un identificador de nodo, otro es un enlace al siguiente elemento de la lista; la lista enlazada representa arcos.

Un grafo no dirigido de orden N con A arcos requiere N entradas en el directorio y $2*A$ entradas de listas enlazadas, excepto si existen bucles que reducen el número de listas enlazadas en 1. Un grafo dirigido de orden N con A arcos requiere N entradas en el directorio y A entradas de listas enlazadas. Dado el siguiente grafo dirigido,



su representación mediante una lista de adyacencia sería la siguiente,



13.3. EJERCICIOS RESUELTOS

13.1. Diseñar las operaciones primitivas para el manejo de un árbol binario de búsqueda utilizando estructuras de tipo array.

Dado que un árbol binario es un árbol donde cada nodo tiene como máximo el grado 2, se puede implementar mediante un array de registros, en el que cada registro está constituido por 3 campos:

- Elemento, de `TipoElemento`, que almacena la información.
- `Izdo`, numérico, que indica la posición del hijo izquierdo.
- `Dcho`, numérico, para indicar la posición del hijo derecho.

El valor 0 en alguno de los campos de tipo puntero —Izdo, Dcho— indicaría que no tiene el hijo correspondiente.

Definición de las estructuras de datos utilizadas

```

const
    Máx = <expresión>
tipo
    registro:      TipoElemento
        .... : ....
        .... : ....
    fin_registro
    registro:      TipoNodo
        TipoElemento : Elemento
        entero       : Izdo, Dcho
    fin_registro
    array[1..Máx] de TipoNodo : Arr
var
    Arr           : a
    TipoElemento : elemento
    entero       : raíz, vacío

```

Procedimientos y funciones

Se considera el array como una lista enlazada y se requiere una lista de vacíos y una variable vacío que apunte al primer elemento de la lista de vacíos. Para almacenar la lista de vacíos podremos utilizar cualquiera de los campos Izdo o Dcho.

Iniciar(a, raíz, vacío)

	elemento	izdo	dcho
raíz			
0			2
vacio			
1			3
			4
			5
			6
			0

```
procedimiento iniciar(S Arr: a; S entero: raíz, vacío)
```

```

var
    entero: i
inicio
    raíz ← 0
    vacío ← 1
    desde I ← 1 hasta Máx - 1 hacer
        a[i].dcho ← i+1
    fin_desde
    a[Máx].dcho ← 0
fin_procedimiento

```

```
lógico función ÁrbolVacio(E entero: raíz)
inicio
```

```

    si raíz=0 entonces
        devolver(verdad)
    si_no
        devolver(falso)
    fin_si
fin_función

lógico función ÁrbolLleno(E entero: vacío)
inicio
    si vacío=0 entonces
        devolver(verdad)
    si_no
        devolver(falso)
    fin_si
fin_función

```

Gráficamente la inserción de un primer elemento en el árbol se realizaría así:

Altas(a, elemento, raíz, vacío)

	raíz	elemento	izdo	dcho
1	1	FFFF	0	0
2				3
3				4
4				5
5				6
6				0

Como se trata de un árbol binario de búsqueda para insertar un segundo elemento habrá que fijarse en si éste es mayor o menor que el primero. Supóngase que, ahora, se desea insertar un segundo elemento menor que el primero; en este caso, se debe colocar como hijo izquierdo.

Altas(a, elemento, raíz, vacío)

	raíz	elemento	izdo	dcho
1	1	FFFF	2	0
2		CCCC	0	0
3				4
4				5
5				6
6				0

Por último, se verá que un tercer elemento, mayor que el primero habría que situarlo como hijo derecho de dicho primer elemento

Altas(a, elemento, raíz, vacío)

	raíz	elemento	izdo	dcho
1	1	FFFF	2	3
2		CCCC	0	0
3		JJJJ	0	0
4				5
5				6
6				0

Para insertar el elemento DDDD, considerado mayor que CCCC y menor que FFFF, los pasos a seguir serían:

- Recorrer la rama donde debiera estar hasta el final —FFFF, CCCC—.
- Si se llega al final y no se ha encontrado, añadirlo allí como hijo derecho o izquierdo según sea su valor comparado con el elemento anterior —se añadiría como un hijo derecho de CCCC—.

Altas(a, elemento, raíz, vacío)

	elemento	izdo	dcho
raíz	FFFF	2	3
1	CCCC	0	4
vacio	JJJJ	0	0
5	DDDD	0	0
4			6
3			
2			
6			0

```

procedimiento buscar(E Arr: A; E entero : raíz;
                      E TipoElemento: elemento; S entero:act,ant)
var
    lógico: encontrado
inicio
    Encontrado _ falso
    act <- raíz
    ant <- 0
    mientras no encontrado y (act<>0) hacer
        si igual(elemento, a[act].elemento) entonces
            encontrado <- verdad
        si_no
            ant <- act
            si mayor(a[act].elemento, elemento) entonces
                act <- a[act].Izdo
            si_no
                act <- a[act].Dcho
            fin_si
        fin_si
    fin_mientras
fin_procedimiento

procedimiento altas(E/S Arr: A; E TipoElemento:elemento;
                      E/S entero: raíz,vacío)
var
    entero: act,ant,auxi
inicio
    si vacío <> 0 entonces
        buscar(a, raíz, elemento, act, ant)
        si act <> 0 entonces
            escribir('Ese elemento ya existe')
        si_no
            auxi <- vacío
            vacío <- a[auxi].Dcho
    fin_si
fin_procedimiento

```

```

a[auxi].elemento ← elemento
a[auxi].Izdo ← 0
a[auxi].Dcho ← 0
si ant = 0 entonces
    raíz ← auxi
si_no
    si mayor(a[ant].elemento, elemento) entonces
        a[ant].Izdo ← auxi
    si_no
        a[ant].Dcho ← auxi
    fin_si
fin_si
fin_si
fin_procedimiento

```

Considérese, con la siguiente situación de partida, que se desea eliminar el elemento CCCC.

		elemento	izdo	dcho
raíz	1	FFFF	2	3
	2	CCCC	0	4
vacio	3	JJJJ	0	0
	4	DDDD	0	0
	5			6
	6			0

Los pasos a seguir serían:

- Buscar el elemento.
- Como tiene un único hijo, su anterior se enlaza con el hijo del que se desea borrar.

Considerando todas las posibilidades el procedimiento bajas habría que diseñarlo de la siguiente forma:

```

procedimiento bajas(E/S Arr: A; E TipoElemento: elemento;
                      E/S entero: raíz,vacío)
var
    entero: act,ant,auxi
inicio
    buscar(a, raíz, elemento, act, ant)
    si act = 0 entonces
        escribir('Ese elemento no existe')
    si_no
        // El nodo a eliminar es una hoja, no tiene hijos
        si (a[act].Izdo = 0) y (a[act].Dcho = 0) entonces
            si ant = 0 entonces
                raíz ← 0
            si_no
                si a[ant].Izdo = act entonces
                    a[ant].Izdo ← 0

```

```

    si_no
        a[ant].Dcho ← 0
    fin_si
    fin_si
    si_no
        // El nodo a eliminar tiene dos hijos
        si (a[act].Izdo <> 0) y (a[act].Dcho <> 0) entonces
            ant ← act
            auxi ← a[act].Izdo
            mientras a[auxi].Dcho <> 0 hacer
                ant ← auxi
                auxi ← a[auxi].Dcho
            fin_mientras
            a[act].Elemento ← a[auxi].Elemento
            si ant = act entonces
                a[ant].Izdo ← a[auxi].Izdo
            si_no
                a[ant].Dcho ← a[auxi].Izdo
            fin_si
            act ← auxi
        si_no
            // El nodo a eliminar tiene hijo derecho
            si a[act].Dcho <> 0 entonces
                si ant = 0 entonces
                    raíz ← a[act].Dcho
                si_no
                    si a[ant].Izdo = act entonces
                        a[ant].Izdo ← a[act].Dcho
                si_no
                    a[ant].Dcho ← a[act].Dcho
                fin_si
            fin_si
        si_no
            // El nodo a eliminar tiene hijo izquierdo
            si ant = 0 entonces
                raíz ← a[act].Izdo
            si_no
                si a[ant].Dcho = act entonces
                    a[ant].Dcho ← a[act].Izdo
                si_no
                    a[ant].Izdo ← a[act].Izdo
                fin_si
            fin_si
        fin_si
    fin_si
    // Incluir un nuevo elemento en la lista de
    a[act].Dcho ← vacío
    vacío ← act
fin_si
fin_procedimiento

```

Por último, tres procedimientos que permiten recorrer el árbol en inorden, preorden y postorden.

```

procedimiento inorden(E Arr: a; E entero: raíz)
inicio
    si raíz<>0 entonces
        inorden(a,a[raíz].Izdo)
        llamar_a proc_escribir(a[raíz].elemento)
        inorden(a,a[raíz].Dcho)
    fin_si
fin_procedimiento
procedimiento preorden(E Arr: a; E entero: raíz)
inicio
    si raíz<>0 entonces
        llamar_a proc_escribir(a[raíz].elemento)
        preorden(a,a[raíz].Izdo)
        preorden(a,a[raíz].Dcho)
    fin_si
fin_procedimiento

procedimiento postorden(E Arr: a; E entero: raíz)
inicio
    si raíz<>0 entonces
        postorden(a,a[raíz].Izdo)
        postorden(a,a[raíz].Dcho)
        llamar_a proc_escribir(a[raíz].elemento)
    fin_si
fin_procedimiento

```

- 13.2.** Diseñar las operaciones primitivas para el manejo de un árbol binario de búsqueda utilizando estructuras dinámicas.

Definición de las estructuras de datos utilizadas

```

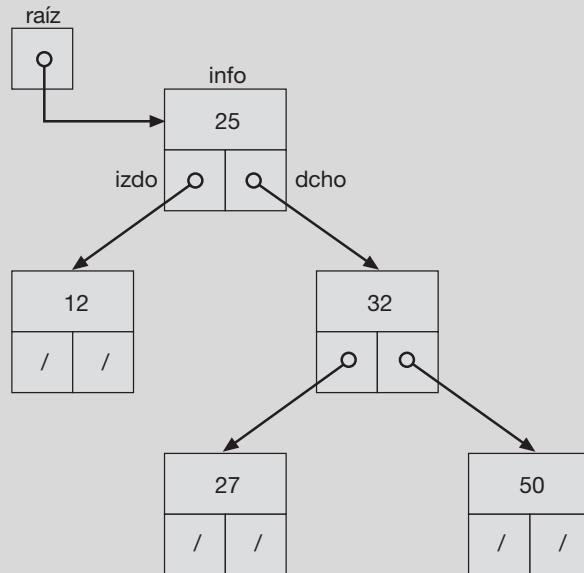
tipo
    puntero_a nodo: punt
    registro : tipo_elemento
    ... : ...
    ... : ...
fin_registro
    registro : nodo
    tipo_elemento : elemento
    punt         : izdo,dcho
fin_registro
var
    raíz : punt
    tipo_elemento : elemento

```

Procedimientos y funciones

```
Altas(raíz,elemento)
```

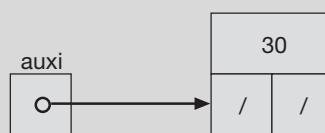
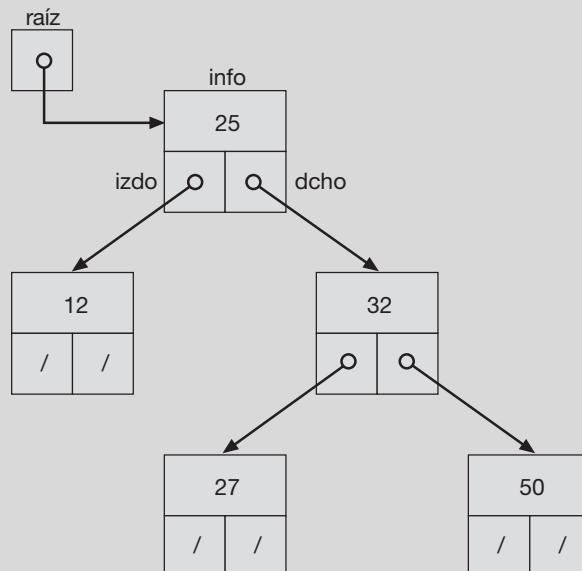
1. Situación de partida



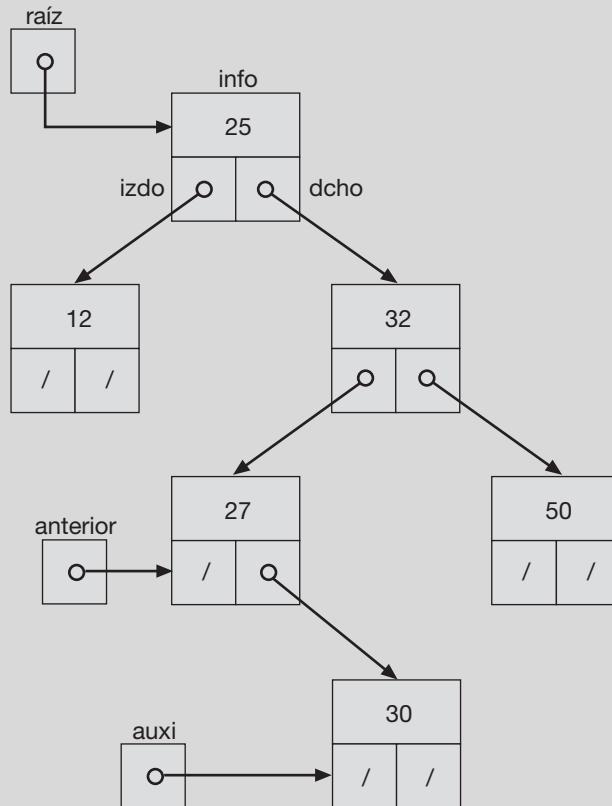
2. **reservar**(auxi)

```

auxi↑.elemento ← elemento
auxi↑.Izdo ← nulo
auxi↑.Dcho ← nulo
  
```



3. Situación final



```

procedimiento buscar( E punt: raíz;   E tipo_elemento: elemento;
                      S punt: actual,anterior)
var
    lógico: encontrado
inicio
    encontrado ← falso
    anterior ← nulo
    actual ← raíz
mientras no encontrado y (actual <> nulo) hacer
    si igual(actual↑.Elemento , elemento) entonces
        encontrado ← verdad
    si_no
        anterior ← actual
        si mayor(actual↑.Elemento , elemento) entonces
            actual ← actual↑.Izdo
        si_no
            actual ← actual↑.Dcho
    fin_si
    fin_si
fin_mientras
fin_procedimiento

```

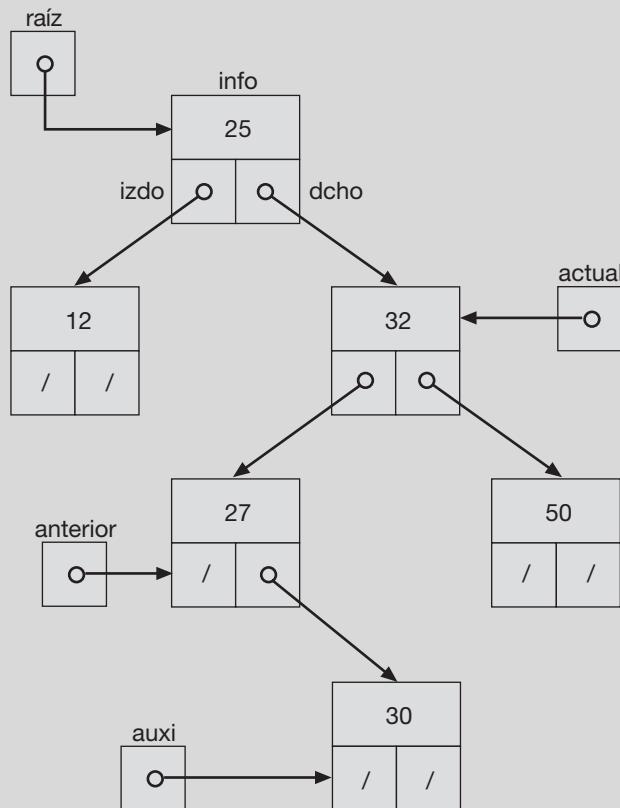
```

procedimiento altas(E/S punt: raíz;  E tipo_elemento: elemento)
var
    punt: auxi,actual,anterior
inicio
    buscar(raíz,elemento,actual,anterior)
    si actual <> nulo entonces
        escribir('Elemento duplicado')
    si_no
        reservar(auxi)
        auxi↑.Elemento ← elemento
        auxi↑.Izdo ← nulo
        auxi↑.Dcho ← nulo
        si anterior=nulo entonces
            raíz ← auxi
        si_no
            si mayor(anterior↑.Elemento, elemento) entonces
                anterior↑.Izdo ← auxi
            si_no
                anterior↑.Dcho ← auxi
            fin_si
        fin_si
    fin_si
fin_procedimiento

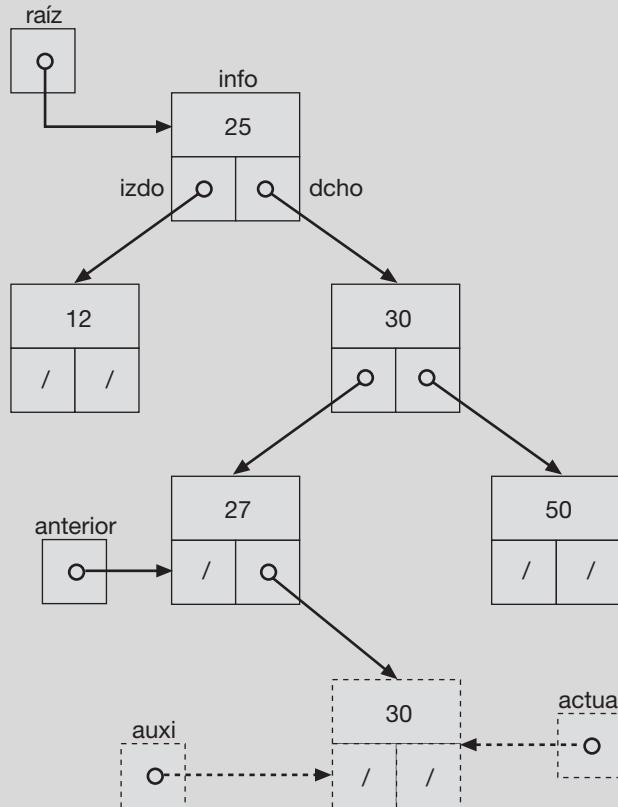
```

Bajas(raíz, 32)

1. Se desea dar de baja al 32 y, para ello, los punteros deben llegar a colocarse como se muestra en el gráfico.



2. Situación final



```

procedimiento bajas (E/S punt: raíz; E tipo_elemento: elemento)
var
  punt: anterior,actual,auxi
inicio
  buscar(raíz,elemento,actual,anterior);
  si actual = nulo entonces
    escribir('No existe')
  si_no
    si (actual^.Izdo=nulo) y (actual^.Dcho=nulo) entonces
      si anterior=nulo entonces
        raíz ← nulo
      si_no
        si anterior^.Dcho=actual entonces
          anterior^.Dcho ← nulo
        si_no
          anterior^.Izdo ← nulo
        fin_si
      fin_si
    si_no
      si (actual^.Dcho <> nulo) y (actual^.Izdo <> nulo) entonces
        // Caso de nuestro ejemplo gráfico
  
```

```

anterior ← actual
auxi ← actual↑.Izdo
mientras auxi↑.Dcho <> nulo hacer
    anterior ← auxi
    auxi ← auxi↑.Dcho
fin_mientras
actual↑.Elemento ← auxi↑.Elemento
si anterior = actual entonces
    anterior↑.Izdo ← auxi↑.Izdo
si_no
    anterior↑.Dcho ← auxi↑.Izdo
    // Caso de nuestro ejemplo gráfico
fin_si
actual ← auxi
si_no
    si actual↑.Dcho <> nulo entonces
        si anterior = nulo entonces
            raíz ← actual↑.Dcho
        si_no
            si actual = anterior↑.Dcho entonces
                anterior↑.Dcho ← actual↑.Dcho
            si_no
                anterior↑.Izdo ← actual↑.Dcho
            fin_si
        fin_si
    si_no
        si anterior = nulo entonces
            raíz ← actual↑.Izdo
        si_no
            si actual = anterior↑.Dcho entonces
                anterior↑.Dcho ← actual↑.Izdo
            si_no
                anterior↑.Izdo ← actual↑.Izdo
            fin_si
        fin_si
    fin_si
fin_si
liberar(actual)
fin_procedimiento

```

Los tres procedimientos recursivos para recorrer el árbol serían los siguientes:

```

procedimiento inorden(E punt: raíz)
inicio
    si raíz <> nulo entonces
        inorden(raíz↑.Izdo)
        llamar_a proc_escribir(raíz↑.elemento)
        inorden(raíz↑.Dcho)

```

```

fin_si
fin_procedimiento

procedimiento preorden(E punt: raíz)
inicio
    si raíz <> nulo entonces
        llamar_a proc_escribir(raíz↑.elemento)
        preorden(raíz↑.Izdo)
        preorden(raíz↑.Dcho)
    fin_si
fin_procedimiento

procedimiento postorden(E punt: raíz)
inicio
    si raíz <> nulo entonces
        postorden(raíz↑.Izdo)
        postorden(raíz↑.Dcho)
        llamar_a proc_escribir(raíz↑.elemento)
    fin_si
fin_procedimiento

```

- 13.3.** *Teniendo en cuenta que nuestro lenguaje de programación no maneja estructuras dinámicas de datos escribir un procedimiento que inserte un nuevo nodo en un árbol binario en el lugar correspondiente según su valor. Escribir otro procedimiento que permita conocer el número de nodos de un árbol binario. Utilizar ambos procedimientos desde un algoritmo que cree el árbol y nos informe sobre su número de nodos.*

Análisis del problema

El procedimiento de inserción será análogo al de altas que aparece en el ejercicio 13.1. Para conocer el número de nodos del árbol se realiza su recorrido, por uno cualquiera de los métodos ya comentados —inorden, preorden, postorden— y se irán contando.

El programa principal comenzará con un proceso de inicialización, a continuación utilizará una estructura repetitiva que permita la inserción de número indeterminado de nodos en el árbol y, por último, llamará al procedimiento para contarlos.

Al no especificarse en el enunciado el tipo de información que se almacena en los registros del árbol, ésta se tratará de forma genérica, recurriendo a procedimientos y funciones auxiliares no desarrollados que permitan manipularla, como por ejemplo leerelemento(elemento), escribirelemento(elemento), distint(elemento, '0').

Diseño del algoritmo

```

algoritmo Ejercicio_13_3
const
    Máx = ....
tipo
    registro: tipoelemento
        .... : ....
        .... : ....
    fin_registro
    registro: tiponodo
        tipoelemento : elemento

```

```

        entero      : izdo,dcho
fin_registro
array[1..Máx] de tiponodo: arr
var
arr          : a
tipoelemento : elemento
entero       : raíz, vacío
inicio
iniciar(a,raíz,vacío)
escribir ('Introduzca nuevo elemento: ')
si no árbolleno(vacío) entonces
    llamar_a leerelemento (elemento)
fin_si
mientras distinto(elemento,'0') y no árbolleno(vacío) hacer
    altas (a, elemento, raíz, vacío)
    si no árbolleno(vacío) entonces
        escribir ('Introduzca nuevo elemento: ')
        llamar_a leerelemento (elemento)
    fin_si
fin_mientras
listado (a,raíz)
fin

procedimiento iniciar(S arr: a;  S entero: raíz, vacío)
var
entero: i
inicio
raíz ← 0
vacío ← 1
desde i ← 1 hasta máx 1 hacer
    a[i].dcho ← i+1
fin_desde
a[máx].dcho ← 0
fin_procedimiento

lógico función árbolleno(E entero: vacío)
inicio
si vacío=0 entonces
    devolver(verdad)
si_no
    devolver(falso)
fin_si
fin_función

procedimiento inorden(E arr: a;  E entero: raíz;  E/S entero: cont)
inicio
si raíz <> 0 entonces
    inorden(a, a[raíz].izdo, cont)
    cont ← cont + 1
    llamar_a escribirelemento(a[raíz].elemento)

```

```
// Además de contar los nodos visualiza la
// información almacenada en ellos
inorden(a, a[raíz].dcho, cont)
fin_si
fin_procedimiento

procedimiento buscar(E arr: a; E tipoelemento:elemento; S entero:act,ant)
var
    lógico: encontrado
inicio
    encontrado ← falso
    act ← raíz
    ant ← 0
    mientras no encontrado y (act<>0) hacer
        si igual(elemento, a[act].elemento) entonces
            encontrado ← verdad
        si_no
            ant ← act
            si mayor(a[act].elemento, elemento) entonces
                act ← a[act].izdo
            si_no
                act ← a[act].dcho
            fin_si
        fin_si
    fin_mientras
fin_procedimiento

procedimiento altas(E/S arr: a; E tipoelemento: elemento;
                     E/S entero: raíz, vacío)
var
    entero: act, ant, auxi
inicio
    si no árbolleno(vacío) entonces
        buscar(a, elemento, act, ant)
        si act<>0 entonces
            escribir('Ese elemento ya existe')
        si_no
            auxi ← vacío
            vacío ← a[auxi].dcho
            a[auxi].elemento ← elemento
            a[auxi].izdo ← 0
            a[auxi].dcho ← 0
            si ant=0 entonces
                raíz ← auxi
            si_no
                si mayor(a[ant].elemento, elemento) entonces
                    a[ant].izdo ← auxi
                si_no
                    a[ant].dcho ← auxi
            fin_si
    fin_si
```

```

        fin_si
    fin_si
fin_si
fin_procedimiento

procedimiento listado (E arr: a; E entero: raíz)
var
    entero:cont
inicio
    escribir ('inorden: ')
    cont ← 0
    inorden (a, raíz, cont)
    escribir('El número de nodos es ', cont)
fin_procedimiento

```

- 13.4.** Trabajando con estructuras dinámicas, escribir un procedimiento que nos permita contar las hojas de un árbol.

Análisis del problema

Se debe recorrer el árbol, contando, únicamente, los nodos que no tienen hijos

```

procedimiento contarhojas(E punt: raíz; E/S entero: cont)
inicio
    si raíz<>nulo entonces
        si (raíz↑.izdo = nulo) y (raíz↑.dcho = nulo) entonces
            cont ← cont+1
        fin_si
        contarhojas(raíz↑.izdo,cont)
        contarhojas(raíz↑.dcho,cont)
    fin_si
fin_procedimiento

```

- 13.5.** Diseñar una función que permita comprobar si son iguales dos árboles cuyos nodos tienen la siguiente estructura:

```

tipo
    puntero_a nodo: punt
    registro : nodo
    entero : elemento
    punt   : izdo,dcho
fin_registro

```

Análisis del problema

Se trata de una función recursiva que compara nodo a nodo la información almacenada en ambos árboles. Las condiciones de salida del proceso recursivo serán:

- Que se termine de recorrer uno de los dos árboles.
- Que terminen de recorrerse ambos.
- Que encontremos diferente información en los nodos comparados.

Si los árboles terminaron de recorrerse simultáneamente es que ambos tienen el mismo número de nodos y nunca ha sido diferente la información comparada, por tanto, la función devolverá verdad; en cualquier otro caso la función devolverá falso.

```
lógico función iguales(È punt: raíz1, raíz2)
inicio
    si raíz1 = nulo entonces
        si raíz2 = nulo entonces
            devolver(verdad)
        fin_si
    si_no
        si raíz2 = nulo entonces
            devolver(falso)
        si_no
            si raíz1↑.elemento <> raíz2↑.elemento entonces
                devolver(falso)
            si_no
                devolver(iguales(raíz1↑.izdo, raíz2↑.izdo)
                    y iguales(raíz1↑.dcho, raíz2↑.dcho))
            fin_si
        fin_si
    fin_si
fin_función
```


14

RECUSIVIDAD

@UfYWfg]j]XUX ffYWfg]DE Yg'Uei Y'Udfcd]YXUX ei Y'dcgYY i b'gi Vdfc[fUa Udcf 'UWU di YXY "U a UfgY U'g' a lga c 'n'z dcf 'hUbhcž fYdYh]fgY' Wb' j UcfYg' X]ZYfYbhYg' XY' dUza Yhfcg" 9 `` i gc' XY' UfY! Wfg]DE dYfa]hYU`cg'dfc[fUa UXcfYg'YgdYWZ]Wf' gc' i WcbYg'bUhi fUYgžgYbW'Užei YgYf #UbžYb Wgc WbhfUF]cz X]ZM]Yg' XY' fYgc' j Yf'" Dcf' YghUWl gU' UfYWfg]DE Yg' i bU'\YffUa]YbhU dcXYfcgU Y' ja ! dcf hUbhY Yb' 'Uf Ygc' i WDE' XY' df cVYa Ug'miYb' Udfc[fUa UWDE"

14.1. CONCEPTO Y TIPOS DE RECUSIVIDAD

Como se explicó en el Capítulo 5 un *subprograma recursivo* es un subprograma que se llama a sí mismo y esta llamada puede efectuarse tanto directa como indirectamente. Existen por tanto dos tipos de recursividad: la recursividad directa que se produce cuando subprograma se llama a sí mismo y la recursividad indirecta que se origina cuando un subprograma hace referencia a otro el cual, a su vez, efectúa una llamada directa o indirecta al primero.

Un ejemplo de recursividad directa puede ser la siguiente función que efectúa la suma de los N primeros números naturales

```
entero funcion suma(E entero: n)
  inicio
    si (n = 1) entonces
      devolver (1)
    si_no
      devolver(n + suma (n - 1))
    fin_si
  fin_función
```

Un ejemplo de recursividad indirecta es el siguiente

```
algoritmo muestra_alfabeto
  inicio
    A('Z')
  fin
```

```

procedimiento A(E caracter: c)
  inicio
    si c>'A' entonces
      B(c)
    fin_si
    escribir(c);
  fin_procedimiento

procedimiento B(E caracter: c)
  inicio
  {
    aCarácter es una función predefinida que devuelve el carácter
    correspondiente al código ASCII que recibe como argumento
    aCódigo es una función predefinida que devuelve el código ASCII
    del carácter recibido como argumento
  }
  A(aCarácter(aCódigo(c)-1))
  fin_procedimiento

```

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

14.2. USO ADECUADO DE LA RECURSIVIDAD

Cualquier problema que se puede resolver recursivamente, se puede resolver también iterativamente (no recursivamente) y la razón para el uso de la recursividad es que existen numerosos problemas complejos que poseen naturaleza recursiva y, en consecuencia, son más fáciles de implementar con algoritmos de este tipo. La recursión debe pues usarse en aquellos casos en los que conduce a soluciones que son mucho más fáciles de leer y comprender que su correspondiente iterativa y debe ser sustituida por la iteración cuando el consumo de tiempo y memoria son decisivos. Si una solución de un problema se puede expresar iterativa o recursivamente con igual facilidad, es preferible la solución iterativa ya que se ejecuta más rápidamente (no existen llamadas adicionales que consumen tiempo de proceso) y utiliza menos memoria (la pila necesaria para almacenar las sucesivas llamadas necesaria en la recursión). Por otra parte, resulta especialmente conveniente convertir en iterativos aquellos algoritmos recursivos en los que distintas llamadas recursivas producen los mismos cálculos. Un ejemplo de esta situación es el cálculo de los números de Fibonacci, que se definen por la relación de recurrencia $fibonacci_{n+1} = fibonacci_n + fibonacci_{n-1}$, es decir cada término se obtiene sumando los dos anteriores excepto $fibonacci_0 = 0$ y $fibonacci_1 = 1$. La definición dada es recursiva y la implementación del cálculo del elemento *n*-ésimo de la serie se muestra a continuación tanto de forma recursiva como iterativa.

```

//forma recursiva
entero función Fibonaccir(E entero : n)
  inicio
    si n < 0 entonces
      devolver(-1)
    si_no

```

```

    si n = 0 o n = 1 entonces
        devolver(n)
    si_no
        devolver(Fibonaccir(n-1) + Fibonaccir(n-2))
    fin_si
    fin_si
fin_función

//forma iterativa
entero función Fibonaccii(ENTERO : n)
var
    entero : i, último, penúltimo
inicio
    si n > 0 entonces
        último ← 1
        penúltimo ← 0
        desde i ← 2 hasta n hacer
            último ← último + penúltimo
            //se prepara el valor de penúltimo para la siguiente iteración
            penúltimo ← último - penúltimo
        fin_desde
        devolver(último)
    si_no
        si n = 0 entonces
            devolver(n)
        si_no
            devolver(-1)
        fin_si
    fin_si
fin_función

```

En general, la recursividad debe ser evitada cuando el problema se pueda resolver mediante una estructura repetitiva con un tiempo de ejecución significativamente más bajo o similar. Teniendo en cuenta esta afirmación, la recursividad debiera evitarse siempre en subprogramas con recursión en extremo final que calculen valores definidos en forma de relaciones de recurrencia simples

14.3. MÉTODOS PARA LA RESOLUCIÓN DE PROBLEMAS QUE UTILIZAN RECURSIVIDAD

Las estrategias de resolución de problemas denominadas «divide y vencerás» y «retroceso» o «backtracking», son dos técnicas muy utilizadas que emplean recursividad y pueden describirse de la siguiente forma:

Divide y vencerás consiste en dividir un problema en dos o más subproblemas, cada uno de los cuales es similar al original pero más pequeño en tamaño. Con las soluciones a los subproblemas se debe poder construir de manera sencilla una solución del problema general. Para resolver cada subproblema generado existen dos opciones, o el problema es suficientemente pequeño o sencillo como para resolverse fácilmente de manera cómoda o si los subproblemas son todavía de gran tamaño, se aplica de forma recursiva la técnica de *divide y vencerás*.

La técnica de *backtracking* se basa en dividir la solución de un problema en pasos, en cada uno de los cuales habrá una serie de opciones entre las que se debe elegir la adecuada para, al final, alcanzar la solución definitiva. En cada paso se busca una posibilidad u opción aceptable y si se encuentra se

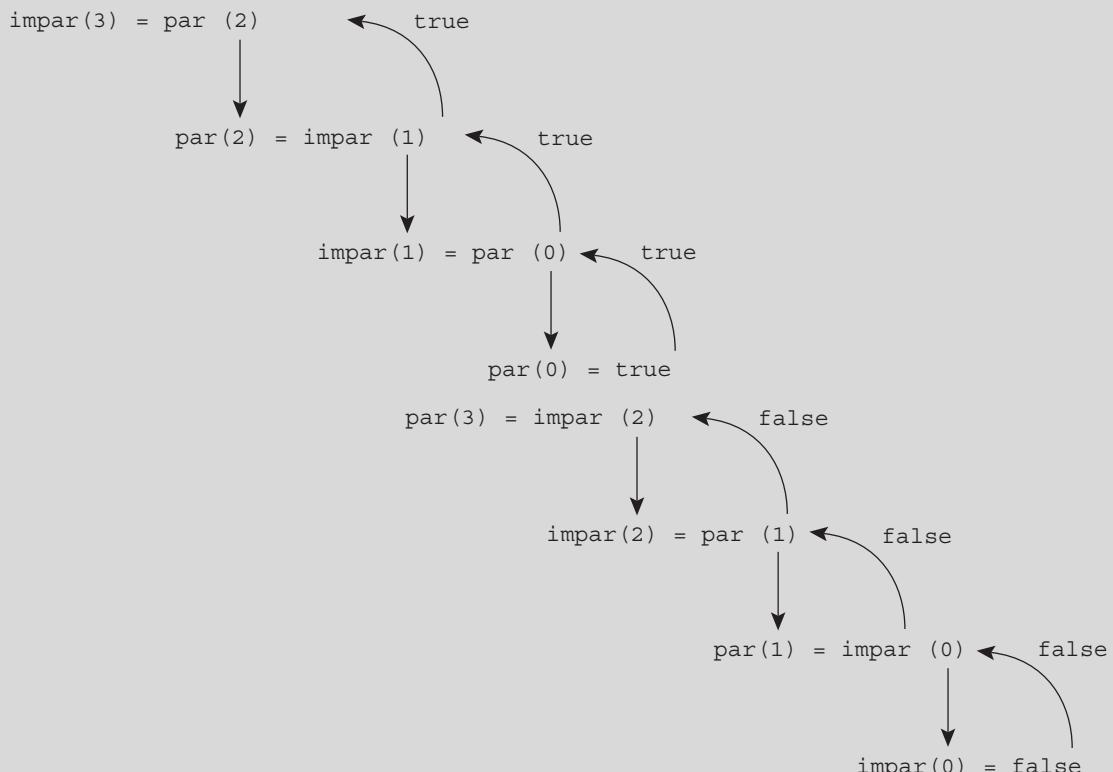
pasa a decidir la del paso siguiente. Si en este siguiente paso no se encuentra ninguna posibilidad aceptable (no existe solución) se retrocede para avanzar a continuación y ver si aparecen posibilidades aceptables tras haber tomado una decisión distinta en el paso anterior. Lógicamente, si en el paso anterior tampoco quedaran posibilidades se retrocede de nuevo.

14.4 EJERCICIOS RESUELTOS

- 14.1.** Implemente un algoritmo para determinar si un número leído de teclado es o no par mediante el uso de dos funciones: *par* e *impar*. Dichas funciones se llamarán una a otra alternativamente y en cada llamada restarán una unidad al número hasta que éste se convierta en cero, momento en el que devolverán la información correspondiente sobre si el número original era par o impar.

Análisis del problema

El ejercicio propuesto es un ejemplo de recursividad indirecta, puesto que las funciones *par* e *impar* se invocan entre sí. La condición de terminación o salida del proceso recursivo es que el número sea o se haya convertido en cero. Como el cero se considera un número par, en este caso la función *impar* devolverá falso y *verdadero* par. Cuando se trate de un número distinto de cero la función *par* invocará a *impar* pasándole como argumento el número decrementado en una unidad y viceversa. La necesidad de una cantidad *par* o *impar* de llamadas hasta que el número se convierta en cero proporcionará la solución. Así, si se empieza invocando a *impar* con un argumento *par*, se necesitará una cantidad *par* de llamadas hasta que el argumento se convierta en cero e *impar* devolverá falso. Análogamente, si se empieza invocando a *par* con un argumento *par*, se necesitará una cantidad *par* de llamadas hasta que el argumento se convierta en cero y *par* devolverá verdad. Cuando el argumento sea *impar* el proceso será:



Codificación

```

algoritmo Ejercicio_14_1
var
    entero: n
inicio
    escribir('Deme un número(negativo para terminar)')
    leer(n)
    mientras n >= 0 hacer
        si par(n) entonces
            escribir(n, 'es par')
        si_no
            escribir(n, 'es impar')
        fin_si
        escribir('Deme un número(negativo para terminar)')
        leer(n)
    fin_mientras
fin

logico function par(E entero: n)
inicio
    si n = 0 entonces
        devolver(true)
    si_no
        devolver(impar(n - 1))
    fin_si
fin_función

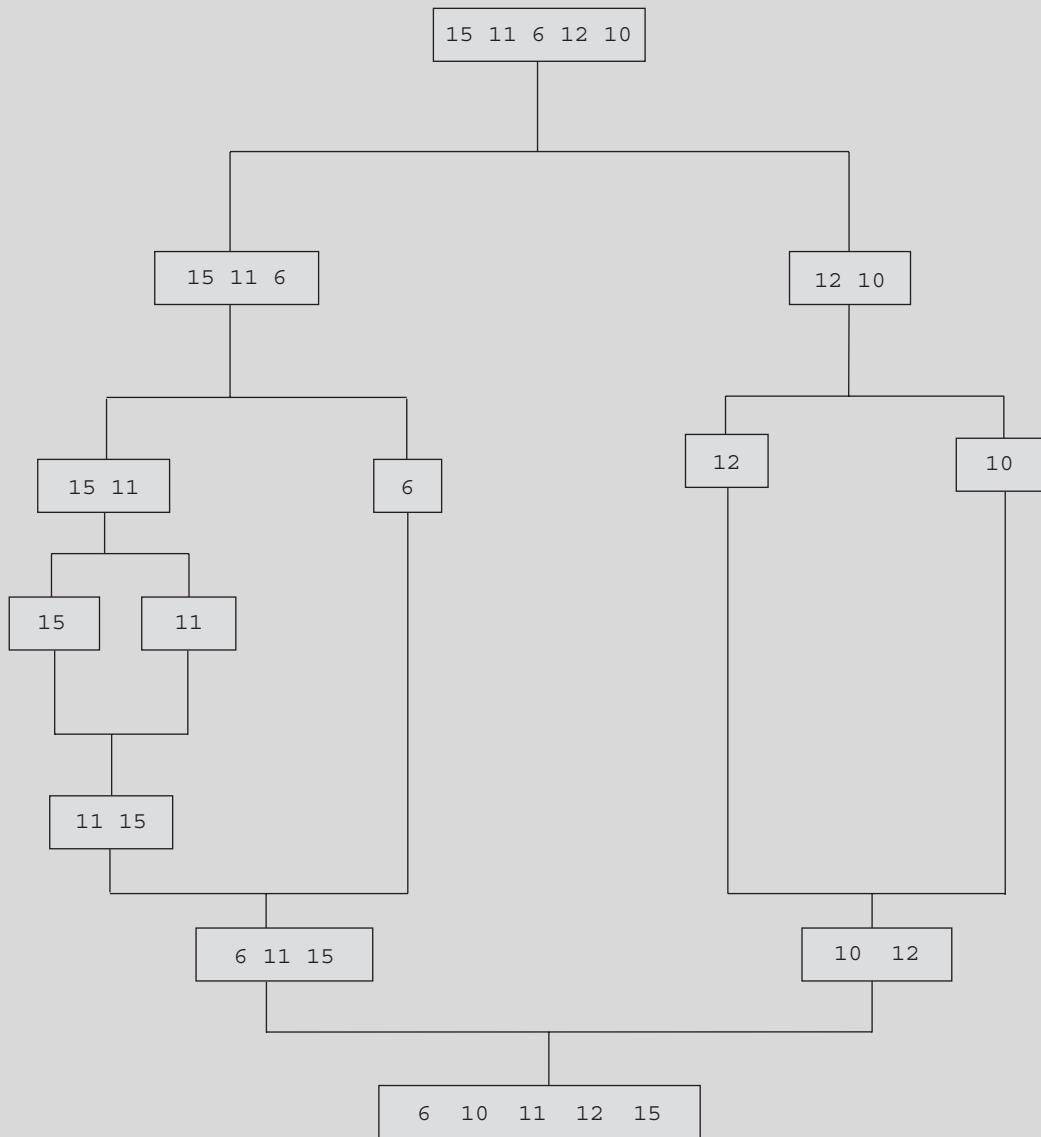
logico function impar(E entero: n)
inicio
    si n = 0 entonces
        devolver(falso)
    si_no
        devolver(par(n - 1))
    fin_si
fin_función

```

- 14.2.** Escriba un algoritmo que almacene en un vector 20 números enteros y los ordene por el método recursivo de mezclas sucesivas, MERGESORT. Este método de ordenación consiste en dividir el vector por su posición central en dos partes y tratar análogamente cada una de ellas hasta que consten de un único elemento. Las partes con un único elemento se consideran ordenadas y su mezcla se puede efectuar como mezcla de dos vectores ordenados para generar otro también ordenado. Las operaciones de mezcla se repiten al retorno de los procesos recursivos de partición, alcanzándose, por último, de esta forma la ordenación definitiva del vector.

Análisis del problema

Este método de ordenación es una aplicación de la técnica *Divide y vencerás* y, para la lista 15, 11, 6, 12, 10, el proceso se puede representar con las siguientes figuras en las que aparecen las divisiones y mezclas posteriores. La mezcla comienza con las sublistas de un solo elemento, que dan lugar a otra sublista del doble de elementos ordenados. El proceso continúa hasta que se construye un única lista ordenada.



Codificación

```

algoritmo Ejercicio_14_2
  const
    t = 20
  tipo
    array[1..t] de entero: arr
  var
    arr: a
    entero: i
  inicio
    desde i ← 1 hasta t hacer
      leer (a[i])
  
```

```

fin_desde
ordenarfusion(a, 1, t)
// t es una constante, pero es necesario pasarla como parámetro
desde i ← 1 hasta t hacer
    escribir(a[i])
fin desde
fin

procedimiento ordenarfusion(E/S arr: a; E entero: inicio, fin)
    var
        entero: central
    inicio
        si inicio < fin entonces
            central ← (inicio + fin) div 2
            ordenarfusion(a, inicio, central)
            ordenarfusion(a, central + 1, fin)
            fusion(a, inicio, central, central + 1, fin)
        fin_si
    fin_procedimiento

procedimiento fusion(E/S arr: a; E entero: inicio1, fin1, inicio2, fin2)
{El procedimiento fusión recibe el vector y los extremos de cada partición.
Devuelve el vector mezclado}
    var
        entero: i, j, k
        arr: b
    inicio
        i ← inicio1
        j ← inicio2
        k ← inicio1
        mientras (i <= fin1) y (j <= fin2) hacer
            si a[i] < a[j] entonces
                b[k] ← a[i]
                i ← i + 1
            si_no
                si a[i] = a[j] entonces
                    b[k] ← a[j]
                    j ← j + 1
                    k ← k + 1
                    b[k] ← a[i]
                    i ← i + 1
                si_no
                    b[k] ← a[j]
                    j ← j + 1
                fin_si
            fin_si
            k ← k + 1
        fin_mientras
        mientras i <= fin1 hacer
            b[k] ← a[i]

```

```

    i ← i + 1
    k ← k + 1
fin_mientras
mientras j <= fin2 hacer
    b[k] ← a[j]
    j ← j + 1
    k ← k + 1
fin_mientras
    desde k ← inicio1 hasta fin2 hacer
        a[k] ← b[k]
    fin_desde
fin_procedimiento

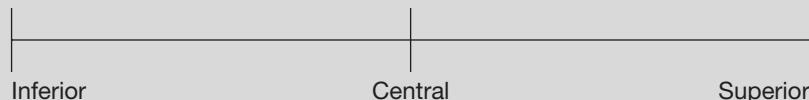
```

14.3. Implemente la búsqueda binaria de forma recursiva.

Análisis del problema

Para aplicar el método de búsqueda binaria es necesario que la lista donde se va a efectuar la búsqueda sea una lista ordenada. El algoritmo de búsqueda binaria se puede describir recursivamente:

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



Central = (inferior + superior)div 2 Comparar A[central] y clave

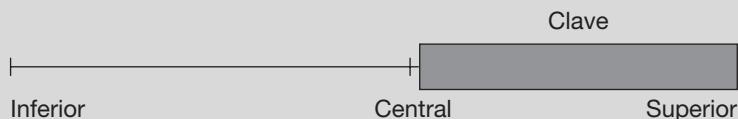
Si se produce coincidencia (se encuentra la clave), se tiene la condición de terminación que permite detener la búsqueda y devolver el índice central. Si no se produce la coincidencia (no se encuentra la clave), dado que la lista está ordenada, se centra la búsqueda en la «sublista inferior» (a la izquierda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

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



Búsqueda en sublistas izquierdas
[Inferior .. central - 1]

Si clave > A[central], el valor buscado sólo puede estar en la mitad derecha de la lista con elementos en el rango de índices, Central + 1 a Superior.



El proceso recursivo continúa la búsqueda en sublistas más y más pequeñas. La búsqueda termina, o con éxito (*aparece la clave buscada*) o sin éxito (*no aparece la clave buscada*), situación que ocurrirá cuando el

límite superior de la lista sea más pequeño que el límite inferior. La condición Inferior > Superior será la condición de salida o terminación y el algoritmo devuelve el índice -1.

Diseño del algoritmo

```

entero funcion búsquedAB( E TipoArray: a; E entero: tamañoArray;
                           E entero: elementoABuscar)
inicio
    devolver(búsquedaBinaria(a,1,tamañoArray, elementoABuscar))
fin_función

entero función búsquedaBinaria( E TipoArray a; E entero: iz, de, elemento)
var
    entero: central
inicio
    si de < iz entonces
        devolver(-1)
    si_no
        central ← (iz + de) div 2
        si elemento < a[central] entonces
            devolver(búsquedaBinaria(a, iz, central-1, c))
        si_no
            si (a[central] < elemento) entonces
                devolver(búsquedaBinaria(a, central+1, de, c))
            si_no
                devolver(central)
            fin_si
        fin_si
    fin_si
fin_función

```

- 14.4.** Implemente el método de Ordenación Rápida (Quick Sort) de forma recursiva. El método comienza eligiendo uno de los elementos de la lista como pivote o elemento de partición y dividiendo la lista a ordenar en dos, de modo que ningún elemento de la sublista izquierda tenga una clave mayor que el pivote y ningún elemento de la sublista derecha una clave menor. Las dos sublistas se dividen sucesivamente utilizando el mismo algoritmo, es decir que se llama recursivamente al propio algoritmo quicksort hasta que las sublistas constan de un único elemento. La lista final ordenada se consigue concatenando la primera sublista, el pivote y la segunda lista, en ese orden, en una única lista.

Análisis del problema

Puesto que se va a seguir el método detallado en el enunciado, lo que queda por establecer es la forma de efectuar la elección del pivote, pues, aunque en principio puede ser cualquiera, interesa que su valor no sea un valor extremo entre los de los elementos que constituyen la lista. En este ejercicio se elegirá como pivote el elemento de valor intermedio entre los situados en las posiciones primera, última y central. Así, si el array fuera

173	140	42	20	175	172	5	78	185	51
-----	-----	----	----	-----	-----	---	----	-----	----

al pivote le correspondería el valor 173

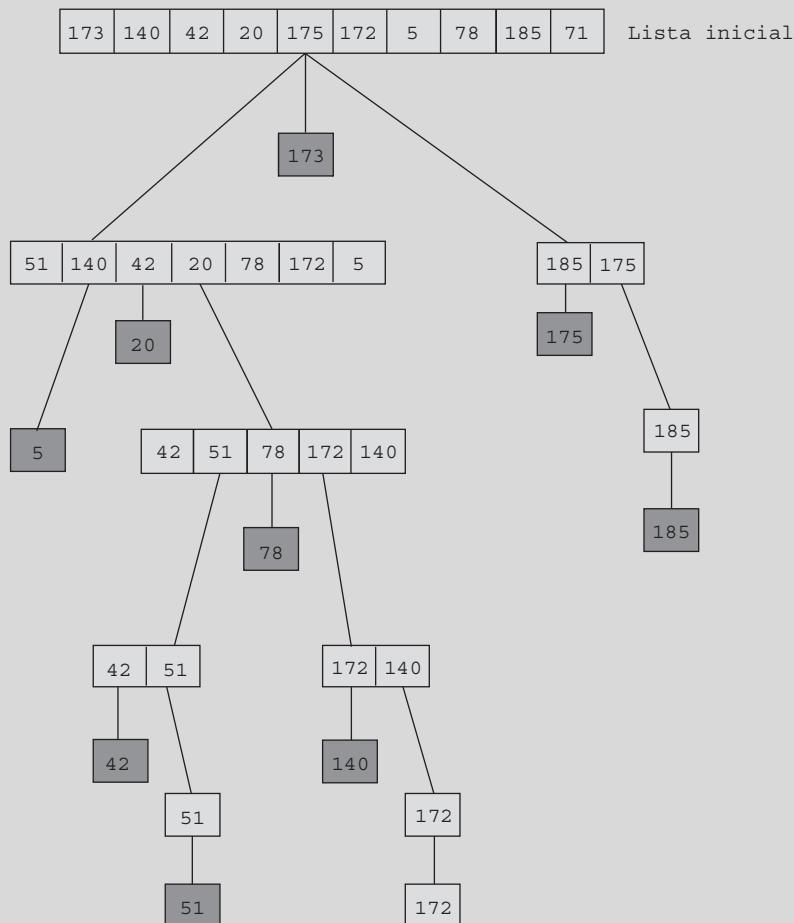
Izq	Der	Centro	Pivote
1	10	5	173

y aplicando lo que indica el enunciado la lista se dividiría en dos y la que tendría que ser considerada a continuación es

51	140	42	20	78	172	5	173	185	175
----	-----	----	----	----	-----	---	-----	-----	-----

Izq	Der	Centro	Pivote
1	7	4	20

El seguimiento completo se efectúa a través de la siguiente figura



Al final, del proceso se devuelve la lista ordenada.

5	20	42	51	78	140	172	173	175	185
---	----	----	----	----	-----	-----	-----	-----	-----

Diseño del algoritmo

```
algoritmo Ejercicio_14_4
const
    nmax=100
```

```

Tipo
    array[1..nmax] de entero: TipoArray
var
    TipoArray: arr
    entero: nn

inicio
    llenar (arr, nn)
    Ordenar(arr,1,nn)
    mostrar(arr, nn)fin

procedimiento Ordenar (E/S TipoArray: arr; E entero: Izq,Der)
var
    entero: centro
    entero: aux, x
    entero: i,j
inicio
    si (der > 1) y (izq >= 1) y (izq <= der) entonces
        centro ← (Izq + Der )div 2
        si arr[Izq] > arr[centro]entonces
            aux ← arr[centro]
            arr[centro] ← arr[izq]
            arr[izq] ← aux
        fin_si
        si arr[izq] > arr[Der] entonces
            aux ← arr[izq]
            arr[izq] ← arr[der]
            arr[der] ← aux
        fin_si
        si arr[centro] > arr[Der] entonces
            aux ← arr[centro]
            arr[centro] ← arr[der]
            arr[der] ← aux
        fin_si
        aux ← arr[centro]
        arr[centro] ← arr[der]
        arr[der] ← aux
        x ← arr[der]
        i ← Izq
        j ← Der
        mientras (i<j) hacer
            mientras (i<j) y (arr[i]<=x) hacer
                i ← i+1
            fin_mientras
            mientras (i<j) y (arr[j]>=x) hacer
                j ← j-1
            fin_mientras
            si i<j entonces
                aux ← arr[i]
                arr[i] ← arr[j]
                arr[j] ← aux

```

```

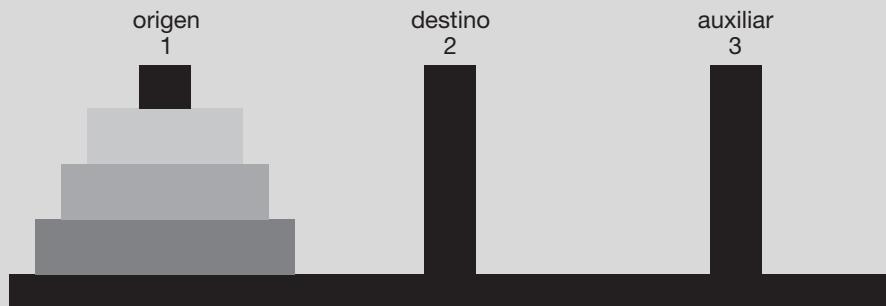
    fin_si
fin_mientras
aux ← arr[i]
arr[i] ← arr[der]
arr[der] ← aux
Ordenar(arr, izq, i-1)
Ordenar(arr, i+1, der)

fin_si
fin_procedimiento

```

- 14.5.** Diseñar un algoritmo que resuelva el problema de las Torres de Hanoi que, supuestas 3 torres y un conjunto de discos de diferentes tamaños situados inicialmente en una de ellas ordenados de mayor a menor, pide determinar los movimientos necesarios para pasar todos los discos a otra torre, utilizando la tercera como auxiliar y cumpliendo las siguientes reglas:

- Los discos han de estar siempre situados en alguna de las torres.
- En cada movimiento sólo puede intervenir un disco.
- En ningún momento puede quedar un disco sobre otro de menor tamaño.



Análisis del problema

Si el número de discos fuera 1, bastaría con mover dicho disco de origen a destino.

Si el número es dos se puede trasladar un disco de origen a auxiliar, otro de origen a destino y, por último, pasar el que se colocó en la torre auxiliar al destino.

Si se trata de 3 discos, el problema se puede resolver si se trasladan dos discos de origen a auxiliar, uno de origen a destino y, por último, se pasan los dos de la torre auxiliar al destino. La solución implicaría los siguientes movimientos:

Mover 2 discos de una torre a otra, en este caso de origen a auxiliar utilizando destino como auxiliar.

1. De origen a destino.
2. De origen a auxiliar.
3. De destino a auxiliar.

Mover un disco de origen a destino.

4. De origen a destino.

Mover 2 discos de una torre a otra, en este caso de auxiliar a destino utilizando origen como auxiliar.

5. De auxiliar a origen.
6. De auxiliar a destino.
7. De origen a destino.

En general, pues, para mover n discos de la torre origen a la destino, hay que pasar $n - 1$ discos a la torre auxiliar, mover 1 de origen a destino y por último pasar $n - 1$ discos de la torre auxiliar al destino.

Diseño del algoritmo

```

algoritmo Ejercicio_14_5
var
    entero: n
inicio
    escribir('Nº de discos:')
    leer(n)
    Mover_torre(n,1,2,3)
fin

procedimiento Mover_torre( E entero: N;   E entero: orig,dest,aux)
inicio
    si N = 1 entonces
        escribir('Paso de ', orig,' a ', dest )
    si_no
        Mover_torre ( N-1, orig, aux, dest )
        escribir('Paso de ', orig,' a ', dest )
        Mover_torre ( N-1, aux, dest, orig )
    fin_si
fin_procedimiento

```

- 14.6.** Diseñar un algoritmo que resuelva el problema de las ocho reinas. que consiste en disponer ocho reinas en un tablero de ajedrez de tal forma que no se amenacen entre sí (recuerde que una reina amenaña a las piezas situadas en su misma fila, columna o diagonal).

Análisis del problema

El problema consiste en colocar las 8 reinas dentro del tablero de ajedrez sin que se coman unas a otras o, en general, de poner n reinas en un tablero de $n \times n$. Este es un ejemplo típico de *backtracking*. Dado que se tendrá una reina y sólo una por fila (y por columna), los pasos en los que puede dividirse el problema podrían considerarse las filas y en cada paso considerar unas posibilidades que son las diferentes columnas. Se trata entonces de elegir, para cada una de las filas, la columna donde se situará la reina de forma que no haya ninguna otra reina ya colocada en dicha columna ni ninguna que se encuentre con la nueva en diagonal.

En vez de colocar las reinas en un *array** de $n \times n$ elementos, se utiliza un vector de n elementos. Cada elemento del vector representa la fila donde está la reina. El contenido representa la columna.

El vector Reinas se inicializa a 0. Para buscar la siguiente solución se comienza situando en Reinas [1] un 1 (la primera reina está en la primera fila, columna 1). La siguiente reina estará en la fila 2 y empezamos a buscar en la posición 1.

Dado que para las casillas situadas en la misma diagonal principal se cumple siempre que

columna-fila = constante

y para las situadas en la misma diagonal secundaria que

columna+fila = constante

no será una posición válida si Reinas [1] = Reinas [2] (implica que están en la misma columna) y tampoco si Reinas [1]+1 = Reinas [2]+2 (implica que están en diagonal) o si Reinas [1]-1 = Reinas [2]-2 (también en diagonal). Si esto ocurre, incrementamos la columna de la nueva reina y se volverá a comprobar.

	1	2	3	4	5	6	7	8
1			3-1		5+1			
2				4-2 4+2				
3			3+3		5-3			
4		2+4				6-4		
5	1+5						7-5	
6								8-6
7								
8								

Diseño del algoritmo

```

algoritmo Ejercicio_14_6
const n = 8
tipo
    array de entero[1..n] : ListaReinas
var
    ListaReinas : REINAS
    entero : i
    lógico : solución
inicio
    // Inicializar el array
    desde i ← 1 hasta n hacer
        reinas[i] ← 0
    fin_desde
    Ensayar(reinas,1,solución)
    si no solución entonces
        escribir('No hay solución')
    si_no
        // se deja al lector hacer la presentación del resultado
    fin_si
fin

lógico función PosiciónVálida(✉ ListaReinas reinas; ✉ entero : i)
var
    entero : j
    lógico : válida
inicio
    válida ← verdad
    desde j ← 1 hasta i - 1 hacer
        // No se ataca en la columna
        válida ← válida y (reinas[i] <> reinas[j])
        // no se ataca en una diagonal
        válida ← válida y (reinas[i] + i <> reinas[j] + j )

```

```

// no se ataca en la otra diagonal
válida ← válida y (reinas[i] - i <> reinas[j] - j)
fin_desde
devolver(Válida)
fin_función

procedimiento Ensayar( E/S ListaReinas: reinas ; E entero : i;
                        S lógico : Solución)
inicio
    si i = n + 1 entonces
        solución ← verdad
    si_no
        solución ← falso
        repetir
            reinas[i] ← reinas[i] + 1
            si PosiciónVálida(reinas,i) entonces
                Ensayar(reinas,i+1,solución)
            hasta que solución o (reinas[i] = n)
            si no solución entonces
                reinas[i] ← 0
            fin_si
        fin_si
    fin_procedimiento

```

	1	2	3	4	5	6	7	8
1	R							
2				R				
3								R
4					R			
5		R						
6						R		
7		R						
8			R					

14.7. Diseñar un algoritmo recursivo que muestre las combinaciones de m elementos tomados de n en n .

Análisis del problema

Se llaman combinaciones de m elementos tomados de n en n ($n \leq m$) a todas las agrupaciones posibles que pueden hacerse con los m elementos de forma que:

- Cada agrupación está formada por n elementos distintos entre sí.
- Dos agrupaciones distintas se diferencian al menos en un elemento y no se tiene en cuenta el orden de los mismos.

El problema de construir una agrupación consistirá en elegir n elementos distintos y se puede dividir en pasos que serán la elección de cada uno de dichos elementos. Además, para obtener todas las agrupaciones habrá que considerar las diferentes posibilidades de selección. Así, cuando se eligen n elementos entre m , hay m formas de efectuar la primera selección, pero cuando se ha seleccionado el primer objeto quedan $m - 1$ para

efectuar la segunda y así sucesivamente. Teniendo en cuenta esto, el procedimiento Combs, que muestra las combinaciones, recibirá como parámetro una cadena S (de longitud inicial m) que por un lado contendrá los elementos seleccionados hasta el momento (S[1] a S[i]) y por otro los que todavía resulta posible seleccionar como i-ésimo + 1 elemento de la agrupación, almacenados en las posiciones comprendidas entre i + 1 y el final de la cadena.

Diseño del algoritmo

```

algoritmo Ejercicio_14_7
var
    entero: N, M, Cont
inicio
    escribir('Indique el número de elementos: ')
    leer(m)
    escribir('Tomados de: ')
    leer(n)
    escribir('Las combinaciones de ', m, ' elementos ',
        'tomados de ', n, ' en ', n, ' son ', NumCombs(m, n))
    Cont ← 0
    Combs(subcadena('123456789', 1, m), n, 0)
    escribir(AvLn) // salto de línea
    escribir(cont, AvLn)
fin

procedimiento Combs(E cadena: S; E entero: n, i)
inicio
    si i = n entonces
        escribir(subcadena(S, 1, i), ' ') //sin cambiar de línea
        cont ← cont + 1
    si_no
        Combs(S, n, i+1)
        mientras longitud(S) > n hacer
            // se elimina de la cadena
            S ← subcadena(S, 1, i) + subcadena(S, i+2, longitud(S)-i-1)
            Combs(S, n, i+1)
    fin_mientras
    fin_si
fin_procedimiento

entero función NumCombs(E entero: M, N)
inicio
    devolver (Factorial(M) div
        (Factorial(N) * Factorial(M-N)))
fin_función

entero función Factorial(E entero: N)
var
    entero: P, i
inicio
    P ← 1
    desde i ← 1 hasta N hacer

```

```

P ← P * i
fin_desde
devolver(P)
fin_función

```

- 14.8.** Suponga que dispone de una colección de productos químicos y una serie de cajas con capacidad suficiente para almacenarlos, no obstante, algunos de estos productos, pueden originar violentas reacciones cuando se guardan juntos, por lo que los denominaremos incompatibles. Diseñe un algoritmo que permita determinar el mínimo número de cajas necesarias para almacenar los mencionados productos teniendo en cuenta que sólo se almacenarán en la misma caja los que sean compatibles.

Análisis del problema

Para resolver el problema supondremos que el máximo número de productos y de cajas es 10.

Las incompatibilidades entre unos productos y otros se representarán mediante una matriz, donde filas y columnas representan los productos de tal manera y si un producto *i* es incompatible con otro *j*, la celda correspondiente de la matriz almacenará verdad y cuando sean compatibles almacenará falso.

La solución será almacenada en un registro con don dos campos, uno de tipo array con tantos elementos como productos donde se guardará la caja correspondiente a cada producto, y otro de tipo entero donde se almacenará el número máximo de cajas.

El problema se divide en pasos, donde se considera cada uno de los productos y en cada paso se analizan las distintas posibilidades. Las posibilidades para un determinado producto *i* son las distintas cajas. La función *aceptable*, se encarga de decidir si es posible guardar el producto *i* en una determinada caja. Será factible si la caja es distinta de la que tengan todos productos incompatibles con él, en cuyo caso se anota, es decir se asigna al producto su caja (borrar anotación, es marcarlo como sin caja). La función *Distinta*, se encarga de decidir si la caja en la que se va a almacenar un producto es una nueva caja, es decir una caja donde aun no se ha almacenado nada, por lo que el número de cajas necesarias para almacenar los productos se incrementará en una unidad. Para encontrar la solución óptima hay que hallar todas las soluciones, es decir todas formas de almacenamiento posibles sin que ocurran reacciones peligrosas, y quedarse con aquella que requiera un menor número de cajas.

Otros procedimientos interesantes son, el procedimiento *inicializar*, que se encarga de almacenar la información sobre productos incompatibles, así como inicializar una solución cuyo número de cajas sea infinito y el procedimiento *EscribeSolución*, que escribe la solución óptima.

Diseño del algoritmo

```

algoritmo Ejercicio_14_8
    const
        Max= 10
        // Máximo número de productos y de cajas
    tipo
        array[1..Max, 1..Max] de lógico: Matriz
        registro: Solucion
            entero: cont
            array[1..max] of integer: caja
            {las cajas son enteros del 1 al 10,
            0 representa sin caja}
        fin_registro
    var
        matriz: m
        solucion: Sol, Soloptima
    inicio
        Inicializar (m, Soloptima, Sol)

```

```

AsignarCaja(1)
EscribeSolucion
fin

procedimiento Inicializar(E/S Matriz: m; E/S Solucion: Soloptima, Sol)
var
    entero: i, j
    caracter: ch
begin
    desde i ← 1 hasta Max hacer
        Soloptima.caja[i] ← 0
        Sol.caja[i] ← 0
        m[i,i] ← false
    desde j ← i + 1 hasta Max hacer
        escribir('Producto ', i ' y producto ', j)
        escribir('0 Compatible y 1 Incompatible')
        leer(ch)
        m[i,j] ← ch = 1
        //falso no incompatibles, verdad si
        m[j,i] ← m[i,j]
    fin_desde
fin_desde
Soloptima.cont ← Maxint
Sol.cont ← 0
fin_procedimiento

procedimiento AsignarCaja(E entero: i)
var
    entero: caja
    logico: sw
inicio
    caja ← 0
    sw ← falso
repetir
    caja ← caja + 1
    si Aceptable(i, caja) entonces
        si Distinta(caja) entonces
            Sol.cont ← Sol.cont + 1
            sw ← verdad
        fin_si
        Sol.caja[i] ← caja
        // se anota la caja donde guardar el producto

        si Sol.cont < Soloptima.cont entonces
            {técnica de podado, si ya se llevaran gastadas más cajas
            que en una solución anterior el camino no conduce a
            a una solución óptima y no interesa seguirlo}

        si i < Max entonces
            AsignarCaja(i + 1)

```

```

        si_no
            si Sol.cont < Soloptima.cont entonces
                Soloptima ← Sol
            fin_si
            fin_si
        si sw entonces
            Sol.cont ← Sol.cont - 1
            sw ← falso
        fin_si
        Sol.caja[i] ← 0
        // Sin caja
    fin_si
hasta que caja = 10
fin_procedimiento

procedimiento EscribeSolucion
    tipo
        array[1..max ] de cadena: salida
    var
        entero: i, j
        caracter: ch
        salida: sal
    inicio
        desde i ← 1 hasta 10 hacer
            sal[i] ← aCarácter(aCódigo('A')+ i - 1)
        fin_desde
        {se supone que escribir no salta a la siguiente línea, excepto
         si se especifica escribir AvLn}
        escribir (' matriz (1 si el producto i es incompatible con el j) ')
        escribir (AvLn)
        escribir('      ')
        desde j ← 1 hasta 10 hasta
            escribir (j, '      ')
        fin_desde
        escribir(AvLn) // salto de línea
        desde i ← 1 hasta Max hasta
            escribir (i, ' - ')
            desde j ← 1 hasta Max hacer
                si m[i,j] entonces
                    escribir ('  1')
                si_no
                    escribir ('  0')
                fin_si
            fin_desde
            escribir(AvLn)
        fin_desde
    si Soloptima.cont < Maxint entonces
        escribir (' Número de cajas necesarias: ' , Soloptima.cont)
        escribir (' Asignación de cajas a los productos ')

```

```

desde i ← 1 hasta Max hacer
    escribir(i:2,' ',sal[ord(Soloptima.caja[i]):3,' ')
fin_desde
fin_si
fin_procedimiento
logico funcion Aceptable(ENTERO: i; ENTERO: caja)
var
    ENTERO: j
    logico: sw
inicio
    sw ← verdad
    j ← 0
repetir
    j ← j + 1
    si (i <> j) y (m[i,j]) entonces
        sw ← caja <> Sol.caja[j]
    fin_si
hasta_que (j = Max) o no sw
devolver(sw)
fin_funcion
logico funcion Distinta(ENTERO: caja)
var
    ENTERO: j
    logico: sw
inicio
    sw ← verdad
    j ← 0
repetir
    j ← j + 1
    si (Sol.caja[j] = caja) entonces
        sw ← falso
    fin_si
hasta_que (j = Max) o no sw
devolver(sw)
fin_función

```

- 14.9.** Dada una tabla de dimensiones $n*n$ cuyas celdas contienen el coste de edificación de un edificio en un determinado solar por una determinada empresa. Diseñe un programa recursivo que, mediante backtracking, obtenga la solución que minimiza el coste total de construcción de los n edificios, teniendo en cuenta que se debe asignar la construcción de un único edificio a cada una de las empresas. Mejore la eficiencia empleando técnicas de poda.

Por ejemplo, para la siguiente tabla

	solar 1	solar 2	solar 3
Empresa A	351.000	685.000	300.000
Empresa B	204.000	600.000	102.000
Empresa C	340.000	980.030	422.050

la solución sería: A-2, B-3, C-1.

Análisis del problema

Como el ejercicio anterior este también es un problema de *backtracking*, donde se buscarán todas las soluciones con la finalidad de encontrar y retener la óptima, que es aquella en la que cada empresa tiene asignado un solar y la suma total de los costes de edificación es menor.

Puesto que cada empresa debe construir un único edificio la obtención de una solución pasa por recorrer las empresas y en cada caso, es decir a cada una de ellas, otorgarle la construcción de un solar que no haya sido ya encargado a otra empresa. Cada solución se almacenará pues en un vector (*d*) con tantos elementos como empresas que contendrán los solares asignados. Por otra parte, el cálculo del coste de la solución obligará a consultar la tabla de $n*n$ elementos (denominada *m*) y acumular los de cada empresa para el solar que tiene asignado. Hay que tener en cuenta que, en un momento determinado, si la empresa considerada es *j*, el solar que le ha sido asignado está en *d[j]* y el valor a acumular se encuentra en *m[j, d[j]]*.

Una vez obtenida una solución hay que buscar otras que pudieran resultar más rentables. Para ello se desasigna el solar adjudicado y se prueba otra posibilidad (otro solar) o, si no quedan posibilidades, se retrocede, y se desasigna y busca nuevo solar para la empresa anterior, a continuación, si resulta conveniente, se avanza de nuevo. El avance hacia una nueva solución no resultará conveniente cuando el coste acumulado de las adjudicaciones actuales supere ya el de alguna solución anterior.

El procedimiento fundamental para resolver el ejercicio es el que se ha denominado recursivo en la solución propuesta.

Diseño del algoritmo

```

algoritmo Ejercicio_14_9
    const
        TEmpresas = 'C'
        TTareas = 3
    tipo
        array['A'..TEmpresas] de entero: vector
        //se admitirá que uno de los subíndices del vector sea de tipo carácter
        array['A'..TEmpresas, 1..TTareas] de entero: costes
        {el tipo base de la matriz de costes, aunque eso depende del lenguaje,
        probablemente deba ser entero largo}
        array[ 1..TTareas] de logico: arrconjunto
    var
        vector: d, dopt
        carácter: i, ch
        entero: j
        entero: s, optima, cont
        //probablemente entero largo
        arrconjunto: c
        costes: matriz

    inicio
        cont ← 0
        s ← 0
        desde j ← 1 hasta TTareas hacer
            c[j] ← falso
        fin_desde
        optima ← 0
        j ← 1
        desde i ← 'A' hasta TEmpresas hacer
            desde j ← 1 hasta TTareas hacer

```

```

        leer( matriz[i, j])
        {se inicializa optima a un valor muy grande, que nunca va a poder
        tener ninguna solución. Cualquiera solución que se elija
        tendrá un coste total menor}
        optima ← optima + matriz[i,j]
    fin_desde
fin_desde
recursivo('A')
imprimir
escribir(cont)
escribir (AvLn)
fin

procedimiento recursivo(E caracter: j)
var
    entero: i
inicio
desde i ← 1 hasta TTareas hacer
    // si el solar no está asignado ya
    si no c[i] entonces
        //Asignar
        c [i] ← verdad
        d[j] ← i
        // en s se acumulan los costes
        s ← s + matriz[j,d[j]]
        si s < optima entonces
            si (j = TEmpresas) entonces
                cont ← cont + 1
                // se guarda la solución optima
                optima ← s
                dopt ← d
            si_no
                recursivo(aCaracter(aCodigo(j) + 1))
            fin_si
        fin_si
        //Desasignar
        c[i] ← falso
        s ← s - matriz[j,d[j]]
        {no hace falta borrar el contenido de d[j]
        porque se sobreescibirá}
    fin_si
fin_desde
fin_procedimiento

procedimiento imprimir
var
    caracter: i
inicio
desde i ← 'A' hasta TEmpresas hacer
    escribir(i, ' ',dopt[i])

```

```

    escribir(' ', matriz[i,dopt[i]])
    escribir(AvLn)
fin_desde
fin_procedimiento

```

- 14.10.** Diseñe un programa que invierta una pila sin utilizar ninguna estructura auxiliar y usando los subprogramas *Pinsertar*, *Pborrar*, *Tope* y *EsPilaVacia*.

- *PilaVacia* Inicializa una pila, crea una pila como vacía.
- *EsPilaVacia* Es una función lógica que indica si una pila está vacía.
- *PInsertar* Inserta un elemento en la pila.
- *Tope* Devuelve el elemento situado en la cima de la pila.
- *PBorrar* Borra un elemento de la pila.

Análisis del problema

Puesto que una pila no se puede recorrer sin vaciarla, para invertir la pila será obligatorio usar estructuras auxiliares y, como el enunciado no permite crearlas, para solucionar el problema utilizaremos recursividad.

En primer lugar se llamará al procedimiento recursivo *invertir*, para que desapile los elementos hasta que la pila original quede vacía. A la salida de la recursividad el procedimiento *invertir* llama a *insertafinal* con cada uno de los elementos que automáticamente va recuperando, ya que *a* es variable local.

El procedimiento *insertafinal* (también recursivo) recibe la pila y el nuevo elemento a insertar, pero no lo puede colocar directamente en ella, pues entonces los elementos quedarían de nuevo en el orden inicial, así que sólo lo coloca abajo del todo. Es decir, que si la pila no está vacía desapila, inserta el nuevo elemento y vuelve a colocar encima los demás, que son recuperados automáticamente a la salida de la recursividad, puesto que *e* también es variable local.

Por ejemplo, imagine una pila en la que se han ido colocando los siguientes elementos 1, 2, 3, 4. Al llamar a *invertir* la pila se vacía y a la salida de la recursividad se llama a *insertafinal* para que coloque de nuevo en la pila el último que desapiló (el 1). A continuación en *invertir*, se recupera el siguiente elemento a colocar (el 2) y se vuelve a llamar a *insertafinal*, pero como la pila no está vacía, *insertafinal* desapila el elemento que hay en ella (el 1), ahora que está vacía, coloca el nuevo (el 2) y después vuelve a apilar el 1 que quito.

Diseño del algoritmo

```

procedimiento invertir(E/S pila: p)
var
    entero: a
inicio
    si no EsPilaVacia(p) entonces
        Tope(p,a)
        Pborrar(p)
        invertir(p)
        //llama a insertafinal a la salida de la recursividad
        insertafinal(p,a)
    fin_si
fin_procedimiento

procedimiento insertafinal (E/S pila: p; E entero: a)
var
    entero: e
inicio
    si no EsPilaVacia(p) entonces

```

```
Tope(p,e)
Pborrar(p)
insertafinal(p,a)
// vuelve a poner los que acaba de desapilar
Pinsertar(p,e)
si_no
    //coloca el nuevo elemento que le ha proporcionado invertir
    Pinsertar(p,a)
fin_si
fin_procedimiento
```

15

INTRODUCCIÓN A LA PROGRAMACIÓN ORIENTADA A OBJETOS

9`a i bXc ei Y`bcg`fcXYU`Yghz ``Ybc`XY`cV``Yhcg`Ubja UXcg`Y`jbUbja UXcg`WbW`Yhcg`mUV`ghf`UW`cg`zf! Vc`Yg`mihUV`Ug`z`U`hca`{`]`Yg`mW`gl`U`@`U`cf`]`YbhU`W`C`U`c`V``Yhcg`Vi`gW`a`c`XY`U`Y`g`c`V``Yhcg`XY`a`i`b! Xc`f`YU`Y`a`d`Ya`YbhU`cg`Yb`Yb[i`U`Yg`XY`df`c[f`Ua`U`W`C`DC`C`Yg`i`b`a`YX`c`d`U`Ua`Y`cf`U`U`df`c`Xi`W`hj`j`XUX`XY`U`df`c[f`Ua`U`W`C`W`bh`f`zb`Xcg`Yb`U`V`ghf`U`W`C`XY`XUhcg`mif`Yi`h`j`n`U`W`C`XY`Wa`dc`b`Ybh`Yg`7`ca`Ybn`CEU`[`Ub`U`dc`di`U`f`]`XUX`U`df`j`b`W`d`c`XY`cg`-`S`m`\cm`Yg`XY`i`gc`[`Yb`YU`U`n`UXc`Yb`U`]`b`Xi`ghf`]`U`m`U`i`bj`Yg`XUX`!`D`c`ei`f`g`Y`b`Y`W`gl`h`U`U`df`Y`b`X`Y`f`DC`C`m`Y`b[i`U`Yg`cf`]`Ybh`U`Xcg`U`c`V``Yhcg`3`!`E`i`f`df`cd`Y`!`XUX`Yg`h`Y`b`Y`U`DC`C`m`ig`g`Yb[i`U`Yg`U`gc`W`U`Xcg`3`!`E`i`f`g`cb`W`U`g`Yg`m`ic`V``Yhcg`3`9`gh`U`g`m`ich`f`U`g`df`Y`i`b`h`U`g`g`Y`hf`U`h`b`XY`f`Y`g`j`Y`f`Y`b`Y`gh`Y`W`d`f`hi`c`m`Y`g`[`i`]`Y`bh`Y`

15.1. MECANISMOS DE ABSTRACCIÓN

Los primeros ingenieros/programadores de software han utilizado diferentes métodos de abstracción antes de llegar al paradigma de la programación orientada a objetos. Desde una perspectiva histórica, el uso de la abstracción por la programación orientada a objetos no es más que la progresión natural de la abstracción, desde funciones, módulos, a tipos abstractos de datos y, a continuación, objetos.

15.1.1. Funciones y procedimientos

Las funciones y procedimientos son los mecanismos de abstracción más antiguos y más ampliamente utilizados por los programas. Las funciones permiten que determinadas tareas se puedan utilizar en muchos sitios, incluso en diferentes aplicaciones, se recogen en un lugar y se reutilizan. Los procedimientos permiten a los programadores organizar tareas repetitivas de una sola vez. Estas dos abstracciones evitan duplicaciones de código.

Las funciones y procedimientos permiten a los programadores la capacidad de implementar la ocultación de la información. Un programador escribe una función o conjunto de funciones que se utilizarán por muchos otros programadores. Otros programadores no necesitan conocer los detalles exactos de la implementación solo necesitan conocer la interfaz. Desgraciadamente las funciones abstractas no son un mecanismo eficiente para ocultación de información. Sólo resuelven el problema parcial de muchos programadores que hacen uso del mismo nombre.

Lenguajes como Pascal, introdujeron un mejor control sobre la visibilidad de nombres mediante los nombres globales y locales. Sin embargo, tampoco esta característica resuelve el problema y se necesita otro mecanismo de abstracción: el *módulo*.

15.1.2. Módulos

Un módulo es un mecanismo abstracto que se utiliza para crear y gestionar espacios de nombres. En su forma básica, un módulo proporciona al programador la posibilidad de dividir el espacio de nombres en dos partes, pública y privada. La parte pública es accesible a cualquier persona , mientras que la parte privada sólo es accesible dentro del módulo. El módulo es un mecanismo abstracto que resuelve el problema de ocultación de la información. Por ejemplo, con un módulo se pueden ocultar los detalles de una pila. Los módulos facilitan el mantenimiento del software ya que el aislamiento permite que el código sea modificado y ampliado y los errores fijados sin el riesgo de introducir efectos laterales innecesarios o no deseados.

Sin embargo, el módulo todavía tiene problemas. Si se desearán manejar dos pilas a la vez, se requerirían dos módulos, no se pueden manejar las dos a la vez; lo mismo podríamos decir si el ejemplo son números complejos (números con parte real e imaginaria y operaciones aritméticas y funciones a realizar sobre sus miembros). No se puede *instanciar* el módulo; dicho de otro modo, no se pueden crear múltiples copias de áreas de datos.

15.1.3. Tipos datos abstractos

Un tipo abstracto de datos es un tipo de dato definido por el programador que se puede manipular de un modo similar a un tipo de dato predefinido. Los programadores pueden crear instancias de tipos abstractos de datos asignando valores legales a las variables. Además se pueden utilizar las funciones para manipular los valores asignados a las variables.

En síntesis, los **TDA** permiten las siguientes tareas:

1. Extender un lenguaje de programación añadiendo tipos de datos definidos por el usuario.
2. Poner a disposición de otro código un conjunto de funciones definidas por el programador que se utilizan para manipular los datos de las instancias de los tipos definidos por el programador.
3. Proteger (ocultar) los datos asociados a las instancias con el tipo y limitar el acceso a los datos, permitiendo el acceso sólo por las funciones definidas por el programador.
4. Hacer instancias (ilimitadas) del tipo de dato definido por el programador.

LENGUAJES CON TIPOS DE DATOS ABSTRACTOS: CLU, Turbo Pascal, Ada.

15.2. MODELADO DEL MUNDO REAL: CLASES Y OBJETOS

El segundo gran problema —y tal vez el más importante— del paradigma procedimental es que la disposición independiente de datos y funciones realiza un pobre modelado de cosas del mundo real donde van asociados. En el mundo físico, se trata con objetos tales como personas y autos. Tales objetos no son ni como los datos ni como las funciones. Los objetos complejos del mundo real tienen *atributos, comportamiento e identidad* y representan cosas concretas del mundo real:

Vuelo IB 1020 de Madrid a Santo Domingo.

Casa n.º 146 de la Calle Real.

Flor Amarilla del jarrón de flores de la habitación.

Un objeto no sólo encapsula su *estado* sino también su *comportamiento*. El comportamiento de un objeto se describe en términos de *servicios*(*operaciones*) proporcionados por ese objeto que modifi-

can o inspeccionan el estado. Estos servicios se invocan *enviando* mensajes del objeto que solicita el servicio al objeto sobre el que se envía el mensaje. Normalmente el único medio para acceder a un objeto es a través de las operaciones. Estas operaciones, por consiguiente, actúan como una *interfaz* al objeto. Un objeto es un conjunto de tres características.

1. objeto = identidad + variables + operaciones
2. objeto = identidad + estado + comportamiento

15.2.1. Atributos

Los atributos son las propiedades de los objetos (a veces denominados *características*). Por ejemplo, el color de los ojos, el título de un empleado, en el caso de personas y en el caso de autos, la potencia, el número de puertas o el modelo. A su vez los atributos del mundo real son equivalentes a los datos de un programa: tienen un cierto valor específico, tal como azul (color de los ojos), 150 CV (potencia en caballos de vapor en automóviles) o cinco (para el número de puertas).

```
@cg:cV^Yhcg:gYWFUWYf]nLb`dcf`i b`Web`i bhc`XY`Uhf]Vi hcg`fdfcdjYXUXYgE"
```

Una mesa tiene patas, un tablero, tamaño, color, etc

Los atributos representan la identificación y propiedades descriptivas tales como un empleado que se asigna a un proyecto o un libro que se presta a un socio. A nivel de lenguaje de programación los objetos que tienen el mismo conjunto de atributos se dice pertenecen a la misma clase.

El *estado* agrupa los valores instantáneos de todos los atributos de un objeto donde un atributo es una información que cualifica al objeto que la contiene. Cada atributo puede tomar un valor en un ámbito de definición dado. El estado de un objeto, en un instante dado, corresponde a una selección de valores, entre todos los valores posibles de los diferentes atributos. El estado evoluciona con el tiempo

Ejemplo: Un auto perteneciente a la clase Auto tiene los atributos: Marca, color, peso y potencia.

```
Un auto : Audi
           Azul cielo
           1.100 Kg.
           150 CV
```

El estado de auto es variable. Algunos componentes son constantes (marca, país de su fabricación, etc.). Los litros del tanque de la gasolina varía a medida que circula; el color puede cambiar si su dueño decide pintarlo con un color distinto al de fábrica.

```
9`Web`i bhc`XY`Uhf]Vi hcg`fYdfYgYbhUY`YghUXc XY`cV^Yhc`m`Ug`cdYfUWcbYg`fYdfYgYbhUb`Y`Wa dcfhUa]Ybhc"
```

```
I b`cV^Yhc`Yg`i bU`Ybh]XUX`ei Y`YbWldgi`U`jbZcfa UWfE`niWa dcfhUa]Ybhc"
```

```
9`YghUXc`XY`i b`cV^Yhc`Wa V]U`Web`Y`h]Ya dc"9`cV^Yhc`bc`Yg`Yghzh]We"
```

```
@U`jbghUbWU`XY`cV^Yhc`gY`WYUgY`UWfU]nU`mYj`Ybhi`Ua`YbhY`gY`XYghfi`nY"
```

15.2.2. Comportamiento

El comportamiento es algo que un objeto del mundo real hace en respuesta a los mismos estímulos. Por ejemplo, si usted solicita a su empresa un aumento de sueldo, normalmente le contestarán si o no. Si se aplica una acción sobre los frenos de un coche, normalmente, se detendrá. Aumentar el sueldo y detener son ejemplos de comportamiento. El comportamiento es como una función: se llama a una función para hacer algo (visualizar el inventario, por ejemplo) y se hace.

El comportamiento agrupa todas las competencias de un objeto y describe las acciones y reacciones de ese objeto. Cada elemento de comportamiento se denomina *operación*. Las operaciones de un objeto se ejecutan a consecuencia de un estímulo externo, representado en forma de un mensaje enviado por otro objeto.

15.2.3. Identidad

La identidad es una propiedad fijada por la cual se identifica a un objeto de otro. Si tiene dos tazas de café del mismo juego, se dice que son iguales pero no idénticas. Ambas son de igual tamaño, color, forma, material, están vacías (o llenas), etc, es decir, son iguales, pero no son idénticas, ya que usted puede elegir para tomar su café una u otra.

Un objeto posee una identidad que caracteriza su propia existencia. La identidad permite distinguir los objetos de forma no ambigua, independientemente de su estado. Esto permite distinguir dos objetos en los que todos los valores de atributos son idénticos.

7UXUcV^Yhc`dcgYY'i bU]XYbh]XUX XY'a UbYfU]a d`WWhU'

15.2.4. Paso de mensajes

El paso de mensajes. Una acción se inicia por una petición de un servicio (mensaje) que se envía a un objeto específico, no como en programación en la que una llamada de función utiliza un dato específico. El estilo de programación imperativo presta especial importancia a las funciones, mientras que el estilo orientado a objetos proporciona especial importancia al objeto (valor). Por ejemplo se puede llamar a la función poner un dato en una pila o bien llamar a una pila para poner un valor en la misma.

El paso de mensajes proporciona la capacidad de sobrecargar nombres y reutilizar software; esto no es posible utilizando el estilo imperativo. Implícito a la idea de mensaje es la idea de la interpretación de un mensaje que puede variar con objetos de diferentes clases. Es decir, el comportamiento dependerá de la clase de objeto que recibe el mensaje. La función poner puede significar una cosa en objetos pila y otra muy distinta en objetos que representan a una cafetería.

Se puede solicitar cierta información de un objeto. Esta información puede ser relativa al estado de la instancia del objeto pero también puede implicar un cálculo de algún tipo.

15.3. EL ENFOQUE ORIENTADO A OBJETOS

La idea fundamental que subyace en los lenguajes orientados a objetos es combinar en una única unidad tanto *datos* como *funciones que operan sobre esos datos*. Tal unidad se denomina *objeto*. Un objeto es una abstracción de algo en un dominio del problema, reflejando la posibilidad de un sistema para mantener información sobre el mismo, interactuar con él en ambos casos.

El mecanismo/concepto fundamental de la programación orientada a objetos es un **objeto**. Un objeto es un paquete de software que consta de los atributos (datos) y los métodos (código) que actúan sobre los datos. Los datos no son accesibles directamente a los usuarios del objeto. El acceso a los

datos se garantiza sólo por los métodos, o código, proporcionado por el objeto (por ejemplo, llamadas de las funciones a sus métodos).

La orientación a objetos se denomina así porque este método ve las cosas que son parte del mundo real como objetos. Un teléfono es un objeto; la silla en que está sentado es un objeto; el libro que está leyendo es un objeto, etc. De igual forma son objetos, una bicicleta, una póliza de seguros, etc.

Los objetos individuales de una clase se llaman **instancias** de esa clase.

Ejemplo: Clase Mesa Instancias Mi mesa Tu mesa, etc.

Estas instancias tienen los mismos atributos y posiblemente diferentes valores.

Los objetos que tienen el mismo conjunto de atributos se dice que pertenecen a la misma clase. Los objetos individuales de una clase se llaman *instancias* de esa clase. Las instancias tienen los mismos atributos pero con valores diferentes.

POO encapsula datos (atributos) y métodos (comportamiento) en objetos. Los datos y métodos de un objeto se conectan juntos. Los objetos tienen la propiedad de la ocultación de la información. Esto significa que aunque los objetos pueden conocer como comunicarse con otros objetos a través de interfaces bien definidas, los objetos normalmente no están autorizados a conocer como se implementan otros objetos —los detalles de implementación se ocultan dentro de los propios objetos.

Se puede conducir un auto eficientemente sin conocer los detalles de cómo funciona internamente el sistema de transmisión, los frenos o el motor.

Los programadores de C se centran en escribir funciones. Los grupos de acciones que realizan algunas tareas se forman en funciones y las funciones se agrupan para formar programas. Los datos son muy importantes como soporte a las acciones que se ejecutan. Los programadores OO se centran en crear sus propios tipos definidos por el usuario denominados clases. Las clases se conocen también como tipos definidos por el programador. Cada clase contiene los datos, así como el conjunto de métodos que manipulan los datos.

Los componentes datos de una clase se llaman *variables de instancia* (se denominan miembros dato en C++). Igual que una instancia de un tipo predefinido tal como int se llama variable, una instancia de un tipo definido por el usuario (por ejemplo, una clase) se denomina objeto.

Los *nombres* de un documento de requisitos ayudan al programador C++/Java a determinar un conjunto inicial de clases con las cuales comenzar el proceso de diseño. Las clases se utilizan entonces para *instanciar* objetos que trabajan juntos para implementar el sistema. Los verbos en un documento de requisitos del sistema ayudan al programador en C a determinar el conjunto de funciones que trabajan juntas para implementar el sistema.

Las funciones de un objeto, denominadas *funciones miembro* en C++, proporcionan el único método para acceder a sus datos. Si se desean leer los datos de un objeto, se llama a una función miembro del objeto. Se accederán a los datos y devuelve el valor. No se puede acceder a los datos directamente. Los datos están *ocultos*, así están protegidos de modificaciones accidentales. Los datos y las funciones se dice que están *encapsulados* en una única entidad.

Si se desea modificar los datos de un objeto, se conoce exactamente cuáles funciones interactúan con las funciones miembro del objeto. Ninguna otra función puede acceder a los datos. Esto simplifica la escritura, depuración y mantenimiento del programa.

I b\df c[f\Ua U\cf]YbhUXc 'U\cV^Yhcg'gY'Wa dc\bcf a U\a YbhY'XY'i b\b•a Yfc 'XY'cV^Yhcg'ei Y'gY
Wa i b\Wb i b\Ug'Wb chf\Ug' 'Ua UbXc 'U'Ug'Zi b\WcbYg'a]Ya Vfcg"

@U'Zi b\WcbYg'a]Ya Vfcg'XY'7\ZgY'XYbca]b\Ub'a f\hcXcg'Yb'chf\cg'`Yb[i U'Yg'hUYg'Wa c'Ga U'hU_"

@cg'YYa Ybhcg'XUhcg'gY'WbcWb'Wa c'Uhf]Vi hcg c'j UF]UV'Yg'XY'bg\hUWbWJU"

@U'Ua UXU'U'i b\UZi b\WcE'a]Ya Vfc'XY'i b'cV^Yhcg'gY'WbcW'Wa c'Ybj]UF'i b'a Ybg'U'YU'cV^Yhcg"

Ejemplo:

Listados con objetos del mundo real que se corresponden con objetos en programación orientada a objetos.

Objetos físicos

- Automóviles en un sistema de simulación de tráficos.
- Camiones en una empresa transportista.
- Componentes eléctricos en un programa de diseño de circuitos.
- Países en un modelo de comercio.
- Aviones en un sistema de tráfico aéreo.

Elementos de un sistema informático

- Unidad central, teclado, impresora, unidad de discos, unidad de DVD-CDRW, etc.
- Menús.
- Ventanas.
- Objetos gráficos (líneas, rectángulos, círculos, etc.).

Estructuras de datos

- Pilas.
- Colas.
- Listas enlazadas.
- Árboles binarios.
- Árboles binarios de búsqueda.
- Dobles colas.

Tipos de datos definidos por el usuario

- El tiempo en formato horario.
- Números complejos.
- Puntos de un plano.
- Ángulos de un sistema.
- Fecha de un día.

Recursos Humanos

- Empleados.
- Clientes.
- Proveedores.
- Socios.
- Vendedores.

Almacenes de datos

- Archivos de personal.
- Archivos de cursos.
- Inventario.
- Diccionario.
- Stock de artículos.

15.4. CLASES

En **POO** se dice que los objetos son miembros de *clases*. ¿Qué significa esto? Casi todos los lenguajes de programación tienen tipos de datos incorporados. Por ejemplo, un tipo de dato entero (`int`) está predefinido en C++ y en Java. Se pueden declarar tantas variables de tipo `int` como sea necesario en su programa:

- `int` día
- `int` mes
- `int` divisor
- `int` salario

De modo similar se pueden definir muchos objetos de la misma clase (*figura*, *mesa*, etc.) que sirven como un plano o un proyecto. Especifica que datos y funciones se incluirán en los objetos de esa clase. La definición de la clase no crea objetos, al igual que la simple existencia de tipos de datos no crea ninguna variable.

Una clase es por consiguiente una descripción de un número determinado de objetos similares. Ricky Martin, Chayanne, Tamara y Shakira son miembros de la clase "cantantes latinos". No hay ninguna persona llamada "Cantante Latino" sin embargo hay personas específicas con nombres específicos que son miembros de esta clase si poseen unas características determinadas.

I b'cV^Yhc 'Yg i bU Ä]bgħUbWUÄ XY i bUWUgY"
 I b'cV^Yhc 'h]YbY i bUa]ga Uf YUW]B 'Wb i bUWUgY ei Y i bUj Uf]UvYh]YbY Wb i b'h]dc 'XY XUhC"

15.4.1. Declaración de clases

```
clase <nombre_de_clase>
    //Declaración de atributos
    //Declaración de operaciones (métodos y constructores)
fin_clase
```

<nombre_de_clase> es un identificador válido.

Ejemplo:

```
clase auto
var
    privado entero: añoDeFabricación
    privado real: mileage
    privado cadena: licenciaConducir

público entero función obtenerAñoFabricación ( )
...
inicio
...
devolver(....)
fin_función

público procedimiento obtenerMileage( )
...
```

```

    inicio
    ...
    fin_procedimiento

    público real función kmRecorridos ( )
    ...
    inicio
    ...
    devolver (...)
    fin_función

    ...

    público constructor auto ( )
    inicio
    ...
    fin_constructor
fin_clase

```

15.5. REPRESENTACIÓN GRÁFICA DE UNA CLASE EN UML¹

En **UML** las clases se definen como un conjunto de objetos con un nombre, atributos y operaciones. Una clase se dibuja en **UML** con un rectángulo que se divide horizontalmente en tres bandas. La banda superior contiene el nombre de la clase; la banda central los atributos de la clase; y la banda inferior contiene las operaciones de la clase.

Los atributos con un signo más (+) delante de ellos son *públicos*, que significa que otras clases pueden acceder a ellos. Los atributos precedidos con un signo menos (-) son *privados*, lo que indica que sólo la clase y sus objetos pueden acceder a ellos. Por último, los atributos *protagonistas* rotulados con el número de signo (#) pueden ser utilizados por la clase y por cualquier descendiente de la clase (especialización).

Nombre de la clase
-Atributos
+Operaciones()

15.5.1. Atributos

Los atributos pueden ser descriptivos, de nombres y referencias. Un atributo es una propiedad de una clase. Describe el rango de valores que la propiedad puede tener en los objetos de esa clase. Una clase puede tener cero o más atributos. Por convenio se suele escribir el nombre del atributo en letras minúsculas. Si el nombre consta de más de una palabra, se unen cada palabra diferente comienza con una letra mayúscula:

```

nombreMarca
nombreModelo
númeroSerie

```

¹ Lenguaje de Modelado Unificado. **Unified Model Language**. Lenguaje universal estándar, muy utilizado en el análisis y diseño orientado a objetos.

capacidadAvión

Un objeto tiene un valor específico para cada uno de los atributos de su clase:

miLavadora: Lavadora

nombreMarca	= Fagor
nombreModelo	= A323
numeroSerie	= GL5674T
capacidad	= 30

Los atributos pueden mostrar su tipo así como su valor por defecto.

Lavadora
-nombreMarca : String = 'Fagor'

15.5.2. Operaciones

Una **operación** es algo que una clase puede hacer o que otra clase puede hacer a una clase. Se nombrá de igual forma que los atributos.

Lavadora
-nombreMarca : String = 'Fagor'
+encender()
+apagar()
+lavar()
+aclarar()
+secar()

Diagrama de clases

1. *Atributos* (estructura de los objetos: sus componentes y la información o datos contenidos en ellos).
2. *Operaciones* (comportamiento de los objetos: operaciones, servicios o métodos, etc.; no se utilizan procedimientos ni funciones).
3. *Restricciones* (condiciones, requisitos y reglas que los objetos deben cumplir).

Ejemplo: Clase Motocicleta .

Motocicleta
-marca : String
-color : String
-cilindrada : double
-velocidadMáxima : double
+arrancar()
+acelerar()
+frenar()

Ejemplo: Clase Factura con atributos.

Factura
+cantidad : double +fecha : double +cliente : String +especificación : String -administrador :String -númeroFactura : int +estado : String = 'Pendiente'

Ejemplo: Clase Cuenta Corriente con atributos y operaciones.

Cuenta Corriente
-número Cuenta : String -saldo : double
+depositar(entrada cantidad : souble) : Boolean +retirar(entrada cantidad : double) : Boolean +transferir() : double

15.5.3. Representación gráfica de una clase

1. Nombre de la clase.
2. Nombre de los atributos.
 - a) Tipo del atributo.
 - b) Valor inicial.
3. Operaciones.
 - a) Parámetros (nombre: tipo = valor inicial).
4. Definición de objetos.

La sentencia

```
nombreClase : obj1, obj2
```

define dos objetos `obj1` y `obj2`. La definición de la clase no crea objetos. Sólo describe cómo se llamarán cuando se crean. La definición de un objeto es similar a la definición de una variable de cualquier tipo de datos. Se reserva espacio en memoria para dichos objetos. La creación física del objeto se denomina *instanciación*. Un objeto se puede inicializar a sí mismo cuando se crea sin necesidad de requerir una llamada a una función miembro o se puede realizar una inicialización automática cuando se ejecuta un método llamado constructor. Un **constructor** es una función miembro que se ejecuta automáticamente cuando se crea un objeto.

```
Contador c1, c2
```

15.5.4. Notación de objetos

La notación **UML** de un objeto es un rectángulo con dos compartimentos. El compartimento superior contiene el nombre de un objeto y el nombre de la clase a la cual pertenece ese objeto. La sintaxis es:

nombreobjeto: nombrecclase

El compartimento inferior contiene la lista de nombres de atributos y sus valores. Los tipos de atributos se pueden mostrar utilizando la sintaxis:

```
nombreatributo: tipo = valor
```

Ejemplo: Objeto c1 de la clase Curso con dos atributos.

<u>c1 : Curso</u>	
Código_curso : String	= FM11
Nombre_curso : String	= Fundamentos de Programación

15.5.5. Reglas para encontrar clases en el análisis

Los conceptos fundamentales para encontrar clases en un documento de análisis de requisitos, se agrupan en cinco grandes categorías que permiten determinar qué tipos de cosas son clases.

Cosas tangibles

Se buscan cosas tangibles o del mundo real en el dominio del problema.

Avión	Fuente de alimentación	Libro	Auto
Reactor nuclear	Circuito de frenos	Perro	Moto
Caballo de carreras	Parque nacional	Gato	Banco
Curso	Grupo de clase	Fotocopia	Barco

El método más sencillo es subrayar los nombres en el documento de requisitos o especificaciones del problema, pero es preciso tener presente que este método es muy general y no siempre ayuda a encontrar los tipos de clases más interesantes.

Roles jugados por personas o instituciones

Doctor	Empleado	Supervisor
Paciente	Gerente	Jefe Departamento
Enfermero	Cliente	Ingeniero de Sistemas
Director	Socio	Analista

Incidentes/Eventos

Los incidentes representan una ocurrencia o evento; algo que sucede en un momento determinado.

Vuelo	Accidente	Rendimiento
Suceso	Rotura de un sistema	Llamada telefónica
Salida de un tren	Despegue de un avión	Llegada de un avión

Interacciones

Generalmente las interacciones tienen una calidad de “transacción” o “contrato” y es relativa a dos o más clases del modelo.

Compra	Arco (entre dos nodos)
Cargo en tarjeta de crédito	Reunión
Intersección	Contrato

Especificaciones

Las clases especificación se utilizan cuando un conjunto de clases comparten ciertas características.

- Producto de seguros
- Artículo de un libro
- Tipo de tarjeta de crédito
- Tipo de crédito

15.6. RESPONSABILIDAD DE UNA CLASE

Cada clase de un sistema debe ser responsable de un aspecto del mismo. Las propiedades localizadas en su área de responsabilidad se deben agrupar en una única clase y no dividirse en diversas clases. Una clase debe contener propiedades que no pertenezcan a su área de responsabilidad. Cada responsabilidad se asigna a una única clase. Cada clase es responsable de un aspecto del sistema total.

Una clase consta de atributos, operaciones, restricciones y relaciones. El principio de coherencia exige que todas las propiedades de una clase deben formar una conexión lógica. Si por ejemplo se desea crear una clase `Cliente` se debe determinar primero su responsabilidad: sobre qué elementos es responsable esta clase.

Ejemplo: Considerar un sistema de administración de clientes y definir las responsabilidades de las diferentes clases implicadas.

Cliente
<ul style="list-style-type: none"> • Administrar todos los datos personales de un cliente • Administrar direcciones, detalles de telecomunicaciones y cuentas de bancos

Direcciones
<ul style="list-style-type: none"> • Administrar y representar una dirección postal • Comprar la dirección frente a las tablas existentes y clases y códigos postales, insofar como esto es responsable o útil

CuentaBanco
<ul style="list-style-type: none"> • Administra y representa una cuenta en una institución financiera • En el caso de las cuentas domésticas del banco, comprueba el código de ordenación frente a una tabla existente de códigos de ordenación

15.7. DECLARACIÓN DE OBJETOS

La sentencia

```
NombreClase: obj1, obj2
```

define dos objetos `obj1` y `obj2`. La definición de la clase no crea objetos. Sólo describe como se llamarán cuando se crean. La definición de un objeto es similar a la definición de una variable de cualquier tipo de datos. Se reserva espacio en memoria para una variable que puede referenciar dichos objetos. La creación física del objeto se denomina *instanciación*. La instancia se efectúa mediante el operador `nuevo`, empleado por ejemplo en una sentencia de asignación de la siguiente forma

```
//llamada al constructor por defecto, sin argumentos  
obj1 ← nuevo NombreClase()
```

15.8. LOS MIEMBROS DE UN OBJETO

Los miembros de un objeto son los elementos dato, a los que también se puede denominar *variables de instancia*, y los *métodos*. Los métodos son acciones que se realizan por un objeto de una clase. Una invocación a un método es una petición al método para que ejecute su acción y lo haga con el objeto mencionado. La invocación de un método se denominaría también *llamar a un método y pasar un mensaje a un objeto*.

Existen dos tipos de métodos, aquellos que devuelven un valor único y aquellos que ejecutan alguna acción distinta de devolver un único valor. La diferencia entre unos y otros en pseudocódigo se destacará mediante el empleo de las palabras reservadas `procedimiento` y `función`. El paso de parámetros se regirá por las normas habituales descritas al tratar procedimientos y funciones. Por ejemplo en una clase `CuentaCorriente` el método `depositar`, que no devuelve ningún valor, se declararía:

```
público procedimiento depositar(E real: cantidad)  
...  
inicio  
...  
fin_procedimiento
```

El método `obtenerSaldo`, también de la clase `CuentaCorriente`, se supone devuelve el saldo y su declaración será:

```
público real función obtenerSaldo ( )  
...  
inicio  
...  
devolver(...)  
fin_función
```

La *llamada* o *invocación* a un método se puede realizar de dos formas, dependiendo de que el método devuelva o no un valor.

1. Si el método devuelve un valor, la llamada al método se trata normalmente como un valor.
2. Si el método realiza una acción distinta a devolver un único valor, una llamada al método debe ser una sentencia.

Así dada la declaración:

```
CuentaCorriente: miCuenta
```

las llamadas a los métodos depositar y obtenerSaldo se podrían efectuar de la siguiente forma:

```
miCuenta.depositar(2400)
saldo ← miCuenta.obtenerSaldo()
```

15.9. CONSTRUCTORES

Un **constructor** es un método que tiene el mismo nombre que la clase, cuyo propósito es inicializar los miembros datos de un nuevo objeto y que se ejecuta automáticamente cuando se crea un objeto de una clase. Sintácticamente es similar a un método. Dependiendo del número y tipos de los argumentos proporcionados, una función o método constructor se llama automáticamente cada vez que se crea un objeto. Si no se ha definido ningún constructor en la clase, el compilador proporciona un constructor por defecto. Cuando se define un constructor no se puede especificar un valor de retorno, un constructor nunca devuelve un valor. Un constructor puede, sin embargo, tomar cualquier número de parámetros (cero o más). A su rol como inicializador, un constructor puede también añadir otras tareas cuando es llamado.

```
constructor <nombre_de_clase>[(<lista_parámetros_formales>)]
//declaración de variables locales
inicio
    //código del constructor
fin_constructor
```

En el pseudocódigo utilizado se supone que, cuando un objeto ya no se necesita y se queda sin referencias, la memoria ocupada por ese objeto se libera automáticamente, sin necesidad de una destrucción explícita.

15.10. ACCESO A LOS MIEMBROS DE UN OBJETO, VISIBILIDAD Y ENCAPSULAMIENTO

Un objeto es una unidad que existe realmente y actúa en el sistema a desarrollar. Cada objeto es una instancia de una clase. Un objeto contiene datos (atributos) y algoritmos (operaciones) que actúan sobre esos datos. Para acceder a los miembros de un objeto se emplea el operador punto (.) .

```
<nombre_objeto>.<nombre_miembro> [ (lista_de_parámetros_actuales) ]
```

Por ejemplo

```
c1.Código_curso
```

Una característica de los objetos es que mantienen unidos o encapsulados sus miembros, es decir sus características y su comportamiento y un principio básico en programación orientada a objetos es la *ocultación de la información* que significa que, a determinados miembros de un objeto, no se puede acceder por funciones externas a la clase. Para controlar el acceso a los miembros de una clase se utilizan tres diferentes *especificadores de acceso*: público, privado y protegido. El mecanismo

principal para ocultar datos es ponerlos en una clase y hacerlos *privados*. A los datos o funciones privados sólo se puede acceder desde dentro de la clase. Por el contrario los datos o funciones *públicos* son accesibles desde el exterior de la clase. Los miembros *protegidos* son accesibles por funciones miembro de la misma clase o de clases derivadas de la misma, así como por amigas. Normalmente una clase debe tener sus campos de datos privados y todos o casi todos sus métodos públicos. Las reglas de visibilidad complementan el concepto de *encapsulamiento* de forma que las estructuras de datos internas utilizadas en la implementación de una clase no pueden ser accesibles directamente al usuario de la clase. Cuando una clase tiene su estructura interna oculta, privada, se facilita el mantenimiento del software, ya que se podrá mejorar la estructura de la clase sin tener que modificar los programas que la utilizan.

15.11. RESUMEN

POO es un método de organización de programas que resuelve limitaciones importantes existentes en la programación estructurada. En particular, la programación POO se organiza alrededor de objetos que contienen datos y las funciones que actúan sobre esos datos.

El lenguaje de modelado unificado (UML) es un método estándar para visualizar una estructura y las operaciones de un programa utilizando diagramas.

Clase. Una clase es la definición de los atributos, las operaciones y la semántica de un conjunto de objetos. Todos los objetos de una clase se corresponden con esa definición. El comportamiento de un objeto se describe por los posibles mensajes capaces de entender. Una clase es una plantilla para generar objetos.

Objeto. Es una unidad que existe realmente y actúa en el sistema a desarrollar. Cada objeto es una instancia de una clase. Un objeto contiene información representada por atributos cuya estructura se define en la clase. Un objeto puede recibir los mensajes definidos en la clase.

Atributos. Un atributo es un elemento (datos) contenido en cada objeto de una clase y se representa en cada objeto con un valor individual. Las clases definen tipos de atributos. Los objetos contienen valores de los *atributos*. Un tipo atributo puede ser un tipo *primitivo* o *predefinido* o puede ser otra clase.

Operaciones. Un objeto contiene datos (atributos) y algoritmos (operaciones) que actúan sobre esos datos. Una operación se declara en una clase. El procedimiento o función que implementa la operación se denomina *método*.

15.12. EJERCICIOS RESUELTOS

- 15.1.** Definir las clases que representan a los siguientes objetos del mundo real (atributos y operaciones) y a continuación dibujar sus representaciones gráficas en UML: a) Un número complejo; b) Un aparato de televisión; c) Una Libreta de Ahorros y una Cuenta corriente, y d) Una estructuras de datos clásicas (lista, pila, cola, árbol binario).

Número complejo

Atributos

Parte real

Parte imaginaria

Operaciones

Sumar()

Restar()

Multiplicar()

Dividir()

	Calcular el módulo()
	Leer el argumento
	Leer la parte real
	Leer la parte imaginaria
Libreta de Ahorros	Depósito (efectuado en...)
Atributos	Cantidad
	Fecha
Operaciones	Ingresar()
	Retirar()
	Transferir()
Lista	
Atributos	Frente
	Final
Operaciones	Primero()
	Último()
	Añadir()
	Quitar()
	Insertar()
	Vacía

15.2. Representar una clase círculo que se desea representar en una pantalla.

Teóricamente un círculo tiene un centro con unas coordenadas x, y , un radio $Radio$ para representarlo en la pantalla, se necesitará visualizar, borrar, cambiar de tamaño, mover, etc. Entre las restricciones que debe tener un círculo es que su radio no puede ser ni igual ni menor que cero.

Atributos: Radio, posición x, y

Operaciones: Visualizar, Ocultar, Cambiar de tamaño, Desplazar, etc.

Restricciones: Radio > 0

```

clase círculo
  var
    privado entero: radio
    privado Punto: puntoCentral

  público procedimiento fijarRadio(E entero: nuevoRadio)
    inicio
      si(nuevoRadio > 0) // restricción
        radio = nuevoRadio
        ...
      fin_si
    fin_procedimiento

  público procedimiento fijarPosición(E Punto: pos)
    ...
  público procedimiento visualizar( )
    ...
  público procedimiento ocultar( )
    ...
fin_clase // círculo
  
```

15.3. Representar mediante una clase una máquina lavadora; una plancha y un frigorífico.

Tomemos un catálogo de electrodomésticos o , simplemente, vayamos a la cocina de nuestra casa.

Máquina lavadora	Nombre de la marca Nombre del modelo Número de serie Capacidad Fecha de compra Garantía
	AñadirRopa: lógico sacarRopa: lógico añadirDetergente: real aclarar: lógico secar: lógico estado: lógico (encendida/apagada)
Plancha	Voltaje y frecuencia de la corriente Potencia Potencia del calderín Capacidad del calderón Tipo de suela de la plancha
	Desplazar Detener Girar
Frigorífico	Capacidad Número de estantes Número de puertas Termostato (lógico)
	Abrir Cerrar Descongelar Fijar temperatura

Realice el lector la definición de la clase y su representación gráfica.

15.4. Se trata de representar un sistema gráfico con dos clases: Punto y Rectángulo. Describir y representar en UML dichas clases.

Vamos a representar una clase especificando los atributos, las responsabilidades de la clase (operaciones previsibles a realizar) y los servicios u operaciones que puede, en consecuencia, ejecutar.

PUNTO**Responsabilidades**

El punto se representa en coordenadas cartesianas 2D (x, y).

Se puede crear un punto determinado con valores dados. Se puede eliminar dicho punto.

Los clientes de la clase pueden obtener y cambiar los valores de las coordenadas.

Está permitido sumar y restar puntos.

Está permitido multiplicar y dividir el punto por un entero.

Se utiliza el signo == para definir la igualdad (resultado, 1, si igual, 0 si es distinto).

Atributos

Abcisa	tipo entero
Ordenada	tipo entero

RECTÁNGULO**Responsabilidades**

Rectángulo en representación cartesiana 2D
 Operadores igual (==) y distinto (!=)
 Se puede crear un rectángulo dados la esquina superior izquierda
 y la esquina superior derecha
 Se puede borrar un rectángulo
 Los clientes pueden obtener valores: anchura, altura, longitud,
 vértice superiorIzquierdo, inferiorDerecho.
 Los clientes pueden solicitar mover el rectángulo un
 desplazamiento x, y
 Los clientes pueden solicitar si un determinado punto está
 dentro o fuera del rectángulo
 Los clientes pueden solicitar la intersección de las diagonales
 del rectángulo
 Los clientes pueden solicitar si se cruza con otro rectángulo

Servicios (operaciones)

```
Rectángulo( )    // crear un rectángulo en 0, 0
Rectángulo ( punto, punto) // un nuevo rectángulo
    Punto: superiorIzquierdo
    Punto: inferiorDerecho
Rectángulo( ) // destruir un rectángulo
Se definen operadores == y !=
Ent anchura( ) // proporcionar anchura del rectángulo
Ent altura( ) // proporcionar altura del rectángulo
Punto superiorIzquierdo( ) // proporcionar vértice superior izquierdo
Mover(ent dx, ent dy) // mover por incrementos
...

```

Atributos

Punto: vértice superior izquierdo
 Punto: vértice inferior derecho

- 15.5.** Definir y dibujar las siguientes clases de objetos: Motocicleta; Número complejo (números con parte real y parte imaginaria) y Aparato de televisión.

Motocicleta

Atributos Marca
 Color
 Cilindrada
 Velocidad Máxima

Operaciones Arrancar()
 Acelerar()
 Frenar()
 HacerCaballitos()

Número Complejo: Es aquel número que consta de una parte real y una parte imaginaria ($a+bi$)

```

Atributos Parte real
          Parte imaginaria

Operaciones CalcularMódulo
            SumarComplejos( )
            RestarComplejos( )
            Multiplicar( )
            Dividir( )
            CalcularArgumentos( )
            LeerParteReal( )
            LeerParteImaginaria( )

Aparato de televisión
Atributos Marca
          Color
          Modelo
          Tamaño en pulgadas
          Digital: Boolean (si o no)
          Hertzios

Operaciones Encender( )
            Apagar( )
            CambiarDeEmisora( )
            RegularVolumen( )
            Sintonizar( )

```

- 15.6.** Diseñe la clase Alumno con los campos Nombre, Nota1 y Nota2 y una función que calcule la media de ambas notas.

Análisis del problema

Al no especificar la visibilidad, se consideran los atributos como privados y los métodos públicos, por lo que, además de la función que calcula la media, se establecen procedimientos para leer y mostrar los atributos.

Codificación

```

clase Alumno
  var
    cadena: nombre
    real: nota1, nota2
  constructor Alumno (cadena: n)
    {El parámetro de un constructor siempre es de entrada}
    inicio
      nombre ← n
    fin_constructor
  procedimiento notas(E real: n1, n2)
    inicio
      nota1 ← n1
      nota2 ← n2
    fin_procedimiento
  real function media()

```

```

    inicio
        devolver ((nota1 + nota2) / 2)
    fin_función
procedimiento modificarnombre(E cadena: n)
    inicio
        nombre ← n
    fin_procedimiento
procedimiento mostrarnombre()
    inicio
        escribir(nombre)
    fin_procedimiento
fin_clase

```

15.7. Diseñe una clase Pila donde almacenar números naturales.

Análisis del problema

La implementación de una pila requiere el diseño de los métodos:

- **Pila**, el constructor. Crea una pila vacía asignando **nulo** al miembro privado **cima** que contiene la referencia a la cima (*tope*) de la pila.
- **vacía**. Determina si la pila está o no vacía comprobando el valor de **cima**; devuelve **true** cuando el valor de **cima** es **nulo** y **falso** en caso contrario.
- **apilar**. Añade un nuevo elemento en la cima de la pila, es decir crea un nuevo nodo cuyo campo **sig** referencia la **cima** de la pila y a continuación asigna a **cima** el nuevo nodo.
- **desapilar**. Comprueba que la pila no está vacía, suprime el nodo de la cima haciendo que **cima** pase a referenciar al siguiente nodo, o a **nulo** si la pila se ha quedado vacía, y devuelve el elemento perteneciente al nodo eliminado.
- **obtenerCima**. Devuelve el número natural almacenado en la cima de la pila o **-1** si la pila está vacía.

Codificación

```

clase Pila
    var
        privado Nodo: cima

    constructor Pila()
    inicio
        cima ← nulo
    fin_constructor

    público logico función vacía()
    inicio
        devolver(cima = nulo)
    fin_función

    público procedimiento apilar(E entero: elemento)
    inicio
        cima ← nuevo Nodo(elemento, cima)
    fin_procedimiento

    público entero función desapilar()
    var

```

```
    entero: aux
inicio
    aux ← -1 //error
    si no vacía() entonces
        aux ← cima.inf
        cima ← cima.sig
    fin_si
    devolver(aux)
fin_función

público entero función obtenerCima()
inicio
    if no vacía() entonces
        devolver(cima.inf)
    si_no
        devolver (-1) //error
    fin_si
fin_función
fin_clase

clase Nodo
    var
        público entero: clave
        público Nodo: sig

constructor Nodo(entero: cl; Nodo: cima)
{no se especifica el tipo de los parámetros ya que sólo pueden
    ser de entrada}
inicio
    clave ← cl
    sig ← cima
fin_constructor

público procedimiento mostrarNodo()
inicio
    escribir(clave)
fin_procedimiento

fin_clase
```


16

RELACIONES: ASOCIACIÓN, GENERALIZACIÓN, HERENCIA

9b YghY Wdhi c gY JbfhfcXi Wb cg WbWdhc Zb XLb YbhUyg XY f YUWcbYg YbfhY WLgYg @U f YU WcbYg a zg ja dcf hUbhYg gcdcf hUXUg dcf Ua Uhcf UXY Ug a YhcXc c Ug XY cf JYbhUWCB UcV Yhcg m Yb dUf hJWUf dcf I A @ gcb Uge VJUJDEz UfYf UMCB m/ YbYfU]nUJDE#YgdYVJU]nUJDE

8Y a cXc YgdYVJU gY JbfhfcXi WY Y WbWdhc XY \ Yf YbVJUWa c Yl dcbyhY X]f YWc XY Uf Y! UJDE XY | YbYfU]nUJDE#YgdYVJU]nUJDE migY a i YghfU Vf c WYUf WLgYg XYf j UXUg @U \ Yf Yb! WU \ UW dcg]VY WYUf ^YfUf ei Ug XY WLgYg f YUWcbUXUg mif YXi W U Wbh]XUX XY WBX] [c f Y! Xi bXUbhY Yb Wa dcbyhYg XY WLgYg "9 gcdcf hY XY U \ Yf YbVJU Yg i bU XY Ug df cd]YXUXYg ei Y XZYf YbVJU cg Yb[i UYg cf JYbhUXcg UcV Yhcg XY cg Yb[i UYg VLgUXcg Yb cV Yhcg m Yb/ i UYg Yghf i Vh f UXcg

@U \ Yf YbVJU Yg U df cd]YXUX ei Y dYf a JhY XYZ]b]f bi Yj Ug WLgYg i gUbXc Wa c VUgY U WLgYg nU Yl JghYbhYg @U b Yj UWLgY fWLgY XYf j UXU \ Yf YXU cg Uhf]Vi hcg mWa dcf hUa]YbhC ei Y gcb YgdY! WZ]Wg XY Y U @U \ Yf YbVJU Yg i bU \ YffUa]YbhU dcXYf cgU ei Y df cdcf WcbUi b a UW UXYWUXc dUf U df cXi W gcZhk UFYZ]UVYZ Wa df Ybg]VYZ VUc WghYZ UXUdhUVY mf Yf h]]nUVY

16.1. RELACIONES ENTRE CLASES

Una relación es una conexión semántica entre clases. Permite que una clase conozca sobre los atributos, operaciones y relaciones de otras clases. Las relaciones que se pueden establecer entre clases son: *asociaciones, agregaciones, dependencias, generalizaciones y especializaciones*.

16.2. ASOCIACIONES

Una **asociación** es una conexión conceptual o semántica entre clases. Cuando una asociación conecta dos clases, cada clase envía mensajes a la otra en un diagrama de colaboración. *Una asociación es una abstracción de los enlaces que existen entre instancias de objetos.* Los siguientes diagramas muestran objetos enlazados a otros objetos y sus clases correspondientes asociadas. Las asociaciones se representan de igual modo que los enlaces. La diferencia entre un enlace y una asociación se determina de acuerdo al contexto del diagrama.

Notación gráfica

La asociación recibe un nombre y una especificación numérica (multiplicidad) de cuantos objetos de un lado de la asociación se corresponden con objetos del otro lado:

La ventana visualiza una o muchas figuras geométricas

I bU'UgcWJUWJD' Yg'i bUfYUWJD' YbfhY cV'Yhcg' XY'i bUc'a zg'WUgYg''@UfYUWJD' XY'UgcWJUWJD' dfcdcfWcbU i bUfYUWJD' YbfhY cV'Yhcg' XY'WUgYg' XUXUg'' Bcfa Ua YbhY gY' Ybj #Ub'a YbgU'Yg YbfhY cV'Yhcg' XY'Ug'UgcWJUWcbYg"

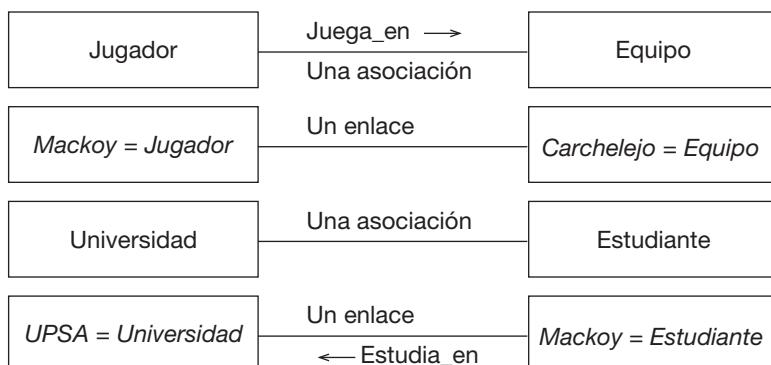


Figura 16.1. Asociación entre clases.

Las asociaciones pueden ser *bidireccionales* o *unidireccionales*. En **UML** las asociaciones bidireccionales se dibujan con flechas en ambos sentidos. Las asociaciones unidireccionales contienen una muestra que muestra la dirección de navegación.

Nombres de roles

En las asociaciones se pueden representar los roles o papeles que juegan cada clase dentro de las mismas. La Figura 16.2 muestra cómo se representan los roles de las clases. Un nombre de rol puede ser especificado en cualquier lado de la asociación. El siguiente ejemplo ilustra la asociación entre la clase Universidad y la clase Persona. El diagrama especifica que algunas personas actúan como **estudiantes** y algunas otras personas actúan como **profesores**. La segunda asociación también lleva un nombre de rol en la clase Universidad para indicar que la universidad actúa como un **empleado** (empleador) para sus profesores. Los nombres de los roles son especialmente interesantes cuando varias asociaciones conectan dos clases idénticas.

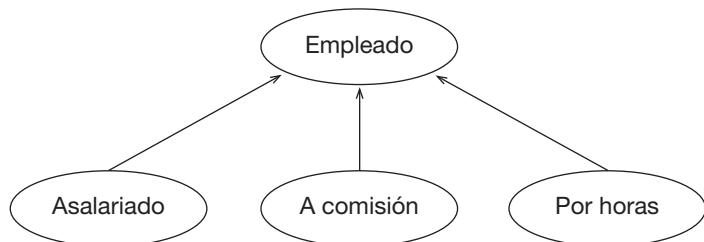


Figura 16.2. Roles en clases.

Multiplicidad

La multiplicidad representa la cantidad de objetos de una clase que se relacionan con un objeto de la clase asociada teniendo presente los roles específicos. La información de multiplicidad aparece en el diagrama de clases a continuación del rol correspondiente. La tabla inferior resume los valores más típicos de multiplicidad.

Tabla 16.1. Multiplicidad en asociaciones

Símbolo	Significado
1	Uno y sólo uno
0 .. 1	Cero o uno
m .. n	De m a n (enteros naturales)
*	De cero a muchos (cualquier entero positivo)
0 .. *	De cero a muchos (cualquier entero positivo)
1 .. *	De uno a muchos (cualquier entero positivo)

Ejemplo 16.1. Diferentes casos de multiplicidad

1	<i>exactamente uno</i>
0 , 1	<i>cero o uno</i>
0 .. 5	<i>entre cero y cinco</i>
3 , 8	<i>bien 3 o bien 8</i>
0 .. *	<i>mayor que o igual a cero (por defecto, si se omite la especificación)</i>
*	<i>muchos</i>
1 .. *	<i>uno o muchos, mayor que o igual a uno</i>
0 .. 4 , 7 , 8 .. *	<i>entre cero y cuatro, o exactamente 7, o mayor que o igual a ocho</i>



Figura 16.3. La multiplicidad representa el número de objetos de una clase que se pueden relacionar con un objeto de la clase asociada.

Ejemplo

Relación de asociación entre Universidad y la clase Persona. La clase Persona puede ser Estudiante y Profesor. Un estudiante normalmente está matriculado en una universidad y un profesor puede impartir clase en una o más universidades.

Las asociaciones pueden ser más complejas que la conexión de clases individuales entre sí. Se pueden conectar varias clases a una sola clase.

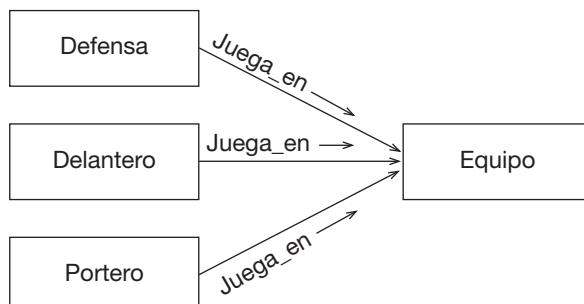


Figura 16.4. Asociación de varias clases a una clase.

Restricciones

En algunas ocasiones las asociaciones pueden establecer una restricción entre las clases. Las restricciones típicas pueden ser {ordenado} {or}.

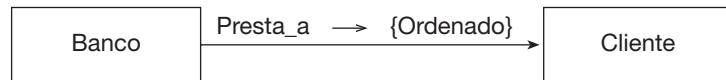


Figura 16.5. Restricciones.

Asociación cualificada

Cuando la multiplicidad de una asociación es de uno a muchos, se puede reducir esta multiplicidad de uno a uno con una cualificación. El símbolo que representa la cualificación es un pequeño rectángulo adjunto a la clase correspondiente.

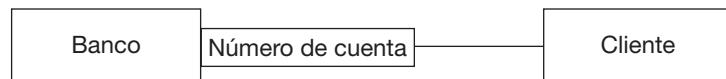


Figura 16.6. Asociación cualificada.

Asociaciones reflexivas

A veces, una clase es una asociación consigo misma. Esta situación se puede presentar cuando una clase tiene objetos que pueden jugar diferentes roles.

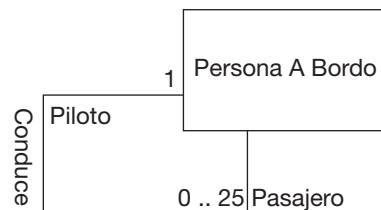


Figura 16.7. Asociación reflexiva.

16.3. AGREGACIONES

Una **agregación** es un tipo especial de asociación que expresa un acoplamiento más fuerte entre clases. Una de las clases juega un papel importante dentro de la relación con las otras clases. La agregación permite la representación de relaciones tales como «maestro y esclavo», «todo y parte de» o «compuesto y componentes». Los componentes y la clase que constituyen son una asociación que conforma un todo.

Las agregaciones representan conexiones bidireccionales y asimétricas. El concepto de agregación desde un punto de vista matemático es una relación que es transitiva, asimétrica y puede ser reflexiva.

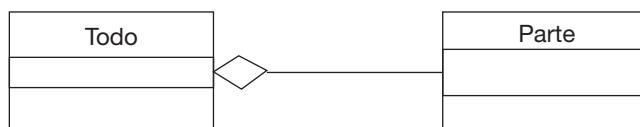


Figura 16.8. Relación de agregación.

Ejemplo 16.2

Una computadora es un conjunto de elementos que consta de una unidad central, teclado, ratón, monitor, unidad de CD-ROM, modém, ratón, altavoces, escáner, etc.

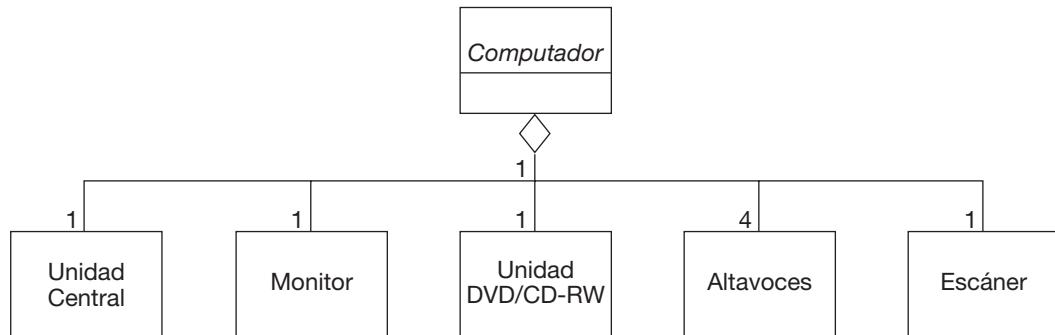


Figura 16.9. Agregación *Computador*.

Restricciones en las agregaciones

En ocasiones el conjunto de componentes posibles en una agregación se establece dentro de una relación **O**. Así, por ejemplo, el menú del día en un restaurante puede constar de: un primer plato (a elegir entre dos-tres platos), el segundo plato (a elegir entre dos-tres platos) y un postre (a elegir entre cuatro postres). El modelado de este tipo se realiza con la palabra reservada **O** dentro de llaves con una línea discontinua que conecte las dos líneas que conforman el todo.

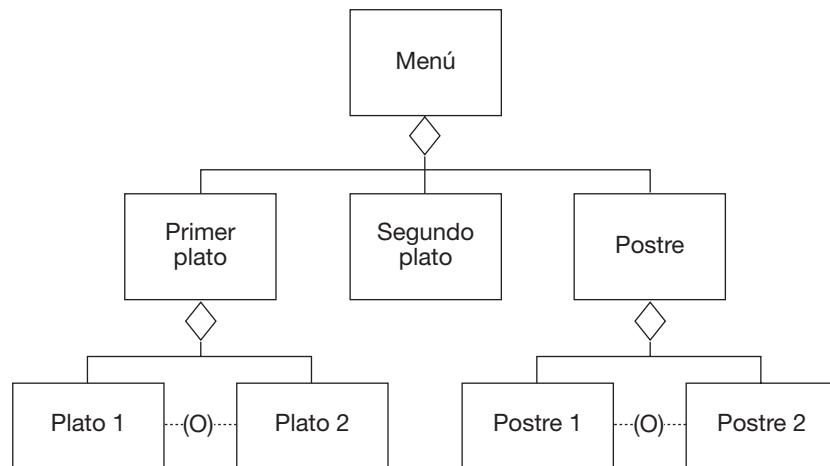


Figura 16.10. Restricciones en las agregaciones.

16.3.1. Composición

Una **composición** es un tipo especial de agregación. Cada componente dentro de una composición puede pertenecer tan sólo a un todo. El símbolo de una composición es el mismo que el de una agregación, excepto que el rombo está relleno.

Ejemplo

Una mesa para jugar al póker es una composición que consta de una superficie de la mesa y cuatro patas.

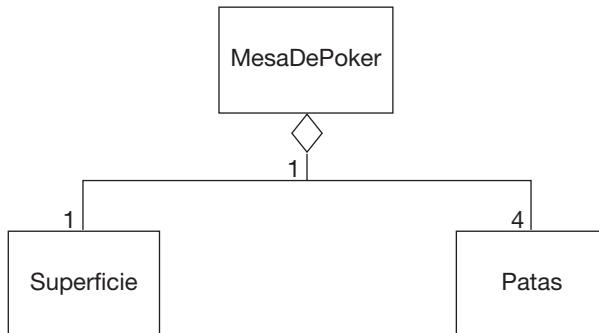


Figura 16.11. Composición.

Ejemplo

Un auto tiene un motor que no puede ser parte de otro auto. La eliminación completa del auto supone la eliminación de su motor.

REGLAS:

- 7UXU cV^Yhc 'Yg'i bU]bgħUbWJU XY'i bUWUgY"
- 5'[i bUg'WUgYg'Í UVghfUWUg'Í 'bc'di YXYb'JbghUbWJUf XjfYVWUa YbhY"
- 7UXU Yb'UV'Yg'i bU]bgħUbWJU XY'i bUUgcWUWJDE"

16.4. JERARQUÍA DE CLASES: GENERALIZACIÓN Y ESPECIALIZACIÓN

La jerarquía de clases (o clasificaciones) hacen lo posible para gestionar la complejidad ordenando objetos dentro de árboles de clases con niveles crecientes de abstracción. Las jerarquías de clase más conocidas son: **generalización** y **especialización**.

La generalización y especialización son principios de abstracción para la estructuración jerárquica de la semántica de un modelo. Generalización (o especialización) es una relación taxonómica entre uno elemento general y uno especial (o viceversa) donde el elemento especial añade propiedad al general y se comporta de un modo compatible con él.

En la generalización o especialización, las propiedades se estructuran jerárquicamente, propiedades de significado general se asignan a las clases más generales (superclases) y las propiedades más especiales se asignan a clases que están subordinadas a las clases generales (subclases). Por consiguiente, las propiedades de las superclases son «bestowed» en las subclases, por ejemplo, las subclases heredan propiedades de sus superclases. Por consiguiente una subclase, contiene tanto sus propias propiedades como las de sus superclases. Las subclases heredan todas las propiedades de sus superclases, pueden modificarlas y ampliarlas, pero no pueden eliminarlas ni suprimirlas.

La diferencia entre las *superclases* y *subclases* se realiza, normalmente, mediante una característica específica, denominada *discriminador*.

Notación

UML define a la generalización como herencia. De hecho, generalización es el concepto y herencia se considera la implementación del concepto en un lenguaje de programación.

Síntesis de generalización/Especialización

1. La generalización es una relación de herencia entre dos elementos de un modelo tal como clase. Permite a una clase heredar atributos y operaciones de otra clase. En realidad es la factorización de elementos comunes (atributos operaciones y restricciones) dentro de un conjunto de clases en una clase más general denominada **superclase**. Las clases están ordenadas dentro de una jerarquía; una superclase es una abstracción de sus subclases.
2. La flecha que representa la generalización entre dos clases apunta hacia la clase más general.
3. La especialización permite la captura de las características específicas de un conjunto de objetos que no han sido distinguidos por las clases ya identificadas. Las nuevas características se representan por una nueva clase, que es una subclase de una de las clases existentes. La especialización es una técnica muy eficiente para extender un conjunto de clases de un modo coherente.
4. La generalización y la especialización son dos puntos de vista opuestos del concepto de jerarquía de clasificación; expresan la dirección en que se extiende la jerarquía de clases.
5. Una generalización no lleva ningún nombre específico; siempre significa "es un tipo de", "es un", "es uno de", etc. La generalización sólo pertenece a clases, no se puede instanciar vía enlaces y por consiguiente no soporta el concepto de multiplicidad.
6. La generalización es una relación no reflexiva: una clase no se puede derivar de sí misma.
7. La generalización es una relación asimétrica: si la clase B se deriva de la clase A, entonces la clase A no se puede derivar de la clase B.
8. La generalización es una relación transitiva: si la clase C se deriva de la clase B que a su vez se deriva de la clase A, entonces la clase C se deriva de la clase A.

HERENCIA

Es la capacidad para crear nuevas clases (descendientes) que se construyen sobre otras existentes. Las clases derivadas heredan el código y los datos de las clases base, añadiendo su propio código especial y datos a ellas, e incluso, cambiando aquellos elementos de las clases base que necesitan sean diferentes. La herencia puede ser *simple* o *múltiple*. En herencia simple, una clase derivada hereda exactamente de una clase base (tiene sólo un parente). La herencia múltiple implica múltiples clases bases, una sola clase derivada tiene varios padres, lo que puede dar origen a ambigüedades. En este libro se trabajará fundamentalmente con herencia simple. Para definir clases descendientes se debe especificar en la cabecera de la clase, tras las palabras `hereda_de`, la clase ascendiente o antepasado, también denominada clase base o superclase.

```
clase <nombre_clase> hereda_de [<especificador_acceso>] <Clase_Base>
    // lista_de_miembros
fin_clase
```

- El *especificador de acceso* que declara el tipo de herencia es opcional (`público`, `privado` o `protegido`). La accesibilidad de los miembros heredados en la clase derivada vendrá dada por la combinación de los modificadores de los miembros de la clase base con el tipo de herencia.

Así, un *especificador de acceso público*, significa que los miembros públicos de la clase base son miembros públicos de la clase derivada. Con herencia privada los miembros públicos y protegidos de la clase base se vuelven miembros privados de la clase derivada.

- La *clase base* (*Clase_Base*) es el nombre de la clase de la que se deriva la nueva clase.
- La *lista de miembros* consta de datos y funciones miembro.

Existe compatibilidad de tipos desde un descendiente a un ascendiente, es decir, a un objeto ascendiente se le puede asignar una instancia cualquiera de sus tipos descendientes. Por ejemplo, si un parámetro formal es de un determinado tipo objeto, el parámetro actual correspondiente podrá ser de dicho tipo objeto o de un tipo descendiente del anterior. La propiedad de la herencia hace las tareas de programación mucho más fáciles y flexibles, no siendo necesario describir cada una de las características explícitamente para cada elemento, ya que los elementos pueden heredar características de otros.

Como ya se comentó en el capítulo anterior, encapsulación es una característica muy potente, y junto con la ocultación de la información, representan el concepto avanzado de objeto, que adquiere su mayor relevancia cuando encapsula e integra datos, más las operaciones que manipulan los datos en dicha entidad. Sin embargo, la orientación a objetos se caracteriza, además de por las propiedades anteriores, por incorporar la característica de *herencia*, propiedad que permite a los objetos ser construidos a partir de otros objetos. Dicho de otro modo, la capacidad de un objeto para utilizar las estructuras de datos y los métodos previstos en antepasados o ascendientes. El objetivo final es la **reutilización** (*reusability*)¹, es decir, reutilizar código anteriormente ya desarrollado.

La herencia se apoya en el significado de ese concepto en la vida diaria. Así, las clases básicas o fundamentales se dividen en subclases. Los animales se dividen en mamíferos, anfibios, insectos, pájaros, peces, etc. La clase vehículo se divide en subclase automóvil, motocicleta, camión, autobús, etc. El principio en que se basa la división de clases es la jerarquía, compartiendo características comunes. Así, todos los vehículos citados tienen un motor y ruedas, que son características comunes; si bien los camiones tienen una caja para transportar mercancías, mientras que las motocicletas tienen un manillar en lugar de un volante.

La herencia supone una *clase base* y una *jerarquía de clases* que contienen las *clases derivadas* de la clase base. Las clases derivadas pueden heredar el código y los datos de su clase base, añadiendo su propio código especial y datos a ellas, incluso cambiar aquellos elementos de la clase base que necesita sean diferentes.

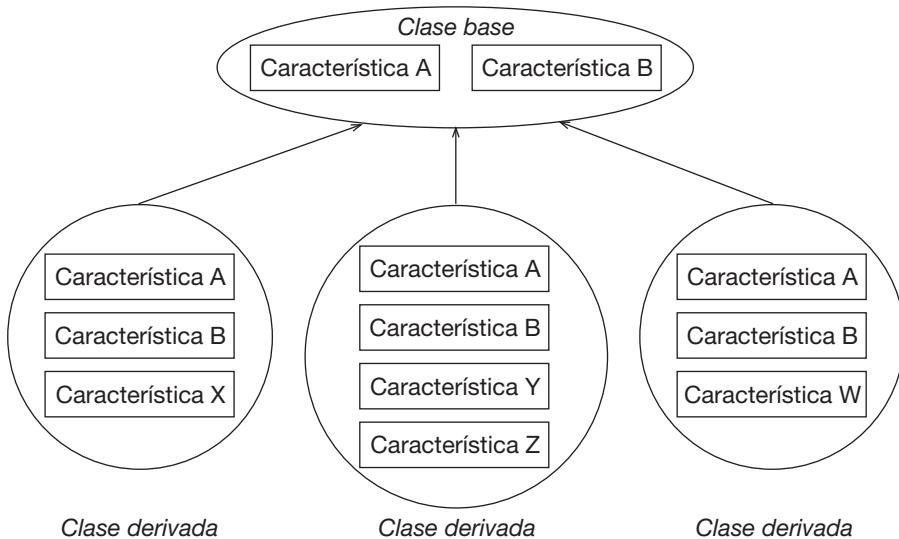
No se debe confundir las relaciones de los objetos con las clases, con las relaciones de una clase base con sus clases derivadas. Los objetos existentes en la memoria de la computadora expresan las características exactas de su clase y sirven como un módulo o plantilla. Las clases derivadas heredan características de su clase base, pero añaden otras características propias nuevas.

I bUWUgY \ Yf YXUgi g\WFUMYf tgh]Wg\fXUhcg'mZl bWcbYgLXY'chf UWUgY"

Así, se puede decir que una clase de objetos es un conjunto de objetos que comparten características y comportamientos comunes. Estas características y comportamientos se definen en una clase base. Las clases derivadas se crean en un proceso de definición de nuevos tipos y reutilización del código anteriormente desarrollado en la definición de sus clases base. Este proceso se denomina *programación por herencia*. Las clases que heredan propiedades de una clase base pueden a su vez servir como definiciones base de otras clases. Las jerarquías de clases se organizan en forma de árbol.

La herencia es un mecanismo potente para tratar con la evolución natural de un sistema con modificación incremental [Meyer, 1988]. Existen dos tipos diferentes de herencia: *simple* y *múltiple*.

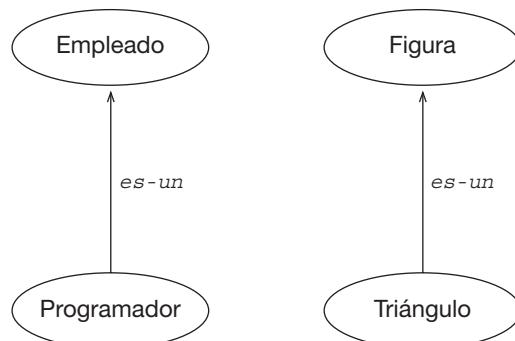
¹ Este término también se suele traducir por *reusabilidad*, aunque no es un término aceptado por el Diccionario de la Real Academia Española. La última edición del DRAE (22.^a edición, Madrid, 2001) incorpora los términos *reutilizar*, *reutilización* y *reutilizable* pero no acepta *reusar*, *reusable* ni *reusabilidad*. Mantenemos el término «*reusable*» en la obra por su gran difusión en la jerga informática.

**Figura 16.12.** Jerarquía de clases.

La relación de herencia se representa mediante una flecha larga con una punta vacía que apunta de la subclase a la superclase (Figura 16.13). Las flechas pueden ser dibujadas de dos formas: directamente de las subclases a las superclases o bien combinadas en una línea común.

Como ya se ha comentado, la herencia es la manifestación más clara de la relación de generalización/especialización y a la vez una de las propiedades más importantes de la orientación a objetos y posiblemente su característica más conocida y sobresaliente. Todos los lenguajes de programación orientados a objetos soportan directamente en su propio lenguaje construcciones que implementan de modo directo la relación entre clases derivadas.

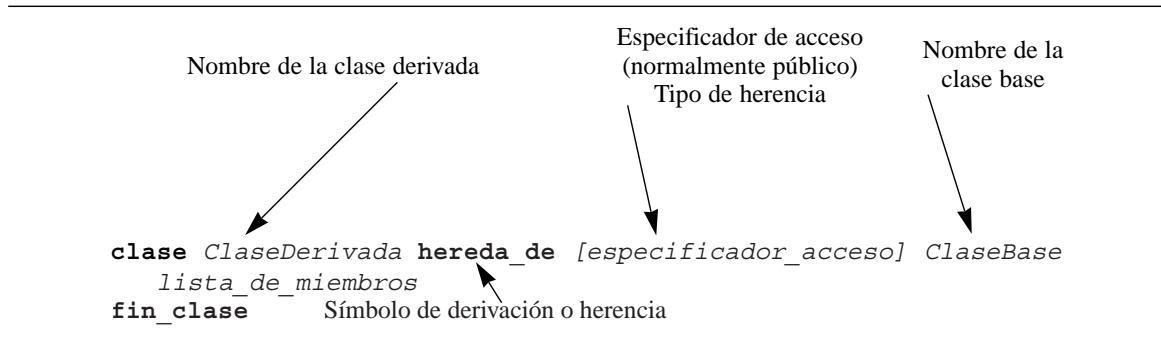
La *herencia* o relación **es-un** es la relación que existe entre dos clases, en la que una clase denominada *derivada* se crea a partir de otra ya existente, denominada *clase base*. Este concepto nace de la necesidad de construir una nueva clase y existe una clase que representa un concepto más general; en este caso la nueva clase puede *heredar* de la clase ya existente. Así, por ejemplo, si existe una clase *Figura* y se desea crear una clase *Triángulo*, esta clase *Triángulo* puede derivarse de *Figura* ya que tendrá en común con ella un estado y un comportamiento, aunque luego tendrá sus características propias. *Triángulo* *es-un* tipo de *Figura*. Otro ejemplo, puede ser *Programador* que *es-un* tipo de *Empleado*.

**Figura 16.13.** Clases derivadas.

Evidentemente, la clase base y la clase derivada tienen código y datos comunes, de modo que si se crea la clase derivada de modo independiente, se duplicaría mucho de lo que ya se ha escrito para la clase base. C++ soporta el mecanismo de *derivación* que permite crear clases derivadas, de modo que la nueva clase *hereda* todos los miembros datos y las funciones miembro que pertenecen a la clase ya existente.

La declaración de derivación de clases debe incluir el nombre de la clase base de la que se deriva y el especificador de acceso que indica el tipo de herencia (*pública*, *privada* y *protégida*). La primera línea de cada declaración debe incluir el formato siguiente:

Sintaxis de la declaración de una clase derivada



El especificador de acceso que declara el tipo de herencia es opcional (*pública*, *privada* o *protégida*); si se omite el especificador de acceso, se considera por defecto *pública*.

Especificador de acceso público significa que los miembros públicos de la clase base son miembros públicos de la clase derivada.

Herencia pública, es aquella en que el especificador de acceso es *público* (*público*).

Herencia privada, es aquella en que el especificador de acceso es *privado* (*privado*).

Herencia protegida, es aquella en que el especificador de acceso es *protégido* (*protégido*).

La *clase base* (*ClaseBase*) es el nombre de la clase de la que se deriva la nueva clase. La *lista de miembros* consta de datos y funciones miembro:

REGLA

En general, se debe incluir la palabra reservada **pública** en la primera línea de la declaración de la clase derivada, y representa herencia pública. Esta palabra reservada produce que todos los miembros que son públicos en la clase base permanecen públicos en la clase derivada.

Tipos de herencia

Existen dos mecanismos de herencia utilizados comúnmente en programación orientada a objetos: *herencia simple* y *herencia múltiple*.

Herencia simple es aquel tipo de herencia en la cual un objeto (*clase*) puede tener sólo un ascendiente, o dicho de otro modo, una subclase puede heredar datos y métodos de una única clase, así como añadir o quitar comportamientos de la clase base. **Herencia múltiple** es aquel tipo de herencia en la cual una clase puede tener más de un ascendiente inmediato, o lo que es igual, adquirir datos y métodos de más de una clase. **Object Pascal**, **Smalltalk**, **Java** y **C#**, sólo admiten herencia simple, mientras que **Eiffel** y **C++** admiten herencia simple y múltiple.

La Figura 16.14 representa los gráficos de herencia simple y herencia múltiple de la clase figura y persona, respectivamente.

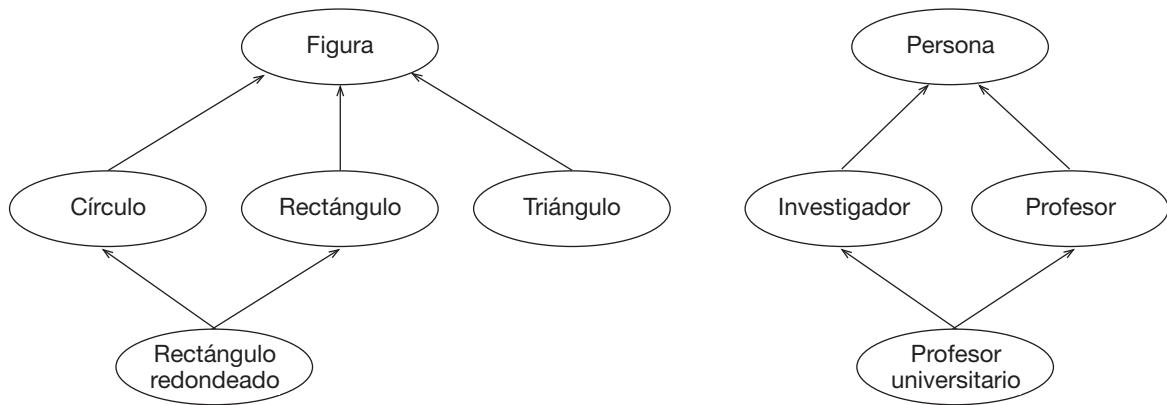


Figura 16.14. Tipos de herencia.

A primera vista, se puede suponer que la herencia múltiple es mejor que la herencia simple; sin embargo, como ahora comentaremos, no siempre será así. En general, prácticamente todo lo que se puede hacer con herencia múltiple se puede hacer con herencia simple, aunque a veces resulta más difícil. Una dificultad surge con la herencia múltiple cuando se combinan diferentes tipos de objetos, cada uno de los cuales define métodos o campos iguales. Supongamos dos tipos de objetos pertenecientes a las clases Gráficos y Sonidos, y se crea un nuevo objeto denominado Multimedia a partir de ellos. Gráficos tiene tres campos datos: tamaño, color y mapasDeBits, y los métodos dibujar, cargar, almacenar y escala; sonidos tiene dos campos dato, duración, voz y tono, y los métodos reproducir, cargar, escala y almacenar. Así, para un objeto Multimedia, el método escala significa poner el sonido en diferentes tonalidades, o bien aumentar/reducir el tamaño de la escala del gráfico.

Naturalmente, el problema que se produce es la ambigüedad, y se tendrá que resolver con una operación de prioridad que el correspondiente lenguaje deberá soportar y entender en cada caso.

```
9b\fYU]XUXZb]`U\YfYbWUgla d'Y'b]`U\YfYbWUa • `h]d`Y`gcb`dYfZWMUg`Yb`hcXcg`cg
Wlgcgz mUa VUg`di YXYb`f Yei Yf ]f` i b`dcW`a zg`XY`V\X][ c`Yl hfU ei YfYdfYgYbhY`V]Yb
`Ug`X]ZYfYbWUg`Yb`Y`a cXc`XY`hfUVU`c"
```

Herencia simple (*herencia jerárquica*)

En esta jerarquía cada clase tiene como máximo una sola superclase. La herencia simple permite que una clase herede las propiedades de su superclase en una cadena jerárquica.

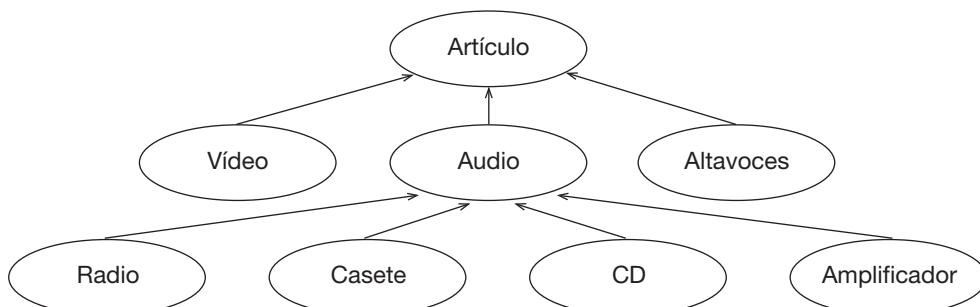


Figura 16.15. Herencia simple.

Herencia múltiple

Herencia simple es aquella en la que cada clase tiene como máximo una superclase. En herencia múltiple, una clase puede tener más de una superclase.

No todos los lenguajes de programación soportan herencia múltiple. C++ sí la soporta, Java y Smalltalk no la soportan.

En la Figura 16.16 la clase C tiene dos superclases, A y D. Por consiguiente, la clase C hereda las propiedades de las clases A y D. Evidentemente, esta acción puede producir un conflicto de nombres, donde la clase C hereda las mismas propiedades de A y D.

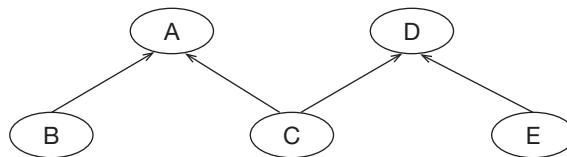


Figura 16.16. Herencia múltiple.

Herencia múltiple es un tipo de herencia en la que una clase hereda el estado (estructura) y el comportamiento de más de una clase base. En otras palabras, hay herencia múltiple cuando una clase hereda de más de una clase; es decir, existen múltiples clases base (*ascendientes o padres*) para la clase derivada (*descendiente o hija*).

La herencia múltiple entraña un concepto más complicado que la herencia simple, no sólo con respecto a la sintaxis sino también al diseño e implementación del compilador. La herencia múltiple también aumenta las operaciones auxiliares y complementarias y produce ambigüedades potenciales. Además, el diseño con clases derivadas por derivación múltiple tiende a producir más clases que el diseño con herencia simple. Sin embargo, y pese a los inconvenientes y ser un tema controvertido, la herencia múltiple puede simplificar los programas y proporcionar soluciones para resolver problemas difíciles. En la Figura 16.17 se muestran diferentes ejemplos de herencia múltiple.

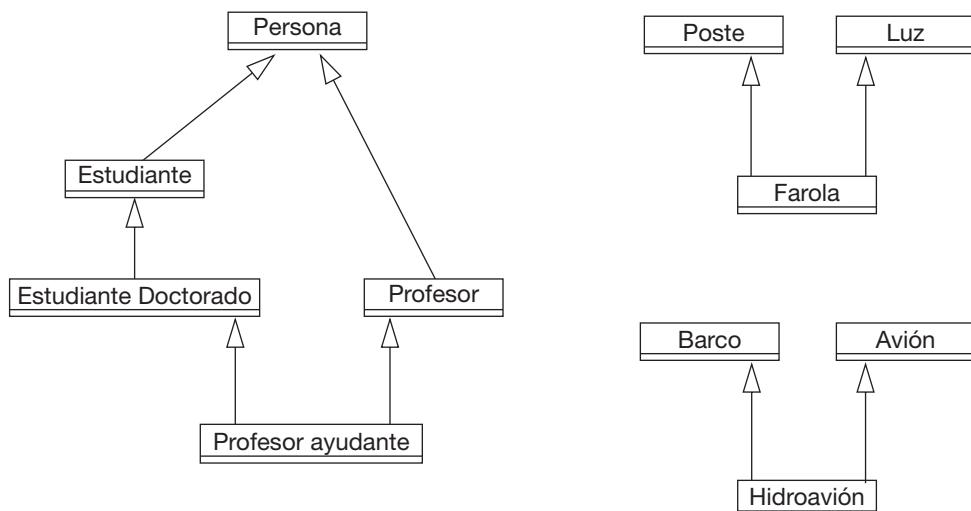


Figura 16.17. Ejemplos de herencia múltiple.

Regla

En herencia simple, una clase derivada hereda exactamente de una clase base (tiene sólo un parente). Herencia múltiple implica múltiples clases bases (tiene varios padres una clase derivada).

En herencia simple, el escenario es bastante sencillo, en términos de concepto y de implementación. En herencia múltiple los escenarios varían ya que las clases bases pueden proceder de diferentes sistemas y se requiere a la hora de la implementación un compilador de un lenguaje que soporte dicho tipo de herencia (C++ o Eiffel). ¿Porqué utilizar herencia múltiple? Pensamos que la herencia múltiple añade fortaleza a los programas y si se tiene precaución en la base del análisis y posterior diseño, ayuda bastante a la resolución de muchos problemas que tomen naturaleza de herencia múltiple.

Por otra parte, la herencia múltiple siempre se puede eliminar y convertirla en herencia simple si el lenguaje de implementación no la soporta o considera que tendrá dificultades en etapas posteriores a la implementación real. La sintaxis de la herencia múltiple es:

```
clase Cderivada hereda_de [especificador_acceso] Base1,  
                  [especificador_acceso] Base2, ...  
  
                  lista_de_miembros  
  
fin_clase
```

<i>CDerivada</i> <i>Base1, Base2, ...</i>	Nombre de la clase derivada Clases base con nombres diferentes
--	---

Funciones o datos miembro que tengan el mismo nombre en *Base1*, *Base2*, *Basen*, etc., serán motivo de ambigüedad.

Ejemplos

```
clase A hereda_de público B, público C  
...  
fin_clase  
  
clase D hereda_de público E, privado F, público G  
...  
fin_clase
```

La palabra reservada **público** ya se ha comentado anteriormente, define la relación «**es-un**» y crea un subtipo para herencia simple. Así en los ejemplos anteriores, la clase A «**es-un**» tipo de B y «**es-un**» tipo de C. La clase D se deriva públicamente de E y G y privadamente de F. Esta derivación hace a D un subtipo de E y G pero no un subtipo de F.

Ejemplo

```
clase Derivada hereda_de Base1, privado Base2  
...  
fin_clase
```

Derivada especifica derivación privada de *Base2* y derivación pública (por defecto u omisión) de *Base1*.

Regla

Asegúrese especificar un tipo de acceso en todas las clases base para evitar el acceso público por omisión. Utilice explícitamente privado cuando lo necesite para manejar la legibilidad.

```
clase Derivada hereda_de público Base1, privado Base2
...
fin_clase
```

Ejemplo

```
clase estudiante
...
fin_clase

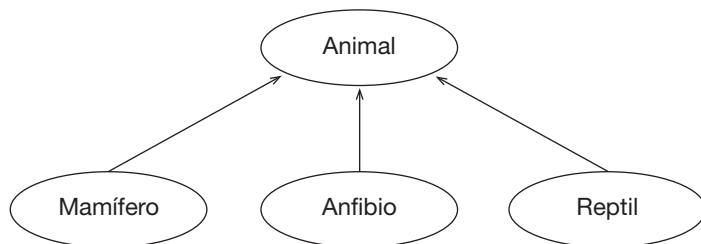
clase trabajador
...
fin_clase
clase estudiante_trabajador hereda_de público estudiante,
                                público trabajador
...
fin_clase
```

Características de la herencia múltiple

La herencia múltiple plantea diferentes problemas tales como la *ambigüedad* por el uso de nombres idénticos en diferentes clases base, y la *dominación* o *preponderancia* de funciones o datos.

Ejemplo 16.2

Dibujar un árbol de jerarquía de herencia que represente los siguientes animales, como mínimo: Anfibios, Mamíferos y Reptiles.



16.5. CLASES ABSTRACTAS

Con frecuencia, cuando se diseña un modelo orientado a objetos es útil introducir clases a cierto nivel que pueden no existir en la realidad pero que son construcciones conceptuales útiles. Estas clases se conocen como clases abstractas. Las clases abstractas son clases en las que algunos o todos los miembros no tienen implementación. Los métodos sin implementación se pueden declarar especificando exclusivamente la cabecera y no pueden ser privados.

```

abstracto clase <clase_base>
  [<modificador_acceso>] abstracto <tipo_dato> funcion <nombre_método>
    ([<parametros_formales>])
  [<modificador_acceso>] abstracto procedimiento <nombre_método>
    ([<parametros_formales>])
  ...
fin_clase

```

Una clase abstracta normalmente ocupa una posición adecuada en la jerarquía de clases que le permite actuar como un depósito de métodos y atributos compartidos para las subclases de nivel inmediatamente inferior. Las clases abstractas no tienen instancias directamente. Se utilizan para agrupar otras clases y capturar información que es común al grupo. Las subclases de clases abstractas, es decir las clases derivadas de una clase abstracta, se encargarán de implementar los métodos abstractos y sí pueden tener instancias. Una clase abstracta es COCHE_TRANSPORTE_PASAJEROS. Una subclase es SEAT, que puede tener instancias directamente, por ejemplo Coche1 y Coche2.

I bU'WUgY'UVghfUMUYg'i bUWUgY'ei Y'glfj Y'Wa c'WUgY'VUgY'Wa •bždYfc 'bc 'hYbXfz 'JbghUbWUg"

Una clase abstracta puede ser una impresora.

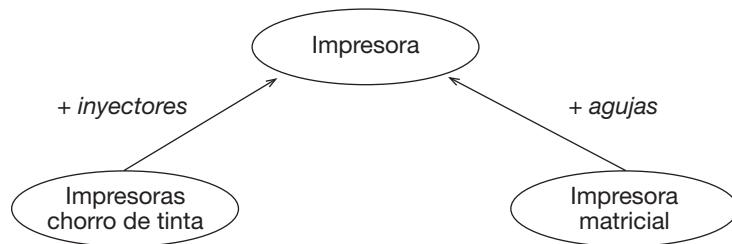


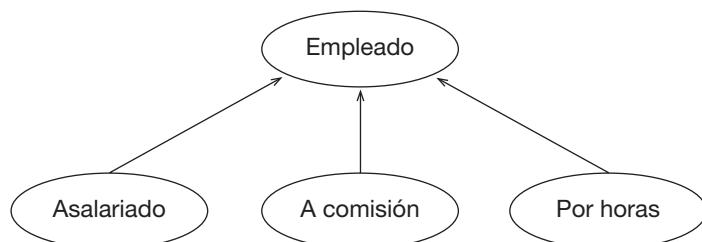
Figura 16.18. La clase abstracta impresora.

Las clases derivadas de una clase base se conocen como *clases concretas*, que ya pueden *instanciarse* (es decir, pueden tener *instancias*).

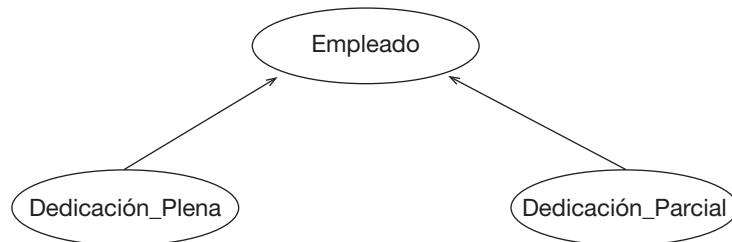
Consideraciones de diseño

A veces es difícil decidir cuál es la relación de herencia más óptima entre clases en el diseño de un programa. Consideremos, por ejemplo, el caso de los empleados o trabajadores de una empresa. Existen diferentes tipos de clasificaciones según el criterio de selección (se suele llamar *discriminador*) y pueden ser: modo de pago (suelto fijo, por horas, a comisión); dedicación a la empresa (plena o parcial) o estado de su relación laboral con la empresa (fijo o temporal).

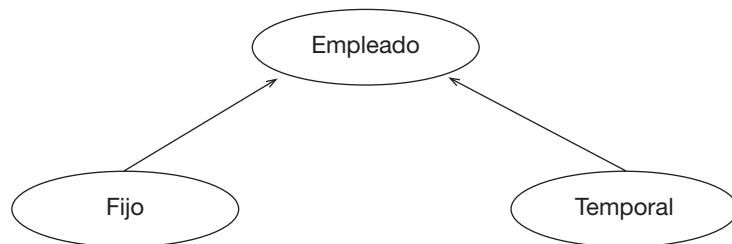
Una vista de los empleados basada en el modo de pago puede dividir a los empleados con salario mensual fijo; empleados con pago por horas de trabajo y empleados a comisión por las ventas realizadas.



Una vista de los empleados basada en el estado de dedicación a la empresa : dedicación plena o dedicación parcial.



Una vista de empleados basada en el estado laboral del empleado con la empresa: fija o temporal.



Una dificultad a la que suele enfrentarse el diseñador es que en los casos anteriores un mismo empleado puede pertenecer a diferentes grupos de trabajadores. Un empleado con dedicación plena puede ser remunerado con un salario mensual. Un empleado con dedicación parcial puede ser remunerado mediante comisiones y un empleado fijo puede ser remunerado por horas. Una pregunta usual es ¿cuál es la relación de herencia que describe la mayor cantidad de variación en los atributos de las clases y operaciones? ¿Esta relación ha de ser el fundamento del diseño de clases? Evidentemente la respuesta adecuada sólo se podrá dar cuando se tenga presente la aplicación real a desarrollar.

El texto original en inglés es:

"A class hierarchy based on employee dedication level (full-time or part-time) is problematic because an employee can belong to multiple groups. An employee with full-time dedication may be paid monthly salary. An employee with part-time dedication may be paid commissions and an employee fixed may be paid by hour. A common question is what inheritance relationship describes the greatest variation in attributes and operations of the classes? This relationship must be the foundation of class design? Obviously the appropriate answer can only be given when the real application is taken into account."

La traducción al español es:

"Un diagrama jerárquico de empleados basada en el estado laboral de la empresa: fija o temporal. Una dificultad a la que suele enfrentarse el diseñador es que en los casos anteriores un mismo empleado puede pertenecer a diferentes grupos de trabajadores. Un empleado con dedicación plena puede ser remunerado con un salario mensual. Un empleado con dedicación parcial puede ser remunerado mediante comisiones y un empleado fijo puede ser remunerado por horas. Una pregunta usual es ¿cuál es la relación de herencia que describe la mayor cantidad de variación en los atributos de las clases y operaciones? ¿Esta relación ha de ser el fundamento del diseño de clases? Evidentemente la respuesta adecuada sólo se podrá dar cuando se tenga presente la aplicación real a desarrollar.

16.6. POLIMORFISMO

Otra propiedad importante de la programación orientada a objetos es el polimorfismo. *Polimorfismo puro*², se produce cuando una única función se puede aplicar a una variedad de tipos o clases de objetos. En polimorfismo puro hay una función (el cuerpo del código) y un número de interpretaciones (significados diferentes). El polimorfismo supone que un mismo mensaje puede producir acciones (resultados) totalmente diferentes cuando se recibe por objetos diferentes.

² Este es el término utilizado por Timoty Budd en la segunda edición de su libro «*Understanding Object-Oriented Programming with Java*», Addison-Wesley, 2000, p. 195.

El polimorfismo, en su expresión más simple, es el uso de un nombre o un símbolo —por ejemplo, un operador— para representar o significar más de una acción. Esta propiedad, en su concepción básica, se encuentra en casi todos los lenguajes de programación, de tal forma que es frecuente que el símbolo + sirva tanto para realizar sumas aritméticas como para concatenar (unir) cadenas. Cuando a un operador existente en el lenguaje, tal como +, = o *, se le asigna la posibilidad de operar sobre un nuevo tipo de dato, se dice que está *sobrecargado*. La *sobrecarga*, que no sólo puede ser de operadores sino también de métodos, es una clase de polimorfismo.

Para implementar el polimorfismo en sentido estricto, un lenguaje debe poder trabajar con métodos virtuales que poseen ligadura dinámica, es decir, cuyas referencias se establecen en tiempo de ejecución. Como las referencias a métodos virtuales se resuelven en tiempo de ejecución, resulta posible ejecutar los métodos definidos en una determinada clase derivada al llamar a los de su clase base, si dicha clase derivada es la del objeto al que en ese momento se está pasando el mensaje. Dado que los lenguajes que siguen estrictamente el paradigma orientado a objetos ofrecen sólo ligadura dinámica, los métodos que se desarrollen como ejemplo presentarán este tipo de ligadura y serán virtuales por defecto.

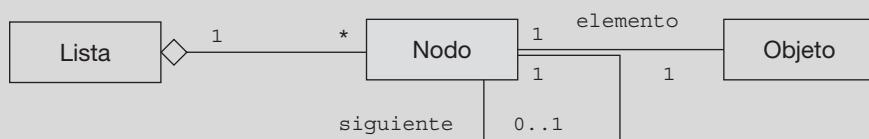
16.7. EJERCICIOS RESUELTOS

16.1. Representar una clase lista enlazada, capaz de contener una serie de objetos.

La lista es un ejemplo de clase que contiene otras clases y a las que se denomina clases contenedoras. En la representación de este tipo de clases se utiliza el símbolo:



que permite especificar la relación de composición y resaltar que la lista está formada por una colección de nodos (desde ninguno hasta muchos). En el diagrama también se refleja que cada nodo contiene una referencia al objeto que almacena y otra al siguiente Nodo (que puede ser nula). La relación entre Nodo y Objeto es de asociación y se representa por la línea continua que une ambas clases y, por encima de la misma, se indica el papel que el Objeto tiene en la asociación.



16.2. Cree un diagrama de clases que represente la estructura de una Facultad, teniendo en cuenta que ésta se divide en una serie de departamentos cada uno de los cuales tiene diversos profesores y además consta de una serie de alumnos.

En el diagrama se deben mostrar tanto los tipos de relación entre las clases, susyos símbolos son:

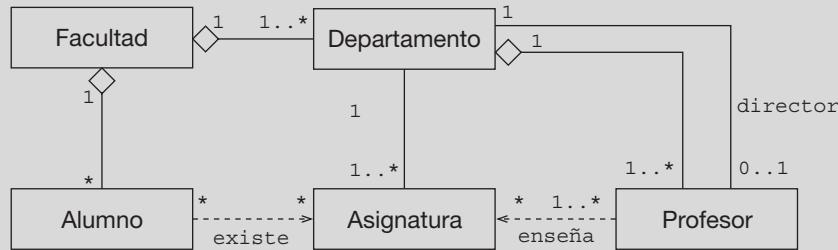
-----> dependencia

————— asociación

◇—— agregación

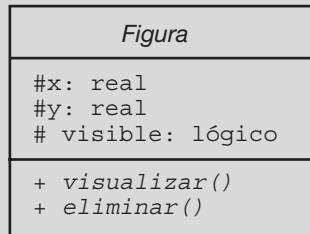
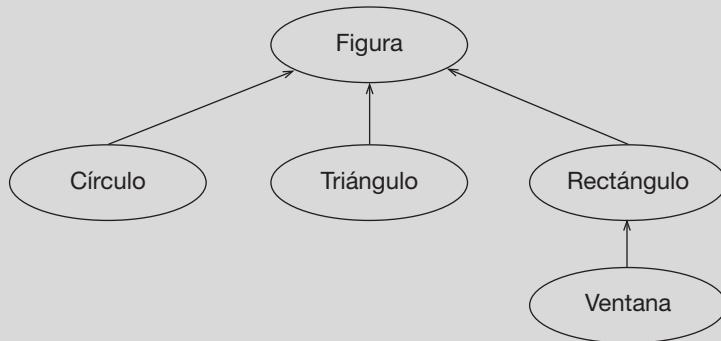
↑
herencia

como su cardinalidad.



- 16.3.** En una pantalla de computadora, las ventanas, círculos, triángulos y rectángulos se visualizan y se mueven. Representar un modelo de jerarquías de clases.

Las clases Círculo, Rectángulo y Triángulo se pueden generalizar y normalmente representan figuras geométricas. Las clases Círculo, Rectángulo y Triángulo, por consiguiente, son la especialización de la superclase común: FiguraGeométrica. El *discriminador* puede ser *tipo de figura*. En la superclases abstracta, las operaciones *visualizar()* y *eliminar()* se especifican abstractas (en UML se remarcá a través del empleo de letra cursiva) ya que todas las figuras geométricas tienen estas operaciones pero sólo se implementan en las subclases concretas.



Los atributos *x*, *y* y *visible*, son parte de todas las figuras geométricas, y por consiguiente, están localizadas en la superclase. El *radio* y los lados *a*, *b* y *c*, en contraste, son propiedades especiales de las figuras geométricas concretas. Los atributos concretos se equipan con las restricciones necesarias para la correspondiente figura. Por ejemplo, en un círculo, los radios no pueden ser igual o menor que cero, y en un triángulo la suma de dos lados cualesquiera debe ser mayor que el tercer lado.

- 16.4.** Declaración de la clases Director y Programador que derivan de Empleado.

Una vez que se ha creado una clase derivada, el siguiente paso es añadir los nuevos miembros que se requieren para cumplir las necesidades específicas de la nueva clase.

```

    clase derivada           clase base
    ↗                         ↘
clase Director hereda_de público Empleado
    declaraciones_de_miembros
fin_clase

```

En la definición de la clase `Director` sólo se especifican los miembros nuevos (funciones y datos). Todas las funciones miembro y los miembros dato de la clase `Empleado` son heredados automáticamente por la clase `Director`. Por ejemplo, la función `calcular_salario` de `Empleado` se aplica automáticamente a los directores:

```

Director: d

d.calcular_salario(325000)

clase Programador hereda_de público Empleado
    // miembros públicos y privados
fin_clase

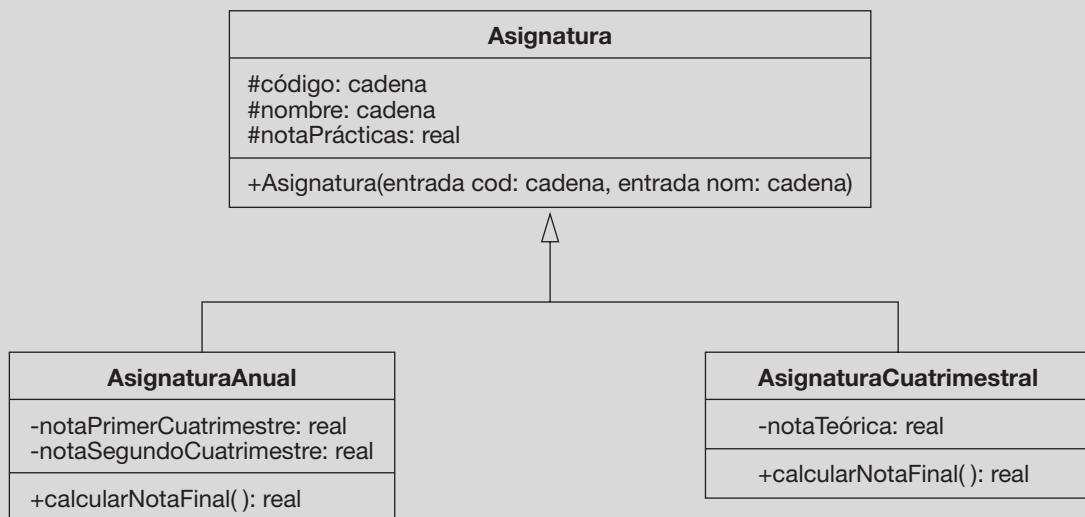
```

- 16.5.** Realice la declaración de una clase `Asignatura`. La clase contendrá atributos para almacenar el código y el nombre de la asignatura. Asimismo deberá tener un atributo de tipo real para almacenar la nota de prácticas. Su constructor deberá inicializar los atributos de código y nombre.

Declare además las clases derivadas `AsignaturaAnual` y `AsignaturaCuatrimestral`. Ambas clases hereden de la clase `Asignatura`. La clase `AsignaturaAnual` tendrá un atributo para la nota teórica del primer cuatrimestre y otra para la del segundo cuatrimestre.

La clase `AsignaturaCuatrimestral` tendrá un atributo para indicar el cuatrimestre (primer o segundo cuatrimestre) y otro para la nota teórica.

Las dos clases derivadas deberán tener un método `calcularNotaFinal`. En el caso de una asignatura cuatrimestral la nota final se calculará teniendo en cuenta que la nota de prácticas vale un 30 % de la nota final y la nota teórica un 60 % de la nota final. En el caso de una asignatura anual la nota teórica final será la media de las notas del primer y segundo cuatrimestre.



Codificación

```

clase Asignatura
var
    protegido cadena : código
    protegido cadena : nombre
    protegido real : notaPrácticas

constructor Asignatura(E cadena: cod, nom)
inicio
    código ← cod
    nombre ← nom
fin_constructor

fin_clase

clase AsignaturaAnual hereda_de Asignatura
var
    protegido real : notaPrimerCuatrimestre
    protegido real : notaSegundoCuatrimestre

real función calcularNotaFinal()
var
    real : notaTeórica
inicio
    notaTeórica ← (notaPrimerCuatrimestre +
                    notaSegundoCuatrimestre) / 2
    devolver(notaTeórica * 0.60 + notaPrácticas * 0.40)
fin_función

fin_clase

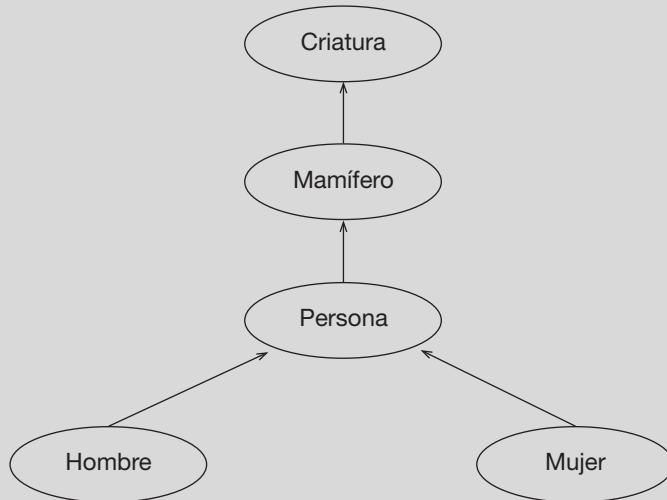
clase AsignaturaCuatrimestral hereda_de Asignatura
var
    protegido real : notaTeórica

real función calcularNotaFinal()
inicio
    devolver(notaTeórica * 0.60 + notaPrácticas * 0.40)
fin_función

fin_clase

```

16.6. Defina la siguiente jerarquía de clases:



Análisis del problema

Como se observa en el gráfico la clase base de toda la jerarquía es Criatura. La clase mamífero se define como *subclase* de animal; la clase persona, como una subclase de mamífero, y hombre y mujer son subclases de persona. Las clases derivadas heredan características de su clase base, pero añaden otras características propias nuevas.

Codificación

```

clase criatura
  var
  // atributos
  cadena      : tipo
  real        : peso
  tipoHabitat : habitat

  // operaciones
  constructor criatura()
  inicio
  ...
  fin_constructor

  procedimiento predadores(E criatura: predador)
  inicio
  ...
  fin_procedimiento

  entero función esperanza_vida()
  inicio
  ...
  fin_función
  ...
  
```

```
fin_clase //fin criatura

clase mamifero hereda_de criatura
    var
    // atributos o propiedades
    real: período_gestación

    // operaciones
    ...

fin_clase //fin mamífero

clase persona hereda_de mamífero
    var
    // propiedades
    cadena : apellidos, nombre
    fecha   : fecha_nacimiento
    // fecha es un tipo de dato compuesto que suele venir
    // predefinido en los lenguajes orientados a objetos modernos
    país     : origen
    ...

    //operaciones
    ...

fin_clase //fin persona

clase hombre hereda_de persona
    var
    //atributos
    mujer: esposa
    ...

    //operaciones
    ...

fin_clase //fin hombre.

clase mujer hereda_de persona
    var
    //propiedades
    hombre: esposo
    cadena: nombre
    ...

    //operaciones
    ...

fin_clase // fin mujer
```

- 16.7. Defina la clase *Círculo* y sus dos clases derivadas *Esfera* y *Cilindro*. Todas estas clases podrán calcular su área a través de un método denominado *Area* y presentarla mediante el método *Mostrar*.

Análisis del problema

Puesto que se suponen métodos virtuales el método que efectúa el cálculo del área se definirá en cada una de las clases, mientras que el procedimiento *Mostrar*, bastará definirlo en la clase base. Cuando se llame a *Mostrar()* el programa determinará el tipo del objeto en tiempo de ejecución y utilizará en cada caso el método adecuado para el cálculo del área.

Codificación

```
clase Círculo
    const
        público real: PI = 3.141592
    var
        protegido real: v1

    constructor Círculo(real: radio)
    inicio
        v1 ← radio
    fin_constructor

    real función Area()
    // virtual
    inicio
        devolver (PI*v1*v1)
    fin_función

    procedimiento Mostrar()
    inicio
        escribir (Area())
    fin_procedimiento

fin_clase

clase Esfera hereda_de Círculo

constructor Esfera(real radio)
inicio
    base(radio)
{ llamada al constructor de la clase base, puesto que no es el
    constructor por defecto y requiere paso de parámetros}
fin_constructor

real función Área()
inicio
    devolver (4 * PI * v1 * v1)
fin_función

fin_clase
```

```

clase Cilindro hereda_de Círculo
var
    protegido real: v2

constructor Cilindro(double radio, double altura)
inicio
    base(radio)
    v2 ← altura
fin_constructor

real función Área()
inicio
    devolver (2* (base.Área()) + 2*PI*v1*v2)
fin_función

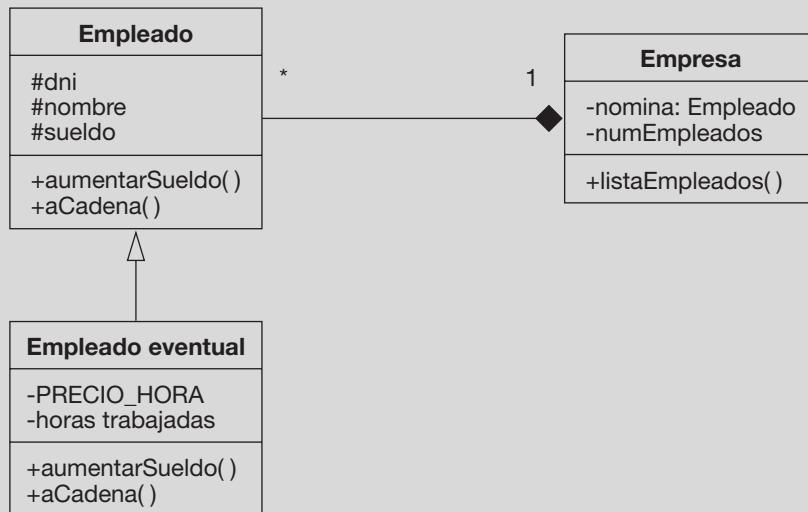
fin_clase

```

- 16.8.** Tenemos una Empresa compuesta por un número variable de empleados. Cada empleado contiene información sobre su DNI, su nombre y su sueldo. Además por cada empleado habrá que poder aumentar el sueldo de dos formas distintas, ya sea aumentándolo en un porcentaje, ya sea aumentándolo en un número determinado de euros.

Algunos empleados son empleados eventuales. Estos empleados tendrán además información sobre las horas trabajadas y sobre el precio que se le paga por hora. La forma de aumentar el sueldo a estos empleados será aumentando el número de horas trabajadas.

Será necesario implementar un mecanismo que permita listar los empleados de la empresa.
Realice un diagrama UML de las clases necesarias y su implementación.



Codificación

```

clase Empleado
var
    protegido cadena : dni
    protegido cadena : nombre
    protegido real : sueldo

```

```
constructor Empleado(cadena : d,n; real : s)
{ en los constructores los parámetros siempre son de entrada
  por lo que no hace falta especificarlo}
inicio
  dni ← d
  nombre ← n
  sueldo ← s
fin_constructor

cadena función aCadena()
inicio
  devolver(dni & " " & nombre & " " & sueldo)
fin_función

procedimiento aumentarSueldo(E real : porCiento)
inicio
  sueldo ← (sueldo * porCiento/100) + sueldo
fin_procedimiento

procedimiento aumentarSueldo(E entero : euros)
inicio
  sueldo ← sueldo + euros
fin_procedimiento

fin_clase

clase EmpleadoEventual hereda_de Empleado
const
  real : PRECIO_HORA = 30
var
  entero: horasTrabajadas

constructor EmpleadoEventual(real : id; cadena : n; entero : h)
inicio
  base(id,n)
  horasTrabajadas ← h
  sueldo ← h * PRECIO_HORA
fin_constructor

procedimiento aumentarSueldo(E entero : horas)
inicio
  sueldo ← sueldo + (horas * PRECIO_HORA)
fin_procedimiento

cadena función aCadena()
inicio
  devolver(super.aCadena() & " " & horasTrabajadas
fin_función
fin_clase
```

```

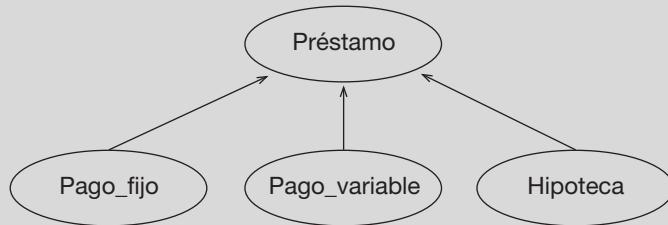
clase Empresa
const
    entero : numEmpleados = 20
var
    array[1..numEmpleados] de Empleado : nomina

constructor Empresa()
var
    entero : i
inicio
    desde i ← 1 hasta numEmpleados hacer
        nomina[i] ← nulo
    fin_desde
fin_constructor

procedimiento listarEmpleados()
var
    entero : i
inicio
    desde i ← 1 hasta numEmpleados hacer
        si nomina[i] <> nulo entonces
            escribir(nomina[i].aCadena())
        fin_si
    fin_desde
fin_procedimiento
fin_clase

```

16.9. Diseñe una clase *Prestamo* y tres clases derivadas de ella : *Pago_fijo*, *Pago_variable* e *Hipoteca*.



Codificación

```

clase abstracta Préstamo
var
    protegida real: capital
    protegida real: tasa_interés

constructor Préstamo(real: p1, p2)
    ...
inicio
    ...
fin_constructor

público abstracto procedimiento crearTablaPagos( E TablaPagos: t)
fin_clase

```

Las variables capital tasa_interes no se repiten en la clase derivada

```
clase Pago_fijo hereda_de público Prestamo
var
    privado real: pago      // cantidad mensual a pagar por cliente

constructor Pago_Fijo (real: p1, p2, p3)
...
inicio
...
fin_constructor

público procedimiento CrearTablaPagos(É TablaPagos: t);
inicio
...
fin_procedimiento
fin_clase

clase Hipoteca hereda_de público Prestamo
var
    privado entero: num_recibos
    privado entero: recibos_por_anyo
    privado real: pago

constructor Hipoteca(entero: p1, p2; real: p3, p4, p5)
...
inicio
...
fin_constructor

público procedimiento CrearTablaPagos(É TablaPagos: t);
inicio
...
fin_procedimiento

fin_clase
```


APÉNDICE A

ESPECIFICACIONES DE LENGUAJE ALGORÍTMICO UPSAM. VERSIÓN 2.0

A.1. ELEMENTOS DEL LENGUAJE

A.1.1. Identificadores

Se pueden formar con cualquier carácter alfabético regional (no necesariamente ASCII estándar), dígitos (0-9) y el símbolo de subrayado (_), debiendo empezar siempre por un carácter alfabético. Los nombres de los identificadores son sensibles a mayúsculas y se recomienda que su longitud no sobrepase los 50 caracteres.

A.1.2. Comentarios

Existen dos tipos de comentarios. Para comentarios de una sola línea, se utilizará la doble barra inclinada (//). Este símbolo servirá para ignorar todo lo que aparezca hasta el final de la línea. Los comentarios también podrán ocupar más de una línea utilizando los caracteres { y }, que indicarán respectivamente el inicio y el final del comentario. Todos los caracteres incluidos entre estos dos símbolos serán ignorados.

A.1.3. Tipos de datos estándar

Datos numéricos

- **Enteros.** Se considera entero cualquier valor numérico sin parte decimal, independientemente de su rango. Para la declaración de un tipo de dato entero se utiliza la palabra reservada `entero`.
- **Reales.** Se considera real cualquier valor numérico con parte decimal, independiente de su rango o precisión. Para la declaración de un tipo de dato real se utiliza la palabra reservada `real`.

Datos lógicos

Se utiliza la palabra reservada `lógico` en su declaración.

Datos de tipo carácter

Se utiliza la palabra reservada **carácter** en su declaración.

Datos de tipo cadena

Se utiliza la palabra reservada **cadena** en su declaración. A no ser que se indique lo contrario se consideran cadenas de longitud variable. Las cadenas de caracteres se consideran como un tipo de dato estándar pero estructurado (se podrán considerar como un array de caracteres).

A.1.4. Constantes de tipos de datos estándar

Numéricas enteras

Están compuestas por los dígitos (0..9) y los signos + y - utilizados como prefijos.

Numéricas reales

Los números reales en coma fija, utilizan el punto como separador decimal, además de los dígitos (0..9), y el carácter de signo (+ y -).

Para los reales en coma flotante, la mantisa podrá utilizar los dígitos (0..9), el carácter de signo (+ y -) y el punto decimal (.). El exponente se separará de la mantisa mediante la letra **E** y la mantisa estará formada por el carácter de signo y los dígitos.

Lógicas

Sólo podrán contener los valores **verdad** y **falso**.

De carácter

Cualquier carácter válido del juego de caracteres utilizados, delimitados por los separadores ' o ".

De cadena

Secuencia de caracteres válidos del juego de caracteres utilizados, delimitados por los separadores ' o ".

A.1.5. Operadores

Operadores aritméticos

Operador	Significado
-	Menos unario.
-	Resta.
+	Más unario.
*	Multiplicación.
/	División real.
div	División entera.
mod	Resto de la división entera.
**	Exponenciación.

El tipo de dato de una expresión aritmética depende del tipo de dato de los operandos y del operador. Con los operadores +, -, * y **, el resultado es entero si los operandos son enteros. Si alguno de los operandos es real, el resultado será de tipo real. La división real (/) devuelve siempre un resultado real. Los operadores **mod** y **div** devuelven siempre un resultado de tipo entero.

A.2. ESTRUCTURA DE UN PROGRAMA

```
algoritmo <nombre_del_algoritmo>
//Secciones de declaraciones
[const
    //declaraciones de constantes]
[tipos
    //declaraciones de tipos]
[var
    //declaraciones de variables]
//Cuerpo del programa
inicio
...
fin
```

A.2.1. Declaración de tipos de datos estructurados

Arrays

```
array[<dimensión>...] de <tipo_de_dato> : <nombre_del_tipo>
```

<dimensión> es un subrango con el índice del límite inferior y el límite superior. Por ejemplo, **array**[5..20] **de** **entero** declararía un array de 16 elementos enteros. Pueden aparecer varios separados por comas para declarar arrays de más de una dimensión.

<tipo_de_dato> es el identificador de cualquier tipo de dato estándar o definido por el usuario.

<nombre_del_tipo> es un identificador válido que se utilizará para referenciar el tipo de dato.

El acceso a un elemento de un array se realizará indicando su índice entre corchetes. El índice será una expresión entera.

Registros

```
registro : <nombre_del_tipo>
    <tipo_de_dato> : <nombre_del_campo>
    ...
fin_registro
```

<tipo_de_dato> identificador de cualquier tipo de dato estándar o definido por el usuario.
<nombre_del_tipo> identificador válido que se utilizará para referenciar el tipo de dato.
<nombre_del_campo> identificador válido que se utilizará para referenciar el campo del registro.
El acceso a un campo de una variable de tipo registro se realizará utilizando el carácter punto (.), por ejemplo, MiRegistro.MiCampo.

Archivos secuenciales

```
archivo_s de <tipo_de_dato> : <nombre_del_tipo>
<tipo_de_dato> identificador de cualquier tipo de dato estándar o definido por el usuario.  

<nombre_del_tipo> identificador válido que se utilizará para referenciar el tipo de dato.
```

Archivos directos

```
archivo_d de <tipo_de_dato> : <nombre_del_tipo>
<tipo_de_dato> es el identificador de cualquier tipo de dato estándar o definido por el usuario.  

<nombre_del_tipo> es un identificador válido que se utilizará para referenciar el tipo de dato.
```

A.2.2. Declaración de constantes

Se realiza dentro de la sección de declaraciones de constantes.

```
<nombre_de_constante> = <expresión>
<nombre_de_constante> identificador válido que se utilizará para referenciar la constante.  

<expresión> Expresión válida. El tipo de la constante será el tipo de dato que devuelva la expresión.
```

A.2.3. Declaración de variables

Se realiza dentro de la sección de declaraciones de variables.

```
<tipo_de_dato> : <nombre_de_variable> [= <expresión>] ...
<tipo_de_dato> es el identificador de cualquier tipo de dato estándar o definido por el usuario.  

<nombre_de_variable> es un identificador válido que se utilizará para referenciar la variable.  

En una declaración es posible declarar varias variables separadas por comas. Es posible inicializar la variable en la declaración, <expresión> es una expresión válida del tipo de dato de la variable.
```

A.2.4. Biblioteca de funciones

Funciones aritméticas

Función	Significado
<code>abs(x)</code>	Devuelve el valor absoluto de la expresión numérica <code>x</code> .
<code>aleatorio()</code>	Devuelve un número aleatorio real mayor o igual que 0 y menor que 1.
<code>arctan(x)</code>	Devuelve la arco tangente de <code>x</code> .
<code>cos(x)</code>	Devuelve el coseno de <code>x</code> .
<code>entero(x)</code>	Devuelve el primer valor entero menor que la expresión numérica <code>x</code> .
<code>exp(x)</code>	Devuelve el valor e^x .
<code>ln(x)</code>	Devuelve el logaritmo neperiano de <code>x</code> .
<code>log10(x)</code>	Devuelve el logaritmo en base 10 de <code>x</code> .
<code>raíz2(x)</code>	Devuelve la raíz cuadrada de <code>x</code> .
<code>sen(x)</code>	Devuelve el seno de <code>x</code> .
<code>trunc(x)</code>	Trunca (elimina los decimales) de la expresión numérica <code>x</code> .

Funciones de cadena

Función	Significado
<code>longitud(c)</code>	Devuelve el número de caracteres de la cadena <code>c</code> .
<code>posición(c, sc)</code>	Devuelve la posición de la primera aparición de la subcadena <code>sc</code> en la cadena.
<code>subcadena(c, ini [, long])</code>	Devuelve una subcadena de la cadena <code>c</code> formada por todos los caracteres a partir de la posición <code>ini</code> . Si se incluye el argumento <code>long</code> , devuelve sólo los primeros <code>long</code> caracteres a partir de la posición <code>ini</code> .

Funciones de conversión de número a cadena

Función	Significado
<code>código(car)</code>	Devuelve el código ASCII del carácter <code>car</code> .
<code>aCarácter(x)</code>	Devuelve el carácter correspondiente al código ASCII <code>x</code> .
<code>valor(c)</code>	Convierte la cadena <code>c</code> a un valor numérico. Si el contenido de la cadena <code>c</code> no puede convertirse a un valor numérico (contiene caracteres alfabéticos, signos de puntuación inválidos, etc.), devuelve 0.
<code>aCadena(x)</code>	Convierte a cadena el valor numérico <code>x</code> .

Funciones de información

Función	Significado
<code>tamaño_de(<variable>)</code>	Devuelve el tamaño en bytes de la variable.

A.2.5. Procedimientos de entrada/salida

`leer(<lista_de_variables>)` lee una o más variables desde la consola del sistema.

`escribir(<lista_de_expresiones>)` escribe una o más expresiones en la consola del sistema.

A.2.6. Instrucción de asignación

```
<variable> ← <expresión>
```

Primero evalúa el valor de la expresión y lo asigna a la variable. La variable y la expresión deben ser del mismo tipo de dato.

A.3. ESTRUCTURAS DE CONTROL

A.3.1. Estructuras selectivas

Estructura selectiva simple y doble

```
si <expresión_lógica> entonces
    <acciones>
[si_no
    <acciones>]
fin_si
```

Estructura selectiva múltiple

```
según_sea <expresión> hacer
    <lista_de_valores> : <acciones>
    ...
[si_no
    <acciones>]
fin_según
```

<expresión> puede ser cualquier expresión válida.

<lista_de_valores> será uno o más valores separados por comas del mismo tipo que *<expresión>*. La estructura verifica si el valor de la expresión coincide con alguno de los valores de la primera lista de valores; si esto ocurre, realiza las acciones correspondientes y el flujo de control sale de la estructura, en caso contrario, evalúa la siguiente lista. Las acciones de la cláusula **si_no** se ejecutarán si ningún valor coincide con la *<expresión>*.

A.3.2. Estructuras repetitivas

Estructura mientras

```
mientras <expresión_lógica> hacer
    <acciones>
fin_mientras
```

Estructura repetir

```
repetir
    <acciones>
hasta_que <expresión_lógica>
```

Estructura `desde`

```
desde <variable> ← <valor_inicial> hasta <valor_final>
      [incremento | decremento <valor_incremento>] hacer
      <acciones>
fin_desde
```

<variable> puede ser cualquier variable en la que se pueda incrementar o decrementar su valor, es decir, todas las numéricas, las de tipo carácter y las lógicas.

<valor_inicial> es una expresión con el primer valor que toma la variable del bucle. Debe ser del mismo tipo que la variable del bucle.

<valor_final> es una expresión con el último valor que toma la variable del bucle. Debe ser del mismo tipo que la variable del bucle. El bucle finaliza cuando la variable toma un valor mayor que este valor inicial.

<valor_incremento> es una expresión con el valor en el que se incrementará o decrementará la variable del bucle al final de cada iteración.

A.4. PROGRAMACIÓN MODULAR

A.4.1. Cuestiones generales

El ámbito de las variables declaradas dentro de un módulo (procedimiento o función) es local, y el tiempo de vida de dicha variable será el tiempo de ejecución del módulo.

A.4.2. Procedimientos

Declaración

```
procedimiento <nombre_procedimiento>([<lista_parámetros_formales>])
  [declaraciones locales]
  inicio
  ...
fin_procedimiento
```

<nombre_procedimiento> debe ser un identificador válido.

<lista_parámetros_formales> son uno o más grupos de parámetros separados por punto y coma. Cada grupo de argumentos se define de la siguiente forma:

```
{E | E/S} <tipo_de_dato> : <lista_de_parámetros>
```

E indica que el paso de parámetros se realiza por valor.

E/S indica que el paso de parámetros se realiza por referencia.

<tipo_de_dato> es un tipo de dato estándar o definido previamente por el usuario.

<lista_de_parámetros> es uno o más identificadores válidos separados por comas.

Llamada a procedimientos

```
[llamar_a] <nombre_procedimiento>([<lista_parámetros_actuales>])
```

La lista de parámetros actuales es una o varias variables o expresiones separadas por comas que deben coincidir en número, orden y tipo con la lista de parámetros formales de la declaración.

A.4.3. Funciones

Declaración

```
<tipo_de_dato> función <nombre_función>([<lista_parámetros_formales>])
[declaraciones locales]
inicio
...
devolver(<expresión>)
fin_función
```

<tipo_de_dato> es un tipo de dato estándar o definido previamente por el usuario. Se trata del tipo del dato que devuelve la función.

<nombre_función> debe ser un identificador válido.

<lista_parámetros_formales> son uno o más grupos de parámetros separados por punto y coma. Cada grupo de argumentos se define de la siguiente forma:

```
{E | E/S} <tipo_de_dato> : <lista_de_parámetros>
```

E indica que el paso de parámetros se realiza por valor.

E/S indica que el paso de parámetros se realiza por referencia.

<tipo_de_dato> es un tipo de dato estándar o definido previamente por el usuario.

<lista_de_parámetros> es uno o más identificadores válidos separados por comas.

<expresión> es el valor de retorno de la función. Debe coincidir con el tipo de dato de la declaración.

Llamada a funciones

```
<nombre_función>([<lista_parámetros_actuales>])
```

La lista de parámetros actuales es una o varias variables o expresiones separadas por comas que deben coincidir en número, orden y tipo con la lista de parámetros formales de la declaración. Al devolver un valor y no existir funciones que no devuelven valores (funciones `void` de C o Java), la llamada debe hacerse siempre dentro de una expresión.

A.5. ARCHIVOS

A.5.1. Archivos secuenciales

Apertura del archivo

```
abrir(<variable_tipo_archivo>,<modo_apertura>,<nombre_archivo>)
```

<var_tipo_archivo> es una variable de tipo archivo secuencial.

<modo_apertura> indica el tipo de operación que se realizará con el archivo. En el caso de archivos secuenciales será:

- '**l**', coloca el puntero al siguiente registro al comienzo del archivo y sólo realiza operaciones de lectura. El archivo debe existir previamente.
- '**e**', coloca el puntero al siguiente registro inmediatamente antes de la marca de final de archivo y sólo realiza operaciones de escritura.

<nombre_archivo> es una expresión de cadena con el nombre que el sistema dará al archivo.

Cierre del archivo

cerrar(<lista_variables_tipo_archivo>)

Cierra el archivo o archivos abiertos previamente.

Entrada/salida

leer(<variable_tipo_archivo>, <variable>)

Leer del archivo abierto para lectura representado por <variable_tipo_archivo> el siguiente registro. El tipo de la variable debe coincidir con el tipo base del archivo definido en la declaración del tipo de dato.

escribir(<variable_tipo_archivo>, <expresión>)

Escribe en el archivo abierto para escritura y representado por la variable de tipo archivo el valor de la expresión. El tipo de la expresión debe coincidir con el tipo base del archivo definido en la declaración del tipo de dato.

A.5.2. Archivos de texto

Se considera el archivo de texto como un tipo especial de archivo compuesto de caracteres o cadenas. La declaración de un tipo de dato de tipo archivo de texto sería por tanto:

```
archivo_s de carácter : <nombre_tipo>
archivo_s de cadena : <nombre_tipo>
```

La lectura de un carácter único en un archivo de texto se haría de la forma **leer(<variable_tipo_archivos>)** que leería el siguiente carácter del archivo. La lectura de una variable de tipo cadena (**leer(<variable_tipo_archivos>)**), leería todos los caracteres hasta el final de línea.

La escritura de datos en un archivo de texto también se podrá hacer carácter a carácter (**leer(<variable_tipo_carácter>)**) o línea a línea (**leer(<variable_tipo_cadena>)**).

La detección del final de línea en un archivo de texto cuando se lee carácter a carácter se realizaría con la función **fd1**:

fd1(<variable_tipo_archivos>)

La función **fd1** devuelve el valor lógico **verdad**, si el último carácter leído es el carácter de fin de línea.

A.5.3. Archivos directos

Apertura del archivo

abrir(<variable_tipo_archivo>, <modo_apertura>, <nombre_archivo>)

<var_tipo_archivo> es una variable de tipo archivo directo.

<modo_apertura> indica el tipo de operación que se realizará con el archivo. En el caso de archivos directos será:

- 'l'
- 'e'
- 'l/e', coloca el puntero al comienzo del archivo y permite operaciones tanto de lectura como de escritura.

<nombre_archivo> expresión de cadena con el nombre que el sistema dará al archivo.

Cierre del archivo

cerrar(<lista_variables_tipo_archivo>)

Cierra el archivo o archivos abiertos previamente.

Acceso secuencial

leer(<variable_tipo_archivo>, <variable>)

Leer del archivo abierto para lectura representado por <variable_tipo_archivo> el siguiente registro. El tipo de la variable debe coincidir con el tipo base del archivo definido en la declaración del tipo de dato.

escribir(<variable_tipo_archivo>, <expresión>)

Escribe en el archivo abierto para escritura y representado por la variable de tipo archivo el valor de la expresión. El tipo de la expresión debe coincidir con el tipo base del archivo definido en la declaración del tipo de dato.

Acceso directo

leer(<variable_tipo_archivo>, <posición>, <variable>)

Lee el registro situado en la posición relativa <posición> y guarda su contenido en la variable.

escribir(<variable_tipo_archivo>, <posición>, <variable>)

Escribe el contenido de la variable en la posición relativa <posición>.

A.5.4. Consideraciones adicionales

Detección del final del archivo

La operación de creación coloca en un archivo la marca de fin de archivo. Esta marca se desplaza con la escritura al añadir nuevos registros al archivo. La función **fda** permite detectar si se ha llegado a dicha marca.

fda (<variable_tipo_archivo>)

Devuelve el valor lógico **verdad**, si se ha intentado hacer una lectura secuencial después del último registro.

Determinar el tamaño del archivo

La función **lda** devuelve el número de bytes del archivo.

lida (<nombre_archivo>)

<nombre_archivo> es el nombre del archivo físico.

Para determinar el número de registros de un archivo se puede utilizar la expresión:

lida (<nombre_archivo>) / **tamaño_de** (<tipo_base_archivo>)

o bien directamente la función **numreg** (<identificador_archivo>).

Otros procedimientos

borrar (<nombre_archivo>)

Elimina del disco el archivo representado por la expresión de cadena <nombre_archivo>. El archivo debe estar cerrado.

renombrar (<nombre_archivo>, <nuevo_nombre>)

Cambia el nombre al archivo <nombre_archivo> por el de <nuevo_nombre>. El archivo debe estar cerrado.

A.6. VARIABLES DINÁMICAS

Declaración de tipos de datos dinámicos

puntero_a <tipo_de_dato> : <nombre_del_tipo>

Declara el tipo de dato <nombre_del_tipo> como un puntero a variables de tipo <tipo_de_dato>.

El valor constante **nulo**, indica una referencia a un puntero nulo.

Referencia al contenido de una variable dinámica

<variable_dinamica>[↑]

Asignación y liberación de memoria con variables dinámicas

reservar(<variable_dinámica>)

Reserva espacio en memoria para una variable del tipo de dato del puntero y hace que la variable dinámica apunte a dicha zona.

liberar(<variable_dinamica>)

Libera el espacio de memoria apuntado por la variable dinámica. Dicha variable queda con un valor indeterminado.

A.7. PROGRAMACIÓN ORIENTADA A OBJETOS

A.7.1. Clases y objetos

Declaración de clases

```
clase <nombre_de_clase>
    //Declaración de atributos
    //Declaración de constructores y métodos
fin_clase
```

<nombre_de_clase> es un identificador válido.

Declaración de tipos de referencias

<nombre_de_clase> : <nombre_de_referencia>

<nombre_de_clase> es el nombre de una clase previamente declarada.

<nombre_de_referencia> es un identificador válido que se utilizará para referenciar a un objeto de dicha clase.

La declaración de una referencia a una clase se hará en la sección de declaraciones de variables o tipos de datos de un algoritmo, o dentro de la sección de variables de otra clase.

Instanciación de clases

nuevo <nombre_de_constructor>([<argumentos_constructor>])

La declaración **nuevo** reserva espacio para un nuevo objeto de la clase a la que pertenece el constructor y devuelve una referencia a un objeto de dicha clase. <nombre_de_constructor> tendrá el mismo nombre de la clase a la que pertenece. La llamada al constructor puede llevar argumentos para la inicialización de atributos (véase más adelante en el apartado de constructores).

La instancia se puede realizar en una sentencia de asignación, o en la propia declaración de la referencia, dentro de la sección de declaraciones del algoritmo.

```
MiObjeto ← nuevo MiClase(arg1,arg2,arg3) //Dentro del código ejecutable
MiClase : MiObjeto = nuevo MiClase(arg1,arg2,arg3) //En la sección var
```

Referencias a miembros de una clase

```
NombreReferencia.nombreDeMiembro //Para atributos
NombreReferencia.nombreDeMiembro([listaParamActuales]) //Para métodos
```

Constructores

```
constructor <nombre_de_clase>[<lista_parametros_formales>]
//Declaración de variables locales
inicio
    //Código del constructor
fin_constructor
```

Existe un constructor por omisión sin argumentos al que se le llama mediante <nombre_de_clase().>. Al igual que con los métodos se admite la sobrecarga dentro de constructores, distinguiéndose los distintos constructores por el orden, número y/o tipo de sus argumentos.

Puesto que la misión de un constructor es inicializar una instancia de una clase, la <lista_parametros_formales> sólo incluyen argumentos de entrada, por lo que se puede omitir la forma en la que se pasan los argumentos.

Destructores

No se considera la existencia de destructores. Las instancias se consideran destruidas cuando se ha perdido toda referencia a ellas (recolector de basura).

Visibilidad de las clases

Se consideran todas las clases como públicas, es decir, es posible acceder a los miembros de cualquier clase declarada en cualquier momento.

Referencia a la instancia desde dentro de la declaración de una clase

Es posible hacer referencia a una instancia desde dentro de una clase con la palabra reservada **instancia** que devuelve una a la instancia de la clase que ha realizado la llamada al método. De esta forma **instancia.UnAtributo**, haría referencia al valor del atributo **UnAtributo** dentro de la instancia actual.

A.7.2. Atributos

Declaración de atributos

La declaración de los atributos de una clase se realizará dentro de la sección de declaraciones **var** de dicha clase.

```
const [privado|público|protegido] <tipo_de_dato> : <nombre_atributo>
    = <valor>
var
    [privado|público|protegido] [estático]
        <tipo_de_dato> : <nombre_atributo> [ = <valor_inicial>]
    ...

```

<nombre_atributo> puede ser cualquier identificador válido.

<tipo_de_dato> puede ser cualquier tipo de dato estándar, definido por el usuario u otra clase declarada con anterioridad.

Es posible dar un valor inicial al atributo mediante una expresión de inicialización que deberá ser del mismo tipo de dato que el atributo.

Visibilidad de los atributos

Por omisión, se considera a los atributos privados, es decir, sólo son accesibles por los miembros de la clase. Para que pueda ser utilizado por los miembros de otras clases de utilizará el modificador **público**. El modificador **protegido** se utiliza para que sólo pueda ser utilizado por los miembros de su clase y por los de sus clases hijas.

Atributos de clase (estáticos)

Un atributo que tenga el modificador **estático** no pertenece a ninguna instancia de la clase, sino que será común a todas ellas. Para hacer referencia a un atributo de una clase se utilizará el nombre de la clase seguido del nombre del atributo (`MiClase.MiAtributoEstático`).

Atributos constantes

La palabra reservada **const** permite crear atributos constantes que no se modificarán durante el tiempo de vida de la instancia.

A.7.3. Métodos

Declaración de métodos

La declaración de métodos se realizará dentro de la clase después de la declaración de atributos sin indicar ninguna sección especial.

```
[estático] [abstracto] [público|privado|protegido] {<tipo_de_retorno>:  
    función|procedimiento}<nombre_del_método>  
    ([<lista_de_parámetros_formales>])  
    //declaración de variables  
    inicio  
        //Código  
        [devolver(<expresión>)]  
    fin_método
```

`<nombre_del_método>` es un identificador válido.

`<tipo_de_retorno>` es cualquier tipo de dato estándar, estructurado o una referencia a un objeto. La declaración **devolver** se utiliza para indicar el dato de retorno que devuelve el método que debe coincidir con el tipo de retorno que aparece en la declaración. Si el método no devuelve valores se utilizará la palabra reservada **procedimiento** y no aparecerá la palabra **devolver**. Si el método devuelve valores, su nombre deberá aparecer precedido por el tipo de retorno y la palabra reservada **función**.

La lista de parámetros formales se declararía igual que en los procedimientos y funciones. El paso de argumentos se realizará como en los procedimientos y funciones normales.

Las variables locales se declararán en la sección **var** entre la cabecera del método y su cuerpo.

Visibilidad de los métodos

Por omisión, se consideran los métodos como públicos, es decir, es posible acceder a ellos desde cualquier lugar del algoritmo. Para que pueda ser utilizado sólo por miembros de su clase se utilizará el modificador **privado**. El modificador **protegido** se utiliza para que sólo pueda ser utilizado por los miembros de su clase y por los de sus clases hijas.

Métodos estáticos

Un método que tenga el modificador **estático** o **estática** no pertenece a ninguna instancia de la clase, sino que será común a todas ellas. Para hacer referencia a un método de una clase se utilizará el nombre de la clase seguido del nombre del método (`MiClase.MiMétodoEstático()`).

Sobrecarga de métodos

Se permite la sobrecarga de métodos, es decir, la declaración de métodos con el mismo nombre pero con funcionalidades distintas. Para que en la llamada se pueda distinguir entre los métodos sobrecargados, número, orden o tipo de sus argumentos deben cambiar.

Ligadura de métodos

La ligadura de la llamada de un método con el método correspondiente se hace **siempre** de forma dinámica, es decir, en tiempo de ejecución, con lo que se permite la existencia de **polimorfismo**.

A.7.4. Herencia

clase <clase_derivada> **hereda_de** <superclase>

<clase_derivada> es un identificador válido.

<superclase> es una clase declarada anteriormente.

La clase derivada:

- Hereda todos los métodos y atributos de la superclase accesibles (atributos públicos y protegidos y métodos públicos y protegidos) presentes sólo en la superclase.
- Sobreescribe todos los métodos y atributos de la superclase accesibles (atributos públicos y protegidos y métodos públicos y protegidos) presentes en ambas clases
- Añade todos los métodos y atributos presentes sólo en la clase derivada.

Es posible acceder a atributos de la superclase o ejecutar sus métodos mediante la palabra reservada **super**.

- Referencia a un miembro de la superclase `super.nombreMiembro()`.
- Referencia al constructor de la superclase: `super()`.

Clases y métodos abstractos

Clases en las que algunos o todos los miembros no tienen implementación, por lo que no pueden instanciarse directamente. Servirán de clase base para clases derivadas.

abstracta **clase** <clase_base>

Aquellos métodos sin implementación se podrían declarar sin inicio ni fin de método.

abstracto {**tipoDato** : **función|procedimiento**} **Nombre** ([paraformales])

En estos casos, las clases hijas deberían implementar el método.

Herencia múltiple

No se considera la existencia de herencia múltiple.

A.8. PALABRAS RESERVADAS

símbolo, palabra	Traducción/Significado
-	Menos unario (negativo)
-	Resta
&	Concatenación
←	Operador de asignación
↑	Referencia a una variable apuntada
*	Multiplicación
.	Cualificador de acceso a registros o a miembros de una clase
.	Separador de decimales
/	División real
//	Comentario de una sola línea. Ignora todo lo que aparezca a continuación de la línea
[]	Índice de array
^	Exponenciación
{	Inicio de comentario multilínea. Ignora todo lo que aparezca hasta encontrar el carácter de final de comentario ()
}	Fin de comentario multilínea. Ignora todo lo que aparezca desde el carácter de inicio de comentario {{}}
'	Comilla simple, delimitador de datos de tipo carácter o cadena
"	Comilla doble, delimitador de datos de tipo carácter o cadena
+	Más unario (positivo)
+	Suma
+	Concatenación
<	Menor que
<=	Menor o igual que
<>	Distinto de
=	Igual a
>	Mayor que
>=	Mayor o igual que
abrir	Abre un archivo
abs(x)	Devuelve el valor absoluto de la expresión numérica x
abstracto	Declaración de métodos abstractos (sin implementación)
aleatorio()	Devuelve un número aleatorio real mayor o igual que 0 y menor que 1
algoritmo	program, inicio del pseudocódigo
añadir	Modo de apertura de un archivo
arctan(x)	Devuelve la arco tangente de x
archivo_d	Declaración de archivos directos
archivo_s	Declaración de archivos secuenciales
array	Declaración de arrays
borrar	Borra un archivo del disco
cadena	string
aCadena(x)	Convierte a cadena el valor numérico de x
carácter	char
aCarácter(x)	Devuelve el carácter correspondiente al código ASCII x
cerrar	Cierra un archivo
cerrar	Cierra uno o más archivos abiertos
clase	Inicio de la declaración de una clase
código(car)	Devuelve el código ASCII del carácter car
const	Inicio de la sección de declaraciones de constantes
const	Declaración de atributos constantes en la definición de clases
constructor	Inicio de la declaración de un constructor

(continúa)

(continuación)

símbolo, palabra	Traducción/Significado
cos (x)	Devuelve el coseno de x
decremento	Decremento en estructuras repetitivas desde
desde	Inicio de estructura repetitiva desde, for
devolver	Indica el valor de retorno de una función
div	División entera
e	Exponente
e	Paso de argumentos por valor
e/s	Paso de argumentos por referencia
entero	integer , int , long , byte , etc.
entero(x)	Devuelve el primer valor entero menor que la expresión numérica x
entonces	then
escribir	Escribe una o más expresiones en un dispositivo de salida (consola, archivo, etc.)
escritura	Modo de apertura de un archivo
estático	Declaración de atributos o métodos de clase o estáticos
existe	Permite determinar la existencia de un archivo
exp(x)	Devuelve el valor e^x
falso	Falso, false
fda	Fin de archivo
fdl	Fin de línea
fin	Fin de algoritmo
fin_clase	Final de la declaración de una clase
fin_constructor	Fin de la declaración de un constructor
fin_desde	Fin de estructura repetitiva desde
fin_función	Fin de la declaración de una función
fin_mientras	Fin de estructura repetitiva mientras
fin_procedimiento	Fin de un procedimiento
fin_registro	Fin de la declaración de registro
fin_según	Fin de estructura selectiva múltiple
fin_si	end if, fin de estructura selectiva simple
función	Inicio de la declaración de una función
hacer	do
hasta	to
hasta_que	Fin de estructura repetitiva repetir
hereda_de	Indica que una clase derivada hereda miembros de una superclase
incremento	Incremento en estructuras repetitivas desde
inicio	Inicio del código ejecutable de un algoritmo, módulo, constructor, etc.
instancia	Referencia a la instancia actual de la clase donde aparece
lدا	Devuelve la longitud en bytes de un archivo
leer	Lee una o más variables desde un dispositivo de entrada (consola, archivo, etc.)
liberar	Libera el espacio asignado a una variable dinámica
ln(x)	Devuelve el logaritmo neperiano de x
log10(x)	Devuelve el logaritmo en base 10 de x
longitud(c)	Devuelve el número de caracteres de la cadena c
llamar_a	Instrucción de llamada a un procedimiento
mientras	while , inicio de estructura repetitiva mientras
mod	Módulo de la división entera
no	Not
nuevo	Reserva espacio en memoria para un objeto de una clase y devuelve una referencia a dicho objeto
nulo	Constante de puntero nulo

(continúa)

(continuación)

símbolo, palabra	Traducción/Significado
o	Or
posición(c, sc)	Devuelve la posición de la primera aparición de la subcadena <i>sc</i> en la cadena <i>c</i>
privado	Modificador de acceso privado a un atributo o método
procedimiento	Inicio de la declaración de un procedimiento
protégido	Modificador de acceso a un atributo o método que permite el acceso a los miembros de su clase y de las clases hijas
público	Modificador de acceso público a un atributo o método
puntero_a	Declaración de tipos de datos de asignación dinámica
raíz2(x)	Devuelve la raíz cuadrada de <i>x</i>
real	float, double, single, real, etc.
registro	record, inicio de la declaración de registro
renombrar	Cambia el nombre de un archivo
repetir	repeat, inicio de estructura repetitiva repetir
reservar	Reserva espacio en memoria para una variable dinámica
según_sea	Inicio de estructura selectiva múltiple, case, select case, switch
sen(x)	Devuelve el seno de <i>x</i>
si	Inicio de estructura selectiva simple / doble, if
si_no	else
subcadena(c, ini[, long])	Devuelve una subcadena de la cadena <i>c</i> formada por todos los caracteres a partir de la posición <i>ini</i> . Si se incluye el argumento <i>long</i> , devuelve sólo los primeros <i>long</i> caracteres a partir de la posición <i>ini</i>
super	Permite el acceso a miembros de la superclase
tamaño_de(x)	Devuelve el tamaño en bytes de la variable <i>x</i>
tipos	Inicio de la sección de declaraciones de tipos de datos
trunc(x)	Trunca (elimina los decimales) de la expresión numérica <i>x</i>
valor(c)	Convierte la cadena <i>c</i> a un valor numérico. Si el contenido de la cadena <i>c</i> no puede convertirse a un valor numérico (contiene caracteres alfabéticos, signos de puntuación inválidos, etc.), devuelve 0
var	Inicio de la sección de declaraciones de variables, o de la declaración de atributos de una clase
verdad	Verdadero, true
y	and

APÉNDICE B

BIBLIOGRAFÍA Y REFERENCIAS WEB

TEORÍA Y PRÁCTICA DE LA PROGRAMACIÓN

Braunstein, Silvia L. y Gioia, Alicia B.: *Introducción a la programación y a las estructuras de datos*. Eudeba, Buenos Aires, 1986.

Castellani, X.: *Método general de análisis de una aplicación informática*. Masson, 1986.

Ceri, Stefano; Madrioli, Dino y Sbatell, Licia. *The Art & Craft of Computing*. Harlow, England: Addison-Wesley, 1998.

Clavel, Biondi: *Introducción a la programación*. Tomo 1, *Algorítmica y lenguajes*. Tomo 2, *Estructura de datos*. Masson, 1985.

Clavel, Jorgesen: *Introducción a la programación*. Tomo 3, *Ejercicios corregidos*. Masson, 1986.

Coleman, D.: *Organización de datos y programación estructurada*. Gustavo Gili, 1986.

Dahl, O. J.; Dijkstra, E. W. y Hoare C. A. R.: *Structured Programming*. Academic Press, 1972.

Dahl; Dijkstra y Hoare: *Structured Programming*, 1972.

Dijkstra, E. W.: «*Go to statement considered harmful*». Comm. Assoc. Comp. Mach., 11 (1986, pp. 147-148).

Dijkstra, E. W.: «*Notes on Structured Programming*». Structured Programming. Academic Press, New York, 1972.

Glenn Nrookshear, J.: *Computer Science an Overview*. 7.^a edición. Boston: Addison-Wesley, 2003.

Goldschlager, Les y Lister, Andrew: *Introducción moderna a la ciencia de la computación*. Prentice-Hall, 1986.

Hoare, C. A. R.: «*Notes on Data Structuring*». Structured Programming. Academic Press, New York, 1972.

Hughes, Juan K.; Michtom, Glen C. y Michtom, Jay I.: *A Structured approach to programming*. 2.^a edición. Prentice-Hall, 1977.

Joyanes Aguilar, Luis: *Metodología de la programación*. McGraw Hill, 1986.

Joyanes Aguilar, Luis: *Fundamentos de Programación*. 3.^a edición, Madrid: McGraw-Hill, 2003.

Knuth, D. E.: *Structured Programming with goto statements*. Comput. Serv. 6(1974), pp. 261-301.

Lasala, Cristobal, Lapena: *Técnicas de análisis y programación para IBM PC (y compatibles)*. Universidad de Zaragoza, 1985.

Levine, Guillermo. *Computación y programación moderna*. México DF: Addison-Wesley, 2001.

Morales Pascual, José Luis: *Programación de ordenadores*. Tomos 1 - 6. ECC, 1986.

- Naylor, Jeff: *Introduction to Programming*. Paradigm Publishing Ltd., 1987.
- Peña, R.: *Diseño de programas. Formalismo y abstracción*. Prentice-Hall, 1998.
- Perry, Greg: *Absolute Beginner's Guide to Programming*. 2.^a edición. Indianapolis (USA): Que, 2001.
- Rodriguez Asensio: *Metodología de análisis y programación*. R. Asensio, 1985.
- Sabatini, Domenico: *Introduzione alla programmazione strutturata*. Bufetti Editore, 1985.
- Tucker, B. Allen y otros, y Joyanes, Luis. *Computación I. Lógica, resolución de problemas, algoritmos y programas*. Madrid: McGraw-Hill, 1999.
- Virgo, Fernando: *Técnicas y elementos de programación*. Gustavo Gili, 1985.
- Wirth, Niklaus: *Introducción a la programación sistemática*. El Ateneo. Buenos Aires, 1984.
- Woodhouse, David; Johnstone, Greg y McDougal, Anne: *Computer Science*. Wiley & Sons, 1982.

ALGORITMOS Y ESTRUCTURA DE DATOS

- Aho, A. M.; Hopcroft, J. E. y Ullman, J. D.: *Estructuras de datos y algoritmos*. Addison-Wesley Iberoamericana, 1988.
- Brassard, G. y Bratley, P.: *Algorítmica. Concepción y análisis*. Masson, 1990.
- Boussard, J. C. y Mahl, R.: *Programmation Avancee: Algorithmique et Structures de Donnees*. Eyrolles, 1983.
- Cairó, O. y Guardati, S.: *Estructuras de datos*. McGraw-Hill, 1993.
- Courtin, J. y Kowarski, I.: *Initiation à l'algorithmique et aux structures de données*. Dunod, 1987.
- Cormen, T. H.; Leiserson, C. E.; Rivest, R. L. y Stein, C.: *Introduction to Algorithms, Second Edition*. MIT Press and McGraw-Hill, 2001.
- Dale, Lilly: *Pascal y estructura de datos*. McGraw-Hill, 1986.
- Dijkstra, E. W.: *A discipline of programming*. Prentice-Hall, 1976.
- Dijkstra, E. W.: *Goto Statement Considered Harmful*. Communications of the ACM. Vol. 11, núm. 3, marzo 1968, 147-148, 538, 541.
- Joyanes Aguilar, Luis y Zahonero Martínez, Ignacio: *Estructura de datos*. Madrid: McGraw-Hill, 2000.
- Joyanes Aguilar, Luis y Zahonero Martínez, Ignacio: *Algoritmos y Estructura de datos: Una visión desde C*. Madrid: McGraw-Hill, 2003.
- Harel, David: *Algorithmics*. Addison Wesley, 1987.
- Helman, Paul y Veroff, Robert: *Intermediate Problem Solving and Data Structures*. Benjamin Cummings, 1986.
- Hoare C. A. R. y Dahl, O.: *Structured Programming*. Academic Press, 1972.
- Horowitz, Ellis y Sahni, Sartaj: *Fundamentals of computer algorithms*. Computer Science, Press, Inc., 1978.
- Jaime Sisa, A.: *Estructuras de datos y algoritmos*. Bogotá: Prentice Hall, 2001.
- Knuth, E. E.: *The art of Computer Programming*. Vol. 1, *Fundamental Algorithms*, 1969. Vol. 2, *Sorting and Searching*, 1972. Addison Wesley.
- Knuth, D. E.: *Structured programming with go-to statements*. ACM Computing Surveys. Vol. 6, 4, 1974, pp. 261-301.
- Kruse, Robert L.: *Data Structures and Program Design*. 2.^a edición. PHI, 1987.
- Lewis, T. G. y Smith, M. Z.: *Estructuras de datos*. Paraninfo, 1985.
- Lipschutz, Seymour: *Estructura de datos*. McGraw-Hill, 1986.
- Lucas; Reyrrin y Scholl: *Algorítmica y representación de datos*. Vol. 1, *Secuencias, autómatas de estados finitos*. Masson, 1985.

- Melhorn, K.: *Data Structures and Algorithms*. Vol. 1, *Sorting and Searching*. Springer-Verlag, 1984. Vol. 2, *Graph Algorithms and WP-Completeness*. Springer-Verlag. Berlín.
- Scholl, P. C.: *Algorítmica y representación de árboles*. Vol. 2, *Recursividades y árboles*. Masson, 1986.
- Tremblay, Jean-Paul y Bunt, Richard B.: *Introducción a la ciencia de las computadoras*. McGraw-Hill, 1982.
- Weiss, M. A.: *Estructuras de datos y algoritmos*. Addison-Wesley, 1995.
- Wirth, Niklaus: *Algoritmo + estructuras de datos = Programas*. Ediciones Castillo.
- *Algoritmos y estructuras de datos*. Prentice-Hall, 1987.
 - *Data Structured and Algorithms*. Scientific American, 1984.

ARCHIVOS (FICHEROS)

- Grosshaus, Daniel: *File Systems*. Prentice-Hall, 1986.
- Joyanes Aguilar, Luis: *Introducción a la teoría de ficheros (archivos)*. UPS, 1987.
- Loomis, Mary E. S.: *Data Management and File Processing*. Prentice-Hall, 1983.
- Walter: *Introduction to Data Management and File Design*. Reston, 1986.

ALGORITMOS DE ORDENACIÓN Y BÚSQUEDA

- Guighur, R.: *Procedimientos de clasificación*. Masson, 1987.
- Knuth, D. E.: *The art of Computer Programming*. Vol. 2, *Sorting and Searching*. Addison Wesley, 1973.
- Roux, M.: *Algorithmies de classification*. Masson, 1985.
- Sedgewick, R.: *Quicksort*. These. Stanford University, 1975.
- Wirth, Niklaus: *Algoritmos + Estructuras de datos = Programas*. Ediciones del Castillo.

LENGUAJES DE PROGRAMACIÓN

- Ghezzi, Carlos y Jazayeri, Mehdi: *Conceptos de lenguajes de programación*. Díaz de Santos, 1986.
- Marcotty, Michael y Ledgar, Henry: *The world of Programming Languages*. Springer-Verlag, 1987.
- Pratt, Terence W.: *Lenguajes de programación*. Prentice-Hall, 1984.
- Smedema, C. H.; Medena, P. y Boasson, M.: *Les langages de programmation*. Masson, 1986.
- Tennent, R. D.: *Principles of Programming Languages*. Prentice-Hall, 1980.
- Terry, Patrick D.: *Programming language translation*. International Computer Science. Addison Wesley, 1986.
- Tucker: *Lenguajes de programación*. McGraw-Hill, 1986. En el año 2003 se ha publicado la 2.^a edición en español por la misma editorial.
- Young, S. J.: *Real Time Languages*. Ellis Horwood-Publishers, 1982.

TÉCNICAS DE PROGRAMACIÓN

- Brassard, G. y Bratley, P.: *Fundamentos de Algoritmia*. Madrid: Prentice Hall, 1997.
- Excelente libro para aprender técnicas algorítmicas básicas y avanzadas utilizando un lenguaje algorítmico (pseudocódigo).

Joyanes Aguilar, Luis.: *Fundamentos de programación*. 2.^a edición. Madrid: McGraw-Hill, 1996.

Libro de nivel iniciación y medio para el aprendizaje del concepto de algoritmos y estructuras de datos mediante el uso de un pseudolenguaje algorítmico UPSAM y con posibilidad de codificación posterior a Pascal, C, FORTRAN, Modula-2 ó C++.

Joyanes Aguilar, Luis.: *Fundamentos de programación*. 3.^a edición. Madrid: McGraw-Hill, 2003.

Libro de nivel iniciación y medio para el aprendizaje del concepto de algoritmos y estructuras de datos mediante el uso de un pseudolenguaje algorítmico UPSAM en su versión 2.0 y con posibilidad de codificación posterior a Pascal, C ó C++ o bien los nuevos lenguajes Java y C#. El libro complementario desde el punto de vista práctico es precisamente la obra que usted tiene en sus manos.

Joyanes, L., Rodriguez, L. y Fernandez, M.: *Fundamentos de programación. Libro de problemas*. 1.^a edición. Madrid: McGraw-Hill, 1997.

Primera edición de esta obra y complementario del anterior con una colección de la mayoría de ejercicios y problemas propuestos en el mismo, además de otra colección complementaria. En esta edición sólo se consideraba la programación estructurada así como las técnicas de algoritmos y estructuras de datos.

Orientación a objetos

Booch, Grady.: *Análisis y diseño orientado a objetos con aplicaciones*. Madrid: Addison-Wesley, 1995.

Libro clave de la metodología de Booch'93, fundamental en el desarrollo de objetos y con fundamentos teóricos de tecnologías de objetos indispensables para su comprensión.

Joyanes Aguilar, Luis.: *Programación Orientada a Objetos*. 2.^a edición. Madrid: McGraw-Hill, 1998.

Nueva edición de un libro sobre programación orientada a objetos con C++ que incluye en este caso, una extensa explicación sobre UML y STL (la biblioteca de plantillas estándar).

Joyanes Aguilar, Luis.: *Programación en Java*. 2.^a edición. Madrid: McGraw-Hill, 2003.

Libro de programación en el que se enseñan técnicas de programación estructurada y orientada a objetos con el lenguaje Java, versión 2.

Joyanes Aguilar, Luis.: *Programación en C++*. Madrid: McGraw-Hill, 2000.

Libro de programación en el que se enseñan técnicas de programación estructurada y orientada a objetos con el lenguaje C++.

Rumbaugh, J.; Blaha, M.; Premerlani, W. L.; Frederik, E. y Lorenzen, W.: *Modelado y diseño orientados a objetos (Metodología OMT)*. 2.^a reimpresión, Madrid: Prentice Hall, 1998.

Libro base de la metodología OMT, posiblemente, ha sido la más utilizada en la década pasada y sigue siendo todavía muy utilizada. Es uno de los soportes sobre los que se ha construido UML (Lenguaje de Modelado Unificado).

Revistas y direcciones de referencia en programación de Internet

PC Magazine

<http://www.pcmag.com>

PC World

<http://www.pcworld.com>

<http://www.pcworld.es>

PC Actual

<http://www.pc-actual.com>

<http://www.vnunet.es>

PC Pro

<http://www.pcpro-es.com>

C/C++ Users Journal
http://www.cuj.com

Dr. Dobb's Journal
www.ddj.com

Dr. Dobb's Journal (edición española)
www.mkm-pi.com

Byte
www.mkm-pi.com

MSDN Magazine
http://msdn.microsoft.com/msdnmag

Sys Admin
www.samag.com

Software Development Magazine
www.sdmagazine.com

UNIX Review
www.review.com

Visual Studio Magazine, Java Pro, .NET Magazine
http://www.ftponline.com/

Windows Developper's Journal
www.wdj.com

Component Strategies
www.componentmag.com

C++ Report
www.creport.com

Journal Object Orientd Programming (JOOP)
www.joopmag.com

Revista Microsoft Systems Journal
http://www.msj.com/msjquery.html

Buscador Altavista
http://www.altavista.digital.com

Buscador Excite
http://www.excite.com

Buscador de la Universidad de Oviedo
http://www.uniovi.es

Buscador Infoseek
http://guide-p.infoseek.com

Buscador Lycos

http://www.lycos.com

Software shareware

http://www.shareware.com

Buscador *Webcrawler*

http://www.webcrawler.com

Buscador Yahoo

http://www.yahoo.com

Buscado Google

http://www.google.com

Tal vez el buscador más popular utilizado hoy día.

Excelente página de orientación a objetos en español

http://www.ctv.es/USERS/pagullo/cpp.htm

Página oficial del fabricante Inprise/Borland

http://www.inprise.com

http://www.borland.com

ÍNDICE

A

- Abstracta, 392, 421
- Acumuladores, 41
- Agregaciones, 382
- Algoritmos, 3
 - estructura, 40, 81
 - representación, 18
- Análisis problema, 3
- Árbol, 307
 - binario de búsqueda, 312
 - binarios, 308
 - formas de implementación, 310
 - general, 309
 - nodos, 308
 - recorridos, 311
- Archivos, 414
 - conversión de claves, 186
 - de datos, 159
 - de texto, 162
 - fusión, 239
 - indexación, 189
 - jerarquización, 160
 - mantenimiento de archivos directos, 187
 - mantenimiento de archivos secuenciales, 163
 - operaciones, 160
 - ordenación externa, 240
 - ordenación por mezcla directa, 241
 - ordenación por mezcla natural, 241
 - organización directa, 185
 - organización secuencial independiente, 187
 - organización secuencial, 161
 - partición de archivos, 240

- registro físico, 159
 - registro lógico, 159
 - tratamiento de sinónimos, 187
- Aritmética modular, 225
- Arrays (arreglos), 106, 263
 - de registros, 111
 - paralelos, 111
- Asignación, 41
- Asociación, 379

B

- Backtracking, 335
- Bicola, 266
- Böhm, 55
- Bus del sistema, 2
- Búsqueda
 - binaria, 224, 340
 - externa, 239
 - interna, 223
 - por transformación de clave, 224
 - secuencial, 223

C

- Cadenas, 149
- Clases, 111, 358, 364, 367, 368, 371, 392, 418
 - acceso miembros, 370
 - atributos, 359, 364, 371, 419
 - clase base, 385,
 - clases derivadas, 386
 - composición, 383
 - constructores, 370, 419
 - declaración, 363
 - miembros, 369, 370, 418

- multiplicidad, 381
- operaciones, 365, 371
- relaciones entre clases, 379

- Colas, 266
- Colisión, 185, 187, 226
- Comentarios, 20, 407
- Composición, 383
- Conjuntos, 109
- Constantes, 16, 149, 410, 420
- Contadores, 41
- continuar, 61
- Controladores, 2

D

- Destructores, 370, 419
- Diagrama de flujo, 18
- Diagrama Nassi-Schneiderman, 19
- Diseño descendente, 55
- Dispositivos de entrada/salida, 2

E

- Encapsulamiento, 370
- Especialización, 384
- Estáticos, 421
- Estructura
 - algoritmo, 40
 - anidadas, 60
 - desde, 59
 - hacer mientras, 59
 - hasta, 59
 - mientras, 58
 - programa, 39
 - repetitiva, 58
 - secuencial, 56
 - selectiva, 56

Estructuras de datos, 105
Expresiones, 16

F

Factor de bloqueo, 159
FIFO, 266
Flujos, 161
Funciones, 17, 80, 162, 186, 357, 369, 411, 414
de cadena, 151
de carácter, 151
de conversión de clave, 186, 224
Fusión o intercalación, 230, 239

G

Generalización, 384
Grafo, 312
dirigido, 313
no dirigido, 313
valorado, 314

H

Herencia, 385, 388, 421
múltiple, 390, 421
simple, 389

I

Identificadores, 21, 407
Implementación, 5
Instancia, 361, 369, 418, 419
Intercalación, 230
Interrumpir, 61
Interrupciones, 2
Interruptores, 42
ir-a, 61

J

Jacopini, 55
Jerarquía de clases, 384
Jerarquización, 160

L

Lenguajes, 2
LIFO, 265
Ligadura dinámica, 395, 421
Listas, 262
circulares, 264
contiguas, 262
de adyacencia, 314
dblemente encadenadas circulares, 265

dblemente encadenadas, 264
enlazadas implementación con arrays, 263
enlazadas implementación con punteros, 263
enlazadas, 263
Literales, 22

M

Matrices, 106
Matriz de adyacencia, 313
Memoria, 1, 223, 261, 366, 370, 418, 419
Mensaje, 360
Métodos, 369, 420
estáticos, 421
sobrecarga, 421
Mezcla, 230, 239
Miembros de una clase, 369, 418
Módulos, 358
Multiplicidad, 381

N

Nassi-Schneiderman, 19
Nodos, 261, 308
nulo, 262

O

Objetos, 261, 310, 359, 371, 418
declaración, 369
identidad, 360
instanciación, 369
Operadores, 22
asignación, 41
concatenación, 150
Ordenación, 227
burbuja, 228
externa, 240
inserción binaria, 228
inserción directa, 228
interna, 227
por mezclas sucesivas, 337
por mezcla directa, 241
por mezcla natural, 241
rápida (*Quick-Sort*), 229, 341
selección, 227
shell, 229

P

Palabras reservadas, 21
Parámetros, 82, 109, 111, 413, 414
correspondencia por nombre explícito, 83

correspondencia posicional, 82
paso por referencia, 84
paso por valor resultado, 84
paso por valor, 83

Partición de archivos, 240

Paso de mensajes, 360

Pilas, 265

Plegamiento, 225

Polimorfismo, 394, 421

POO, 360, 371

Privado, 364, 370, 388, 392, 420

Procedimientos, 81, 357, 369, 413

Programa

propio, 55

estructura, 39

Programación modular, 79

Protegido, 364, 370, 388, 420

Pseudocódigo, 20

Público, 364, 370, 388, 392, 420

Puntero, 261, 417

Puntuadores, 22

Q

Quick-Sort, 229

R

Recursividad, 84, 266, 307, 333
Registros, 111, 159

S

Sentencias de salto, 61
Shell, 229
Sobrecarga, 395, 421
Software, 2
Subalgoritmos, 79

T

Tabla, 106
Tipos de datos, 15
estándar, 407
estructura, 111
estructurados, 105, 409
referencia, 418

Tipos de recursividad, 334

Tratamiento de sinónimos, 187, 226

Truncamiento, 225

U

UCP (*CPU*), 2
UML, 364, 371

V

Variables, 16, 410
de cadena, 149

de instancia, 361, 369
dinámica, 262, 417
locales y globales, 84

Verificación de algoritmos, 5
Virtual, 395
Visibilidad, 370, 419

Segunda Edición

Libro de problemas

Fundamentos de programación I

Algoritmos, Estructuras de datos y Objetos I

Luis JOYANES AGUILAR

Este libro es una obra práctica de aprendizaje para la *introducción a la programación de computadoras*, aunque también ha sido escrito pensando en ser libro complementario de la obra *Fundamentos de programación* (Joyanes, 3^a ed. McGraw-Hill, 2003). Para este fin, el contenido del libro coincide casi en su totalidad con los diferentes capítulos de la citada obra. De esta manera, se pueden estudiar conjuntamente ambos libros y así conseguir no sólo un aprendizaje más rápido sino también, y sobre todo, mejor formación teórico-práctica y mayor rigor académico y profesional en la misma. La parte teórica de este libro es suficiente para aprender los problemas y ejercicios de programación resueltos y proponer su propias soluciones, sobre la base de que muchos ejercicios propuestos en el libro de teoría ofrecen la solución en este libro de problemas.

Para conseguir los objetivos del libro utilizaremos fundamentalmente el lenguaje algorítmico, con formato de pseudocódigo, herramienta ya probada y experimentada, no sólo en la primera edición de esta obra, sino también en las tres ediciones de la obra complementaria *Fundamentos de programación*, y muy utilizada en numerosas universidades y centros de formación de todo el mundo.

El libro contiene los temas más importantes de la programación tradicional tales como estructuras de control, funciones, estructuras de datos, métodos de ordenación y búsqueda, junto con los conceptos fundamentales de orientación a objetos tales como clases, objetos, herencia, relaciones, programación orientada a objetos y recursividad.

Objetivos del libro

El libro pretende enseñar a programar utilizando conceptos fundamentales tales como:

- † *Algoritmos*: conjunto de instrucciones programadas para resolver una tarea específica.
- † *Datos*: una colección de datos que se proporcionan a los algoritmos que han de ejecutarse para encontrar una solución: los datos se organizarán en *estructuras de dato*).
- † *Objetos*: el conjunto de datos y algoritmos que los manipulan, encapsulados en un tipo de dato nuevo conocido como objeto.
- † *Clases*: tipos de objetos con igual estado y comportamiento, o dicho de otro modo, con los mismos atributos y operaciones.
- † *Estructuras de datos*: conjunto de organizaciones de datos para tratar y manipular eficazmente datos homogéneos y heterogéneos.
- † *Temas avanzados*: archivos, recursividad y ordenaciones/búsquedas avanzadas.

Características

- † *Introducción a la informática y a las ciencias de la computación usando algoritmos y el lenguaje algorítmico, basado en pseudocódigo.*

McGraw-Hill Interamericana
de España, S.A.U.

A Subsidiary of The McGrawHill Companies

