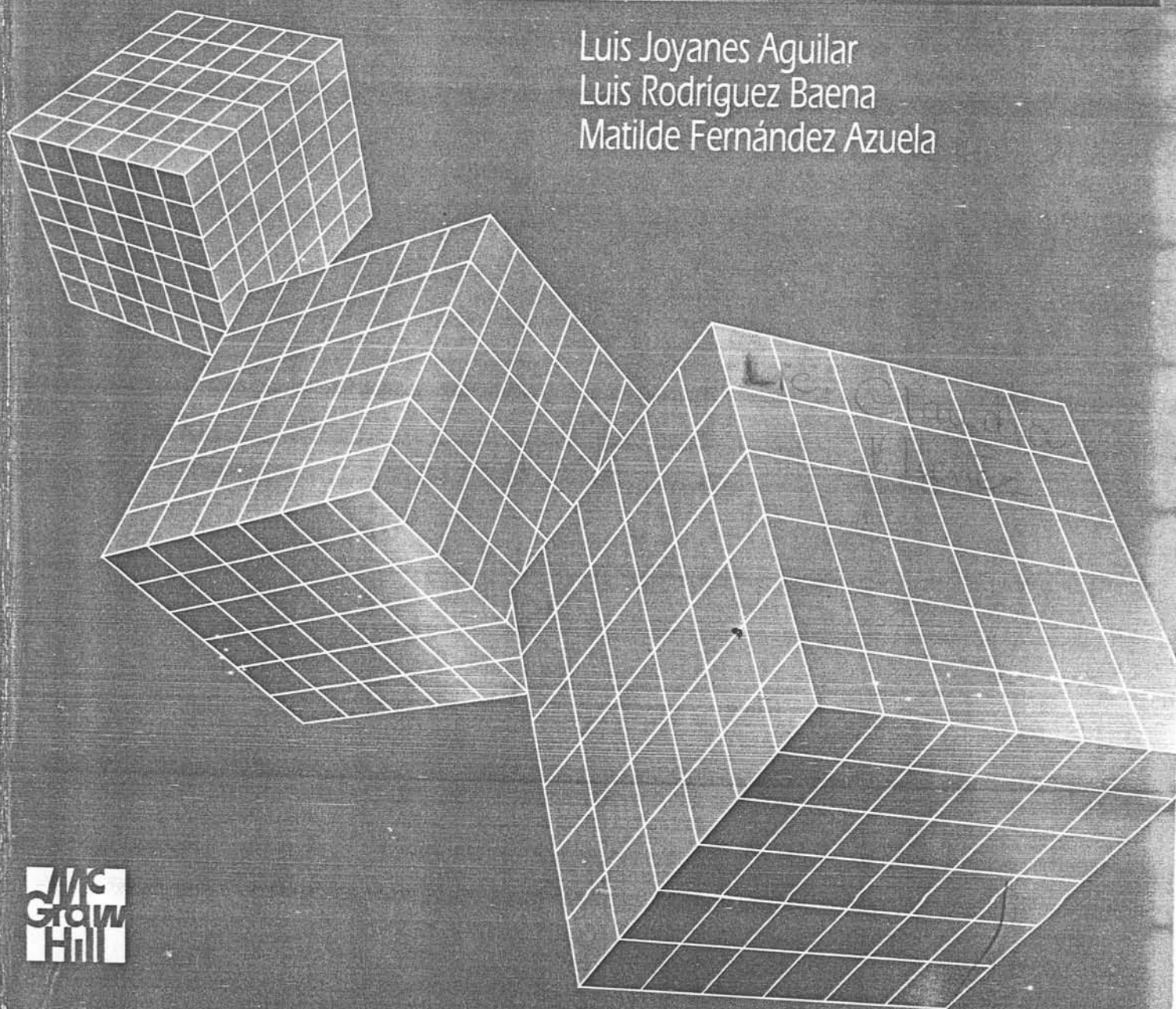


FUNDAMENTOS DE PROGRAMACIÓN

LIBRO DE PROBLEMAS

Luis Joyanes Aguilar
Luis Rodríguez Baena
Matilde Fernández Azuela



FUNDAMENTOS DE PROGRAMACIÓN

Libro de problemas

Lic. Claudia Núñez

CONSULTORES EDITORIALES
ÁREA DE INFORMÁTICA Y COMPUTACIÓN

Antonio Vaquero Sánchez
Catedrático de Lenguajes y Sistemas Informáticos
Escuela Superior de Informática
Universidad Complutense de Madrid
ESPAÑA

Gerardo Quiroz Vieyra
Ingeniero en Comunicaciones y Electrónica
por la ESIME del Instituto Politécnico Nacional
Profesor de la Universidad Autónoma Metropolitana
Unidad Xochimilco
MÉXICO

FUNDAMENTOS DE PROGRAMACIÓN

Libro de problemas

Luis Joyanes Aguilar
Luis Rodríguez Baena
Matilde Fernández Azuela

Departamento de Lenguajes y Sistemas Informáticos
Facultad y Escuela Universitaria de Informática
Universidad Pontificia de Salamanca en Madrid

LIBRO DE CORTESIA



MEXICO

McGraw-Hill

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

FUNDAMENTOS DE PROGRAMACION. Libro de problemas

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 © 1996, respecto a la primera edición en español, por McGRAW-HILL/INTERAMERICANA DE ESPAÑA, S. A. U.

Edificio Valrealty, 1.ª planta
Basauri, 17
28023 Aravaca (Madrid)

ISBN: 84-481-0700-4
Depósito legal: M. 11.667-2000

Editora: Concepción Fernández Madrid
Cubierta: Juan García
Compuesto por los autores
Impreso en EDIGRAFOS, S. A.

IMPRESO EN ESPAÑA - PRINTED IN SPAIN

Contenido

Prólogo	XI
1. Algoritmos y programas	1
1.1. Resolución de problemas por computadoras	1
1.1.1. Fase de resolución del problema	1
1.1.1.1. Análisis del problema	2
1.1.1.2. Diseño del algoritmo	3
1.1.1.3. Verificación de algoritmos	4
1.1.2. Fase de implementación	4
1.2. Datos	4
1.2.1. Constantes	5
1.2.2. Variables	5
1.2.3. Expresiones	5
1.2.4. Funciones	7
1.2.5. Reglas para la construcción de identificadores	7
1.3. Ejercicios resueltos	8
2. La resolución de problemas con computadoras y las herramientas de programación	23
2.1. Herramientas de programación	23
2.2. Diagrama de flujo	24
2.3. Diagrama Nassi-Schneiderman	25
2.4. Pseudocódigo	25
2.5. Ejercicios resueltos	27
3. Estructura general de un programa	39
3.1. Escritura de algoritmos	39

3.2. Contadores, acumuladores e interruptores	40
3.2.1. Contadores	40
3.2.2. Acumuladores	41
3.2.3. Interruptores	41
3.3. Estilo recomendado para la escritura de algoritmos	41
3.4. Ejercicios resueltos	42
 4. Introducción a la programación estructurada	57
4.1. Programación estructurada	57
4.2. Teorema de Böhm y Jacopini	57
4.3. Estructuras de control	58
4.3.1. Estructuras secuenciales	58
4.3.2. Estructuras selectivas	58
4.3.3. Estructuras repetitivas	60
4.3.4. Estructuras anidadas	61
4.4. Ejercicios resueltos	62
 5. Subprogramas (subalgoritmos), procedimientos y funciones	81
5.1. Programación modular	81
5.2. Funciones	82
5.2.1. Declaración de funciones	82
5.3. Procedimientos	83
5.3.1. Declaración de procedimientos	83
5.4. Estructura general de un algoritmo	84
5.5. Paso de parámetros	84
5.6. Variables locales y globales	86
5.7. Recursividad	86
5.7.1. Algoritmos recursivos	87
5.7.2. Algoritmos con retroceso	88
5.8. Ejercicios resueltos	89
 6. Estructuras de datos (arrays y registros)	109
6.1. Datos estructurados	109
6.2. Arrays	109
6.2.1. Arrays unidimensionales	110
6.2.2. Arrays bidimensionales	111
6.2.3. Recorrido de los elementos del array	112
6.2.4. Arrays como parámetros	113
6.3. Registros	113
6.3.1. Arrays de registros y arrays paralelos	114
6.4. Ejercicios resueltos	114

7. Las cadenas de caracteres	157
7.1. Cadenas.....	157
7.2. Ejercicios resueltos	158
 8. Archivos (ficheros). Archivos secuenciales	 167
8.1. Conceptos generales sobre archivos	167
8.1.1. Jerarquización	168
8.1.2. Clasificación de los archivos según su función	168
8.1.3. Operaciones básicas	168
8.1.4. Otras operaciones usuales	169
8.1.5. Soportes	169
8.2. Organización secuencial	169
8.2.1. Archivos de texto	170
8.2.2. Mantenimiento de archivos secuenciales	170
8.3. Ejercicios resueltos	171
 9. Archivos directos	 195
9.1. Organización directa	195
9.1.1. Funciones de conversión de clave	196
9.1.2. Tratamiento de sinónimos	197
9.1.3. Mantenimiento de archivos directos	197
9.2. Organización secuencial indexada	198
9.3. Modos de acceso	199
9.3.1. Ficheros indexados	200
9.4. Ejercicios resueltos	200
 10. Ordenación, búsqueda e intercalación	 235
10.1. Búsqueda	235
10.1.1. Búsqueda secuencial	235
10.1.2. Búsqueda binaria	236
10.1.3. Búsqueda por transformación de claves	236
10.1.3.1. Funciones de conversión de clave	236
10.1.3.2. Resolución de colisiones	238
10.2. Ordenación	240
10.2.1. Ordenación interna	240
10.2.1.1. Selección	240
10.2.1.2. Burbuja	241
10.2.1.3. Inserción directa	241
10.2.1.4. Inserción binaria	242
10.2.1.5. Shell	242
10.2.1.6. Ordenación rápida	242

10.3. Intercalación	244
10.4. Ejercicios resueltos	245
11. Búsqueda, ordenación y fusión externas (archivos)	253
11.1. Conceptos generales	253
11.2. Búsqueda externa	253
11.3. Fusión	254
11.4. Ordenación externa	254
11.4.1. Partición de archivos	254
11.4.1.1. Partición por contenidos	254
11.4.1.2. Partición en secuencias de longitud 1	255
11.4.1.3. Partición en secuencias de longitud N	255
11.4.1.4. Partición en secuencias de longitud N con clasificación interna de dichas secuencias	255
11.4.1.5. Partición según el método de selección por sustitución	255
11.4.1.6. Partición por el método de selección natural	255
11.4.2. Ordenación por mezcla natural	256
11.4.3. Ordenación por mezcla directa	256
11.5. Ejercicios resueltos	255
12. Estructuras dinámicas lineales de datos (listas enlazadas, pilas, colas)	277
12.1. Estructuras dinámicas	277
12.2. Listas	278
12.3. Pilas	282
12.3.1. Aplicaciones de las pilas	283
12.4. Colas	283
12.4.1. Doble cola	284
12.4.2. Aplicaciones de las colas	284
12.5. Ejercicios resueltos	284
13. Estructuras de datos no lineales (árboles y grafos)	331
13.1. Estructuras no lineales	331
13.2. Árboles	331
13.2.1. Terminología	332
13.2.2. Árboles binarios	333
13.2.2.1. Conversión de un árbol general en binario	333
13.2.2.2. Implementación	334
13.2.2.3. Recorridos de un árbol binario	336
13.2.2.4. Árbol binario de búsqueda	337
13.3. Grafos	337
13.3.1. Terminología	338
13.3.1. Representación de los grafos	338
13.4. Ejercicios resueltos	340

14. Tablas de decisión	359
14.1. Definición y estructura	359
14.2. Tipos de tablas de decisión	361
14.3. Tipos de reglas de decisión	362
14.4. Escritura de una tabla de decisión	363
14.5. Transformación de una tabla en diagrama de flujo	363
14.6. Ejercicios resueltos	364
 Apéndice A. Especificaciones del lenguaje algorítmico UPSAM	 371
A.1. Elementos del lenguaje	371
Identificadores	371
Comentarios	371
Tipos de datos estándar	372
Constantes estándar	372
Operadores	372
Definición de tipos	373
Definición de constantes	374
Definición de variables	374
Funciones aritméticas de biblioteca	374
Funciones de cadena de biblioteca	375
Funciones de conversión de número a cadena	375
Funciones de información	375
Estructura de un programa	375
A.2. Estructuras de control	375
Estructuras selectivas	375
Estructuras repetitivas	376
A.3. Programación modular	376
Inclusión de archivos o módulos	376
Procedimientos	376
Funciones	377
Cuestiones generales	377
Procedimientos de entrada/salida	377
A.4. Archivos	377
Archivos secuenciales (texto)	377
Archivos directos (relativos)	378
Archivos indexados	378
A.5. Variables dinámicas	379
A.6. Palabras reservadas, operadores, caracteres especiales, funciones y procedimientos estándar	379
 Apéndice B. Bibliografía	 383
 Índice	 389

Prólogo

Los *curricula* actuales de las carreras universitarias de Informática, Ciencias de la Computación o Sistemas de Computación y la mayoría de Ciencias e Ingeniería no específicas de Informática, contienen en España y en toda Latinoamérica una asignatura común a todas las carreras y con contenido similar, que trata de enseñar las técnicas de iniciación a la programación. El nombre de la asignatura citada es diferente según la carrera **Introducción, Fundamentos o Metodología de la programación**, pero el contenido siempre contempla un curso de introducción a la programación que enseña al estudiante/lector los conceptos básicos de algoritmos y de estructuras de datos, junto con el aprendizaje para el diseño y construcción de algoritmos, «**algoritmia**» («Ciencia del cálculo aritmético y algebraico; teoría de los números». Definición del DRAE —Diccionario de la Real Academia Española— en su 21.^a edición de 1992). Como consecuencia de los objetivos anteriores, la asignatura citada adquiere un carácter fundamental en los estudios citados que requiere una preparación minuciosa por parte del lector para conseguir unos conocimientos sólidos que le permitan afrontar en años sucesivos otras asignaturas claves en la carrera, tales como Algoritmos y Estructuras de datos, Bases de datos, Programación de Gestión y de Sistemas, Programación científica, etc.

Estas causas nos han llevado a diseñar esta obra de modo que nuestros lectores y estudiantes pudieran seguir el curso de introducción a la programación con garantía de éxito. Nuestra experiencia pedagógica en los últimos diez años con alumnos universitarios de primero y segundo curso (primero a cuarto cuatrimestre/semestre) de las carreras de Diplomatura y Licenciatura en Informática, nos ha llevado a la conclusión de que la enseñanza de la algoritmia como soporte de la programación debía apoyarse en el problema a resolver y no en el lenguaje de programación en el que se desea producir el programa.

El primer paso a considerar es la selección de una herramienta de programación adecuada que permita expresar los algoritmos, dado que la dificultad mayor que se encuentra el alumno novato en el arte de la programación es pasar del problema a su solución algorítmica. Haciendo abstracción de las herramientas clásicas tales como diagramas de flujo y diagramas N-S, hemos optado además de explicar estas herramientas en describir y utilizar como herramienta fundamental el pseudocódigo. Para conseguir un aprendizaje gradual y eficiente decidimos elegir un *pseudolenguaje* (lenguaje de especificación de algoritmos) ya experimentado, rodado y probado con un gran número de alumnos universitarios en la última década y que ha sido contrastado por numerosos profesores de nuestra universidad y de otras universidades españolas y latinoamericanas. Este lenguaje de especificación

conocido en sus orígenes como UPS, ha sido bautizado finalmente con el nombre **UPSAM** (Universidad Pontificia de Salamanca en Madrid) dado que ha sido en su Facultad y Escuela Universitaria de Informática del campus de Madrid, donde se ha experimentado este pseudolenguaje en diferentes asignaturas impartidas por el Departamento de Lenguajes y Sistemas Informáticos e Ingeniería de Software, tales como Fundamentos de Programación, Tecnología de la Programación, Metodología de la Programación, etc.

El pseudocódigo es un lenguaje de expresión de algoritmos en español (o «castellano») basado en lenguajes de programación estructurados clásicos fundamentalmente Pascal, C y Ada. Este lenguaje de especificación de algoritmos permitirá pasar del problema a su solución algorítmica y posteriormente a la codificación en un lenguaje de programación.

Para la elaboración del algoritmo formalizaremos progresivamente el enunciado del problema analizando los elementos contenidos en el enunciado, en particular el vocabulario y la sintaxis junto con las tareas y objetos a manipular. La filosofía del diseño será el análisis descendente que conduce a un algoritmo estructurado. Aunque la etapa de codificación en un lenguaje de programación requiere cumplir escrupulosamente la sintaxis del mismo, pensamos que será fácil para el lector esta etapa si ha aprendido bien a diseñar algoritmos y estructuras de datos mediante pseudocódigo en español o más completo el lenguaje UPSAM.

Organización del libro

Este libro tiene un objetivo fundamental claramente práctico. Por esta causa su organización es similar en todos los capítulos: síntesis teórico-práctica y ejercicios resueltos. La síntesis teórico-práctica tiene como propósito describir y resumir los conceptos fundamentales del capítulo junto con la descripción de algoritmos clásicos y solución de ejercicios sencillos que muestran métodos de resolución de problemas frecuentes. Los ejercicios resueltos contienen un gran número de problemas propuestos junto con su solución y siguiendo siempre la misma metodología: *análisis del problema*, con descripción del mismo junto a variables de entrada y salida y operaciones básicas, *algoritmo* resuelto mediante diagrama de flujo y/o diagrama N-S y en la mayoría de los casos mediante pseudocódigo (el lenguaje UPSAM).

El lenguaje algorítmico de programación UPSAM

El lenguaje algorítmico UPSAM o pseudocódigo español utilizado como referencia básica en el texto ha sido creado, diseñado y experimentado por todos los profesores que han impartido las asignaturas de *Fundamentos de Programación* en las carreras universitarias *Técnicos en Informática Fundamental (TIF)*, *Diplomados en Informática* y *Licenciados en Informática* y *Metodología de la Programación* en los cursos de posgrado Master, Técnicos y Expertos en Informática de la Universidad Pontificia de Salamanca en Madrid y en su Departamento de Lenguajes y Sistemas Informáticos e Ingeniería de Software.

La guía de sintaxis del lenguaje algorítmico UPSAM se describe en el Apéndice A. Ha sido redactada por los profesores Matilde Fernández, Luis Rodríguez y Luis Joyanes y revisada y corregida por los profesores Luis Villar, Antonio Muñoz y Joaquín Abeger que han aportado sugerencias y mejoras al borrador original. Asimismo han contribuido a la redacción final y

pertenecen al equipo creador del lenguaje UPSAM los siguientes profesores de la citada Universidad:

- Ángel Hermoso.
- Ángela Carrasco.
- Francisco Oliva.
- Isabel Torralvo.
- Lucas Sánchez.
- M.^a del Mar García.
- Mercedes Vargas.
- Rafael Ojeda.
- Víctor Martín.
- Miguel Sánchez.

Profesores de asignaturas tales como Programación I y II, Estructura de Datos, Programación de Sistemas, Metodología de la Programación, Programación de Gestión, Sistemas Operativos y otras han utilizado el pseudocódigo UPSAM en una u otra forma y han dado sugerencias también que se han incorporado a la redacción final. Así y con el riesgo de olvidar algún nombre de algún colega y amigo —a quién pedimos disculpas por anticipado—, destacamos a:

- Amalia Oñate.
- Andrés Castillo.
- Ángel González.
- Antonio Reus.
- Daniel García.
- David Juanes.
- Enrique Torres.
- Fernando Davara.
- Francisco Agüera.
- Francisco Mata.
- José González.
- José Luis Andrés.
- Juan Ignacio Pérez.
- Juan Manuel Dodero.
- Juan Sanz.
- Julián Margolles.
- Luis Doreste.
- M.^a Luisa Díez.
- Manuel Redón.
- Manuel Riquelme.
- Paloma Centenera.
- Rosa Hernández.
- Virgilio Yagüe.
- Carlos de la Fuente.
- Sergio Ríos.

Libro complementario

El carácter eminentemente práctico de esta obra puede requerir en ocasiones un conocimiento teórico más profundo del descrito en la misma. Por esta causa y aunque se han explicado los conceptos teóricos fundamentales necesarios para la resolución de cualquier problema planteado en la obra, puede ser necesario para algunos lectores ampliar los conceptos teóricos planteados. En base a ello el libro se ha organizado con una estructura similar a la obra *Fundamentos de programación*, 2.^a edición (Luis Joyanes, McGraw-Hill, 1996) y con una correspondencia capítulo a capítulo muy parecida, de modo que ambos libros pueden considerarse complementarios y así pueden ser estudiados/leídos de modo independiente o bien de modo conjunto uno para formación teórica y otro para formación práctica.

Contenido

Esta obra se ha estructurado en catorce capítulos y un apéndice con la organización descrita en la Sección «Organización del libro».

El capítulo 1 presenta una breve descripción del concepto de algoritmo y programa. Introduce el concepto de algoritmo y las primeras construcciones y elementos de un lenguaje de programación. Se describen los tipos de datos y operaciones básicas que se realizan en cualquier lenguaje.

El capítulo 2 analiza los métodos para resolver 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.

En el capítulo 3 se describe 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.

La programación estructurada y sus técnicas es uno de los conceptos clave que se describen y utilizan a lo largo del texto. El capítulo 4 explica las técnicas de programación empleadas en la programación modular y estructurada; se amplía el concepto de flujo de control desarrollado en el capítulo 3 y se describen con una gran cantidad de ejemplos y ejercicios aclaratorios de las estructuras clásicas empleadas en programación estructurada: secuenciales, selectivas y repetitivas.

El capítulo 5 trata de los subprogramas (subalgoritmos): procedimientos y funciones. Los temas descritos incluyen la correspondencia de argumentos y parámetros, así como 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*. El concepto de *recursión* o *recursividad* se introduce en el capítulo como técnica fundamental de diseño de algunos problemas de naturaleza recursiva.

El concepto de **array** (*tabla, matriz o vector*, «arreglo» en Latinoamérica) es el tema central del capítulo 6. El proceso de arrays de una y dos dimensiones se examina junto con el recorrido de los elementos pertenecientes a un array. Asimismo se describe el concepto de registro junto con el de array de registros.

El procesamiento de datos tipo carácter y las cadenas de caracteres se describen en el capítulo 7, así como las operaciones básicas con cadenas.

El capítulo 8 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 las operaciones típicas que se realizan con ellos son tema central del capítulo junto con las organizaciones secuenciales. El capítulo 9 describe las organizaciones y archivos aleatorios o de acceso directo y las indexadas así como los algoritmos de manipulación de los mismos.

Las aplicaciones más importantes y frecuentes en cualquier algoritmo son la ordenación, búsqueda y mezcla o intercalación de información. Los algoritmos más populares y reconocidos relativos a arrays unidimensionales se describen en el capítulo 10. Estos algoritmos se consideran internos dado que el procesamiento de los mismos se realiza en memoria central, para diferenciarlos de la manipulación de los archivos que se consideran externos y que se almacenan en soportes de almacenamiento externo. La ordenación, búsqueda y fusión externa se describen en el capítulo 11 mientras que las operaciones internas correspondientes se describen en el capítulo 10.

El capítulo 12 ofrece una introducción al estudio de las estructuras de datos lineales; se examinan estructuras de datos simples tales como listas enlazadas, pilas y colas. El capítulo 13 estudia las estructuras de datos no lineales: árboles y grafos. El tipo de árbol más usual es el árbol binario y se describe en profundidad.

Una de las herramientas clásicas empleadas en el análisis de un problema es la tabla de decisión. En el capítulo 14 se analiza el concepto, tipos, construcción y conversión de tablas de decisión en programas. La tabla de decisión es una herramienta de gestión eficaz que se complementa de modo excelente con el diseño de algoritmos en la resolución de problemas sobre todo en el campo de gestión.

El libro se completa con las especificaciones del lenguaje de especificación de algoritmos UPSAM que soporta el pseudocódigo y las reglas de sintaxis del lenguaje utilizado en todo el texto.

Agradecimientos

Nuestro agradecimiento al Departamento de Lenguajes y Sistemas Informáticos e Ingeniería de Software de la Universidad Pontificia de Salamanca en Madrid, cuyo lenguaje algorítmico UPSAM ha sido utilizado en todo el libro.

De modo especial agradecemos a los profesores actuales de Fundamentos, Tecnología y Metodología de la Programación, así como de Estructuras de datos y de Programación I y D, de Sistemas y de Gestión, que utilizan el lenguaje UPSAM y a quienes se debe su definición y construcción.

Madrid, verano de 1996

Los autores.

Algoritmos y programas

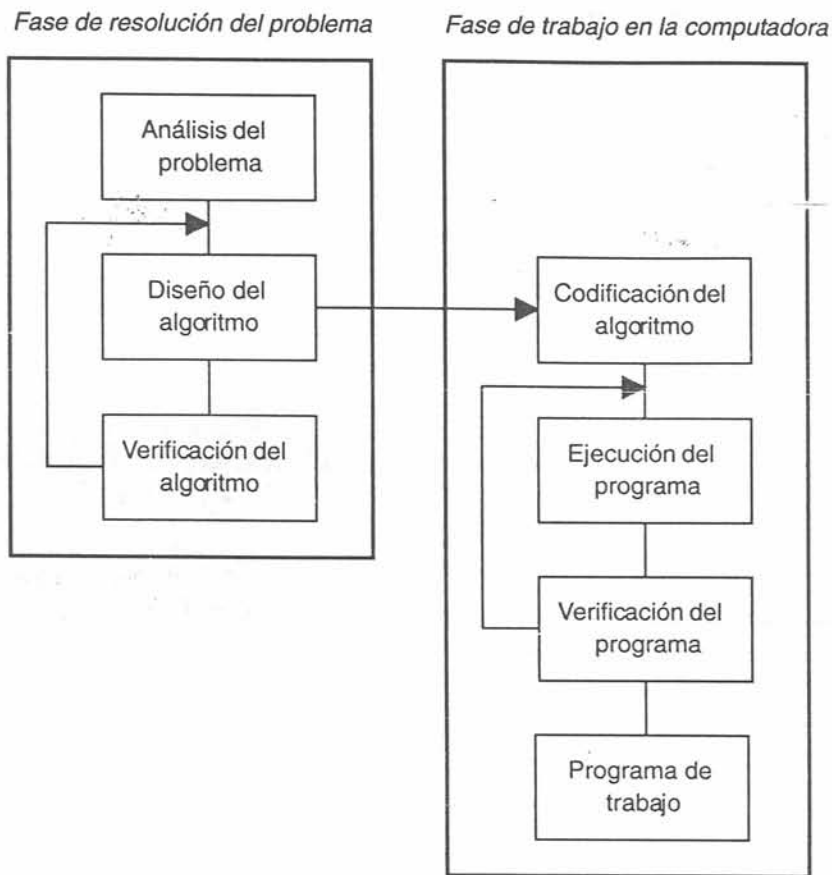
1.1. Resolución de problemas por computadoras

La principal razón para que las personas aprendan lenguajes de programación es utilizar la computadora como una herramienta para la resolución de problemas. Dos fases pueden ser identificadas en el proceso de resolución de problemas ayudados por computadora:

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

1.1.1. *Fase de resolución del problema*

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



1.1.1.1. 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.



1.1.1.2. Diseño del algoritmo

La palabra algoritmo deriva del nombre del famoso matemático y astrónomo árabe Al-Khōwarizmi (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 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.
- **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.

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?
- ¿Cuántos 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 diagrama de flujo, diagrama Nassi-Schneiderman o pseudocódigo, comentados más adelante.

1.1.1.3. 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.

1.1.2. Fase de implementación

Una vez que el algoritmo está diseñado, representado gráficamente mediante una herramienta de programación (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 el ordenador.

1.2. Datos

Dato es la expresión general que describe los objetos con los cuales opera el algoritmo. Los datos podrán ser 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 consola.

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 así, utilizando directamente el identificador o nombre del tipo. Además el usuario podrá definir sus propios tipos de datos.

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 a éste se le denominaría tipo base. Al establecer un tipo para un dato hay que considerar las operaciones que vamos a realizar con él y los instrumentos disponibles. Al establecer el tipo se determina la forma de representación en memoria. Los datos pueden venir expresados como constantes, variables, expresiones o funciones.

1.2.1. *Constantes*

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

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.

1.2.2. *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. 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.

1.2.3. *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(/), exponenciación (^), 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 ($\neq$), menor que (<), mayor que (>), mayor o igual ($\geq$), menor o igual ($\leq$). 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.

<i>a</i>	<i>b</i>	<i>no a</i>	<i>a y b</i>	<i>a o b</i>
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:

^	Exponenciación
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.

1.2.4. 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:

<i>Funcion</i>	<i>Descripción</i>	<i>Tipo de argumento</i>	<i>Resultado</i>
abs(x)	valor absoluto de x	entero o real	igual que el argumento
arctan(x)	arcotangente de x	entero o real	real
cos(x)	cosenode 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
ln(x)	logaritmo neperioano de x	entero o real	real
log10(x)	logaritmo base 10 de x	entero o real	real
raíz2(x)	raíz cuadrada de x	entero de x	real
redondeo(x)	redondea x al entero más proximo	real	entero
sen(x)	seno de x	entero o real	real
trunc(x)	parte entera de x	real	entero

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

1.2.5. Reglas para la construcción de 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 admitirá un máximo de 32 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.

1.3. 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 llegemos 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,...)
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, rellenar 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 transferencia 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

```

14 Fundamentos de Programación. Libro de Problemas

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

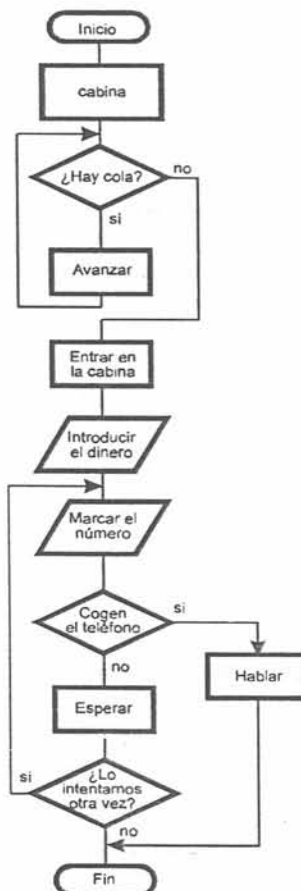
1.10. Realizar una llamada telefónica desde un teléfono público.

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



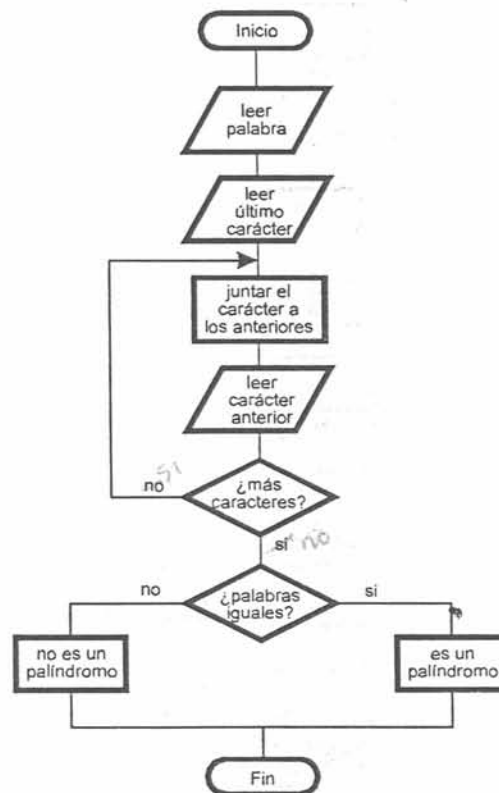
1.11. Averiguar si una palabra es un palíndromo. Un palíndromo es una palabra que se lee igual de izquierda a derecha que de derecha a izquierda, como por ejemplo, 'radar'.

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



- 1.12.** 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

```

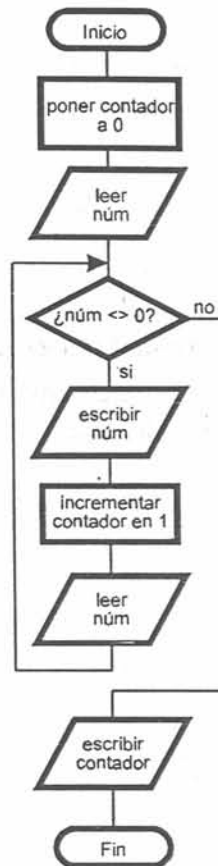
inicio
  leer (a,b)
  mientras a mod b <> 0
    resto ← a mod b
    a ← b
    b ← resto
  mcd ← b
  escribir (mcd)
fin
  
```

- 1.13.** 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

1.14. 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
  suma ← 0
  núm ← 3
  mientras núm ≤ 99 hacer
    escribir (núm)
    suma ← suma + núm
    núm ← núm + 3
  escribir (suma)
fin

```

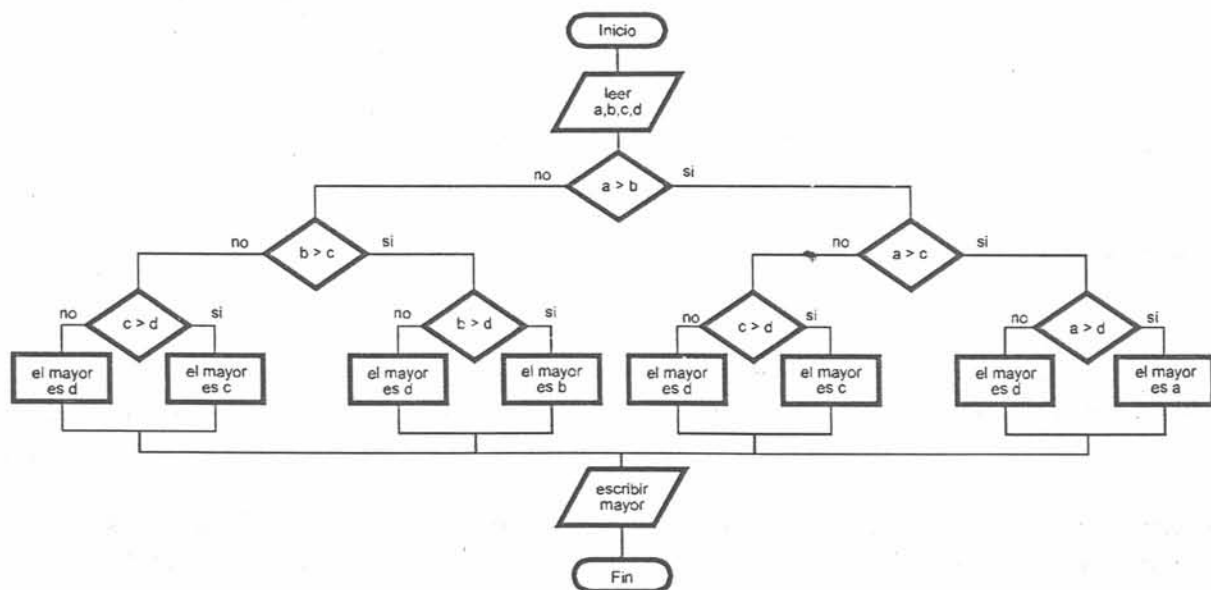
1.15. Escribir un algoritmo que lea cuatro números y, a continuación, escriba el mayor de los cuatro.

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

- 1.16. 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 $\text{tiempo} = \text{ss} + \text{mm} * 60$. La velocidad se hallará con la fórmula $\text{velocidad} = \text{distancia} / \text{tiempo}$.

Diseño del algoritmo

```

inicio
  distancia ← 1500
  leer (mm,ss)
  mientras mm = 0 y ss = 0 hacer
    tiempo ← ss + mm * 60
    v ← distancia / tiempo
    escribir (mm,ss,v)
    leer (mm,ss)
fin
  
```

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

Análisis del problema

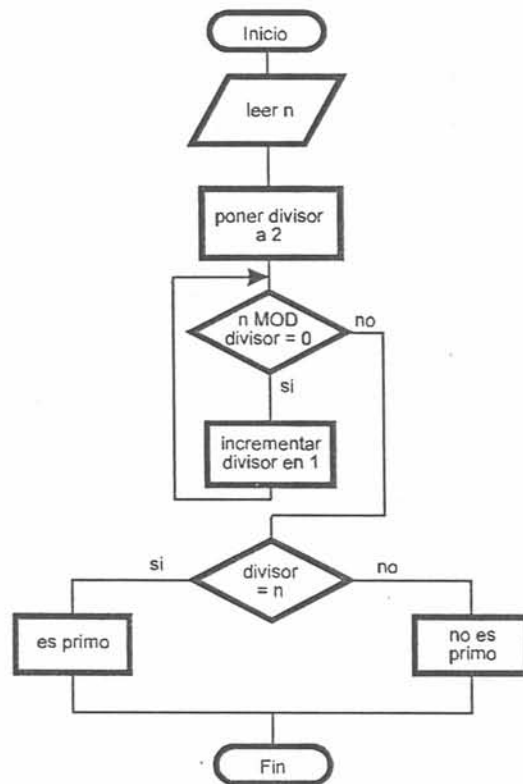
DATOS DE SALIDA: El mensaje que nos indica si es o no primo
 DATOS DE ENTRADA: n
 DATOS AUXILIARES: divisor (es el número por el que vamos a dividir n para averiguar si es primo)

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 lo 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 encuentra un divisor, es decir hasta que $n \bmod \text{divisor} = 0$. Si al salir del bucle $\text{divisor} = n$, el número será primo.

Diseño del algoritmo



1.18. 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)
   $s \leftarrow b * a / 2$ 
  escribir (s)
fin
```

La resolución de problemas con computadoras y las herramientas de programación

2.1. Herramientas de programación

Un algoritmo puede ser escrito en castellano narrativo, pero esta descripción suele ser demasiado prolija 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

2.2. 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.

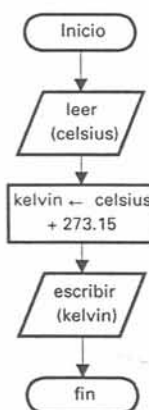
En la actualidad se siguen usando los diagramas de flujo del sistema, pero ha decaído el uso de los diagramas de flujo de detalle al aparecer otros métodos de diseño estructurado más eficaces para la representación y actualización de los algoritmos.

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.



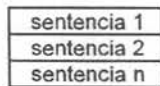
2.3. Diagrama Nassi-Schneiderman

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 leen siempre de arriba-abajo y se documentarán de la forma adecuada.

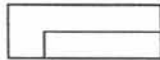
En los diagramas N-S las tres estructuras básicas de la programación estructurada, secuenciales, selectivas y repetitivas, encuentran su representación propia. La programación estructurada será tratada en capítulos posteriores.

Símbolo

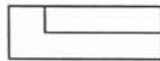
Tipo de estructura



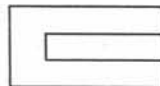
Secuencial



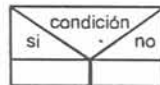
Repetitiva de 0 a n veces *MIENTRAS*



Repetitiva de 1 a n veces *REPETIR HASTA*

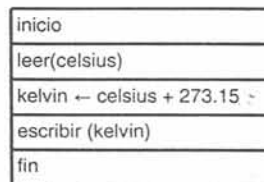


Repetitiva n veces *FOR i = 1 HASTA n*



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:



2.4. 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. En nuestros pseudocódigos usaremos determinadas palabras en español como palabras reservadas.

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.

El pseudocódigo comenzará siempre con la palabra **inicio** y terminará con la palabra **fin**. Cuando se coloque un comentario de una sola línea se escribirá precedido de **//**. Si el comentario es multilínea, lo pondremos entre **{}**. Para introducir un valor o serie de valores desde el dispositivo estándar y almacenarlos en una o varias variables utilizaremos **leer(<lista_de_variables>)**.

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>)**

Las distintas estructuras se representarán de la siguiente forma:

```

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 opcionalidad
                        <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
  
```

inicio, **fin**, **leer**, **escribir** y las palabras que aparecen en negrita en las distintas estructuras se consideran palabras reservadas y no deberán utilizarse en su lugar otras similares.

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. 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 un 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 $\text{numero} * 2 * 25$.

Diseño del algoritmo:

```
algoritmo ejercicio_2_3
    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 asignamos el resultado de sumar las cuatro variables y dividir las entre cuatro. Como el operador suma tiene menos prioridad que el operador división, la será necesario encerrar la suma entre paréntesis. También podríamos haber dividido directamente la variable suma entre cuatro.

Las 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
    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
  constante libra = 0.453592
  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) $A*B \div C$ | e) $A+B \bmod C$ |
| c) $A \div B + C$ | f) $A \div B \div C$ |

Los resultados serían:

a) $(6-2)+3 = 7$

b) $(6*2) \text{ div } 3 = 4$

c) $(6 \text{ div } 2)+3 = 6$

d) $(6*2) \text{ mod } 3 = 0$

e) $6+(2 \text{ mod } 3) = 8$

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?

$$A \leftarrow 5$$

$$B \leftarrow A + 6$$

$$A \leftarrow A + 1$$

$$B \leftarrow A - 5$$

Las variables irían tomando los siguientes valores:

$$A \leftarrow 5 \quad , \quad A = 5, B \text{ indeterminado}$$

$$B \leftarrow A + 6 \quad , \quad A = 5, B = 5 + 6 = 11$$

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

$$B \leftarrow A - 5 \quad , \quad A = 6, B = 6 - 5 = 1$$

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?

$$A \leftarrow 3$$

$$B \leftarrow 20$$

$$C \leftarrow A + B$$

$$B \leftarrow A + B$$

$$A \leftarrow B$$

Las variables tomarían sucesivamente los valores:

$$A \leftarrow 3 \quad , \quad A = 3, B \text{ y } C \text{ indeterminados}$$

$$B \leftarrow 20 \quad , \quad A = 3, B = 20, C \text{ indeterminado}$$

$$C \leftarrow A + B \quad , \quad A = 3, B = 20, C = 3 + 20 = 23$$

$$B \leftarrow A + B \quad , \quad A = 3, B = 3 + 20 = 23, C = 23$$

$$A \leftarrow B \quad , \quad A = 23, B = 23, C = 23$$

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

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

$$A \leftarrow 10$$

$$B \leftarrow 5$$

$$A \leftarrow B$$

$$B \leftarrow A$$

El resultado sería el siguiente:

$$A \leftarrow 10, A = 10, B \text{ indeterminada}$$

$$B \leftarrow 5, A = 10, B = 5$$

$$A \leftarrow B, A = 5, B = 5$$

$$B \leftarrow A, A = 5, B = 5$$

con lo que A y B tomarían el valor 5.

2.10. Escribir las siguientes expresiones en forma de expresiones algorítmicas

$$a) \frac{M}{N} + 4$$

$$b) M + \frac{N}{P - Q}$$

$$c) \frac{\sin x + \cos y}{\tan x}$$

$$d) \frac{M + N}{P - Q}$$

$$e) \frac{P + \frac{N}{P}}{Q - \frac{r}{5}}$$

$$f) \frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

$$a) M / N + 4$$

$$b) M + N / (P - Q)$$

$$c) (\sin(X) + \cos(X)) / \tan(X)$$

$$d) (M + N) / (P - Q)$$

$$e) (M + N / P) / (Q - R / 5)$$

$$f) ((-B) + \text{raíz2}(B^2 - 4 * A * C)) / (2 * A)$$

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
  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
  entero: cuadr
inicio
  cuadr ← 821 * 821
  escribir (cuadr)
fin
```

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

Análisis del problema

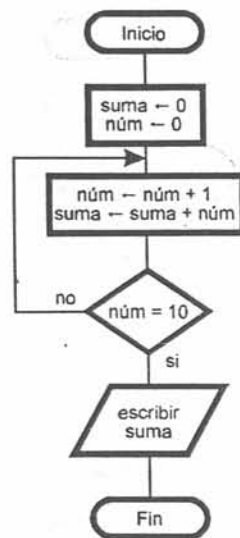
DATOS DE SALIDA: suma (contiene la suma requerida)
DATOS AUXILIARES: núm (será una variable que vaya tomando valores entre 1 y 10 y se acumulará en suma)

Hay que ejecutar un bucle que se realice 10 veces. En él se irá incrementando en 1 la variable `núm`, 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, núm



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

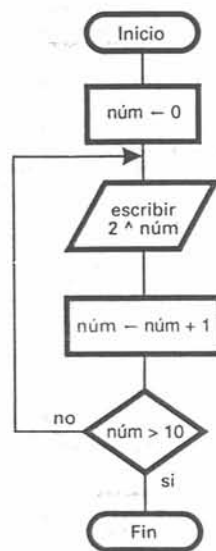
Análisis del problema

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 *núm*. También dentro de él se visualizará el resultado de la operación $2^{\text{núm}}$.

Diseño del algoritmo

TABLA DE VARIABLES:

entero: *núm*



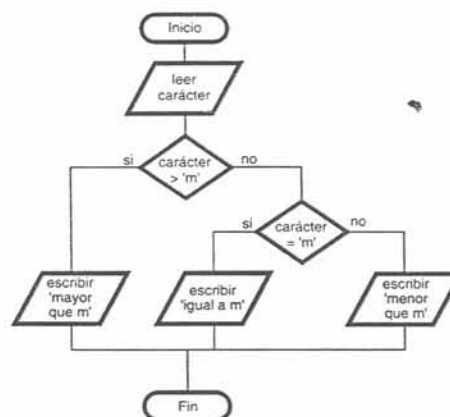
2.18. Leer un carácter y deducir si está situado antes o después de la «m» en orden alfabético.

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



2.19. 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_19
  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
```

2.20. 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_20
  carácter: c
inicio
  leer (c)
```

```
si (c >= 'I') y (c <= 'M') entonces
    escribir ('Está entre la I y la M')
si_no
    escribir ('no se encuentra en ese rango')
fin_si
fin
```

Estructura general de un programa

3.1. Escritura de algoritmos

En la escritura de algoritmos se hará necesario el uso de alguna herramienta de programación. La más adecuada es el pseudocódigo, y debe quedar todo lo más claro posible, de modo que se facilite al máximo su posterior codificación en un lenguaje de programación.

Al escribir un algoritmo deberemos considerar las siguientes partes:

- La cabecera
- El cuerpo del algoritmo.

La cabecera contiene el nombre del algoritmo completo precedido por la palabra **algoritmo**.

El cuerpo del algoritmo contiene a su vez otras dos secciones: 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. Se recomienda seguir este orden.

Para declarar las constantes con nombre el formato será

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

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. Todo esto se comentará con más detalle en capítulos posteriores.

Al declarar las variables tendremos que listar sus nombres y especificar sus tipos, lo que haremos de la siguiente forma:

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

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/fin.** La primera instrucción de este bloque será siempre la de inicio y la última la de fin.
- **de asignación.** Esta instrucción 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.

- **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. Cuando el dispositivo sea el dispositivo estándar de entrada, escribiremos:

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

- **de escritura.** Envía datos a un dispositivo. Cuando los datos los enviemos al dispositivo estándar el formato para la instrucción será:

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

- **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, resulta recomendable que los algoritmos lleven comentarios.

3.2. Contadores, acumuladores e interruptores

Entre las variables que utilizan los algoritmos merecen una especial mención contadores, acumuladores e interruptores.

3.2.1. Contadores

Un contador es una variable cuyo valor se incrementa o decrementa 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.

3.2.2. Acumuladores

Son variables cuyo valor se incrementa o decrementa en una cantidad variable. 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>`

3.2.3. Interruptores

Un interruptor, bandera o *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.

3.3. Estilo recomendado para la escritura de algoritmos

Como se dijo anteriormente, la herramienta de escritura de algoritmos más adecuada es el pseudocódigo. Con el empleo del pseudocódigo los algoritmos deberán presentar el siguiente aspecto:

```
algoritmo <nombre_algoritmo>
const
  <nombre_de_constant1> = 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

```

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 contendrían 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

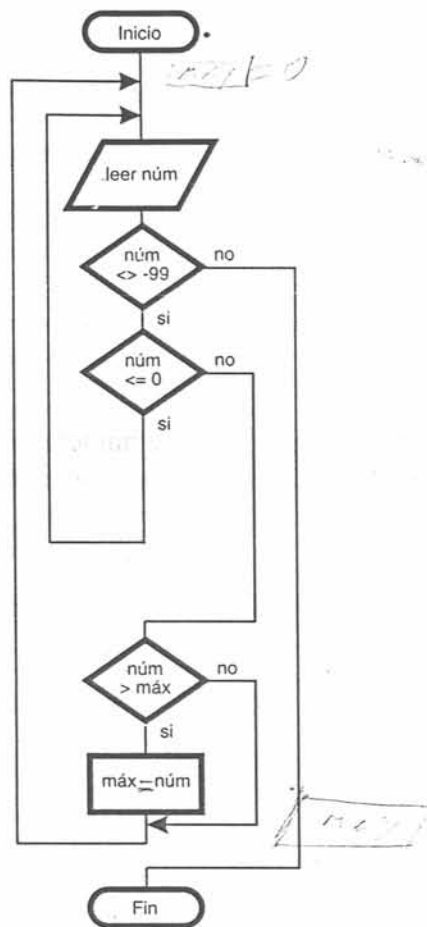
DATOS DE SALIDA: máx (el número mayor de la serie)
DATOS DE ENTRADA: núm (cada uno de los números que introducimos)

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 sea mayor que el máximo, en cuyo caso el nuevo máximo será el propio número.

Diseño del algoritmo

TABLA DE VARIABLES:

entero : máx, núm



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

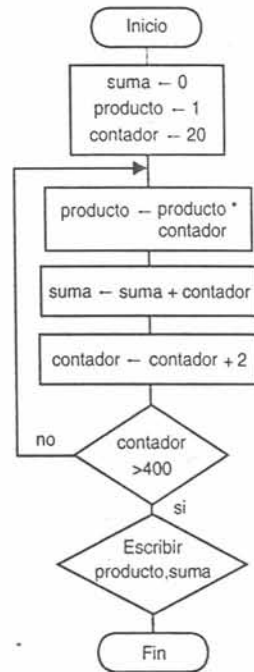
Análisis del problema

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: $\text{suma} \leftarrow \text{suma} + \text{contador}$ y $\text{producto} \leftarrow \text{producto} * \text{contador}$. Una vez realizadas las operaciones se debe incrementar el contador. Como se desea utilizar sólo los números pares, se usará una expresión como $\text{contador} \leftarrow \text{contador} + 2$.

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)

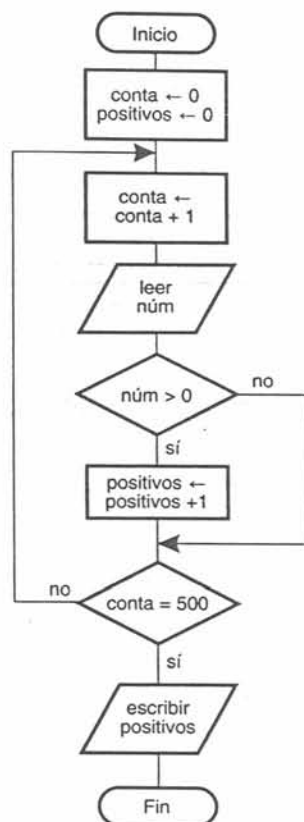
DATOS AUXILIARES: conta (se encarga de contar la cantidad de números introducidos)

Se ha de hacer un bucle que se ejecute 500 veces, controlado por la variable conta. Dentro del bucle se lee el número y comprobaremos 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 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 ($\text{neto} * 0.12$) y el bruto ($\text{neto} + \text{iva}$). Si el bruto es mayor que 50.000 pesetas, se le descuenta un 5% ($\text{bruto} \leftarrow \text{bruto} * 0.95$).

Al multiplicar el 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  $\leftarrow$  precio * unidades
    iva  $\leftarrow$  neto * 0.12
    bruto  $\leftarrow$  neto + iva
    si bruto < 50000 entonces
        bruto  $\leftarrow$  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

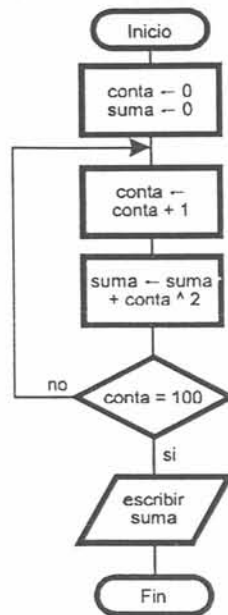
DATOS DE SALIDA: suma (acumula los cuadrados del número)
 DATOS AUXILIARES: conta (contador que controla las iteraciones del bucle)

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

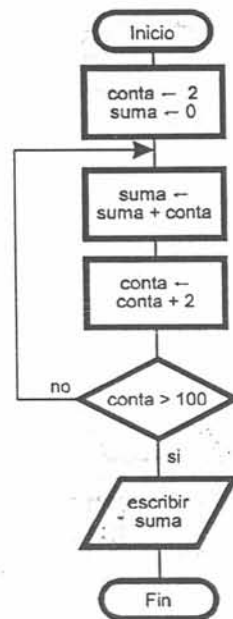
DATOS DE SALIDA: suma (contiene la suma de los números pares)
 DATOS AUXILIARES: conta (contador que va sacando los números pares)

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

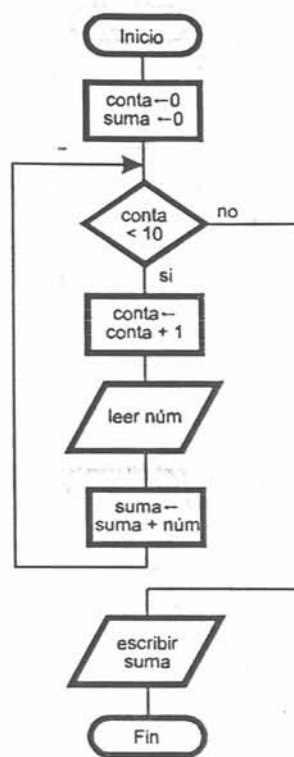
DATOS DE SALIDA: suma (suma de los números)
DATOS DE ENTRADA: núm (los números que introducimos por teclado)
DATOS AUXILIARES: conta (contador que controla la cantidad de números introducidos)

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 o igual 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: núm (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)

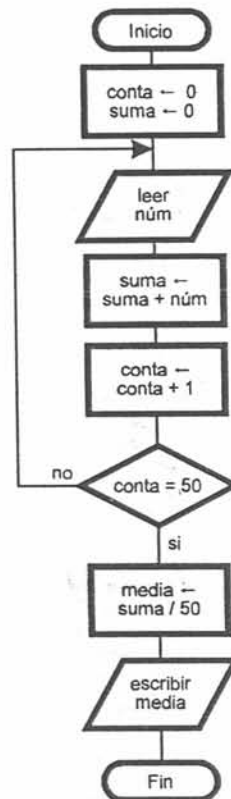
Después de inicializar conta y suma, se realiza un bucle que se repetirá 50 veces. En dicho bucle se lee un número (núm), se acumula su valor en suma y se incrementa el contador conta. El bucle se repetirá hasta que conta sea igual a 50. Una vez fuera del bucle se calcula la media ($\text{suma} / 50$)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

Diseño del algoritmo

TABLA DE VARIABLES

entero : núm, 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 que múltiplo vamos a visualizar)
 DATOS AUXILIARES: conta (contador que servirá para calcular los múltiplos de 4)

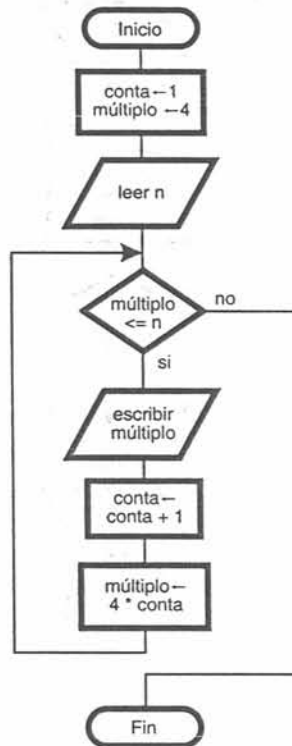
Para obtener los múltiplos de cuatro se pueden utilizar varios métodos. Un método consiste en sacar números correlativos entre 4 y N y para cada uno comprobar si es múltiplo de 4 mediante el operador `mod`. Otro método sería utilizar un contador que arrancando en 4 se fuera incrementado de 4 en 4. Aquí 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

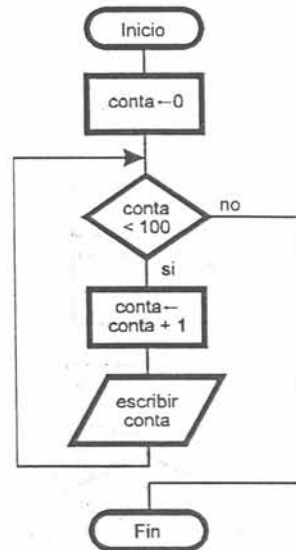
DATOS DE SALIDA: Los números enteros entre 1 y 100

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: num (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.

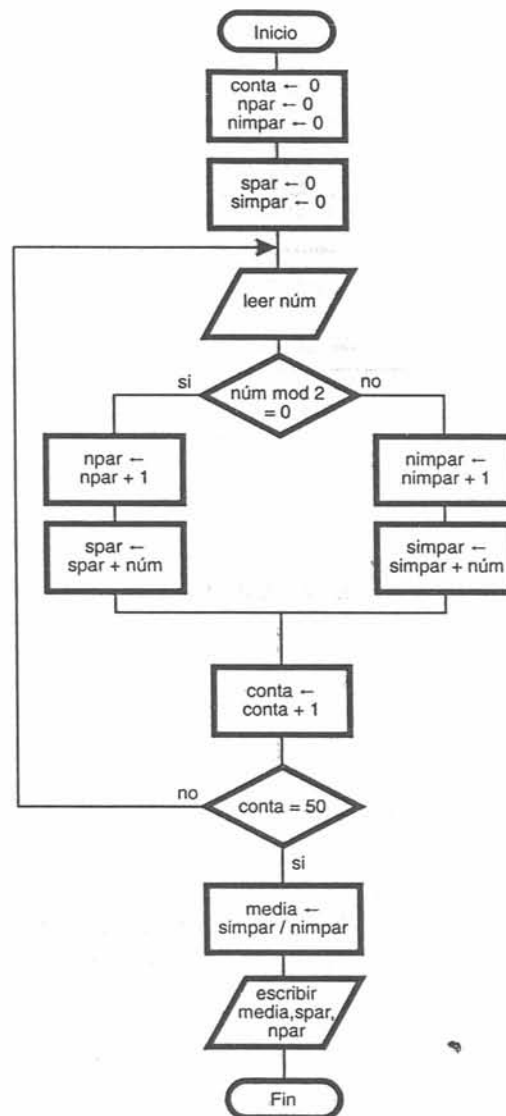
Ya fuera del bucle se calcula la media de impares ($\text{simpar} / \text{nimpar}$) y se escribe *spar*, *npar* y *media*.

Diseño del algoritmo

TABLA DE VARIABLES

entero : conta, núm, 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 distinto.

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.

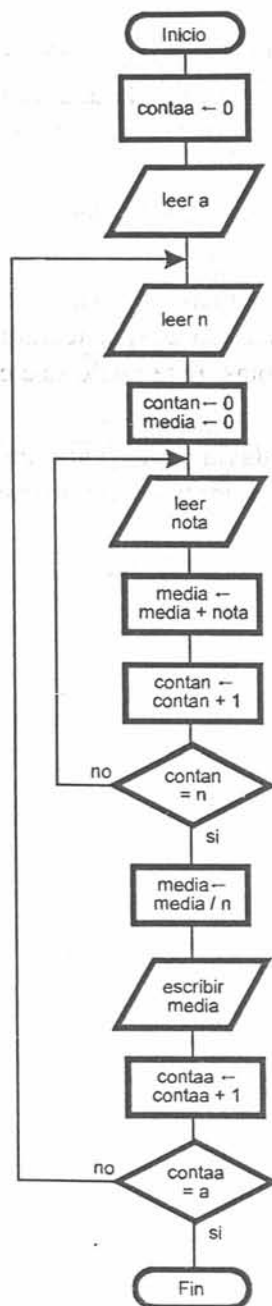
En el bucle de los alumnos se lee el número de notas e inicializar las variables media y contan a 0. Aquí comenzará el bucle de las notas en el que leeremos una nota, se acumula en la variable media y se incrementa el contador de notas. Este bucle se ejecutará hasta que el contador de notas sea igual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



Introducción a la programación estructurada

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. Estructuras de control

A las estructuras secuencial, selectiva y repetitiva se las denomina estructuras de control debido a que controlan el modo de ejecución del programa.

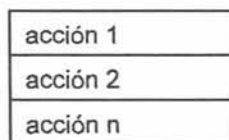
4.3.1. Estructuras secuenciales

Se caracterizan porque una acción se ejecuta detrás de otra. El flujo del programa coincide con el orden físico en el que se han ido poniendo las instrucciones.

Diagrama de flujo



Diagrama N-S



Pseudocódigo

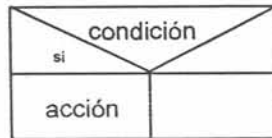
acción 1
acción 2
acción n

4.3.2. Estructuras selectivas

Se ejecutan unas acciones u otras según se cumpla o no una determinada condición; pueden ser *simples*, *dobles* o *múltiples*.

simples

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.

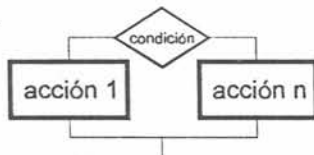
Diagrama de flujo***Diagrama N-S******Pseudocódigo***

```

si <condición> entonces
    acción
fin_si
  
```

dobles

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***

```

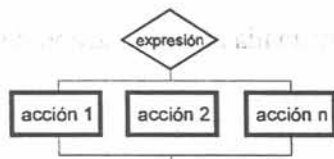
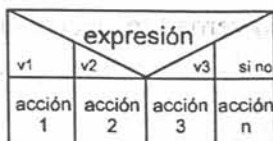
si <condición> entonces
    acción 1
si_no
    acción 2
fin_si
  
```

múltiple

Se ejecutarán unas acciones u otras según el resultado que se obtenga al evaluar una expresión. 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, cuente con el obtenido al evaluar la expresión. Cuando el valor obtenido al evaluar la expresión no esté presente en ninguna lista de valores se ejecutarían las acciones establecidas en la cláusula **si_no**, si existiese dicha cláusula.

Diagrama de flujo**Diagrama N-S****Pseudocódigo**

```

según_sea <expresión> hacer
    <lista1>: <acciones1>
    <lista2>: <acciones2>
    .....
    [si_no
      <accionesN>]
fin_según

```

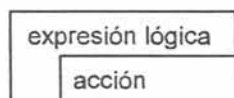
4.3.3. Estructuras repetitivas

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 anómalo 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**, **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**

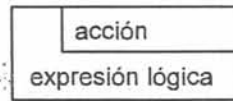
```

mientras <expresión_lógica> hacer
    <acciones>
fin_mientras

```

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**

```

repetir
    <acciones>
hasta_que <expresión_lógica>
  
```

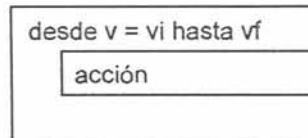
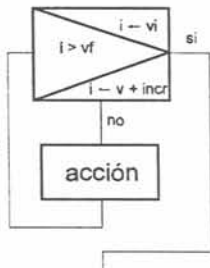
desde

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.

Diagrama de flujo**Diagrama N-S****Pseudocódigo**

```

desde v ← vi hasta vf
    ➡ [incremento | decremento incr]
    ➡ hacer
    <acciones>
fin_desde
  
```

4.3.4. Estructuras anidadas

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

```

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:

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

4.4. 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 ahorran comparaciones (sólo utilizamos 5). Si se utilizan comparaciones con operadores lógicos del tipo $A < B$ **and** $B > C$, 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 `raízcua` si se considera implementada; en caso contrario, deberá utilizarse la exponenciación.

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

$$\text{var mod } 2 = 0$$

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

$$\text{resto} = \text{dividendo} - \text{divisor} * \text{ent}(\text{dividendo} / \text{divisor})$$

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

$$\text{var} - 2 * \text{ent}(\text{var} / 2) = 0$$

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

o, siguiendo el mismo método si

$$\text{var} / 2 = \text{var div } 2$$

$$\text{var} / 2 = \text{ent}(\text{var} / 2)$$

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 ($\text{precio} = \text{distancia} * 2 * 2.5$). 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.

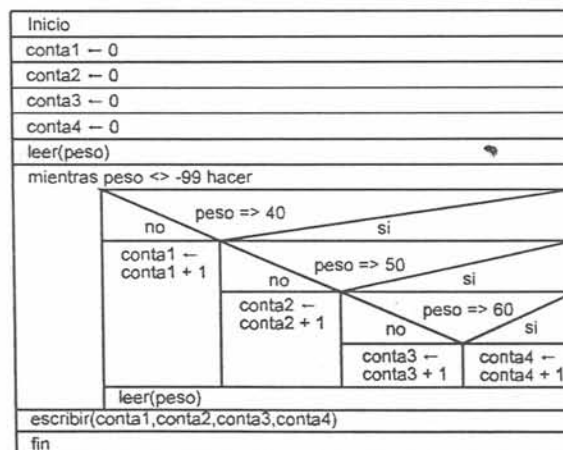
Nótese que ha de leer el peso una vez antes de entrar en el bucle y otra vez como última instrucción de éste, pues de lo contrario se incluirá el centinela (-99) en la estadística la última vez que se ejecute el bucle.

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)

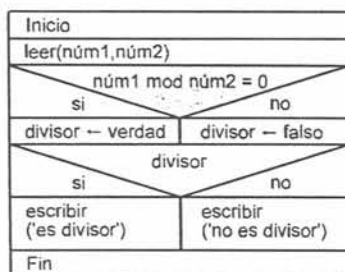
En este problema se utiliza el operador `mod` para comprobar si núm2 es divisor de núm1. Si el resto de dividir los dos números es 0 se establece la variable `divisor` como cierta, y en caso contrario como falsa.

Preguntando por el valor del divisor se obtiene un mensaje que nos indicará si núm2 es o no divisor de núm1.

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**:

desde $i \leftarrow 1$ hasta T hacer


```
leer(número)
suma ← suma + número
fin_desde
```

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

Con una estructura **mientras**:

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

Con una estructura **repetir**:

```
.
.
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: núm (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 : núm

Inicio	
leer(núm)	
núm >= 10 y núm <= 100	
si	no
escribir ('está entre 10 y 100')	
Fin	

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

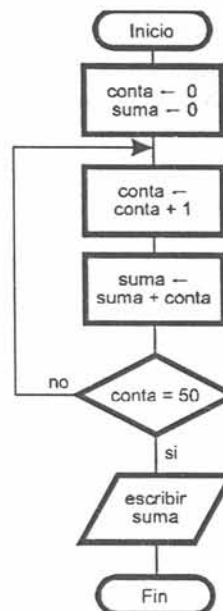
acumulador (suma). El bucle se ejecutará hasta que el contador sea igual a 50.

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 periodo)

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:

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

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:

X puede ser cualquier número real distinto de 0

n puede ser cualquier entero positivo, negativo o nulo

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

Análisis del problema

DATOS DE SALIDA: potencia (resultado de elevar X a la n)
DATOS DE ENTRADA: X (base de la exponenciación), 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
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 : núm
inicio
repetir
    leer(núm)
hasta_que núm < 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 lo único que vamos a hacer es una operación de lectura indicando que se lee desde un archivo:

```
leer(nombre del archivo, lista de variables)
```

Diseño del algoritmo

```
algoritmo ejercicio_4_19
var
  real : núm
inicio
  repetir
    leer(archivo, núm)
    si núm <> 0 entonces
      escribir(núm ^ 2)
    fin_si
  hasta_que núm = 0
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 depuración en cuanto a 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 que la semana comienza 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, dl
  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
    bisiestro ← 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 bisiestro 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 bisiestro 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
```

Subprogramas (subalgoritmos), procedimientos y funciones

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; por ejemplo:

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

a lo largo 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))`. O, 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. Este debe incluir una instrucción mediante la que la función tomará 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, para que, cuando se llame a la función se pueda establecer 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 sería

```
<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 quieren decir que se elija sólo una entre las distintas opciones que aparecen separadas por una barra. En las funciones habitualmente será **E**. Todo esto se detallará más adelante. Los corchetes indican no obligatoriedad. El tipo de dato debe ser estándar o haber sido definido de antemano. Podemos 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)]`

Como se puede apreciar con respecto a la lista de parámetros no existe obligatoriedad.

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 sería:

```
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 de la siguiente forma:

`{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

```

algoritmo <nombre_algoritmo>
const
    <nombre_de_constantel>=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 cómo se desea que se comporte 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.

Todo esto afectará tanto a procedimientos como a funciones. De lo que se deduce que 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. Consideraremos como variables globales las 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 él. 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 encontremos 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, recursividad directa. También es posible que un procedimiento o función haga referencia a otro el cual contenga, a su vez, una referencia directa o indirecta al primero, recursividad indirecta.

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 el factorial de un número n , que se define matemáticamente como:

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

El problema general se resuelve en términos de otro caso del mismo, hasta llegar a uno que se resuelve de forma no recursiva. Existe un acercamiento paulatino al caso no recursivo. Para comprender la recursividad hay que tener en cuenta que:

- 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 las 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 necesitado para almacenar los valores crece pues conforme a los niveles de anidamiento de las llamadas.
- El cuerpo del procedimiento o función debe disponer de una o varias instrucciones selectivas donde establecer la condición o condiciones de salida.

Todo algoritmo recursivo puede ser implementado en forma iterativa utilizando una pila.

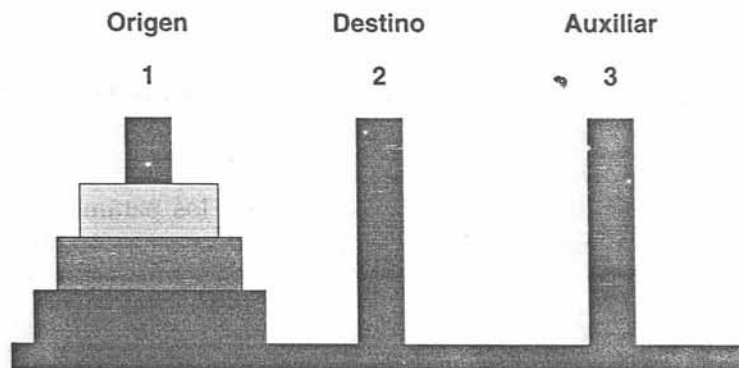
5.7.1. Algoritmos recursivos

El algoritmo de Ordenación Rápida (Quick Sort) tiene una definición recursiva. Ordena un array dividiéndolo en dos particiones más pequeñas del mismo, de forma que todos los elementos de una de ellas son menores que cada uno de los de la segunda y todos los de la otra mayores que cada uno de los de la primera. Estas particiones se subdividirán por separado hasta obtener particiones de un sólo elemento, terminación del proceso recursivo. Este algoritmo se desarrollará más adelante, cuando se trate el apartado de ordenación interna.

La forma iterativa de este algoritmo se verá en el capítulo que estudia los métodos de ordenación interna. Dejamos al lector la comparación entre dicha forma iterativa y la recursiva que aparece desarrollada en los ejercicios del presente capítulo.

El problema de las Torres de Hanoi es un problema clásico de recursión. Se tienen 3 torres y un conjunto de discos de diferentes tamaños. Cada disco tiene una perforación en el centro que le permite ensartarse en cualquiera de las torres. Los discos han de encontrarse siempre situados en alguna de las torres. Inicialmente todos están en la misma torre, ordenados de mayor a menor, como se muestra en el dibujo. Se deben averiguar los movimientos necesarios para pasar todos los discos a otra torre, utilizando la tercera como auxiliar y cumpliendo las siguientes reglas:

- En cada movimiento sólo puede intervenir un disco.
- No puede quedar un disco sobre otro de menor tamaño.



Para $N = 3$ la solución del problema implicaría los siguientes movimientos

1. De 1 a 2
2. De 1 a 3
3. De 2 a 3
4. De 1 a 2
5. De 3 a 1
6. De 3 a 2
7. De 1 a 2

algoritmo Hanoi

var

entero: n

inicio

escribir('Nº de discos:')

leer(n)

llamar_a 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

llamar_a Mover_torre (N-1, orig, aux, dest)

escribir('Paso de ', A, ' a ', B)

llamar_a Mover_torre (N-1, aux, dest, orig)

fin_si

fin_procedimiento

5.7.2. Algoritmos con retroceso

Hay un gran número de problemas cuya solución no puede obtenerse mediante una serie de cálculos más o menos complejos y su resolución requiere la aplicación de métodos de tanteo de forma sistemática, es decir ir probando todas las opciones posibles. Existen dos posibilidades: que la opción escogida forme parte de la solución buscada o que no, en cuyo caso se retrocederá para probar con otra posible opción. En estos problemas resulta muy útil la recursividad.

Un ejemplo de este tipo es 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 amenaza a las piezas situadas en su misma fila, columna o diagonal). La tarea más importante a realizar en este algoritmo, cuyo pseudocódigo se incluye en un ejercicio al final de este capítulo, es seleccionar adecuadamente la estructura de datos para representar el problema y los parámetros de los procedimientos recursivos.

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 intercambiar. 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 no recursiva que permita obtener el término n de la serie de Fibonacci.

Análisis del problema

La serie de Fibonacci se define como:

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

Por lo tanto se deben sumar los elementos mediante un bucle que debe ejecutarse desde 2 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 una 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 ← 2 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)

$$x = \text{radio} * \cos(\text{ángulo})$$

$$y = \text{radio} * \text{sen}(\text{ángulo})$$

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

$$\text{Factorial}_n = n * n-1 * n-2 * \dots * 1$$

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
        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

```

real 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 π mediante las series matemáticas siguientes:

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

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

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

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. En el caso (c) lo que se debe acumular son los productos; el numerador serán los números pares y el denominador los impares.

Diseño del algoritmo

(a)

real función Pi(E entero : n)

var

real : serie

entero : i, denom

lógico : par

inicio

denom ← 1

serie ← 0

par ← falso

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

```

(b)

```

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 pasaremos 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 i mod 2 = 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:

Suma de divisores de 284 : $1+2+4+71+142 = 220$

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:

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

Diseñar una función que permita calcular el número combinatorio $\binom{m}{n}$

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 $\sin(x)$ a partir de las series siguientes:

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

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

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

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 $\sin(x)$, 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 ←  $x^i / \text{Factorial}(i)$ 
    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 ←  $x^i / \text{Factorial}(i)$ 
        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
```

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:

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

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:

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

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 caracter: 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 $0! = 1$. 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)
  si_no
    devolver(n * fact(n-1))
  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...

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 = 0 o n = 1 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 7, 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. Diseñar un algoritmo que resuelva el problema de las ocho reinas.

Análisis del problema

El problema consiste en colocar las 8 damas dentro del tablero de ajedrez sin que se coman unas a otras, en general, se tratará de poner n damas en un tablero de $n \times n$.

En vez de colocar las damas 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 Damas se inicializa a 0. Para buscar la siguiente solución se comienza situando en Damas[1] un 1 (la primera dama está en la primera fila, columna 1). La siguiente dama estará en la fila 2 y empezamos a buscar en la posición 1. No será una posición válida si Damas[1] = Damas[2] (están en la misma columna); Damas[1]+1 = Damas[2]+2 (están en una diagonal) y Damas[1]-1 = Damas[2]-2 (está en la otra diagonal). Si esto ocurre, incrementamos la columna de la nueva dama y se volverá a comprobar.

Diseño del algoritmo

```
algoritmo Ejercicio_5_24
const n = 8
tipos array de entero[1..n] : ListaDamas
```

* El concepto de array se verá en el capítulo siguiente.

```

var
  ListaDamas : D
  entero : i
  lógico : solución
inicio
  // Inicializar el array
  desde i ← 1 hasta n hacer
    d[i] ← 0
  fin_desde
  Ensayar(d,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(E ListaDamas d ; E 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 (d[i] <> d[j])
    // no se ataca en una diagonal
    válida ← válida y (d[i] + i <> d[j] + j )
    // no se ataca en la otra diagonal
    válida ← válida y (d[i] - i <> d[j] - j )
  fin_desde
  devolver(Válida)
fin_función

procedimiento Ensayar( E/S ListaDamas : d ; E entero : i ;
                      S lógico : Solución)
inicio
  si i = n + 1 entonces
    solución ← verdad
  si_no
    solución ← falso
  repetir
    d[i] ← d[i] + 1
    si PosiciónVálida(d,i) entonces
      Ensayar(d,i+1,solución)
  hasta_que solución o (d[i] = n)
  si no solución entonces
    d[i] ← 0
  fin_si
  fin_si
fin_procedimiento

```

Estructuras de datos (arrays y registros)

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, en las estructuras de datos dinámicas el espacio ocupado en memoria puede ser modificado en tiempo de ejecución. Estructuras de datos estáticas son los arrays y los registros y las estructuras dinámicas son listas, árboles y grafos (estas estructuras no son soportadas en todos los lenguajes).

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*

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

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

utilizar un índice, que especifique su posición relativa en el array. Los arrays podrán ser

- Unidimensionales, también llamados Vectores
- Bidimensionales, 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

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

Array tridimensional

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

6.2.1. Arrays unidimensionales

Todo dato que se vaya 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

$$\langle \text{limsup} \rangle - \langle \text{liminf} \rangle + 1$$

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 $\langle \text{nombre_del_vector} \rangle$ fuera *vect* al elemento *n*ésimo del array se le referenciaría por *vect* [*n*], siendo *n* un valor situado entre el límite inferior y el límite superior. Los elementos podrán ser procesados como cualquier otra variable que fuera de dicho tipo_base.

6.2.2. Arrays bidimensionales

También se les denomina *matrices* o *tablas*. Un array bidimensional es un vector de vectores. 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	...	1,j
1	7,5	2,3	...	...
2	3,4	6,3	...	...
...	...	...	...	...
i				m[i,j]

Su declaración se haría de la siguiente forma:

```
tipo
  array[lInf..lSup, lInf..lSup] de tipo_base: nombre_del_tipo
var
  <nombre_del_tipo>: <nombre_de_la_matriz>
```

que aplicado a la matriz dibujada más arriba sería:

```
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 manejaremos los subíndices del array. Si se trata de un vector, bastará con una estructura repetitiva. Para el recorrido de una matriz o tabla se necesitarán dos estructuras repetitivas anidadas, una que controle las filas y otra las columnas. Además, en las matrices se consideran dos posibilidades:

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 primero el contenido de los elementos de la primera fila, a continuación el contenido de los de la segunda, etc.

```
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, etc.

```
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. Por 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. 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.

Ejemplo de un dato de este tipo 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

```

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.

6.3.1. 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.

Array de registros		1	Arrays paralelos			
Nombre	Edad		Nombres		Edades	
'Ana'	28	a[1]	'Ana'	Nombres[1]	28	Edades[1]
'Carlos'	36	a[2]	'Carlos'	Nombres[2]	36	Edades[2]
...	...	...	...	...	...	...
'Juan'	34	a[n]	'Juan'	Nombres[n]	34	Edades[n]

Para acceder al campo nombre del 2º elemento del array A se escribiría a[2].nombre. Por ejemplo,

escribir (a[2].nombre)

presentaría en consola la cadena Carlos.

6.4. 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] + M[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	5
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 rellenar)

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 **para**, 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

```

```

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 rellenar 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 x 4).

Análisis del problema

DATOS DE SALIDA: tabla (la tabla a escribir)

DATOS AUXILIARES: i,j (índices de las 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 la 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: VENTAS (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_si

```



```

    fin_si
    fin_desde
    EscribirVector(LISTA1, conta1)
    EscribirVector(LISTA2, conta2)
    EscribirVector(LISTA3, conta3)
fin

procedimiento EscribirVector(E vector: v ; E 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 rellenar), 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
         $A[i] \leftarrow i$ 
    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 el

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), núm (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 núm, 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 ($\text{suma} / \text{conta}$) 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 : núm,suma,media,desviación
inicio
  i ← 0
  suma ← 0
  leer(núm)
  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 estándar)
 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 estándar 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( E vector : v ; E entero : n)
inicio
    devolver(SumaReal( v,n) / n)
fin_función

real función SumaReal( E vector : v ; E 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 x 3.

Análisis del problema

DATOS DE ENTRADA: matriz (la tabla o matriz de 3 x 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 lo 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 $CIUDAD[i]$. En el bucle interno se debe comprobar el contenido de $ENLACE[i, j]$. Si éste es distinto de 0, se escribe la ciudad con la que está enlazada ($CIUDAD[j]$).

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 utilizó 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 variable `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 ← V
    // 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 traspuesta de una matriz M de 6 x 7 elementos.

Análisis del problema

DATOS DE SALIDA: MT (matriz traspuesta de M)
 DATOS DE ENTRADA: M (matriz original)
 DATOS AUXILIARES: i, j (índice de las matrices)

Una matriz traspuesta (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 dadas dos matrices pasadas como parámetros, comprobar si 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(E vector :A,B ; E 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 lo 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 la 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 <> O entonces
        error ← V
    si_no
        error ← F
        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 $\text{suma} \leftarrow \text{suma} + \text{tabla}[i, j]$. 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 n 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 lo 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 lo 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 : tabla ; 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 t[i,j] = t[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

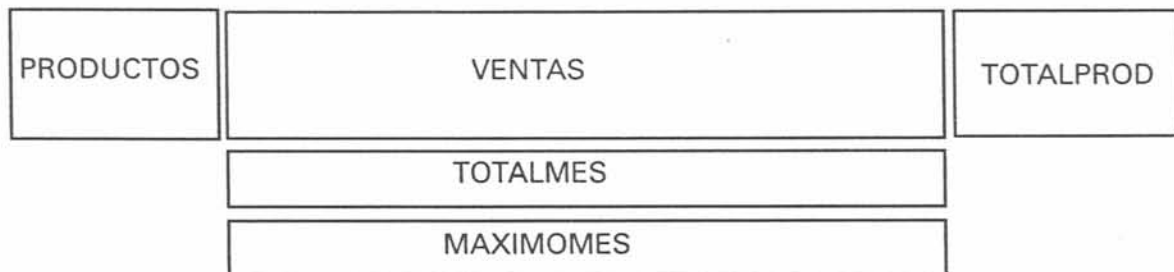
Como resultado final, se desea realizar un listado con el siguiente formato:

	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:

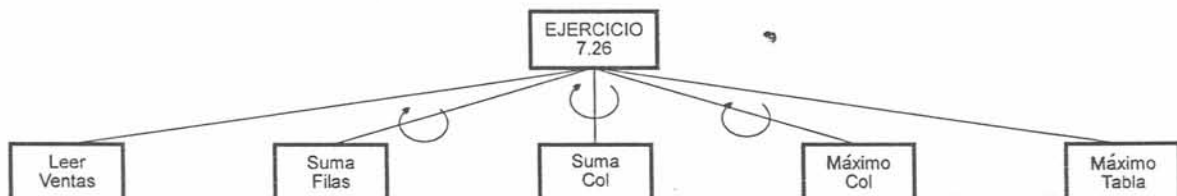
Análisis del problema

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

```

```

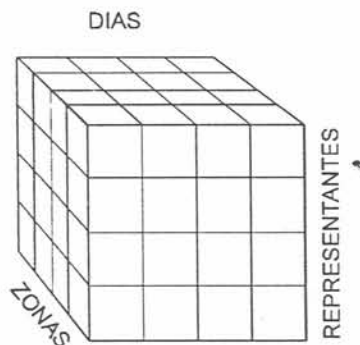
procedimiento MáximoTabla( E tabla : v ; E entero : m,n ;S entero : f,c)
var
    entero : i, j
inicio
    f ← 1
    c ← 1
    desde i ← 1 hasta m hacer
        desde j ← 1 hasta n hacer
            si t[i,j] > t[f,c] entonces
                f ← i
                c ← j
            fin_si
        fin_desde
    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


```

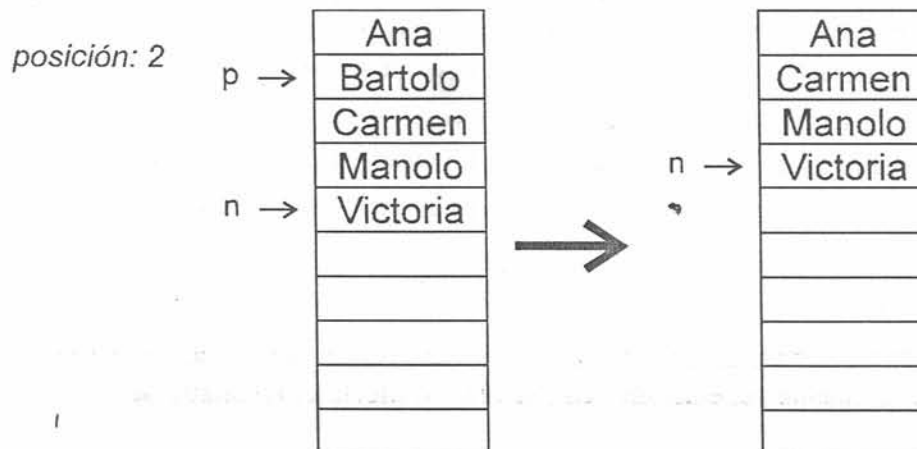
var
  entero : i
inicio
  // consideramos como error el que la lista esté llena o
  // o que la posición a insertar mas allá del número
  // de elementos + 1
  error ← (n = MáxLista) o (p > n+1)
  si no error entonces
    desde i ← n hasta p-1 incremento -1 hacer
      lista[i+1] ← lista[i]
    fin_desde
    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
  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 matriz[i,r]

Multiplicar fila i por matriz[i+1,r]/matriz[i,r] y restarlo a la i+1

Multiplicar fila i por matriz[i+2,r]/matriz[i,r] y restarlo a la i+2

.....

Multiplicar fila i por matriz[m,r]/matriz[i,r] y restarlo a la m

Se deberá almacenar en una variable auxiliar, cs, el contenido de matriz[i+x, r], para que, en las siguientes operaciones con la fila, no afecte el resultado de

$$\text{matriz}[i+x, r+y] \leftarrow \text{matriz}[i+x, r+y] - \text{matriz}[i, r+y] * (\text{cs} / \text{matriz}[i, r])$$

a

$$\text{matriz}[i+x, r] \leftarrow \text{matriz}[i+x, r] - \text{matriz}[i, r] * (\text{matriz}[i+x, r] / \text{matriz}[i, r])$$

- 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

escribir

// Pasar 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
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.

Análisis del algoritmo

```

tipo
  registro : Fecha
    entero :dd, mm, aa
  fin
registro : Medicamento registro
  cadena : Código , Nombre, Descripción
  entero : Precio
  real : IVA
  entero : Stock
  Fecha : Caducidad
fin
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 registro
  cadena : Código , Nombre, Descripción
  entero : Precio
  real : IVA
  entero : Stock
  registro : Caducidad
    entero : dd, mm, aa
  fin
fin

```

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 : Cuadrilátero
        Punto: Inicio, Lado1, Lado2, Lado3, Lado4
    fin

```

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:

<i>Hasta de 1.500.000</i>	<i>0,00%</i>
<i>Más de 1.500.000 y hasta 2.150.000</i>	<i>13,75%</i>
<i>Más de 2.150.000 y hasta 2.900.000</i>	<i>16,50%</i>
<i>Más de 2.900.000 y hasta 3.350.000</i>	<i>17,60%</i>
<i>Más de 3.350.000</i>	<i>18,85%</i>

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

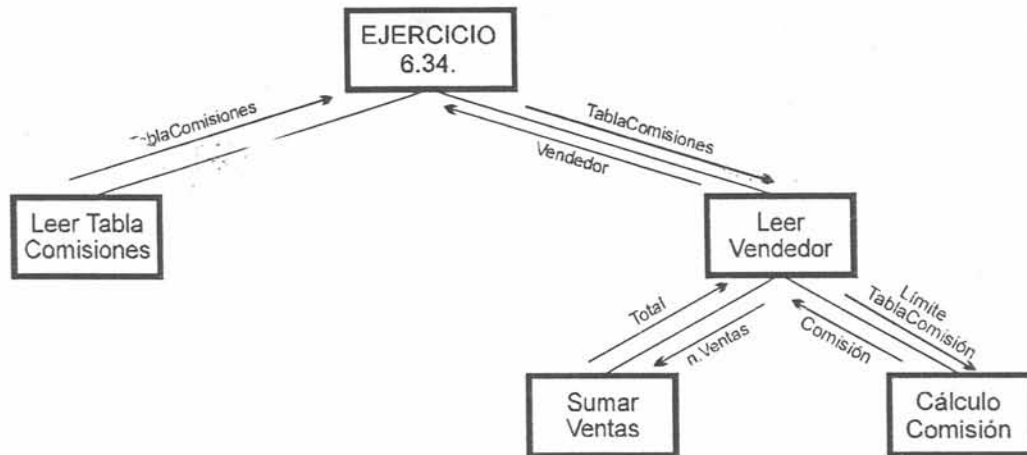
En este caso los campos del registro vendedor serán: DNI de tipo cadena, Nombre de tipo cadena, Apellidos de tipo cadena, Zona también de tipo cadena, Sueldo de tipo entero, Ventas que será un array de enteros, Total de tipo entero y Comisión que será real. Se define también un registro para la tabla de comisiones. Aunque el enunciado hable 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 lo tanto, la arquitectura de nuestro algoritmo podría quedar de la forma siguiente:



Diseño del algoritmo

algoritmo ejercicio_6_35

tipo

array[1..12] de entero : vector

registro : RegistroComisiones

entero : Límite, Porcentaje

fin

registro : Vendedores

cadena : DNI, Apellidos, Nombre, Zona

vector : Ventas

entero : Sueldo, Total, Comisión

fin

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

- 6.36. 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 lo 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 rellenarán 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 : Línea
        Punto: Inicio, 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 C P[1].Inicio
```

```
fin_procedimiento
```

```
procedimiento LeerPunto( S Punto : p)
```

```
inicio
```

```
    leer(p.x)
```

```
    leer(p.y)
```

```
fin_procedimiento
```

Las cadenas de caracteres

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 punteros con dos campos, que contienen la longitud máxima y la longitud actual. Las **cadenas de longitud indefinida** se representan mediante listas enlazadas, listas que se unen mediante puntero.

7.2. Ejercicios resueltos

7.1. Suponiendo que en nuestro 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 el 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

Analizaremos cada una de las funciones:

- 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.
- 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.
- 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 extrae 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 que 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, longitud(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

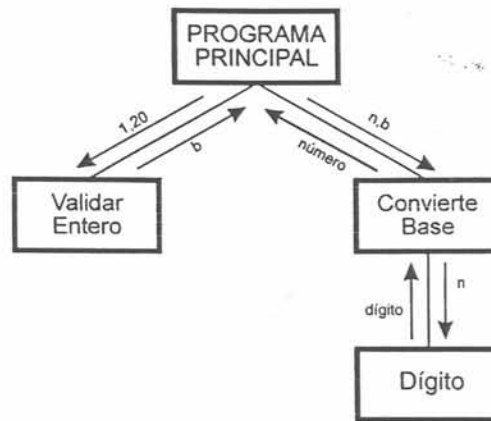
devolver(c)

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, debemos 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))
    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 `código()`) y se obtendrá el carácter correspondiente a dicho código utilizando la función `carácter()`. El proceso para convertir a minúsculas es igual, pero sumando 32 al código ASCII, en vez de restando.

Esto es válido para los caracteres ASCII estándar (hasta el 127). Para el resto de los caracteres como la ñ, 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] ← carácter(código(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] ← 'A'
                'é','è','ê','ë' : c[i] ← 'E'
                'í','ì','î','ï' : c[i] ← 'I'
                'ó','ò','ô','ö' : c[i] ← 'O'
                'ú','ù','û' : c[i] ← 'U'
                '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] ← carácter(código(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

logico 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
```

Archivos (ficheros)

Archivos secuenciales

8.1. Conceptos generales sobre archivos

Los archivos 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* lo define 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** serían un conjunto de registros relacionados entre sí. Un **registro** es un conjunto de campos. Un **campo** puede estar dividido en subcampos. Cuando 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 ficheros 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

Las operaciones básicas que se realizan al trabajar con archivos son:

Creación	Consiste en definirlos mediante un nombre y unos atributos, nombre de archivo y de dispositivo, tamaño, tamaño de bloque y organización.
Apertura	Define un identificador lógico que permite la comunicación de la CPU con el dispositivo de soporte físico del archivo; de esta manera los registros se vuelven accesibles. La operación de abrir archivos se puede aplicar para operaciones de entrada, salida o entrada/salida.
Clausura	Cerrar la conexión entre el identificador y el dispositivo de almacenamiento externo.
Lectura de datos	Copia los registros del archivo sobre variables en memoria central.
Escritura de datos	Copia la información contenida en variables sobre un registro del fichero.
Borrado del fichero	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 o 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. 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 suele 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>), donde *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 colocará 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 **r**. 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>), donde <lista_de_variables>, serán variables del tipo base del archivo.

leer(<id_archivo>, <lista_de_variables>), donde <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.

8.2.1. Archivos de texto

Los archivos de texto son un caso particular de archivos con organización secuencial. Un archivo de texto es un archivo en el que cada registro es del tipo cadena de caracteres. Se podrá detectar en ellos el fin de línea, mediante la función **fdl**(<id_archivo>). También contaremos con la función **leercar**(<id_archivo>, <var_de_tipo_carácter>) para leer carácter a carácter.

8.2.2. *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 crear 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.

8.3. *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 último_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,1,'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,1,'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 `PERSO-` `NAL`.

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,1,'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, personaaux
  lógico         : encontrado
inicio
  abrir(f, 1, '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 personaaux.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, 1, '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 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**, **TÍTULO**, **AUTOR** y **PRECIO**. El fichero está ordenado ascendentemente 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,1,'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, 1, '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

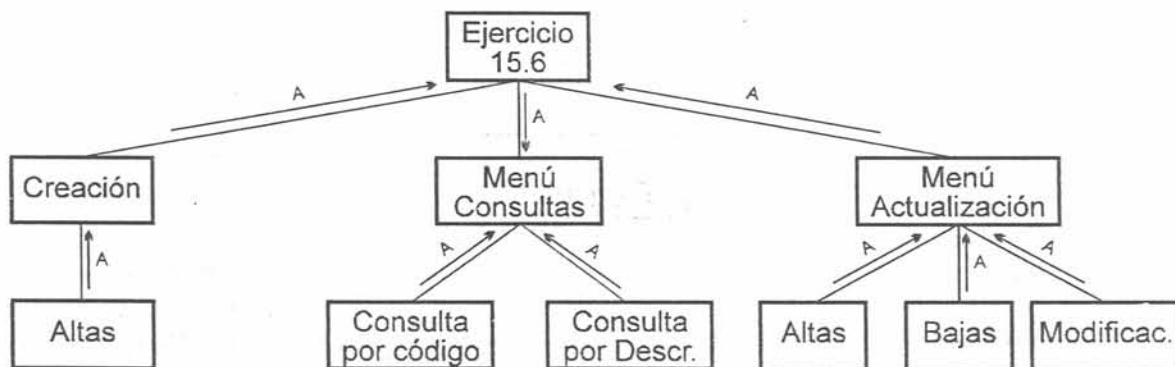
```


- 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.

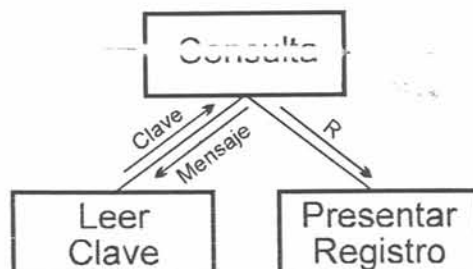
El programa se va a dividir en una serie de procedimientos llamados desde el programa principal a base de menús. Las opciones del menú serán Creación, Consultas, Actualización y Fin del Programa. Las consultas llamarán a otro menú con las opciones de Consulta por Código y Consulta por Descripción. La 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:



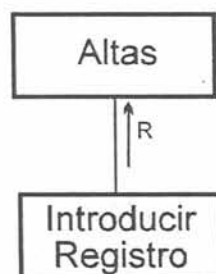
Para la opción de Creación, el archivo se pasará como parámetro de salida. Primero se comprueba si el archivo ya existe. Para ello los distintos lenguajes de programación pueden actuar de forma distinta. Se podría ver si existe el archivo abriéndolo para lectura, con lo que daría un error si no existe. 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

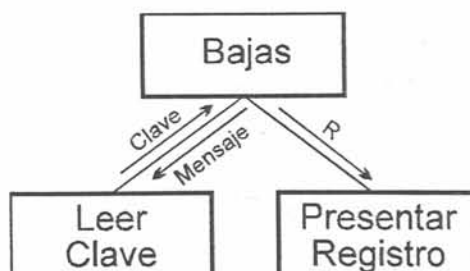
registro deseado. Al final, si no se ha encontrado el registro, saldrá un mensaje advirtiéndolo; en caso contrario, se visualizan los datos mediante el procedimiento `PresentarRegistro`. La descomposición modular de cualquiera de los dos modos de consulta podría quedar de la siguiente forma:



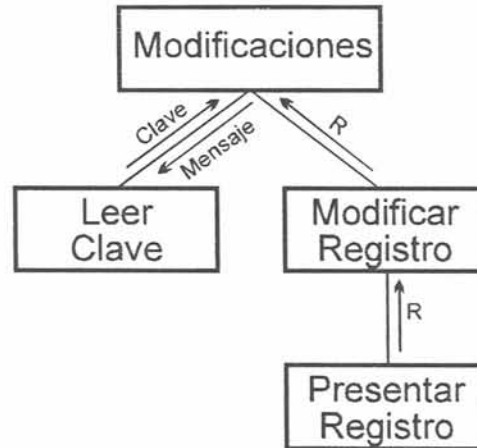
En las actualizaciones también se utiliza un menú que llamará a los procedimientos `Altas`, `Bajas` y `Modificaciones`. Para las altas, se abre el archivo para añadir y mediante un bucle controlado por centinela se introducen los datos de los registros mediante el procedimiento `IntroducirRegistro`. 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_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,1,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,1,NombreArchivo)
        LeerClave(clave, 'Descripción: ')
        leer(A,R)
        mientras no FDA(A) y R.Descr <> Clave hacer
            leer(A,R)
        fin_mientras
        si R.Descr = 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.Descr)
    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.Descr)
    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, 1, NombreArchivo)

 abrir(A-Aux, e, 'AUXILIAR')

 LeerClave(clave, 'Código: ')

 leer(A,R)

 encontrado ← F

 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 ← V

 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, 1, NombreArchivo)
  abrir(A-Aux, e, 'AUXILIAR')
  LeerClave(clave, 'Código: ')
  leer(A,R)
  encontrado ← F
  mientras no FDA(A)hacer
    si R.Código = Clave entonces
      ModificarRegistro(R)
    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.Descr)
      '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, 1, '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, 1, '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, 1, '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
```

Archivos directos

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.

En los archivos con organización directa 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>*, *l/e*, *<nombre_en_disco>*), los archivos se abren siempre para lectura/escritura (*l/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*.

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.

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:

$$\text{lda}(\text{id_archivo}) / \text{tamaño_de}(\text{var_tipo_base})$$

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 crear 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.

Area de índices
clave dirección

24	0010
41	0020
56	0030
92	0040
...	...
245	0100

Area de datos
clave datos

0010	6	
0011	18	
0012	20	
...	...	
0019	24	
0020	28	
0021	29	
...	...	
0029	41	
0030	43	
...	...	
...	...	
0090	...	
...	...	
0100	245	

Se puede acceder a la información de forma secuencial o a través del índice. Secuencialmente, recorreremos 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 coincida 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.

```
lda(<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. *Ficheros 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. Interesará 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 $(\text{índice} - 1) * \text{tamaño_de}(\text{datos_personales})$.

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 claves 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, registros 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_campon
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(personaux.cód)
    posi ← HASH(personaux.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_campos(personaaux)
    persona ← personaaux
    persona.ocupado ← '*' //Al dar un alta marcaremos el campo ocupado
    escribir(f, persona, posi)
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_campon
    fin
    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 <> personaaux.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 = 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_datol> : cód // Podría no ser necesario
                  // su almacenamiento, en el caso

```

```

                                // de que coincidiera con el índice
                                ..... : .....
                                <tipo_datoN> : nombre_campon
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(personaux.cód)
  posi ← HASH(personaux.cód)
  leer(f, persona, posi)
  si persona.ocupado <>'*' o persona.cód <> personaaux.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= personaaux.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_datol> : 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, personaaux

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 <> personaaux.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 = personaaux.cód entonces
encontrado ← verdad

fin_si

fin_mientras

si_no

encontrado ← verdad

fin_si

si encontrado entonces

llamar_a leer_otros_campos(personaux)

personaaux.ocupado = '*'

escribir(f, personaaux, 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° de contrato (3 dígitos), Nombre y apellidos, DNI, y N° de la Seguridad Social. En el otro se encuentran: N° 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° 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. Recurriremos 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(f3,1/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(f1,1,'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(f2,1,'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
  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,l/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 = 'O' 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 <> 'O' // hasta encontrar un registro libre
    fin_si
    // tenga o no colisiones
    LeerDatos(RegAux) // rellenamos los campos del registro auxiliar
    RegAux.estado ← 'O' // 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: $\text{hash}(\text{clave}) = \text{clave} \bmod 97 + 1$.

```

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, 1, 'SECUENCIAL')
    crear(A, 'PRODUCTOS')
    abrir(A, 1/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(altas), 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, 1, 'MOVIMIENTOS')
  abrir(AI, e, 'INCIDENCIAS')
  abrir(A, 1/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.II. Diseñar un algoritmo que permita efectuar ALTAS en un archivo indexado.

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_datol> : cód
    entero      : posic
fin_registro
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_datol> : 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[primero].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_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

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: *CODIGO* (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, posi, 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)

 búsqueda_binaria(índice,n,códigoant,posi, encontrado)

 si encontrado entonces

 nreg ← índice[posi].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,posi)

 búsqueda_binaria(índice,n,rdatos.cód, posi,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
    búsqueda_binaria(índice, n, rdatos.cód, posi, encontrado)
fin_mientras
alta_índice(índice, n, posi, 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: posi ; E cadena: codbusc ; E entero: núm)
var
  entero: i
inicio
  desde i ← n hasta posi decremento 1 hacer
    índice[i+1] ← índice[i]
  fin_desde
  índice[posi].cód ← codbusc
  índice[posi].nreg ← núm
  n ← n + 1
fin_procedimiento

procedimiento baja_índice(E/S arrínd: índice ; E/S entero: n ; E entero:
  posi)
var
  entero: i
inicio
  desde i ← posi 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: índ
inicio

```

```

abrir(índ,1,'Libínd.dat')
i ← 1
mientras no fda(índ) hacer
    leer(índ, índice[i].cód, índice[i].nreg)
    i ← i + 1
fin_mientras
n ← i - 1
cerrar(índ)
fin_procedimiento

procedimiento guardar_índice(E arrínd: índice ; E entero: n)
var
    entero: i
inicio
    crear(índ,'Libínd.dat')
    abrir(índ,e,'Libínd.dat')
    desde i ← 1 hasta n hacer
        escribir(índ,índice[i].cód, índice[i].nreg)
    fin_desde
    cerrar(índ)
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 1500. 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-Indice registro
    cadena : clave
    entero : NúmReg
  fin_registro
  array [0..MáxReg] de R-Indice : Tabla-Indice
  archivo_s de R-Farmacia : A-Secuencial
var
  A-Farmacia : AFar
  A-Secuencial : AS
  R-Farmacia : RFar
  R-Secuencial : RS
  Tabla-Indice : Tabla
  entero : n
  lógico : lleno
inicio
  crear(AFar, 'FARMACIA')
  abrir(AFar, 1/e, 'FARMACIA')
  abrir(AS, 1, 'SECUENCIAL')
  CrearIndexado(Indice,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-Indice : Indice ; 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-Indices : 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 Indice : t ; E entero : n)
```

var

entero : i

registro : R-Cabecera

entero: Ultimo

fin_registro

archivo_s de R-Indice, R-Cabecer : A-Indice

inicio

crear(A-Indice, 'INDICE')

abrir(A-Indice, e, 'INDICE')

R-Cabecera.último ← n

escribir(A-Indice, R-Cabecera)

desde i ← 1 hasta MáxReg hacer

escribir(A-Indice, t[i])

fin_registro

cerrar(A-Indice)

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 si-

guientes 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 $\neq$ 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-Indice registro
        cadena : clave
        entero : NúmReg
    fin_registro
    array [0..MáxReg] de R-Indice : Tabla-Indice
    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-Indice    : Indice
  entero          : n, p
  lógico         : lleno
inicio
  abrir(AFar, 1/e, 'FARMACIA')
  abrir(AM, 1, 'MOVIMIENTOS')
  crear(AInc, 'INCIDENCIAS')
  abrir(AInc, e, 'INCIDENCIAS')
  CargarTabla(Indice,n)
  leer(AM, RM)
  mientras no FDA(AS) hacer
    RFar ← RM
    p ← BuscarIndexado(Indice, 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
        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: Ultimo
    fin_registro
    archivo_s de R-Indice, R-Cabecer : A-Indice
inicio
    abrir(A-Indice, 1, 'INDICE')
    leer(A-Indice, R-Cabecera)
    n ← R-Cabecera.último
    leer(A-Indice, R-Indice)
    i ← 0
    mientras no FDA(A-Indice) hacer
        i ← i + 1
        t[i] ← R-Indice
        escribir(A-Indice, R-Indice)
    fin_registro
    cerrar(A-Indice)
fin_procedimiento

```

```

entero función BuscarIndexado(E Tabla-Indice : t ;
                                E R-Indice : r ; E entero : n)
// función de búsqueda binaria
fin_función

```

```

procedimiento BajaIndexado(E/S Tabla-Indice : t ;
                            E/S R-Indice : 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

```

Ordenación, búsqueda e intercalación

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 encuentra 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 ascendentemente, o cuando sea mayor si se busca en un array ordenado descendentemente.

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 ascendentemente si para todo $i < j$ se cumple siempre que $X[i] \leq X[j]$. El vector estará ordenado descendientemente 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.

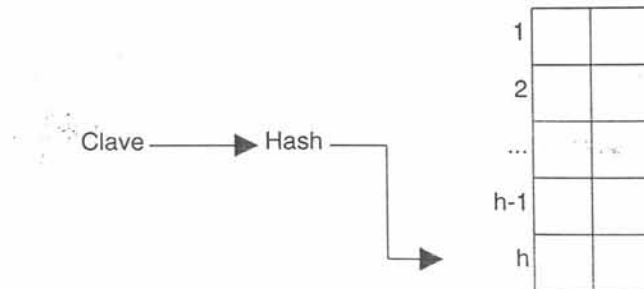
```
entero función hash(E entero: clave)
inicio
    devolver(clave)
fin_función
```

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^omatrícula = año(aa) + curso + n^o_de_alumno

donde cada curso tiene un máximo de 700 alumnos

Curso	Clave		Dirección
1º	961001	961001 menos 961000	1
	961002	961002 menos 961000	2
	...		
	961700	961700 menos 96100	700
2º	962001	962001 menos 96100 mas 300	701
	962002	9620021 menos 96100 mas 300	702

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:

```

4 1 5 -3 8
 1  4 5 -3 8
 1  4 5 -3 8
-3  4 5  1 8

```

con lo que el elemento menor queda seleccionado en a[1] y se compara a[2] con los siguientes elementos:

```

4 5 1 8
 4  5 1 8
 1  5 4 8

```

con los que el elemento menor queda seleccionado en a[2]. Se compara el elemento a[3] con los siguientes:

```

5 4 8
 4  5 8

```

con lo que el elemento menor queda en a[3]. se compara el elemento a[4] con los siguientes:

```

5 8
 5  8

```

con lo que 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:

-3 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 van tomando los siguientes valores:

```

4  1  5 -3  8
1  4  5 -3  8
1  4  5  5  8
1  4  4  5  8
1  1  4  5  8
-3 1  4  5  8

```

10.2.1.4. Inserción binaria

El algoritmo de inserción directa se mejora fácilmente. Para ello se recurre a un método de búsqueda binaria —en lugar de una búsqueda secuencial— para encontrar más rápidamente el lugar de inserción. Este método se conoce como inserción binaria.

2.1.5. Shell

Se basa en realizar comparaciones entre elementos no consecutivos, separados por saltos o intervalos mayores que 1. Estos saltos sufrirán sucesivos decrementos. Es un método elaborado que resulta más eficiente cuando las listas son grandes.

Vector a				
a[1]	a[2]	a[3]	a[4]	a[5]
4	1	5	-3	8
4	-3	5	-3	8
4	-3	5	1	8
-3	-3	5	1	8
-3	4	5	1	8
-3	4	1	1	8
-3	4	1	5	8
-3	1	1	5	8
-3	1	4	5	8

estos son los valores que van tomando los elementos del vector durante el proceso de clasificación.

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, cuando tratemos recursividad
// lo veremos de forma 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
    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_procedimiento

procedimiento ordenarburb(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 $\leftarrow$ 2 hasta N hacer

auxi $\leftarrow$ a[k]

i $\leftarrow$ k-1

encsitio $\leftarrow$ falso

mientras (no encsitio) y(i $\geq$ 1) hacer

si auxi < a[i] entonces

a[i+1] $\leftarrow$ a[i]

i $\leftarrow$ i - 1

si_no

encsitio $\leftarrow$ verdad

fin_si

fin_mientras

a[i+1] $\leftarrow$ auxi

fin_desde

fin_procedimiento

procedimientos ordenarinsbin(E/S arr: a)

var

entero: k, j, prim, últ, centr

entero: auxi

inicio

desde k $\leftarrow$ 2 hasta N hacer

auxi $\leftarrow$ a[k]

prim $\leftarrow$ 1

últ $\leftarrow$ k - 1

mientras prim $\leq$ últ hacer

centr $\leftarrow$ (prim + últ) div 2

si auxi < a[centr] entonces

últ $\leftarrow$ centr - 1

si_no

prim $\leftarrow$ centr + 1

fin_si

fin_mientras

desde j $\leftarrow$ k - 1 hasta prim decremento 1 hacer

a[j+1] $\leftarrow$ a[j]

fin_desde

a[prim] $\leftarrow$ 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(E 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(E arr: a ; E entero:n ; E 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], elemento1) 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

```

Búsqueda, ordenación y fusión externas (archivos)

11.1. Conceptos generales

Para realizar este tipo de operaciones en archivos, se puede recurrir a cargar los registros en arrays 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 quepa 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 descendientemente 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, con ligeras modificaciones debido a las operaciones de lectura y escritura de registros en el disco.

Si el archivo es de organización secuencial, es necesario emplear otros métodos, que consiguieran 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.

11.4.1.3. *Partición en secuencias de longitud N*

Consiste en partir el fichero inicial en dos, 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 archivos.

11.4.1.5. *Partición según el método de selección por sustitución*

Este método divide el archivo inicial en otros dos 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, $r_a[m]$, 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 2º 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 dos ficheros.

11.4.2. *Ordenación por mezcla natural*

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 dos copiando alternativamente en cada uno de ellos secuencias de longitud N .
3. Los dos archivos resultantes mezclan los registros en dicha secuencia N , de tal forma que se obtenga una secuencia ordenada de longitud $2 * N$. Las sucesivas mezclas de las particiones sobrescriben el archivo original.
4. La longitud de la secuencia, N , se multiplica por dos y se vuelve al 2º 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 directa*

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 dos 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 1º y 2º 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, 1, '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 conmuta 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, 1, '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. Consistirá 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.


```

    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:

F:	3	31	14	42	10	15	8	13	63	18	50
F1								F2			
	3	31	14	42	3				13	50	8 18 8

10	31	14	42	10	13	50	8	18	13
15	31	14	42	14	13	50	8	18	18
15	31	8	42	15	13	50	8	18	50
13	31	8	42	31	13	50	8	18	
13	63	8	42	42					
13	63	8	18	63					
13	50	8	18						

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, 1, '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	<table border="1"><tr><td>14</td><td>10</td><td>8</td><td>63</td><td>50</td></tr></table>	14	10	8	63	50	<table border="1"><tr><td>31</td><td>42</td><td>15</td><td>13</td><td>10</td></tr></table>	31	42	15	13	10
14	10	8	63	50								
31	42	15	13	10								

Fusión de secuencias de longitud 1

F	<table border="1"><tr><td>3</td><td>31</td><td>14</td><td>42</td><td>10</td><td>15</td><td>8</td><td>13</td><td>18</td><td>63</td><td>50</td></tr></table>	3	31	14	42	10	15	8	13	18	63	50
3	31	14	42	10	15	8	13	18	63	50		

Partición en secuencias de longitud 2

F1		F2										
3	<table border="1"><tr><td>31</td><td>10</td><td>15</td><td>18</td><td>63</td></tr></table>	31	10	15	18	63	<table border="1"><tr><td>14</td><td>42</td><td>8</td><td>13</td><td>50</td></tr></table>	14	42	8	13	50
31	10	15	18	63								
14	42	8	13	50								

Fusión de secuencias de longitud 2

F	<table border="1"><tr><td>3</td><td>14</td><td>31</td><td>42</td><td>8</td><td>10</td><td>13</td><td>15</td><td>18</td><td>50</td><td>63</td></tr></table>	3	14	31	42	8	10	13	15	18	50	63
3	14	31	42	8	10	13	15	18	50	63		

Partición en secuencias de longitud 4

F1		F2										
3	<table border="1"><tr><td>14</td><td>31</td><td>42</td><td>18</td><td>50</td><td>63</td></tr></table>	14	31	42	18	50	63	<table border="1"><tr><td>8</td><td>10</td><td>13</td><td>15</td></tr></table>	8	10	13	15
14	31	42	18	50	63							
8	10	13	15									

Fusión de secuencias de longitud 4

F	<table border="1"><tr><td>3</td><td>8</td><td>10</td><td>13</td><td>14</td><td>15</td><td>31</td><td>42</td><td>18</td><td>50</td><td>63</td></tr></table>	3	8	10	13	14	15	31	42	18	50	63
3	8	10	13	14	15	31	42	18	50	63		

Partición en secuencias de longitud 8

F1		F2										
3	<table border="1"><tr><td>8</td><td>10</td><td>13</td><td>14</td><td>15</td><td>31</td><td>42</td></tr></table>	8	10	13	14	15	31	42	<table border="1"><tr><td>18</td><td>50</td><td>63</td></tr></table>	18	50	63
8	10	13	14	15	31	42						
18	50	63										

Fusión de secuencias de longitud 8

F	<table border="1"><tr><td>3</td><td>8</td><td>10</td><td>13</td><td>14</td><td>15</td><td>18</td><td>31</td><td>42</td><td>50</td><td>63</td></tr></table>	3	8	10	13	14	15	18	31	42	50	63
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 `fd_a(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 maximales 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:

F: 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 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
        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,1,<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 FUSION 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(f1,1,'FICHA')
abrir(f2,1,'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_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, l/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

```


Estructuras dinámicas lineales de datos (listas enlazadas, pilas, colas)

12.1. Estructuras dinámicas

Hasta ahora se ha trabajado con estructuras para las que hemos reservado memoria en tiempo de compilación (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 son generadas mediante un tipo de dato conocido con el nombre de 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 eliminen 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 : pun a

```

El *<tipo_dato>* podrá ser simple o estructurado. Con los punteros se pueden hacer las siguientes operaciones:

- **Inicialización.** $p \leftarrow \text{nulo}$, a *nulo* para indicar que no apunta a ninguna variable.
- **Comparación.** $p = q$, con los operadores = ó <>.
- **Asignación.** $p \leftarrow q$, 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_puntero↑*. Las variables *nombre_variable_tipo_puntero↑*, por ejemplo *p↑*, 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 *p↑.sig* y *p↑.elemento*.

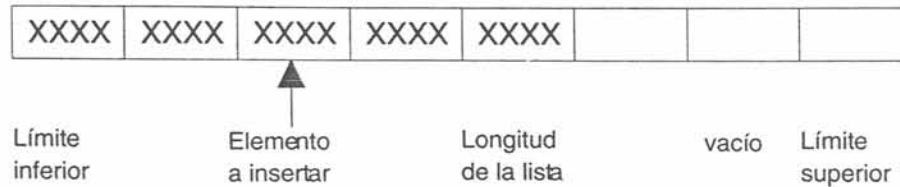
12.2. Listas

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 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.

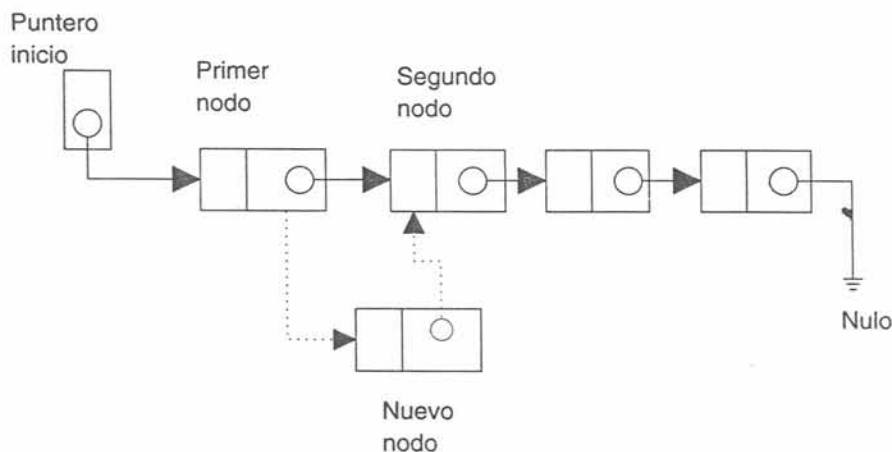


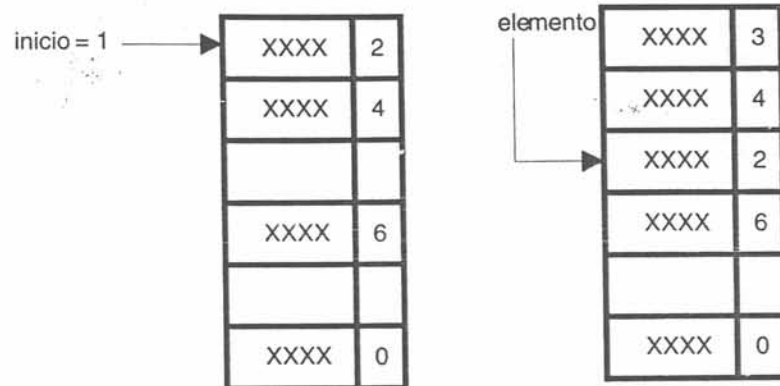
Enlazadas

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 implementado 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

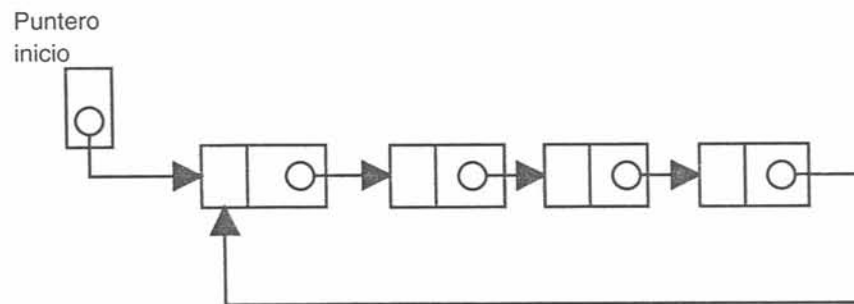


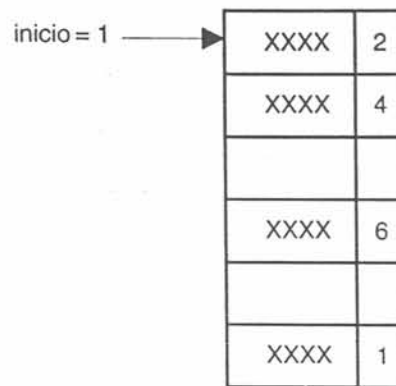
Implementación con arrays

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.

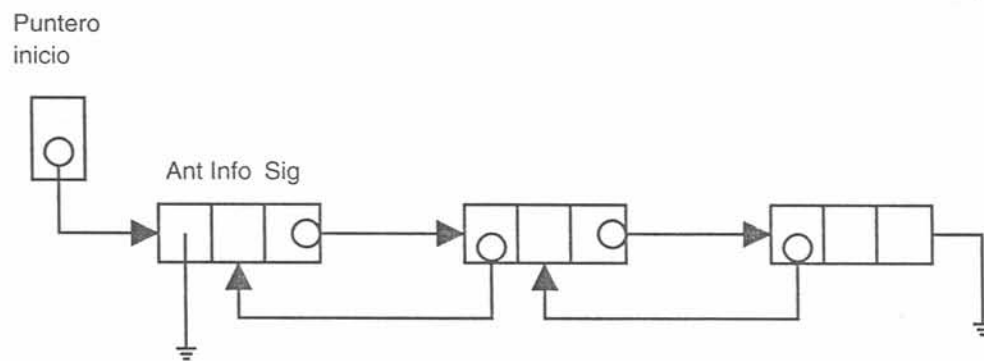


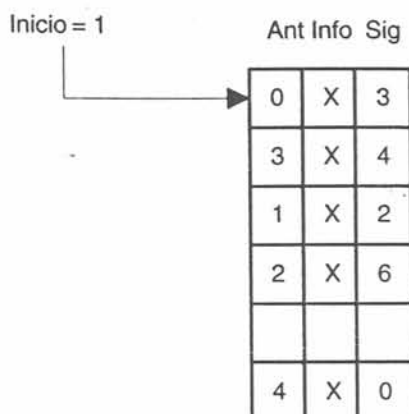


Se debe diseñar un nodo especial, *cabecera*, permanentemente asociado a la existencia de la lista y cuyo campo para almacenar información no se utiliza. Al efectuar un recorrido de la lista, el nodo *cabecera* permitirá detectar cuando han sido visitados todos los demás nodos.

Doblemente 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.





Listas doblemente encadenadas circulares

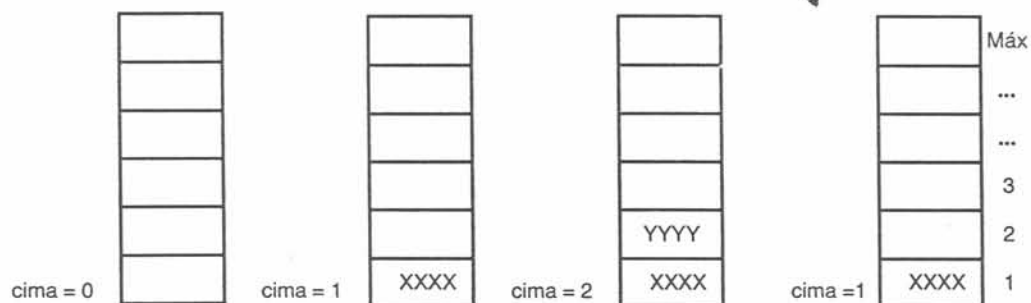
En este tipo de listas el campo *ant* del primer nodo de la lista apunta al último y el campo *sig* del último nodo al primero.

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.



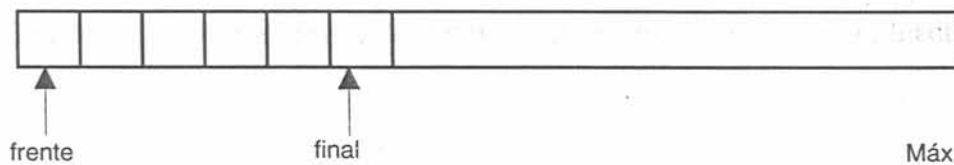
12.3.1. Aplicaciones de las pilas

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 numéricas (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 $\leftarrow$ 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 $\leftarrow$ 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.

	elemento	sig
inic = 0	1	2
vacío = 1	2	3
	3	4
	4	5
	5	6
	6	0

```

procedimiento inicializar(S entero: inic) // lista de elementos
inicio
  inic  $\leftarrow$  0
fin_procedimiento

```

```

procedimiento iniciar(S arr: m ; S entero: vacío) // lista de vacíos
var
  entero: i
inicio
  vacío  $\leftarrow$  1
  desde i  $\leftarrow$  1 hasta Máx 1 hacer
    m[i].sig  $\leftarrow$  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**.

A continuación del primero 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.

	elemento	sig
inic = 1 vacío = 2	1	XXXX 0
	2	3
	3	4
	4	5
	5	6
	6	0

	elemento	sig
inic = 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 vacío: entero)
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:

		elemento	sig
inic = 1	1	XXXX	2
vacío = 5	2	XXXX	3
	3	****	4
	4	XXXX	0
	5		6
	6		0

La posición del elemento a eliminar es la 3 ($\text{posic} \leftarrow 3$), el elemento anterior al 3 ocupa en el array la posición 2 ($\text{anterior} \leftarrow 2$), el primer vacío está en 5 ($\text{vacío} \leftarrow 5$). Al suprimir el elemento 3 la lista quedaría:

		elemento	sig
inic = 1	1	XXXX	2
vacío = 3	2	XXXX	4
	3	****	5
	4	XXXX	0
	5		6
	6		0

es decir, $m[2].\text{sig} \leftarrow 4$. Mediante el procedimiento `liberar(...)` se añade el nuevo elemento vacío en la lista de vacíos ($m[3].\text{sig} \leftarrow 5$ y $\text{vacío} \leftarrow 3$). Como el que se suprime no es el primer elemento de la lista, el valor de `inic` no varía ($\text{inic} \leftarrow 1$).

```

procedimiento liberar(E entero: posic ; E/S arr: m ; E/S entero: va-
cío)
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:

	elemento	sig
inic = 1 vacío = 3	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.

	elemento	sig
inic = 1 vacío = 5	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 ($\text{auxi} \leftarrow 3$, $\text{vacío} \leftarrow 5$ y $m[3].\text{elemento} \leftarrow \text{nuevo elemento}$). Como se desea insertar el nuevo elemento a continuación del primero de la lista, su anterior será el apuntado por **inic** ($\text{anterior} \leftarrow 1$, $m[3].\text{sig} \leftarrow 2$ y $m[1].\text{sig} \leftarrow 3$)

```

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 ascendentemente 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
        // Recurrir 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 ascendentemente 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

```

```

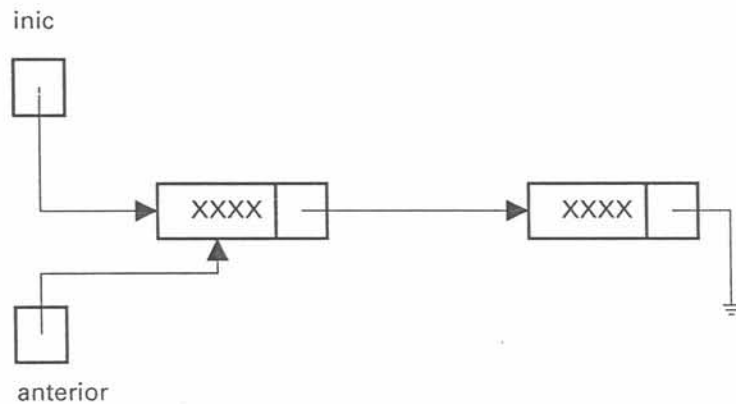
    encontrado ← falso
  fin_si
fin_procedimiento

```

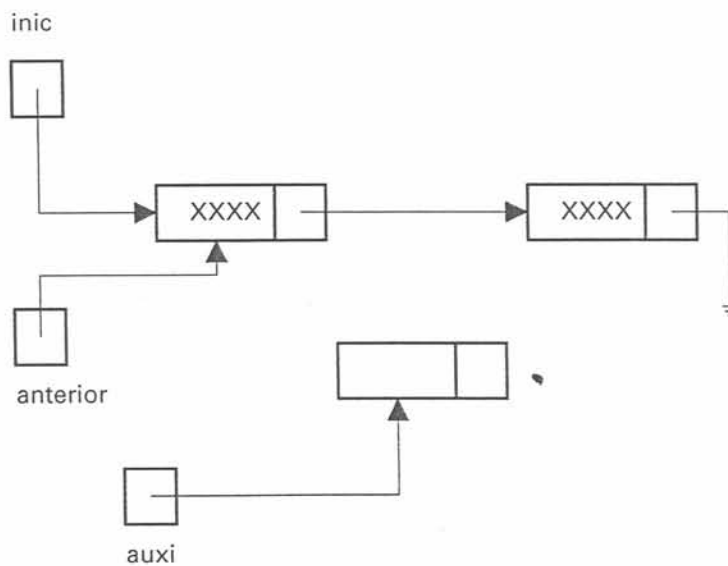
El procedimiento Insertar, inserta un nuevo elemento a continuación de anterior, si anterior fuera nulo significa que ha de insertarse al comienzo de la lista.

Insertar(inic,anterior,elemento)

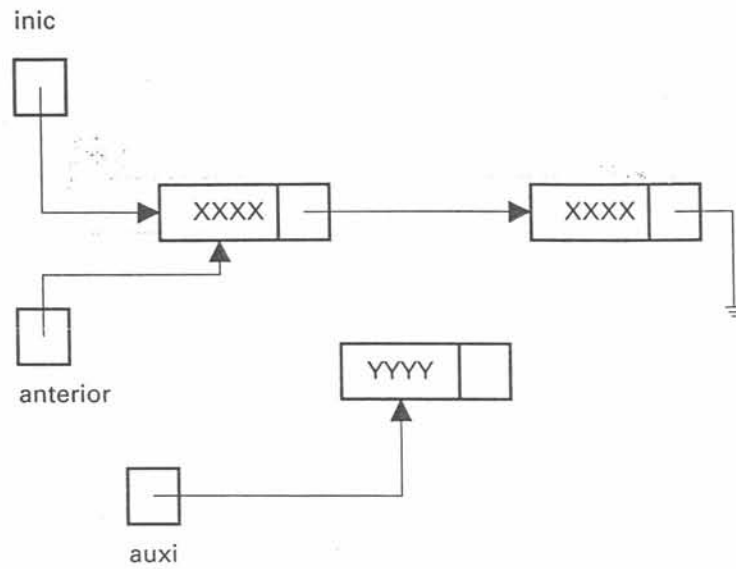
1. Situación de partida



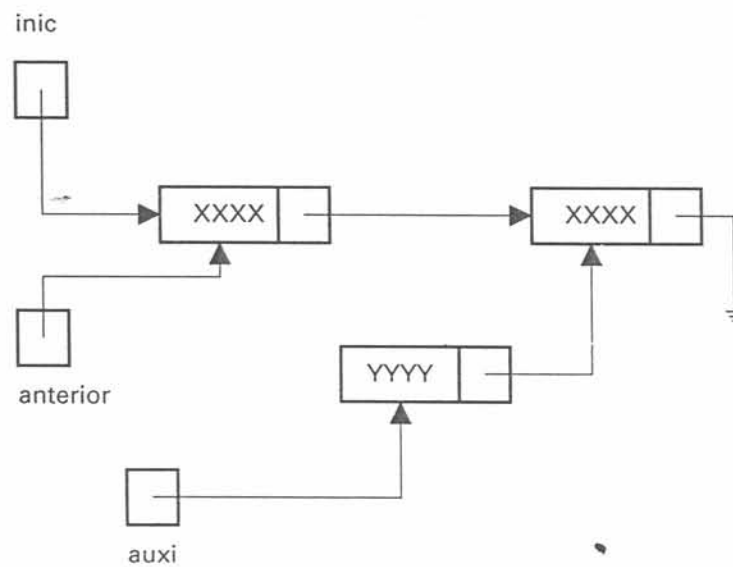
2. Reservar auxi



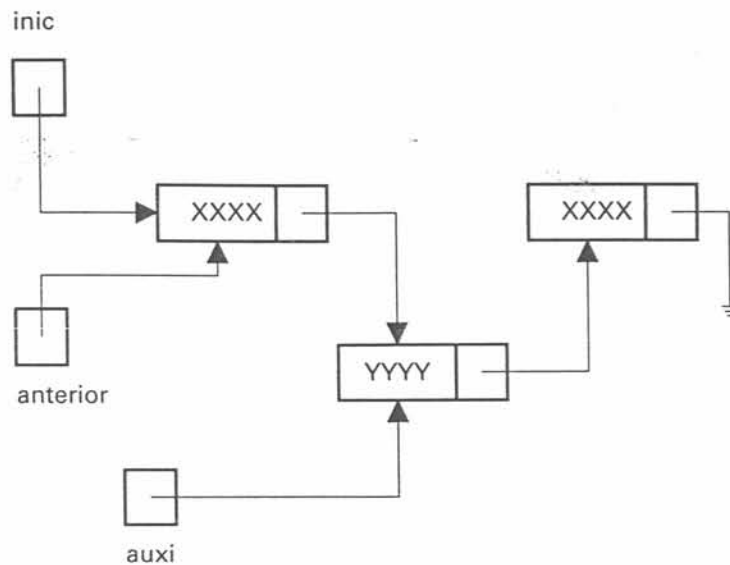
3. Introducir la nueva información en auxi↑.elemento



4. Hacer que `auxi↑.sig` apunte a donde lo hacía `anterior↑.sig`



5. Conseguir que `anterior↑.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

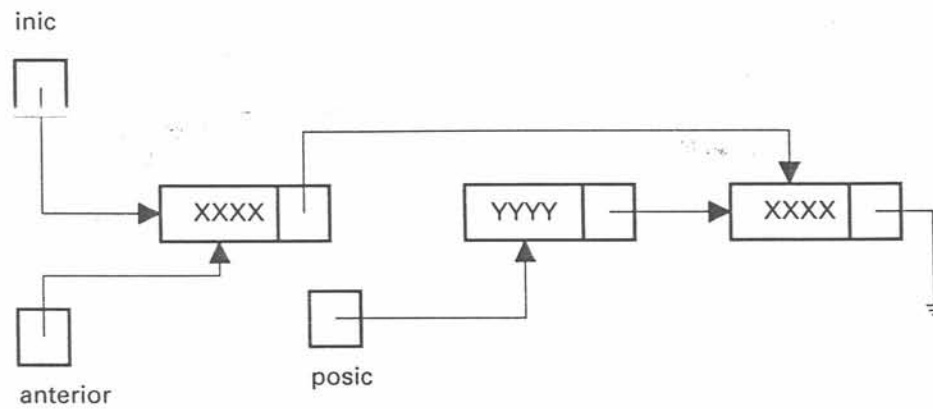
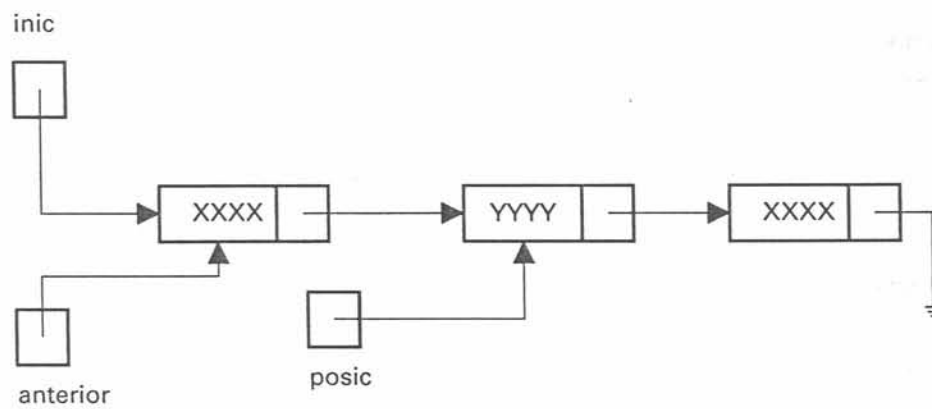
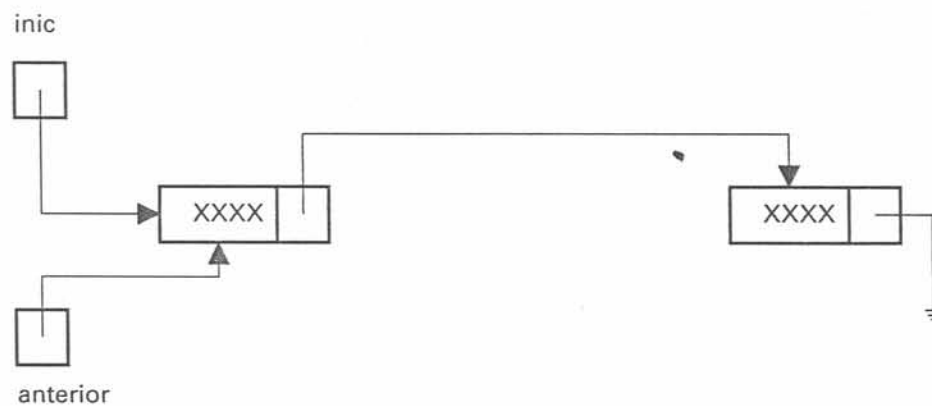
fin_si

anterior ← auxi // Opcional

fin_procedimiento

Suprimir(inic,anterior,posic)

1. Situación de partida

2. $\text{anterior} \uparrow .\text{sig}$ apunta a donde $\text{posic} \uparrow .\text{sig}$ 3. **liberar**(posic)

```

procedimiento suprimir(E/S punt: inic,anterior,posic)
inicio
    si anterior = nulo entonces
        inic ← posic↑.sig
    si_no
        anterior↑.sig ← posic↑.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↑.elemento)
        posic ← posic↑.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

```

tipo
  registro: tipo_nodo
    carácter : elemento
    entero   : sig
  fin_registro
  array[1..Máx] de tipo_nodo: lista
var
  entero: inic, inicauxi 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:vacío ; E/S lista: m)
var
    carácter: c
    entero : anterior
inicio
    Inicializar(inic)
    leercar(c)
    anterior ← 0

```

```

mientras (c<>car(13)) y no llena(vacío) hacer
    escribir(c)
    insertar( inic, anterior, c, m, vacío)
    si no llena(vacío) 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↑.sig ← inic
    inic ← auxi
  si_no

```

```
    auxi↑.sig ← anterior↑.sig
    anterior↑.sig ← 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 ← nulo
  mientras c <> 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 ← inic
  mientras posic <> nulo hacer
    escribir(posic↑.elemento)
    posic ← posic↑.sig
  fin_mientras
  escribir()
fin_procedimiento

entero función longitud_cad(E cade: inic)
var
  cade: posic
  entero: contador
inicio
  posic ← inic
  contador ← 0
  mientras posic <> nulo hacer
    contador ← contador + 1
    posic ← posic↑.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.Coeff)
    // Cuando el coeficiente sea 0 el término se anula,
    // por lo 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)
            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_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_según
        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

```

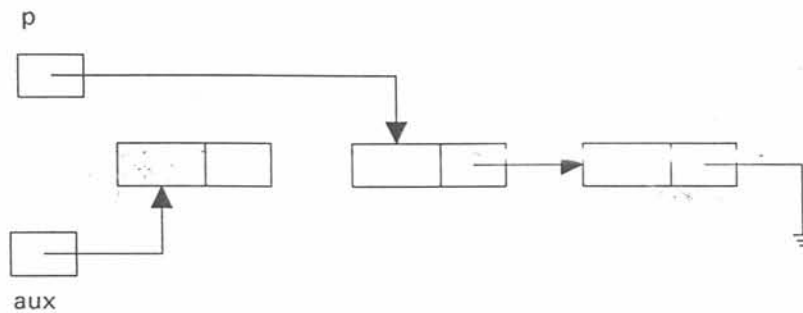
Para averiguar en este caso si la pila está vacía, lo único que se debe hacer es comprobar su valor con la constante *nulo*.

```

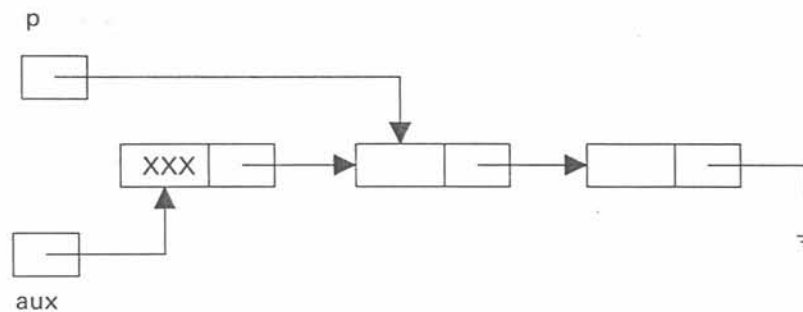
lógico función EsPilaVacía(E pila : p)
inicio
  devolver(p = nulo)
fin_función

```

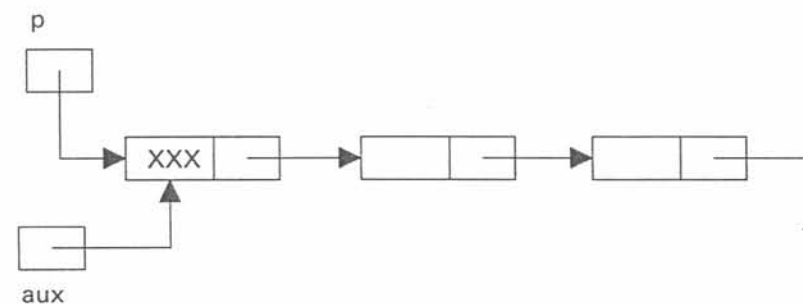
Para insertar un elemento, primero se ha de reservar espacio para un nuevo elemento mediante el procedimiento predefinido *reservar* y hacer que una variable de tipo pila apunte a dicha posición.



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 PInsertar(E/S pila : p ; E TipoElemento : e)
var
  pila : aux
inicio
  Reservar(aux)
  aux↑.sig ← p
  aux↑.info ← e
  p ← aux
fin_procedimiento
  
```

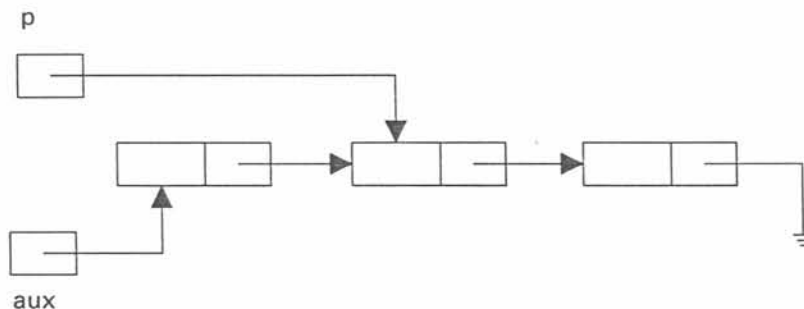
El procedimiento Tope, se limitará a tomar la información del campo info del elemento apuntado por la pila.

```

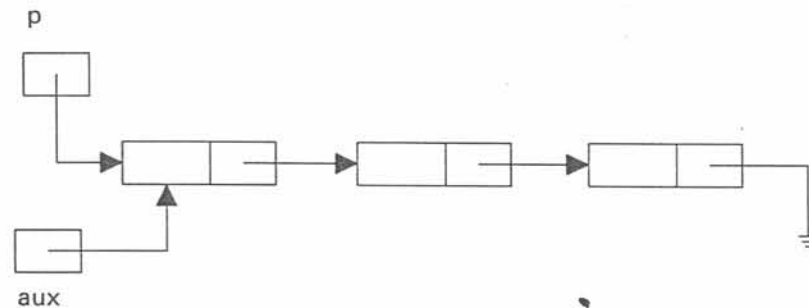
procedimiento Tope(E pila : p ; S TipoElemento : e)
inicio
  si EspilaVacía(p), entonces
    // Error, la pila está vacía
  si_no
    e ← p↑.info
  fin_si
fin_procedimiento

```

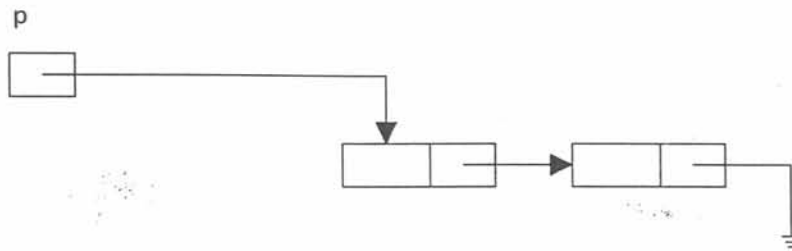
Para borrar un elemento de la pila, es necesario «puentear» el primer elemento, es decir, hacer que pila apunte al nodo al que apunta el primer elemento. Para ello se utilizará una variable auxiliar (aux) que apunte al primer nodo.



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:

ColaVacía	Inicializa una cola, crea una cola como vacía.
EsColaVacía	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 lo 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 lo 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 pila 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 lo 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

```

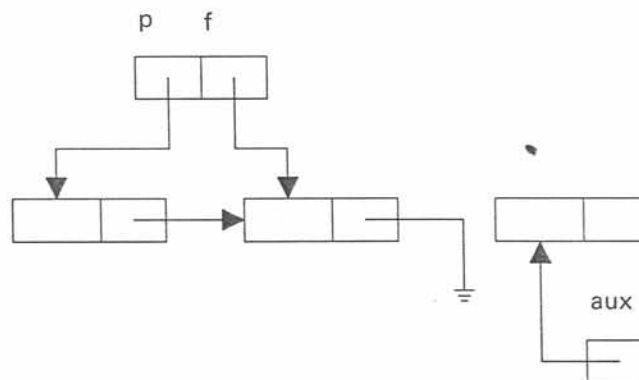
Se sabrá si la cola está vacía si el puntero **c.p** o **c.f** están a nulo.

```

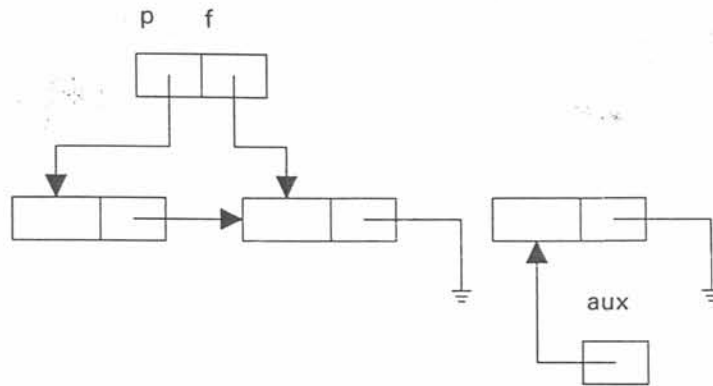
lógico función EsColaVacía(E cola : c)
inicio
  devolver(c.f = nulo)
fin_función

```

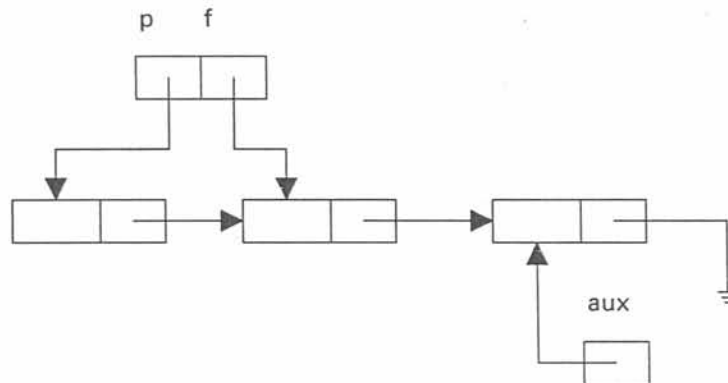
Para insertar un elemento, primero hay que reservar espacio para él.



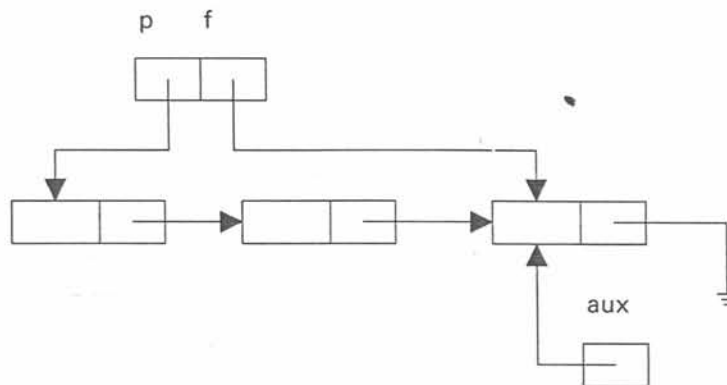
Después se introduce en el campo `info` el elemento a insertar y, como se trata del último elemento de la estructura, se pone su campo `sig` a `nulo`.



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 EsColaVacía(c) entonces
        c.p ← aux
    si_no
        c.f↑.sig ← aux
    fin_si
    c.f ← aux
fin_procedimiento

```

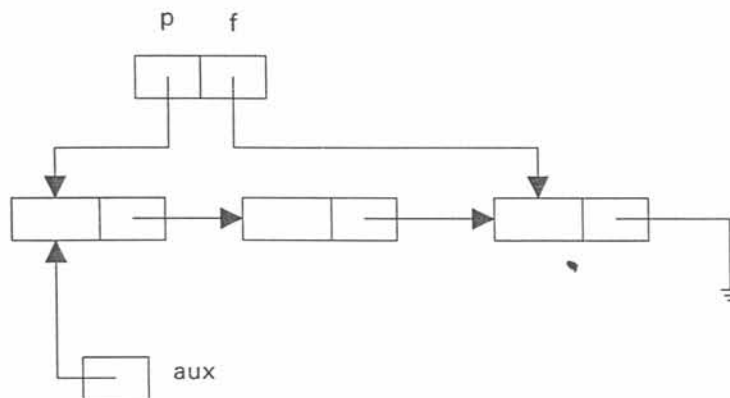
Al primer elemento se accede simplemente mediante el campo `info` del elemento apuntado por `c.p`.

```

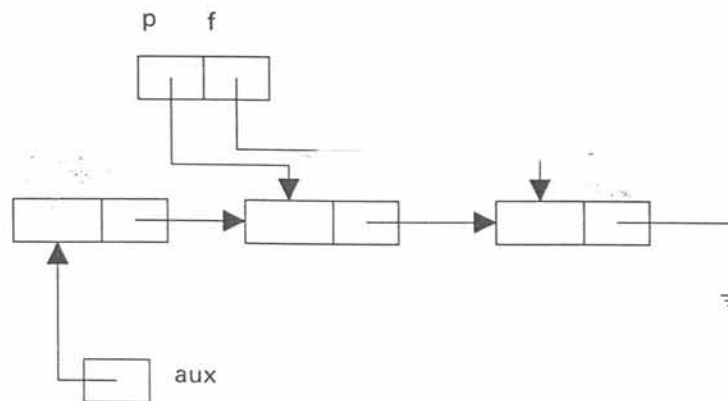
procedimiento Primero(E cola : c ; S TipoElemento : e)
inicio
    si EsColaVacía(p) entonces
        // Error, la cola está vacía
    si_no
        e ← c.p↑.info
    fin_si
fin_procedimiento

```

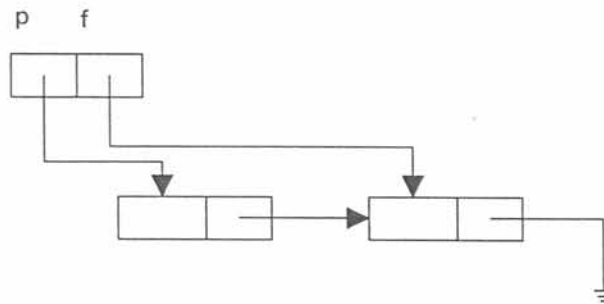
Por último, para borrar un elemento, se asigna a una variable auxiliar la dirección del nodo a borrar (el apuntado por `c.p`).



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 EsColaVacía(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 rellenar otra pila con los mismos elementos y en el mismo orden. Por lo 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 lo 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
```

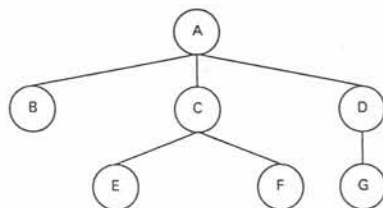
Estructuras de datos no lineales (árboles y grafos)

13.1. Estructuras no lineales

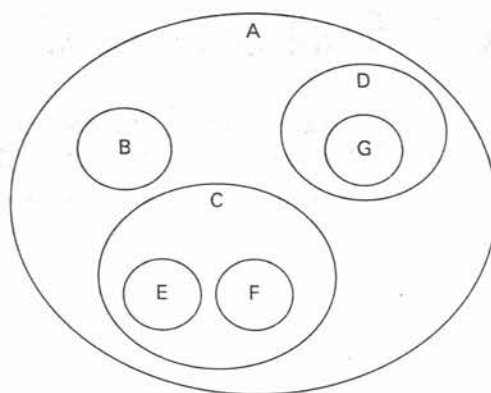
Las estructuras vistas hasta el momento son *lineales* —estáticas y dinámicas. Ahora se va a considerar la existencia de otro tipo de estructuras, *no lineales*: *árboles* y *grafos*. Estas estructuras se caracterizan por no tener una estructura secuencial: *jerárquica* (árboles) o *no jerárquica* (grafos).

13.2. 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 la figura siguiente:



(a)



(b)

Representación gráfica de árboles: (a) mediante árboles; (b) mediante conjuntos.

13.2.1. 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**.

13.2.2. Árboles binarios

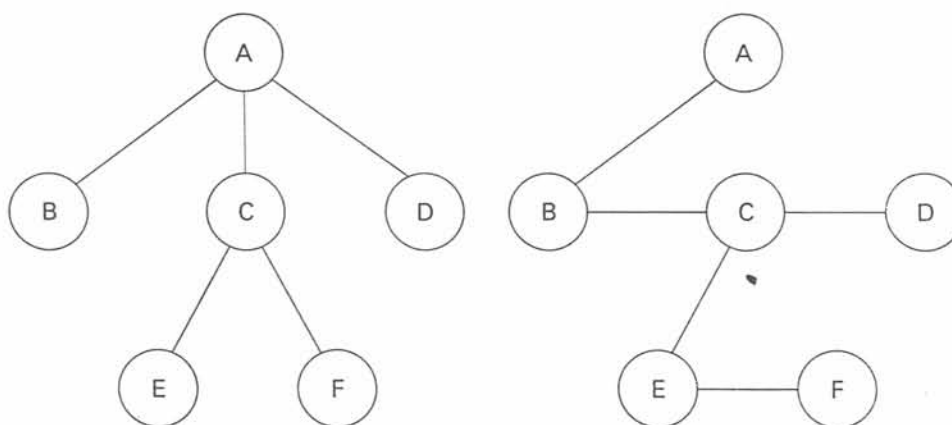
Árbol binario es aquél en el que cada nodo tiene como máximo el grado 2. Un árbol binario es *equilibrado* si para todos sus nodos la altura de sus dos subárboles se diferencia en 1 como máximo y es *completo* si todos sus nodos, excepto las hojas, tienen exactamente dos hijos. Un árbol binario en el 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 diferente entre uno y otro. Dos árboles binarios son *iguales* o *equivalentes* cuando son similares y además los nodos contienen la misma información.

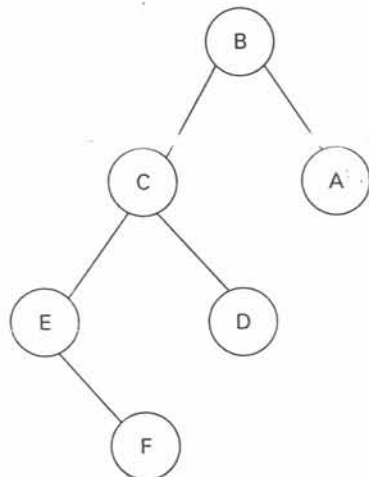
13.2.2.1. Conversión de un árbol general en árbol binario

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.



Conversión de árboles generales en árboles binarios.



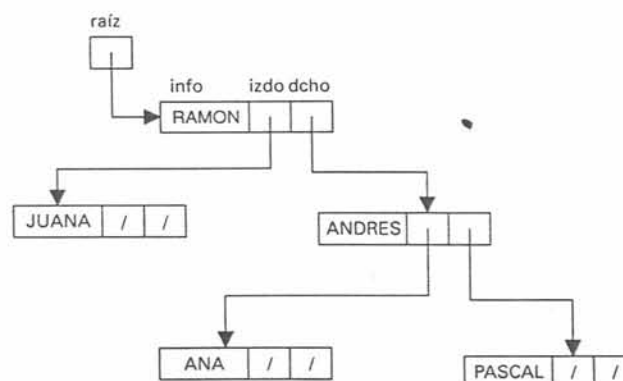
Conversión de árboles generales en árboles binarios.

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.

13.2.2.2. Implementación

Los árboles se deben implementar como estructuras dinámicas, mediante punteros, pues el número de nodos podrá aumentar o disminuir durante la ejecución del programa. No obstante, si el lenguaje no tiene punteros, se pueden utilizar 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.



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:

raíz = 3	1			
	2	PASCAL	0	0
	3	RAMON	5	9
	4			
	5	JUANA	0	0
	6			
	7	ANA	0	0
	8			
	9	ANDRES	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 numéricos 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

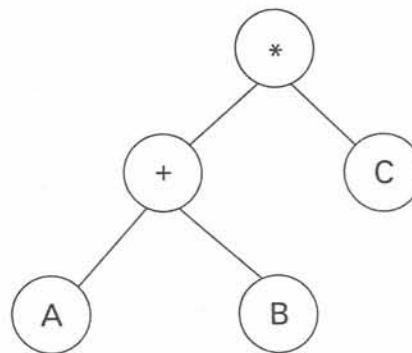
```

13.2.2.3. 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 siguiente árbol binario:



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 prece- dencia 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

$$a \ b \ + \ c \ *$$

13.2.2.4. 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.

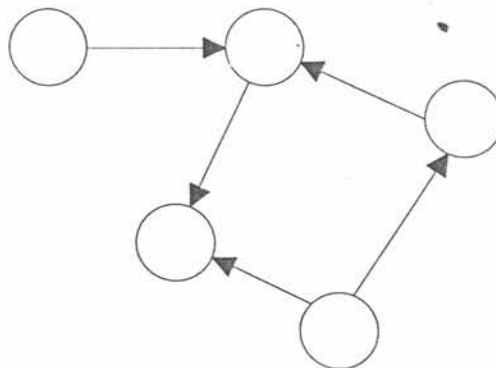
El borrado o eliminación de un elemento requerirá, una vez buscado, considerar las siguientes posibilidades:

- 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.

13.3. 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.



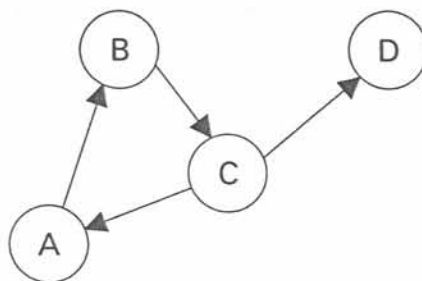
13.3.1. Terminología

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. Debemos 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.

13.3.2. Representación de los grafos

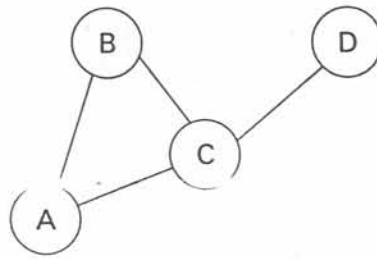
Existen dos técnicas estándar para representar un grafo: la *matriz de adyacencia* y la *lista de adyacencia*. La matriz de adyacencia consiste en un array bidimensional en el que se representan las conexiones entre pares de vértices. Por ejemplo, dado este grafo dirigido,



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	F
C	V	F	F	V
D	F	F	F	F

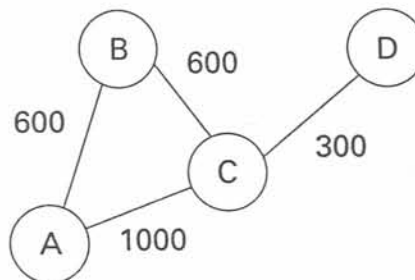
Si el grafo no fuera dirigido su matriz de adyacencia resultaría simétrica, pudiéndose ocupar sólo media matriz en el caso de que no existieran lazos (un lazo es un vértice que se apunta a sí mismo). Dado el siguiente grafo no dirigido,



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,



su matriz de adyacencia sería,

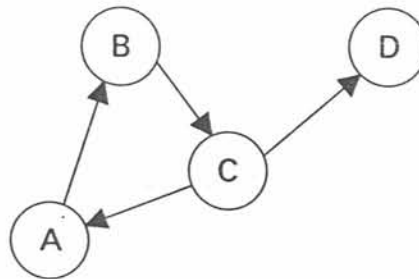
	A	B	C	D
A	0	600	1000	0
B	600	0	600	0
C	1000	600	0	300
D	0	0	300	0

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.

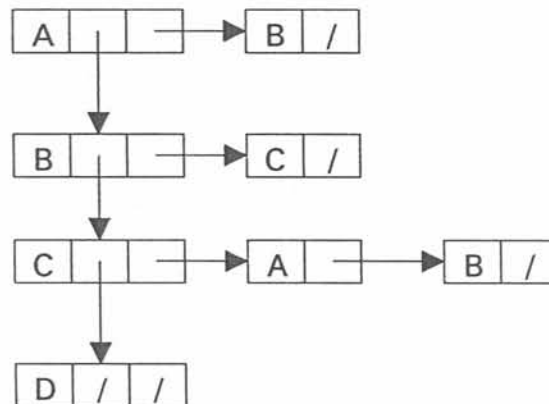
El grafo completo incluye dos partes: un directorio y un conjunto de listas enlazadas. Hay una

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 $2 \cdot A$ 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,



13.4. 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.

raíz				
	0	elemento izdo dcho		
vacío	1	1		2
		2		3
		3		4
		4		5
		5		6
		6		0

Iniciar(a, raíz, vacío)

```

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 ArbolVacío(E entero: raíz)
inicio
  si raíz=0 entonces
    devolver(verdad)
  si_no
    devolver(falso)
  fin_si
fin_función

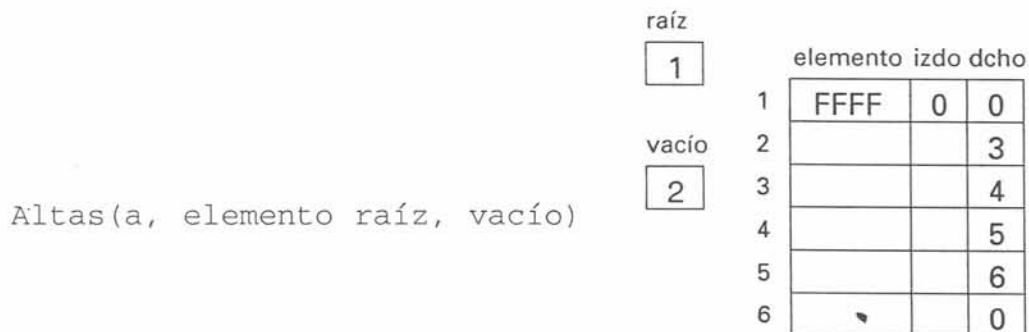
```

```

lógico función ArbolLleno(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:



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)

	elemento	izdo	dcho
raíz = 1	1 FFFF	2	0
vacio = 3	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)

	elemento	izdo	dcho
raíz = 1	1 FFFF	2	3
vacio = 4	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 = 1	1 FFFF	2	3
vacio = 5	2 CCCC	0	4
	3 JJJJ	0	0
	4 DDDD	0	0
	5		6
	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, elemento, raíz, 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_procedimiento

```

Considérese, con la siguiente situación de partida, que se desea eliminar el elemento CCCC.

	elemento	izdo	dcho
raíz = 1	1	FFFF	2 3
vacío = 5	2	CCCC	0 4
	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, elemento, raíz, 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
    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

```

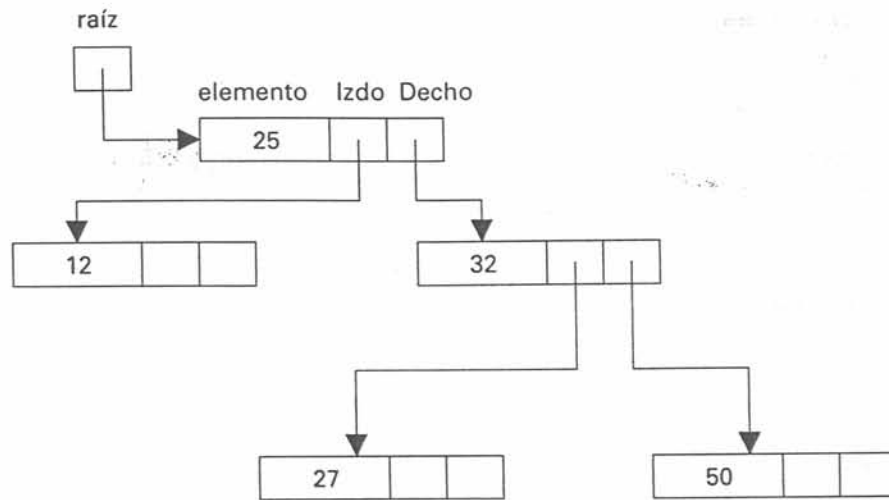
tipo
  puntero_a nodo: punt
  registro : tipo_elemento
  ... : ...
  ... : ...
  fin_registro
  registro : nodo
  tipo_elemento : elemento
  punt : izdo,dcho
  fin_registro
var
  raíz : punt
  tipo_elemento : elemento

```

Procedimientos y funciones

Altas(raíz,elemento)

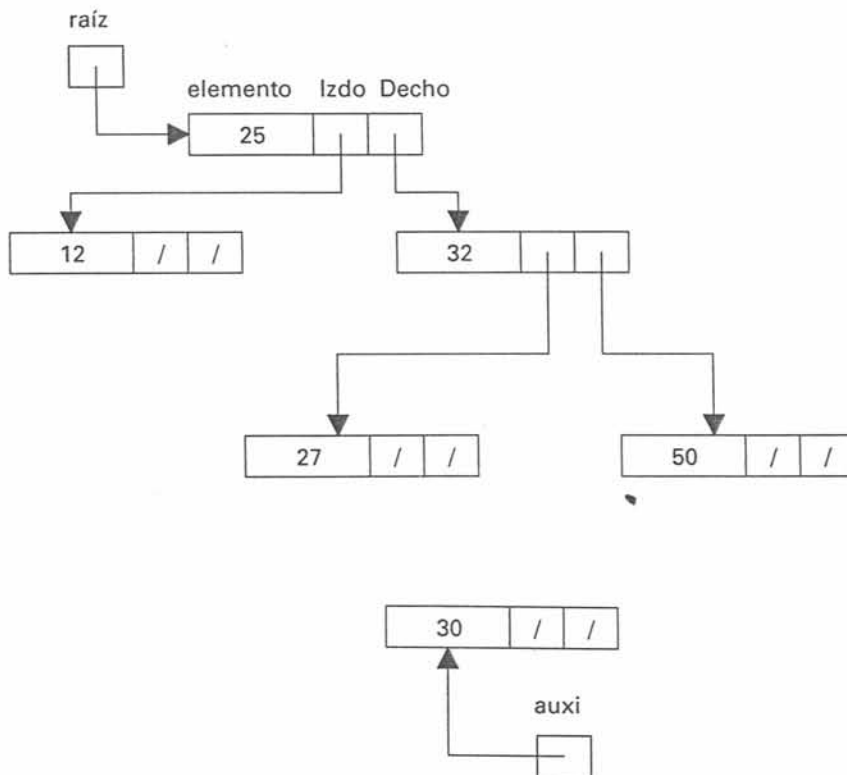
1. Situación de partida



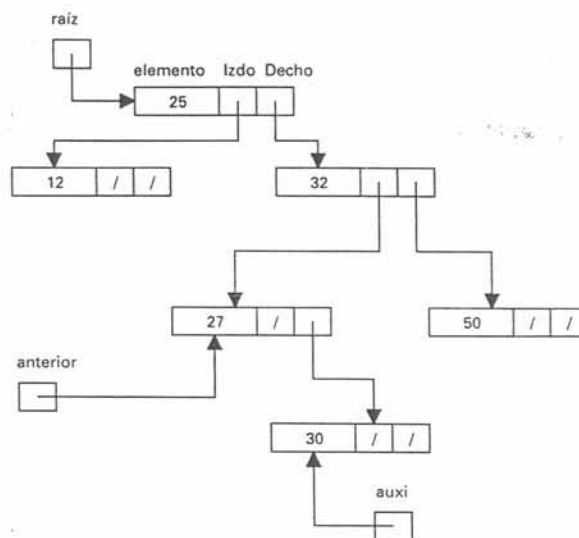
2. reservar(auxí)

```

auxí↑.elemento ← elemento
auxí↑.Izdo ← nulo
auxí↑.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

```

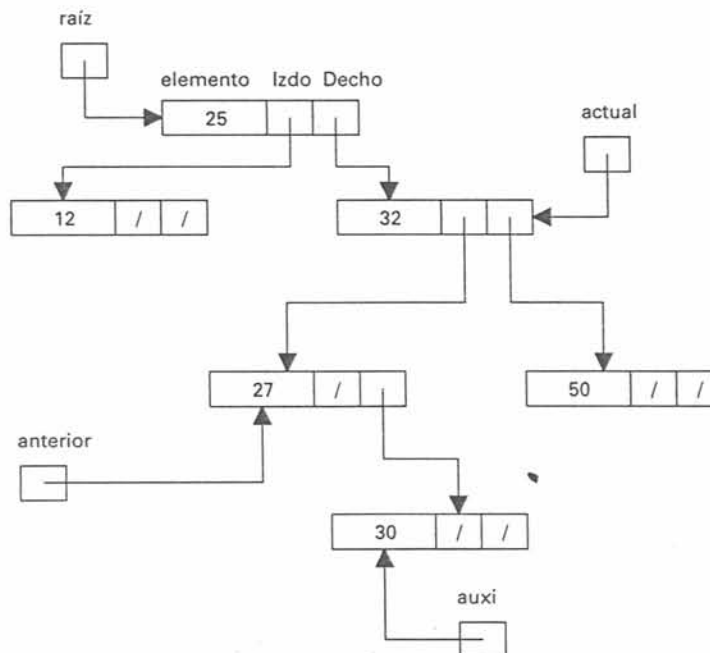
```

    escribir('Elemento duplicado')
si_no
    reservar(auxí)
    auxí↑.Elemento ← elemento
    auxí↑.Izdo ← nulo
    auxí↑.Dcho ← nulo
    si anterior=nulo entonces
        raíz ← auxí
    si_no
        si mayor(anterior↑.Elemento, elemento) entonces
            anterior↑.Izdo ← auxí
        si_no
            anterior↑.Dcho ← auxí
        fin_si
    fin_si
fin_si
fin_procedimiento

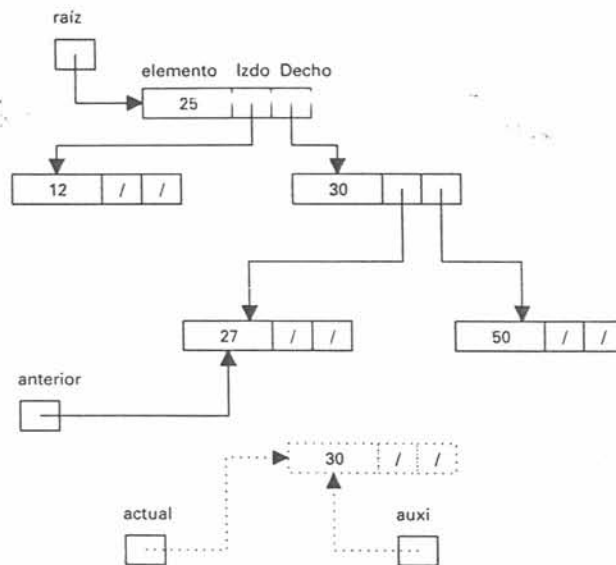
```

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
    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)`, `distinto(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 árbollleno(vacío) entonces
      llamar_a leerelemento (elemento)
    fin_si
    mientras distinto(elemento,'0') y no árbollleno(vacío) hacer
      altas (a, elemento, raíz, vacío)
      si no árbollleno(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 árbollleno(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 escribir elemento(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 tipo elemento: 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 tipo elemento: elemento ;
                    E/S entero: raíz, vacío)
```

```
var
```

```
  entero: act, ant, auxi
```

```
inicio
```

```
  si no árbol lleno(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
  cont:entero
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(E 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

```

Tablas de decisión

14.1. Definición y estructura

Se denomina tabla de decisión a una representación tabular de la lógica de un problema, en la que se presentan variadas situaciones y diferentes alternativas para cada una de ellas. Las tablas de decisión sirven como elemento de comunicación entre analistas, programadores y usuarios.

En una tabla de decisión se pueden distinguir 4 zonas

Lista de condiciones	Entrada de condiciones
Lista de acciones	Salida de acciones

Lista de condiciones	Columna donde figuran las condiciones que intervienen en el problema. Dichas condiciones podrán colocarse en cualquier orden.
Lista de acciones	Columna en la que aparecen las acciones a realizar en el orden en que se tengan que ejecutar.
Entrada de condiciones	Es una matriz con tantas filas como condiciones y tantas columnas como situaciones distintas se pueden presentar.
Salida de acciones	Matriz con tantas filas como acciones y tantas columnas como situaciones distintas se pueden presentar.

Condición 1	N	N	N	N	N	N	N	N	S	S	S	S	S	S	S	S
Condición 2	N	N	N	N	S	S	S	S	N	N	N	N	S	S	S	S
Condición 3	N	N	S	S	N	N	S	S	N	N	S	S	N	N	S	S
Condición 4	N	S	N	S	N	S	N	S	N	S	N	S	N	S	N	S
Acción 1	X		X		X							X	X	X		
Acción 2			X	X		X	X				X				X	X
Acción 3				X	X					X					X	
Acción 4		X				X			X				X	X		
Acción 5								X				X				X

- Cada columna de la entrada de condiciones se denomina **situación** y sus celdas tendrán los valores S ó N.
- Cada columna de la salida de acciones se denomina **tratamiento** y las acciones que la componen se marcan con X.
- Podremos definir **regla de decisión** como el conjunto formado por una situación y su tratamiento. Cada regla de decisión equivale a una ruta o camino en un diagrama de flujo.

Dado un número N de condiciones el número de reglas de decisión para cubrir todos los casos posibles sin repeticiones ni omisiones sería 2^n , ya que cada condición podrá tomar los valores verdad o falso (S/N).

Las tablas cuyo número de reglas cubren todos los casos posibles se denominan **propias**, pero puede ocurrir que varias combinaciones de condiciones den lugar a las mismas acciones y la tabla se pueda simplificar dando lugar a una **tabla reducida**. En una tabla reducida existirán reglas en las que una o más condiciones no precisen ser evaluadas. Diremos entonces que la situación tiene una **indiferencia** en esa condición y se representará mediante un guión. La regla correspondiente se denominará **compuesta**.

	R1	R2	R3	R4
Condición 1	N	N	S	S
Condición 2	N	S	N	S
Acción 1				X
Acción 2	X	X		
Acción 3	X	X		
Acción 4			X	X
Acción 5			X	

	R1-2	R3	R4
Condición 1	N	S	S
Condición 2	-	N	S
Acción 1			X
Acción 2	X		
Acción 3	X		
Acción 4		X	X
Acción 5		X	

Una regla compuesta es equivalente a 2^n de indiferencias reglas simples.

14.2. Tipos de tablas de decisión

Las tablas de decisión se pueden clasificar atendiendo a dos criterios. Según los tipos de entradas de condición tenemos:

- limitadas** Son aquellas en las que la evaluación de todas las condiciones está limitada a dos posibles valores: SI, NO o la reducción de ambas, el guión (-)
- ampliadas** Son aquellas tablas en donde las distintas condiciones pueden tener un rango de alternativas superiores a dos valores. Por ejemplo una tabla de decisión que refleje los distintos precios a los que se pagará la hora extraordinaria a los empleados de una empresa según su categoría (A | B | C) y la sección a la que pertenecen (COMPRAS | ALMACEN | VENTAS | ADMINISTRACION).

SECCION	VENTAS			COMPRAS			ALAMACEN			ADMINISTR.		
CATEGORIA	A	B	C	A	B	C	A	B	C	A	B	C

- mixtas** Se denominan así las tablas que resultan de la combinación de las tablas vistas en los párrafos anteriores.

La comparación entre estos tipos nos lleva a las siguientes conclusiones:

- Las TD (Tabla de decisión) de entradas ampliadas son más legibles que las TD limitadas por ser menos voluminosas que sus equivalentes. La razón reside en el hecho de que una condición de una tabla de entradas ampliadas corresponde a varias condiciones de la tabla de entradas limitadas equivalente. Para el ejemplo anterior las condiciones a considerar en la tabla de entradas limitadas correspondiente serían:

- | | |
|-----------------------------|------------------|
| 1) SECCION = VENTAS | 5) CATEGORIA = A |
| 2) SECCION = COMPRAS | 6) CATEGORIA = B |
| 3) SECCION = ALMACEN | 7) CATEGORIA = C |
| 4) SECCION = ADMINISTRACION | |

- Las TD de entradas limitadas son más fáciles de codificar en programa (las reglas de decisión constituyen caminos de un diagrama de flujo) que las tablas ampliadas.

Para convertir una TD de entradas ampliadas en una TD de entradas limitadas, será necesario ejecutar las siguientes tareas:

- Formular tantas condiciones binarias (S/N) como alternativas consideradas existan para cada condición de la tabla de entrada ampliada.
- Formular tantas acciones binarias (S/N) como posibilidades consideradas en las reglas para cada una de las acciones de la tabla de entrada ampliada.

Cuando un problema es complejo se divide en problemas secundarios de manera que cada uno de ellos de lugar a la creación de una tabla particular. Posteriormente las diferentes tablas se encadenan o enlazan para la solución del problema global. Atendiendo a este criterio, las tablas pueden ser:

abiertas Sin retorno a la tabla que las llamó.

cerradas Devuelven el control a la tabla que las llamó

14.3. Tipos de reglas de decisión

Las reglas básicas que se siguen en la construcción de una tabla de decisión son:

- | | |
|----------------------------|---|
| Reglas Y (AND) | En cada situación se necesita que se satisfaga la primera condición Y la segunda Y la tercera, ... (operador lógico AND). |
| Reglas O (OR) | Para que se satisfaga una regla, basta con que se cumpla cualquiera de las condiciones de la situación (operador lógico OR). |
| Reglas ELSE (si_no) | Entre las diversas situaciones posibles, sólo algunas de ellas conducen a tratamientos diversos, mientras que el resto de las reglas conducen a un único tratamiento. Las reglas que conducen a un tratamiento único se agrupan en una sola regla denominada ELSE (si_no o en otro caso). Se suele colocar al final de la TD sin evaluar sus condiciones. |

Las reglas de decisión con las que se viene son reglas AND. Cuando en el análisis de un problema se plantea una regla O hay que tener en cuenta que una regla O se transforma en n reglas Y, siendo n el número de condiciones que intervienen en la regla (las n reglas se suponen con indiferencias, ya que sin ellas se tendría $2^n - 1$).

Supóngase la siguiente regla O de una TD de cuatro condiciones.

- si no se cumple la primera condición;
 o si se cumple la segunda;
 o si no se cumple la tercera;

o si se cumple la cuarta.

Esta regla OR equivale a las siguientes cuatro reglas AND

R1	R2	R3	R4
N	S	S	S
-	S	N	N
-	-	N	S
-	-	-	S

14.4. Escritura de una tabla de decisión

La escritura de una tabla de decisiones requiere las siguientes etapas:

1. Listar las condiciones.
2. Listar las acciones en el orden en el que se deben realizar.
3. Escribir todas las situaciones posibles. Para n condiciones 2^n .
4. Marcar con una cruz, aspa, las acciones que deben ejecutarse en cada situación.

14.5. Transformación de una tabla en diagrama de flujo

La conversión de una tabla de decisión en un diagrama de flujo requiere tener presente las siguientes consideraciones y reglas prácticas:

- Comprobar que está completa, no tiene redundancias ni contradicciones. Completa quiere decir que considera todas las situaciones posibles. Sin redundancias implica que no existen reglas repetidas. Sin contradicciones significa que no aparecen las mismas situaciones con tratamientos diferentes.
- Simplificar la tabla agrupando las reglas simples en compuestas. Dos reglas pueden simplificarse siempre que se produzcan las mismas acciones y la situación se diferencie sólo en el valor de una condición. Podremos repetir el proceso y agrupar reglas compuestas en otra compuesta con más indiferencias. Aunque, cuando son más de dos reglas las que contienen acciones iguales podremos aplicar el «TEOREMA DEL PARAGUAS», denominado así por el tipo de dibujo necesario para la obtención de la simplificación: «Se unen mediante un arco aquellas reglas en que sólo cambia una condición. Si todos los arcos forman un circuito cerrado, se puede obtener una regla compuesta que sustituya a las reglas simples».

- Aplicar la técnica de Pollack, que consiste en:
 1. Escribir debajo de cada regla compuesta a cuántas reglas simples equivale. Este número se da como peso a las indiferencias de dicha regla.
 2. A cada condición se le calcula un coeficiente, para lo que se recorre la condición buscando las indiferencias que aparecen en ella y se van sumando los pesos de cada una de ellas. El coeficiente mide el número de reglas en que la condición en estudio no interviene (su valor será tanto más importante cuanto menor sea su valor).
 3. Se evalúa en primer lugar la condición cuyo coeficiente sea menor. Si existen varias condiciones con una cuenta mínima igual, se elige un nuevo valor (coeficiente) que es el que mide en valor absoluto la diferencia entre el número de SI y NO que aparecen en dichas condiciones, eligiéndose el de mayor valor absoluto. Si subsiste la igualdad, se elige indiferentemente cualquiera de las condiciones.
 4. Se repite el proceso (pasos A a C) hasta la última condición.

14.6. Ejercicios resueltos

14.1. Diseñar una tabla de decisión y su diagrama de flujo correspondiente que permita calcular el salario total a pagar a un empleado de una empresa de acuerdo a las siguientes proposiciones:

- Si el empleado está autorizado a trabajar horas extras y ha trabajado más de 40 horas semanales se le debe pagar el salario normal y el importe de las horas extraordinarias.
- Si no ha trabajado más de 40 horas, sólo se le pagará el salario normal.
- Un empleado no autorizado no puede trabajar más de 40 horas semanales.

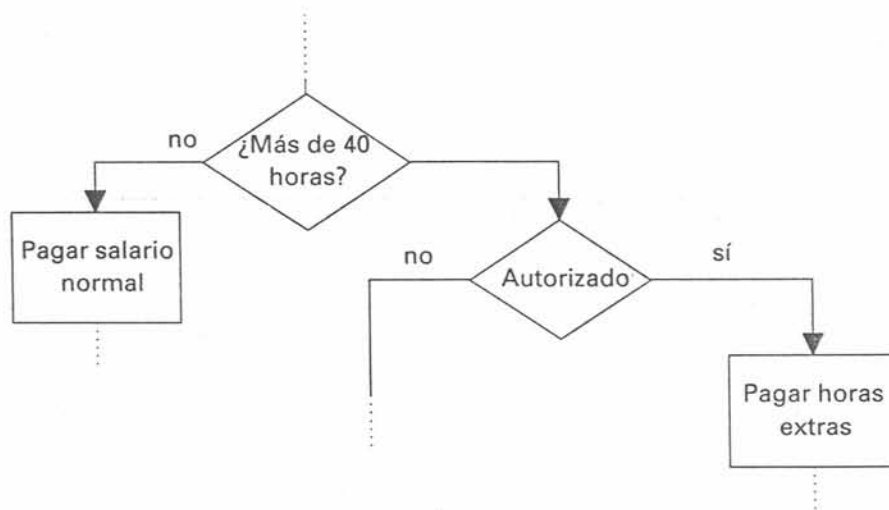
La tabla de decisión que resuelve el problema anterior es:

	R1	R2	R3	R4
Empleado autorizado	N	N	S	S
Trabaja más de 40 horas	N	S	N	S
Pagar salario normal	X		X	X
Pagar horas extras				X

Como es imposible que un trabajador no autorizado trabaje más de 40 horas, la regla 2 es imposible. Además, la regla 1 y la 3 realizan las mismas acciones: siempre que un empleado no trabaje más de 40 horas, sólo cobrará el salario normal. Por lo tanto, la tabla de decisión simplificada quedaría de la siguiente forma:

	R1-2	R4
Empleado autorizado	N	S
Trabaja más de 40 horas	-	S
Pagar salario normal	X	X
Pagar horas extras		X

A la hora de diseñar el diagrama de flujo, se debe comenzar a preguntar por la condición que más peso tenga, es decir, por la que intervenga en menos reglas. En este caso sería la primera, que no tiene ninguna indiferencia. Por lo tanto el diagrama de flujo sería el siguiente:



14.2. Un mecanismo de seguridad dispone de cuatro interruptores P, Q, R y S que funcionan de la siguiente manera:

- Si el interruptor P está pulsado, el mecanismo funciona con independencia de los restantes interruptores.
- Si el interruptor Q está pulsado, funciona siempre que también esté pulsado el interruptor S.
- Si el mando R está pulsado funciona si también está pulsado S.
- Si el mando S está pulsado, funciona si también están pulsados R y Q.

Diseñar la tabla de decisión y el diagrama de flujo correspondiente que pueda resolver este problema.

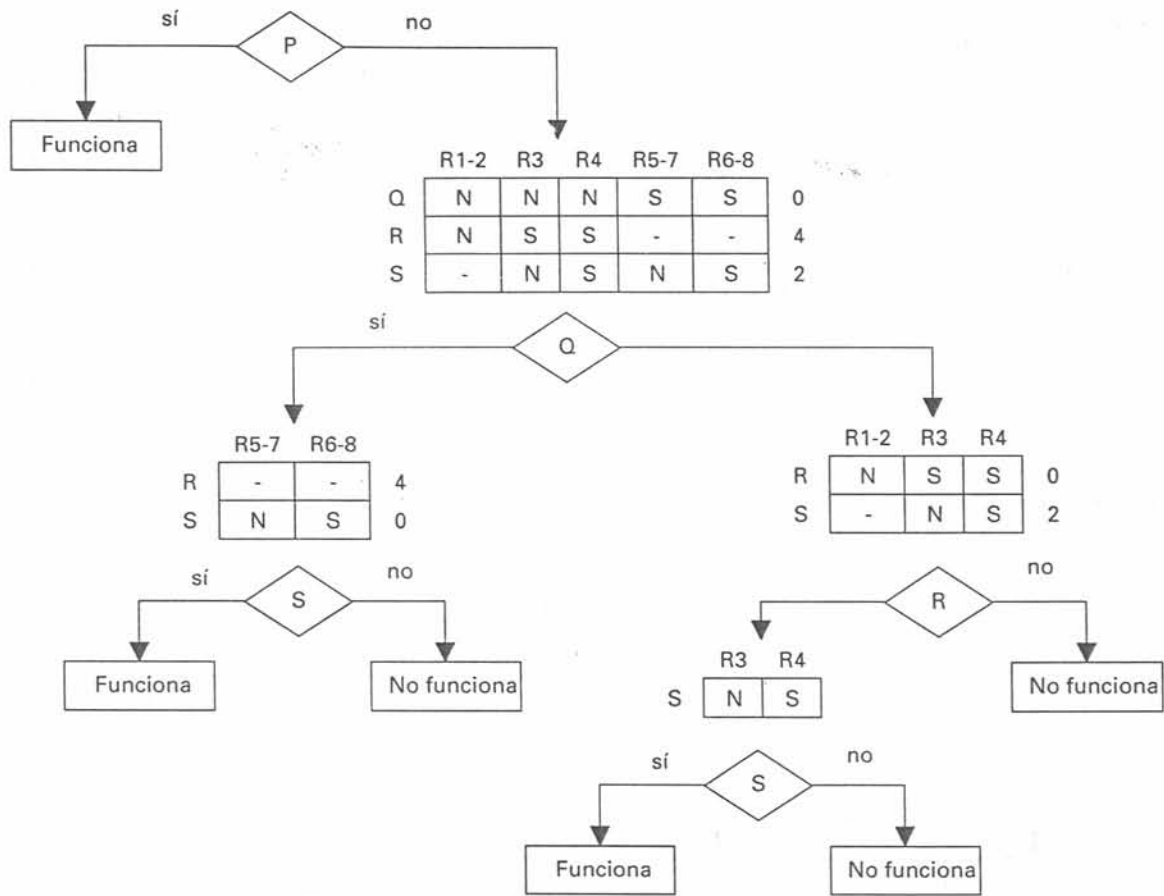
La tabla de decisión completa sería la siguiente:

	R1	R2	R3	R4	R5	R6	R7	R8	R9	R1	R1	R1	R1	R1	R1	R1
P	N	N	N	N	N	N	N	N	S	S	S	S	S	S	S	S
Q	N	N	N	N	S	S	S	S	N	N	N	N	S	S	S	S
R	N	N	S	S	N	N	S	S	N	N	S	S	N	N	S	S
S	N	S	N	S	N	S	N	S	N	S	N	S	N	S	N	S
Funciona				X		X		X	X	X	X	X	X	X	X	X
No funciona	X	X	X		X		X									

La tabla de decisión simplificada con el peso de cada regla y los coeficientes de cada condición serían los siguientes:

	R1-2	R3	R4	R5-7	R6-8	R9-10-11-12-13-14-15-16	
P	N	N	N	N	N	S	0
Q	N	N	N	S	S	-	8
R	N	S	S	-	-	-	12
S	-	N	S	N	S	-	12
Funciona			X		X	X	
No funciona	X	X		X			
	2	1	1	2	2	8	

con lo que queda claro que se debe comenzar por la condición P.



14.3. Una compañía de seguros de automóviles desea realizar un programa que emita los recibos de las primas de sus asegurados. El importe del recibo se calculará a partir de la prima base, a la que se harán modificaciones según el siguiente baremo:

- A todos los asegurados menores de 27 años, se les incrementará la prima en un 15%.
- Si un asegurado no ha tenido ningún siniestro se le aplicará un descuento del 20%.
- Si ha tenido más de dos siniestros, la prima se incrementará en un 5%.
- Si ha tenido más de 5 siniestros, la prima se incrementará en un 10%.
- Cualquier otra situación no modificará el importe de la prima base.

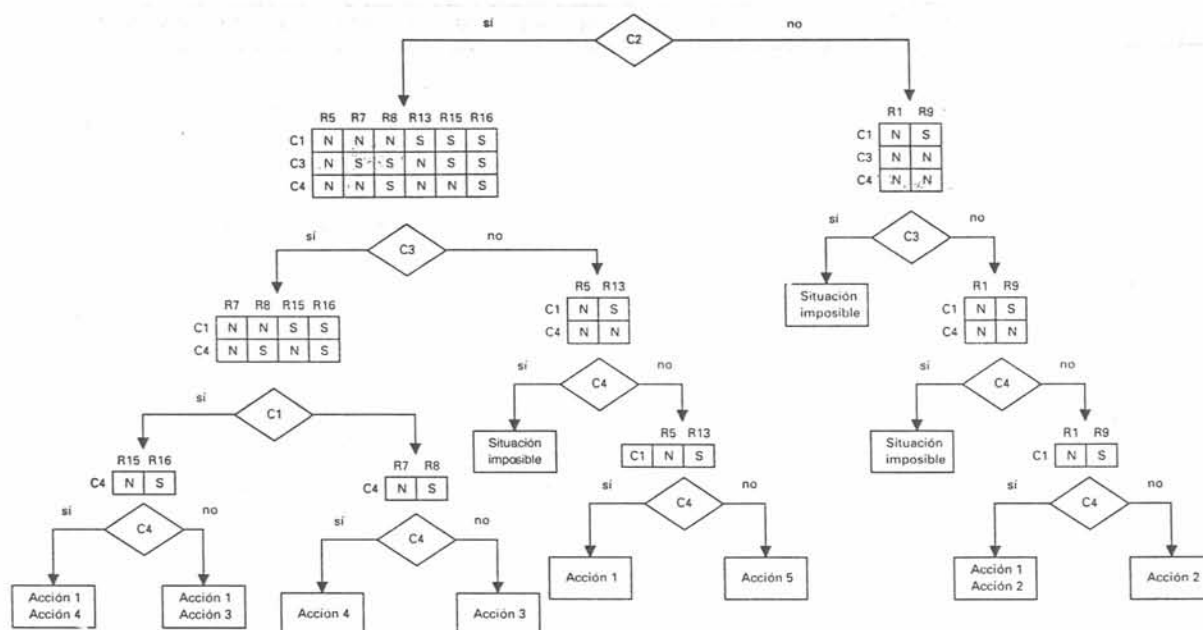
Mediante una tabla de decisión, diseñar el diagrama de flujo correspondiente para solucionar el problema.

	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12	R13	R14	R15	R16
Mayor de 27 años	N	N	N	N	N	N	N	N	S	S	S	S	S	S	S	S
No tiene siniestros	N	N	N	N	S	S	S	S	N	N	N	N	S	S	S	S
Tiene más de 2 siniestros	N	N	S	S	N	N	S	S	N	N	S	S	N	N	S	S
Tiene más de 5 siniestros	N	S	N	S	N	S	N	S	N	S	N	S	N	S	N	S
Incremento del 15%									X				X		X	X
Descuento de 20%	X								X							
Incremento del 5%							X								X	
Incremento de 10%								X								X
No modifica la base					X											

Como se observa en la tabla de decisión anterior, las reglas 2, 3, 4, 6, 10, 11, 12 y 14 son imposibles, por lo que se eliminan de la tabla.

	R1	R5	R7	R8	R9	R13	R15	R16
Mayor de 27 años	N	N	N	N	S	S	S	S
No tiene siniestros	N	S	S	S	N	S	S	S
Tiene más de 2 siniestros	N	N	S	S	N	N	S	S
Tiene más de 5 siniestros	N	N	N	S	N	N	N	S
Incremento del 15%					X	X	X	X
Descuento de 20%	X				X			
Incremento del 5%			X				X	
Incremento de 10%				X				X
No modifica la base		X						

La tabla no tiene indiferencias, por lo que todas las condiciones tienen el mismo coeficiente. Cuando ocurre esto, se selecciona la condición en la que el valor absoluto de la diferencia entre condiciones verdaderas y condiciones falsas sea mayor. En este caso hay dos: la segunda y la cuarta, por lo que empezaremos a preguntar por cualquiera de ellas.



4.4. Una universidad desea informatizar la nómina de sus profesores a partir de su sueldo base mediante el siguiente criterio:

- Si el profesor es tutor y da clases en el ciclo superior, el sueldo base se incrementará un 3%.
- Si es tutor y encargado de asignatura, el incremento será del 5%.
- Si es tutor, no es encargado de asignatura, pero tiene dedicación exclusiva, se le incrementará un 4%.
- Si no es tutor, pero tiene dedicación exclusiva, su sueldo se incrementará un 3,5%.
- Si no es tutor, no es encargado de asignatura y da clases en el ciclo superior, el sueldo se incrementará un 2%.
- Cualquier otra circunstancia no modificará el sueldo.

Realizar el diseño del diagrama de flujo que de la solución óptima al problema.

La tabla de decisión completa que muestra el problema es la siguiente:

	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12	R13	R14	R15	R16
Profesor tutor	N	N	N	N	N	N	N	N	S	S	S	S	S	S	S	S
Encargado de asignatura	N	N	N	N	S	S	S	S	N	N	N	N	S	S	S	S
Imparte ciclo superior	N	N	S	S	N	N	S	S	N	N	S	S	N	N	S	S
Dedicación exclusiva	N	S	N	S	N	S	N	S	N	S	N	S	N	S	N	S
Incremento del 3%											X	X			X	X
Incremento del 5%													X	X	X	X
Incremento del 4%										X		X				
Incremento del 3,5%		X		X		X		X								
Incremento del 2%			X	X												
No modifica	X				X		X		X							

Al simplificar, la tabla con el peso de cada regla y el coeficiente de cada condición quedaría de la siguiente forma:

	R1-9	R2	R3	R4	R5-7	R6-8	R10	R11	R12	R13-14	R15-16	
Profesor tutor	-	N	N	N	N	N	S	S	S	S	S	2
Encargado de asignatura	N	N	N	N	S	S	N	N	N	S	S	0
Imparte ciclo superior	N	N	S	S	-	-	N	S	S	N	S	4
Dedicación exclusiva	N	S	N	S	N	S	S	N	S	-	-	4
Incremento del 3%								X	X		X	
Incremento del 5%										X	X	
Incremento del 4%							X		X			
Incremento del 3,5%		X		X		X						
Incremento del 2%			X	X								
No modifica	X				X							
	2	1	1	1	2	2	1	1	1	2	2	

por lo que habría que empezar a preguntar por la segunda condición.



Especificaciones del lenguaje algorítmico UPSAM¹

A.1. Elementos del lenguaje

Identificadores

- Longitud: máximo 32 caracteres.
- Se admiten todos los caracteres alfabéticos (estándar y extendidos), dígitos y el símbolo de subrayado.
- No se hace distinción entre mayúsculas y minúsculas.
- Se debe empezar siempre por un carácter alfabético.

Comentarios

- Se recomienda el uso de la doble barra inclinada (//) al comienzo del comentario. Puede ir al principio de una línea del pseudocódigo, o después de alguna instrucción.
- Para comentarios multilínea, se puede encerrar el comentario entre llaves ({}).

¹ Las especificaciones del pseudolenguaje UPSAM (su denominación original fue UPS) han sido experimentadas desde el año 1990, en el Departamento de Lenguajes y Sistemas Informáticos de la Facultad y Escuela Universitaria de Informática de la Universidad Pontificia de Salamanca en Madrid y, con anterioridad, desde 1986 en el CESIES de la misma Universidad.

Las primeras especificaciones del pseudocódigo se publicaron en 1986 en nuestro libro *Metodología de la Programación*, editado por McGraw-Hill. El resultado que ahora se publica es fruto de todos los profesores que han impartido la asignatura de *Fundamentos de Programación*, y esta síntesis ha sido redactada por los profesores Luis Rodríguez, Matilde Fernández y Luis Joyanes, con las sugerencias y mejoras de los profesores Luis Villar y Joaquín Abeger. Han colaborado también los profesores Antonio Muñoz, Ángel Hermoso, Isabel Torralbo, Ángela Carrasco, Miguel Sánchez, Víctor Martín, M.^a del Mar García, Rafael Ojeda y Lucas Sánchez.

Tipos de datos estándar

- Numérico (entero y real).
- Lógicos (logico).
- Caracteres (character).
- Cadenas (cadena). Se considera como un dato estándar, pero estructurado, es decir, como un array de caracteres acabado en un carácter nulo.
- Se admiten cadenas abiertas o limitadas a un número determinado de caracteres, de la forma `cadena nombre_variable [n]`

Constantes estándar

- Numéricas enteras: están compuestas por los dígitos (0..9), y el signo (+, -).
- Reales en coma fija: están compuestas por los dígitos (0..9), el signo (+, -) y la coma decimal (,).
- Reales en coma flotantes: están compuestas por los dígitos (0..9), el signo (+, -) y la coma decimal (,) de la mantisa, la letra **E**, antes del exponente, los dígitos (0..9) y el signo (+, -).
- Lógicos: son las constantes que representan a verdadero (**Verdad**) y falso (**Falso**).
- Caracteres: cualquier carácter del juego de caracteres utilizado, delimitado por el separador de caracteres (').

Operadores

Tabla A.1. Operadores aritméticos

-	Menos unario
*	Multiplicación
/	División real
↑	Exponenciación
+	Adición
-	Resta
mod	Módulo de la división entera
div	División entera

Tipos de resultados de las expresiones aritméticas:

La división real siempre da un resultado real. Los operadores `mod` y `div` sólo operan con enteros y el resultado es entero. El menos unario devuelve el mismo tipo que el operando. En el resto de operadores aritméticos, si alguno de los operadores es real, el resultado es real.

Tabla A.2. Operadores de relación

=	Igual a
<>	Distinto de
<	Menor que
>	Mayor que
<=	Menor o igual que
>=	Mayor o igual que

Tipos de resultados de expresiones de relación:

Siempre devuelven un dato de tipo lógico. Los operandos deben ser del mismo tipo.

Tabla A.3. Operadores lógicos

no	Negación lógica
y	Multiplicación lógica
o	Suma lógica
o_x	O exclusivo
eqv	Equivalencia
imp	Implicación

Tipos de resultados de expresiones lógicas:

Siempre devuelven un dato de tipo lógico. Los operandos deben ser del mismo tipo lógico.

Tabla A.4. Prioridad de los operandos

Paréntesis	()
exponenciación	↑
Operadores unarios	no, -
Operadores multiplicativos	*, /, div, mod, y, ↑
Operadores auditivos	+, -, o, o_x
Operadores de relación	=, <, >, < >, >=, <=

Las operaciones se evalúan siempre de izquierda a derecha.

Operador de asignación: (←). Sólo se pueden asignar datos del mismo tipo.

Definición de tipos

- Registros:

```
registro: <nombre_del_tipo>
         <tipo>: <nombre_del_campo>
         ....
fin_registro
```

Una línea por campo.

- Archivos de organización secuencial

`archivo_s de <tipo_de_dato> : <nombre_del_tipo>`

- Archivos de organización relativa

`archivo_d de <tipo_de_dato> : <nombre_del_tipo>`

- Arrays

`array[dimensiones] de <tipo_de_dato>: <nombre_del_tipo>`

dimensiones sería un subrango.

- Conjunto

`conjunto de <tipo_de_dato> : <nombre_del_tipo> [es (<lista de valores>)]`

- Enumerados

`<nombre_del_tipo> es (<lista de valores>)`

- Subrangos

`subrango de <tipo_de_dato> : <nombre_del_tipo> es [<inf>..`

Definición de constantes

- `const <nombre_de_constante> = <expresion>`

Definición de variables

- `<tipo_de_dato> : <lista_de_variables>`

Funciones aritméticas de biblioteca

<code>abs()</code>	<i>valor absoluto de un número</i>
<code>ent()</code>	<i>valor entero (se redondea al entero menor)</i>
<code>trunc()</code>	<i>valor entero (se truncan los decimales)</i>
<code>redondeo()</code>	<i>redondea un número</i>
<code>aleatorio()</code>	<i>valor aleatorio</i>
<code>exp()</code>	e^x
<code>ln()</code>	<i>logaritmo neperiano (base e)</i>
<code>log10()</code>	<i>logaritmo decimal (base 10)</i>
<code>raiz2()</code>	<i>raíz cuadrada</i>
<code>cuadrado()</code>	<i>cuadrado de</i>

<code>sen()</code>	<i>seno</i>
<code>cos()</code>	<i>coseno</i>
<code>arctan()</code>	<i>arco tangente</i>
<code>tipo()</code>	<i>convierte la expresión en el tipo base especificado</i>

Funciones de cadena de biblioteca

<code>longitud()</code>	<i>Longitud de la cadena</i>
<code>posicion(<c>, <cb>)</code>	<i>Posición de cb dentro de c</i>
<code>subcadena(<c>, <ini>, [<long>])</code>	<i>Extrae de c, a partir del carácter ini una subcadena de longitud long</i>

Funciones de conversión de número a cadena

<code>código()</code>	<i>Devuelve el código ASCII de un carácter</i>
<code>car()</code>	<i>Devuelve el carácter asociado a un código ASCII</i>
<code>valor()</code>	<i>Convierte una cadena en un valor numérico</i>
<code>cad()</code>	<i>Convierte un número en cadena</i>

Funciones de información

<code>tamaño_de()</code>	<i>Devuelve la longitud en bytes de una variable o tipo de dato</i>
--------------------------	---

Estructura de un programa

```

algoritmo <nombre_algoritmo>
[incluir <modulo>]...
// declaraciones de constantes, tipos y variables
[const <declaraciones de constate>]
[tipo <declaraciones de tipos de datos>]
[var <declaraciones de variables>]
// Se recomienda declarar primero las constantes,
// luego los tipos y por ultimo las variables
// Cualquier objeto debe ser declarado antes de ser utilizado
inicio
...
fin
[declaraciones de procedimientos y funciones]

```

A.2. Estructuras de control

Estructuras selectivas

```

si condicion entonces
    <acciones>

```

```

[si_no
  <acciones>]
fin_si
segun_sea <expresion_ordinal> hacer
  <lista_de_valores_ordinales> : <acciones>
  ...
[si_no
  <acciones>]
fin_segun

```

- La expresión que se evalúa debe ser de tipo ordinal.
- Las listas de valores, son uno o más valores ordinales separados por comas.

Estructuras repetitivas

```

mientras <expresion_logica> hacer
  <acciones>
fin_mientras

repetir
  <acciones>
hasta_que <expresion_logica>

desde <variable_ordinal> ← <valor_final>
      incremento|decremento <valor>] hacer
  <acciones>
fin_desde

iterar
  <acciones>
  [salir_si]
fin_iterar

```

A.3. Programación modular

A.3.1. Inclusión de archivos o módulos

- `incluir <modulo>`, permite incluir archivos con subprogramas.

A.3.2. Procedimientos

- Declaración

```

procedimiento <nombre_proc>[lista_de_parametros_formales]
[declaraciones locales]
inicio
...
fin_procedimiento

```

<lista_de_argumentos> es uno o más argumentos de la siguiente forma:

{{E|S|E/S}¹ <tipo_de_dato> : <lista_de_argumentos>}...

- El tipo de dato debe estar definido por anticipado.
- Llamada a un procedimiento:

`[llamar_a] <nóm_proc>[lista_parametros_actuales]`

- Los parámetros deben coincidir en posición, tipo y número.

A.3.3. Funciones

- Declaración

```
<tipo_de_dato> funcion
<nombre_fun>[(lista_de_parametros_formales)]
[declaraciones locales]
inicio
    ...
    devolver (<expresion>)
fin_funcion
```

- Lista de parámetros es igual que en los procedimientos.
- `devolver` nos dice el valor de retorno de la función.
- Sólo se pueden devolver datos estándar.

A.3.4. Cuestiones generales

- El ámbito de las variables es igual que en Pascal.
- No se admiten subprogramas anidados.
- Los archivos se pasan siempre como E/S.

A.3.4. Procedimientos de entrada/salida

- `leer` (<lista_de_variables>)
- `escribir` ([<lista_de_expresiones>])
- `leercar` (<variable_tipo_caracter>)

A.4. Archivos

A.4.1. Archivos secuenciales (texto)

- Crear y abrir es distinto. Para abrir es necesario tener creado el archivo (esto también es válido para indexados y directos). Crear inicializa el archivo; abrir, permite acceder a un archivo existente.

¹ E: Entrada; S: Salida; E/S: Entrada/Salida.

- **crear** (<var_tipo_archivo>, <nombre_fisico>), nombre físico es una expresión de cadena.
- **abrir** (<var_tipo_archivo>, <modo>, <nombre_fisico>), nombre físico es una expresión de cadena.
- <modo> en archivos secuenciales puede ser: **l** (coloca el puntero de datos al inicio del archivo y abre para la lectura) o **e** (coloca el puntero de datos al final del archivo y abre para escritura).
- Escritura: **escribir** (<var_tipo_archivo>, <lista_de_expresiones>).
- Lectura: **leer** (<var_tipo_archivo>, <lista_de_variables>).
- Lectura: **leercar** (<var_tipo_archivo>, <var_tipo_caracter>).
- Fin de línea: **fdl** [(var_tipo_archivo)]
- Fin de archivo: **fda** [(var_tipo_archivo)]. Es verdadero si se ha intentado la lectura después del último registro.
- Cerrar un archivo **cerrar** (<lista var_tipo archivo>).
- Otros procedimientos útiles pueden ser **borrar** (<nombre_fisico>), que borra un archivo que debe estar cerrado y **renombrar** (<nombre_fisico>, <nombre_fisico>), que renombra el archivo primero por el nombre del segundo. Ambos deben estar cerrados.
- Otra función útil puede ser **lda** (<nombre_fisico>), que devuelve la longitud del archivo en bytes.

A.4.2. Archivos directos (relativos)

- **crear** (<var_tipo_archivo>, <nombre_fisico>), nombre físico es una expresión de cadena.
- **abrir** (<var_tipo_archivo>, <modo>, <nombre_fisico>), nombre físico es una expresión de cadena.
- <modo> en archivos directos sólo es **l/e** para lectura y escritura.
- Escritura: **escribir** (<var_tipo_archivo>, <var_tipo_base>, <posicion>). Escribe una variable del tipo base del archivo en la posición relativa especificada.
- Lectura: **leer** (<var_tipo_archivo>, <var_tipo_base>, <posicion>). Lee una variable del tipo base del archivo en la posición relativa especificada.
- Fin de línea y Fin de archivo, no tienen efecto en archivos relativos (*si consideramos la lectura secuencial, Fin de archivo sí tendría sentido*).
- También se admiten los procedimientos **borrar** y **renombrar**, así como la función **lda**.
- El número de registros del archivo se puede obtener con

`lda(<var_tipo_archivo>) / tamaño_de(var_tipo_base_del_archivo).`

A.4.3. Archivos indexados

- Definición:

```

archivo_i de <tipo_dato>: <nombre_del_tipo>
               clave_p <lista_de_campos> //Clave primaria sin duplicados
               [clave_s <lista_de_campos> [duplicada]]...

```
- **crear** (<var_tipo_archivo>, <nombre_fisico>), nombre físico es una expresión de cadena.

- **abrir** (<var_tipo_archivo>, <modo>, <nombre_fisico>), nombre físico es una expresión de cadena.
- <modo> en archivos indexados sólo puede ser **l/e**.
- Escritura: **escribir** (<var_tipo_archivo>, <var_tipo_base>). Escribe el registro en la posición que tenga en ese momento la clave primaria. Da error si existe, o si tiene claves secundarias sin duplicados.
- Lectura directa: **leer** (<var_tipo_archivo>, <var_tipo_base>, [<campo_clave>]). Por omisión lee el registro cuyo valor coincida con el que en ese momento tenga la clave primaria. Para leer por claves alternativas se incluirá el argumento <campo_clave>.
- Una función **Existe** (<var_tipo_archivo>) devolvería verdadero si la lectura ha sido correcta.
- Lectura secuencial: **leersec** (<var_tipo_archivo>, <var_tipo_base>, [<campo_clave>]). Lectura secuencial del siguiente registro de la clave especificada. Por omisión, lee la siguiente clave secuencial.
- Fin de archivo: **fda** [(var_tipo_archivo)]. Es verdadero si se ha intentado la lectura después del último registro.
- Otros procedimientos útiles pueden ser: **borrar** (<nombre_fisico>), que borra un archivo que debe estar cerrado y **renombrar** (<nombre_fisico>, <nombre_fisico>), que renombra el archivo primero por el nombre del segundo. Ambos deben estar cerrados.
- Otra función útil puede ser **lda** (<nombre_fisico>), que devuelve la longitud del archivo en bytes.

A.5. Variables dinámicas

- Declaración de tipos dinámicos
`puntero_a <tipo_de_dato> : <nombre_del_tipo>`
- Acceso al contenido de una variable
`<var_tipo_ptr> ↑`
- Asignar memoria a una variable de tipo puntero: **reservar** (<var_tipo_ptr>).
- Liberar memoria de una variable de tipo puntero: **liberar** (<var_tipo_ptr>).
- Puntero nulo: **nulo**.

A.6. Palabras reservadas, operadores, caracteres especiales, funciones y procedimientos estándar

Tabla A.5. Palabras reservadas, símbolos y operadores

Carácter / Palabra reservada	Significado / Traducción
'	delimitador de caracteres
→	variable apuntada
←	asignación
*	multiplicación

Tabla A.5. Palabras reservadas, símbolos y operadores (Cont.)

Carácter / Palabra reservada	Significado / Traducción
+	suma, concatenación
+	signo positivo
-	signo negativo
, (coma decimal)	.
-	menos unario
-	resta
/	división real
//	inicio de comentario
:	dos puntos
<	menor que
< =	menor o igual que
< >	distinto de
=	igual a
>	mayor que
> =	mayor o igual a
^	exponenciación
{	inicio de comentario
}	fin de comentario
&	concatenación
abrir	<i>open</i>
abs ()	<i>abs</i>
aleatorio ()	<i>random</i>
algoritmo	<i>program</i>
arctan ()	<i>atm</i>
archivo_d	definición de archivos directos
archivo_i	definición de indexados
archivo_s	definición de archivos secuenciales
array	<i>array</i>
borrar	<i>delete</i>
cad ()	<i>str</i>
cadena	<i>string</i>
car ()	<i>chr</i>
caracter	<i>char</i>
clave_p	clave primaria de archivos indexados
clave_s	clave secundaria de archivos indexado
codigo ()	<i>asc</i>
conjunto	<i>set</i>
const	<i>const</i>
cos ()	<i>cos</i>
crear ()	inicializa un archivo
cuadrado ()	<i>sqr</i>
de	<i>of</i>
decremento	decremento para bucles desde
desde	<i>for</i>
devolver	<i>return</i>
div	<i>div</i>

Tabla A.5. Palabras reservadas, símbolos y operadores. (Cont.)

Carácter / Palabra reservada	Significado / Traducción
duplicada	se admiten duplicados en claves Secundarias de archivos indexados
E	inicio del exponente en un real de coma flotante
e	modo de apertura de archivos para escritura
E	parámetro de entrada
E/S	parámetro de entrada/salida
ent()	<i>int</i>
entero	<i>integer</i>
entonces	<i>then</i>
eqv	<i>eqv</i>
es	<i>is</i>
escribir	<i>write</i>
existe()	indica en archivos indexados si una operación de lectura ha sido correcta
exp()	<i>exp</i>
Falso	<i>false</i>
fda	<i>eof</i>
fdl	<i>eoln</i>
fin	<i>end</i>
fin_desde	<i>next, end for</i>
fin_funcion	<i>end function</i>
fin_iterar	<i>loop</i>
fin_mientras	<i>end while</i>
fin_procedimiento	<i>end sub</i>
fin_según	<i>end select</i>
fin_si	<i>end if</i>
funcion	<i>function</i>
hacer	<i>do</i>
hasta	<i>to</i>
hasta_que	<i>until</i>
imp	<i>imp</i>
incluir	<i>include</i>
incremento	para bucles desde
inicio	<i>begin</i>
iterar	<i>do</i>
l/d	<i>I-O</i>
lda	<i>lof</i>
l	<i>input</i>
leer	<i>read</i>
leercar	<i>readkey, input\$</i>
leersec	lectura secuencial en archivos indexados
liberar	<i>dispose</i>
ln()	<i>log</i>
log10()	<i>log10</i>

Tabla A.5. Palabras reservadas, símbolos y operadores (Cont.)

Carácter / Palabra reservada	Significado / Traducción
logico	<i>boolean</i>
longitud()	<i>len, length</i>
llamar_a	<i>call</i>
mientras	<i>while</i>
mod	<i>mod</i>
no	<i>not</i>
nulo	<i>nil</i>
o	<i>or</i>
o_x	<i>xor</i>
posicion (<c>, <cb>)	<i>pos, index, instr</i>
procedimiento	<i>procedure</i>
puntero_a	declaración de tipos de puntero
raiz2	<i>sqrt</i>
real	<i>real</i>
redondeo ()	<i>round ()</i>
registro	<i>record, struct</i>
renombrar	<i>rename</i>
repetir	<i>repeat</i>
reservar	<i>new</i>
S	parámetro de salida
salir	<i>exit</i>
segun_sea	<i>case of</i>
sen ()	<i>sin</i>
si	<i>if</i>
si_no	<i>else</i>
subcadena (<c>, <ini>, [lond>])	<i>mid, copy</i>
subrango	declaración de subrangos
Tamaño_de	<i>sizeof</i>
tipo	<i>type</i>
trunc ()	<i>trunc</i>
var	<i>var</i>
valor ()	<i>val</i>
verdad	<i>true</i>
y	<i>and</i>



Bibliografía

TEORÍA DE PROGRAMACIÓN EN GENERAL

- BRAUNSTEIN, Silvia L. GIOIA, y 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.
- 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, y E. W. HOARE C.A.R.: *Structured Programming*. Academic Press, 1973.
- DAHL; DIJKSTRA, y HOARE: *Structured Programming*, 1972.
- DIJKSTRA, E. W.: «Go to statement considered harmful». *Comm. Assoc. Comp. Mach.*, 11 (1968, pp. 147-148).
- DIJKSTRA, E. W.: «Notes on Structured Programming». *Structured Programming*. Academic Press, Nueva York, 1972.
- GHEZI, Carlo, y JAZAYER Meh: *Conceptos de lenguajes de programación*. Díaz de Santos, 1986.
- GOLDSCHLAGER; LES, y LISTER, Andrew: *Introducción moderna a la ciencia de la computación*. Prentice-Hall, 1986.
- HOARE, Car.: «Notes on Data Structuring». *Structured Programming*. Academic Press, Nueva York, 1972.
- HUGHE, Juan K.; MIGHTOM, Glen C., y MIGHTOM, Jay I.: *A Structured approach to programming*. 2.^a edición. Prentice-Hall, 1977.
- JOYANES, Luis: *Metodología de la Programación*, McGraw-Hill, 1986; *Problemas de Metodología de la Programación*, McGraw-Hill, 1990.
- JOYANES, Luis, RODRÍGUEZ, Luis, y FERNÁNDEZ, Matilde: *Fundamentos de programación. Libro de problemas*. McGraw-Hill, 1996.
- KNUTH, D. E.: *Structured Programming with goto statements*. *Comput. Serv.* 6(1974), pp. 261-301.
- LASALA; CRISTÓBAL, y LAPENA: *Técnicas de análisis y programación para IBM PC (y compatibles)*. Universidad de Zaragoza, 1985.
- MORALES PASCUAL, José Luis: *Programación de ordenadores*, tomos (1 a 6). ECC, 1986.
- NAYLOR, Jeff: *Introduction to Programming*. Paradigm Publishing Ltd., 1987.

- RODRÍGUEZ ASENSIO: *Metodología de análisis y programación*. R. Asensio, 1985.
 SABATINI, Domenico: *Introduzione alla programmazione strutturata*. Bufetti Editore, 1985.
 VIRGO, Fernando: *Técnicas y elementos de programación*. Gustavo Gili, Barcelona, 1985.
 WIRTH, Niklaus: *Introducción a la programación sistemática*. El Ateneo. Buenos Aires, 1984.
 WOODHOUSE, David; JOHNSTONE, Greg, y MCDUGAL, Anne: *Computer Science*. Wiley & Sons, 1982.

ALGORITMOS Y ESTRUCTURA DE DATOS

- BRASSARD, G.; BRATLEY, P.: *Algorithmique*. Masson, 1987.
 BOUSSARD, J. C., y MAHL, R.: *Programmation Avancee: Algorithmique et Structures de Donnss*. Eyrolles, 1983.
 COURTIN, J., y KOWARSKI, I.: *Initiation a l'algorithmique et aux structures de donnss*. Dunod, 1987.
 DALE, y 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.
 HAREL, David: *Algorithmics*. Addison Wesley, 1987.
 HELMAN, Paul, y VEROFF, Robert: *Intermediate Problem Solving and Data Structures*. Benjamín 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.
 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. 161-301.
 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.
 LUCAS; REYRIN, y SCHOLL: *Algorítmica y representación de datos*. Vol. 1, *Secuencias, autómatas de estados finitos*. Masson, 1985.
 MELHORN, N.: *Data Structures and Algorithms*. Vol. 1, *Sorting and Searching*. Springer-Verlag, 1984. Vol. 2, *Graph Algorithms and WP-Completeness*. Springer-Verlag. Berlín.
 LIPSCHUTZ, Seymour: *Estructura de datos*. McGraw-Hill, 1986.
 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.
 WIRTH, Niklaus: *Algoritmos + 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 System*. 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

- GUIHUR, R.: *Procedimientos de clasificación*. Masson, 1987.
- KNUTH, D. E.: *The art of Computer Programming*. Vol. 2, *Sorting and Searching*. Addison Wesley, 1973.
- ROUX, M.: *Algorithmes 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, Terrence 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 traslation*. International Computer Science. Addison Wesley, 1986.
- TUKER: *Lenguajes de programación*. McGraw-Hill, 1986.
- YOUNG, S. J.: *Real Time Languages*. Ellis Horwood-Publishers, 1982.

LENGUAJE ADA

- ALABAN, M.: *Le langage Ada. Theorie et pratique*, Eyrolles, 1995.
- BARNES, J. G. P.: *Programación en Ad.*, Díaz de Santos, 1987.
- FELDMAN, M. B., y KOFFMAN, E. B.: *Ada 95. Problem Solving and Program Design*, Addison-Wesley, 1996.
- LUCAS, Angel: *Ada. Metodología y Lenguaje*. Paraninfo, 1990.
- SAVITCH, Walter J., y PETERSEN, C. G.: *Ada. An Introduction to the Art and Science of Programming*. Benjamin Cummings, 1992.

LENGUAJE BASIC

- ETTLIN, y SOLBERG: *Manual de MBASIC*. McGraw-Hill, 1985.
- GOTTFRIED: *Programación BASIC*, 3ª edición. McGraw-Hill, 1986.
- GRAHAM, y FIELD: *IBM/PC. Guía del IBM/PC*. McGraw-Hill, 1986.
- HERGERT, Douglas: *Microsoft Quick BASIC*. Microsoft Press, 1987. *Using Turbo BASIC*. McGraw-Hill, 1987.
- JOYANES AGUILAR, Luis: *Programación BASIC para microcomputadoras*. 3ª edición. McGraw-Hill, 1992.
- *IBM PC. Manual de BASIC*. McGraw-Hill, 1986.
- *Introducción a la programación estructurada en BASIC (True BASIC)*. UPS, 1987.
- *BASIC Avanzado. Tratamiento y diseño de archivos*. McGraw-Hill, 1987.
- *Apple II. Manual de BASIC*. McGraw-Hill, 1986.
- *ZX Spectrum. Guía del usuario*. McGraw-Hill, 1986.
- *Programación en QuickBasic/QBasic*, 2ª edición, McGraw-Hill, 1995.
- JOYANES Luis, y VILLAR Luis: *QuickBASIC avanzado*, McGraw-Hill, 1993.
- KURTZ, y KEMMENY: *Structured BASIC Programming*. Wiley, 1987.

LENGUAJE C

- DELANNOY, C.: *El libro del C. Primer lenguaje*. Eyrolles, 1995.
 KERNIGHAM, B. M., y RITCHIE D. M.: *El lenguaje de programación C*. 2.^a edición. Prentice-Hall, 1991.
 PLAUGER, P. J.: *The Standard C Library*. Prentice-Hall, 1992.
 PLAUGER, P. J., y BRODIE, J.: *C estándar*. Anaya, 1990.
 RADCLIFFE, Robert A.: *Encyclopédia C*. Sybex, 1991.

LENGUAJE C++

- AHMERAL, Lendert: *C++ for Programmers*. Wiley, 1991.
 ANDERSON, Arthur, y HEINZE, William: *C++ Programming and Fundamentals Concepts*. Prentice-Hall, 1992.
 AT & T C++: *Reference Manual*, 1989.
 ATKINSON & ATKINSON: *Using C/C++*. Quie, 1993.
 BARKAKATI: *Object-Oriented Programming in C++*. Sams, 1991.
 BERRY, John Tomas: *C++ Programmings*. Second Edition. The Waite Groups's. Sams, 1992.
 BOOCH, Grady: *Object-Oriented Design with Applications*. Benjamin Cummings, 1991.
 BUDD, Timothy: *An Introduction to Object-Oriented Programming*. Addison-Wesley, 1991.
 BORLAND C++: *Programmer's Guide, Library Reference, User's Guide, Tools Utilities Guide, Borland C++ 4*. Borland International, 1991, 1992, 1993.
 CANTÚ, Marco, y TENDON, Steve: *Borland C++ 3.1. Object-Oriented Programming*. Bantam Books, 1992.
 COX, Brad J., y NOVOBILSKI, Andrew: *Object-Oriented Programming. An Evolutionary Approach*. Addison-Wesley, 1991.
 CHRISTIAN, Kaare: *The Microsoft Guide to C++ Programming*. Microsoft Press, 1992.
 DEWHURST & STARK: *Programming in C++*. Prentice-Hall, 1989.
 ECKEL, Bruce: *Aplique C++*. Osborne McGraw-Hill, 1991. *C++ Inside & Out*. McGraw-Hill, 1993.
 EGE, Raimund K: *Programming in a Object-Oriented Environment*. Academic Press, 1992.
 ELLIS & STROUTUP: *The annotated C++ Reference Manual*. Addison-Wesley, 1990.
 ENSTMINGER, Gary: *The Tao of Objects*. M. & T Books, 1991.
 FAISON: *Borland C++4. Objects Oriented Programming*. Sams, 1994.
 FLAMING, Bryan: *Turbo C++. A Self-Teaching Guide*. Wiley, 1991, *Turbo C++. Step-by-step*. Wiley, 1991.
 GORLEN; ORLOW, y PLEXICO: *Conception et Programmation Orientées Object en C++*. Dunod, 1991.
 HOLUB, Allen I: *C+ C++. Programming with Objects in C and C++*. McGraw-Hill, 1992.
 JOYANES, Luis. *Turbo C++. Manual de bolsillo*. McGraw-Hill, 1992.
 — *C++. Manual de Bolsillo*. McGraw-Hill, 1993.
 — *Borland C++. Manual de bolsillo*. McGraw-Hill, 1993.
 — *C++ a su alcance. Un enfoque orientado a objetos*. McGraw-Hill, 1994.
 — *Microsoft C/C++. Manual de bolsillo*. McGraw-Hill, 1994.
 — *Programación orientada a objetos*. McGraw-Hill, 1996.
 — *Turbo C++. Iniciación y referencia*. McGraw-Hill, 1996.
 — *Visual C++. Iniciación y referencia*. McGraw-Hill, 1996.
 — *Curso de programación con Turbo++*. McGraw-Hill, 1996.
 KERNIGHAM & RITCHIE: *El Lenguaje de Programación C*. Segunda edición. Prentice-Hall, 1991.
 LAFORE, Robert: *Object-Oriented Programming in Turbo C++*. Waite Group Press, 1991.
 LIPPMAN, Stanley B: *C++ Primer*. Primera y segunda edición. Addison-Wesley, 1989, 1991.
 MCCORD: *Borland C++ 3.1. Programmer's Reference*. Que, 1992.
 Microsoft Languages Library: *Microsoft C. Run-Time Library Reference*. Microsoft Press, 1990.
 — *Run-Time Library Reference. Microsoft C/C++*. 1991.
 — *Class Libraries User's*. 1991.
 — *Guide. C++ Language Reference*. 1991.

- PAPAS & MURRAY: *Microsoft C/C++ 7. The Complete Reference*. McGraw-Hill, 1992.
- POHL, Ira: *C++ for C Programmers*. Benjamin/Cummings, Redwood City, CA, 1989.
- PRATA, Stephen: *C++ Primer Plus*. Waite Group Press, 1991.
- RANADE, Jay & ZAMIR, Saba: *C++ Primer for Programmers C*. McGraw-Hill, 1992.
- SHAMMAS: *Borland C++4. Programmer Should Know*, Sams, 1994.
- SCHILDT: *Aplique Turbo C++*. McGraw-Hill, 1991.
- SKINNER, M. T: *The Advance C++ Book*. Prentice-Hall, 1992. M & T Books, 1990.
- SMITH, Michael A: *Object-Oriented software in C++*. Chapman & Hall, 1993.
- STROUSTRUP, Bjarne: *The C++ Programming Language*. Second Edition. Addison-Wesley, 1991.
- SWAN, Ton: *Learning C++*. Sams, 1991. *Mastering Borland C++*. Sams, 1992.
- VOSS, Greg: *Object-Oriented Programming*. McGraw-Hill, 1991.
- WAITE: *Turbo C++ Bible*. Sams, 1990.
- WAITE & PRATA: *New C Primer Plus*. Second Edition. Sams, 1993.
- WEISKAMP & FLAMING: *C++ The Complete Primer*. Academic Pres, 1991.
- WIENNER & PINSON: *An Introduction to Object-Oriented Programming and C++*. Addison-Wesley, 1988.
- WINBLAD; EDWARDS, y KING: *Object-Oriented software*. Addison-Wesley, 1990.

LENGUAJE COBOL

- DE LA FUENTE, Ciro: *Lenguaje COBOL*, 10.^a edición, El autor, 1993.
- GALÁN PASCUAL, C.: *Programación con Lenguaje COBOL*, Paraninfo, 1986
- GARCÍA CERRO: *Programación COBOL, Algoritmos estructurados*, McGraw-Hill, 1994.
- HURTADO MELERO, T.: *Lenguaje COBOL con programación estructurada*, Paraninfo, 1988.
- *Programación COBOL*, Paraninfo, 1988.
- JARILLO, P., Y JARILLO, M.L.: *COBOL*, Anaya, 1995.
- MÁRQUEZ, T.: *Diseño de programas en COBOL con programación estructurada y pseudocódigo*, McGraw-Hill, 1995.
- NEWCOMER, L.: *Programación avanzada en COBOL estructurado*, McGraw-Hill, 1989.
- *Programación en COBOL estructurado*, McGraw-Hill, 1988.
- PHILIPPAKIS, G.: *COBOL estructurado*, 3.^a edición, McGraw-Hill, 1958.

LENGUAJE FORTRAN

- DAVIS: *FORTRAN 77*. McGraw-Hill, 1986.
- ETTER, D. M.: *Structured FORTRAN 77*. Benjamin Cummings, 1987.
- HAMMOND; ROGER, y CRITTENDEN: *Introduction to FORTRAN 77 and the personal Computer*. McGraw-Hill, 1987.
- JOYANES AGUILAR, Luis: *Introducción al FORTRAN 77*. UPS, 1988.
- JOYANES Luis, y ZAHONERO, Ignacio: *Programación en FORTRAN 77 y 90*, Los autores, Distribución Librería Paraninfo (Argüelles, Madrid), 1993.
- KOFFMAN, y FRIEDMAN: *Problem Solving and Structured Programming in FORTRAN 77*, 3^a edición. Addison Wesley, 1987.
- POLLACK, Seymour V.: *Structures FORTRAN 77 Programming*. Boyd & Fraser Computer Science Series, 1982.

LENGUAJE MODULA-2

- GREENFIELD, Stuart B.: *Invitation to Modula-2*. Pretocelli Books, Inc. 1985.
- KAARE, Christian: *A guide to Modula-2*. Springer-Verlag 1986.
- KELLY, y BOOTLE: *Modula-2*. Primer, 1987.

- MOORE, y MCKAY: *Modula-2. Text and Reference*. Reston, 1987.
 OGILVIE, John W. L.: *Modula-2 Programming*. McGraw-Hill, 1985.
 TERRY, Patrick D.: *An Introduction Programming with Modula-2*. Addison Wesley, 1987.
 WIRTH, Niklaus: *Programming in Modula-2*. Springer-Verlag, 1985.

LENGUAJE PASCAL

- DELGADO, Alonso: *Sistema para programación en Pascal sobre microordenadores*. Gustavo Gili, 1986.
 DALE, y LILLY: *Pascal y estructura de datos*. McGraw-Hill, 1986.
 DALE, y ORSHALICK: *Pascal*. McGraw-Hill, 1986.
 GARLAND STEPHEN, J.: *Introduction to Computer Science with applications in Pascal*. Addison Wesley, 1986.
 GOTTFRIED, Byron: *Programación en Pascal*. McGraw-Hill, 1985.
 GROGONO: *Programación en Pascal*. Fondo Educativo Interamericano. McGraw-Hill, 1984.
 HOARE, y WIRTH: *An Axiomatic Definition of the Programming Language Pascal*, Acta Informática, 2, núm. 4, 1973, 335-355.
 JAMSA, y NAMEROFF: *Turbo Pascal. Biblioteca de programas*, 1987.
 JENSEN, K, y WIRTH, N.: *Pascal. Manual de l'utilisateur et report*. 3ª edición. Eyrolles.
 JENSEN, y WIRTH, Niklaus: *Pascal. Manual del usuario e informe*. El Ateneo, Buenos Aires, 1985.
 JOYANES, Luis: *Programación en Turbo Pascal*. 2ª edición.
 — *Turbo Pascal 6.0 a su alcance*. McGraw-Hill.
 — *Turbo Pascal 7. Manual de bolsillo*. McGraw-Hill 1995.
 — *Turbo Pascal 6. Manual de bolsillo*. McGraw-Hill, 1993.
 JOYANES Luis; ZAHONERO, Ignacio, y HERMOSO, Ángel: *Pascal y Turbo Pascal. Un enfoque práctico*, McGraw-Hill, 1995.
 KONVALENA, y WILEMAN: *Programming with Pascal*. McGraw-Hill, 1987.
 LLESTMA, Sanford, y NYHOFF, Larry: *Pascal*. McMillan, 1984.
 LINZ: *Programming Concept and Problem Solving*. Benjamin Cummings, 1983.
 MILLER, Lawrence, H.: *Advanced Programming. Design and Structure using Pascal*. Addison Wesley, 1986.
 TISSERANT, A.: *Pascal ISO/AFNOR*. Mason, 1986.

TERMINOLOGÍA INFORMÁTICA

- OXFORD: *Diccionario Oxford de Informática*. Díaz de Santos, 2ª edición, 1993.
 ROSEMBERG: *Diccionario of Computers. Data Processing & Telecomunications*. Wiley, 1985.
 SPILL, Charles, J.: *Macmillan Dictionary of Microcomputing*. Macmillan, 1985.
 VAQUERO, y JOYANES: *Informática. Glosario de términos*. McGraw-Hill, 1985.

Índice

A

Acumulador, 40-41

Algoritmo, 1

- características, 3
- con retroceso, 88
- diseño, 3
- especificaciones, 3
- estructura, 39
- recursivo, 87
- verificación, 4

Apuntador, 277 (*véase* puntero)

Árbol, 331

- altura, 332
- binario, 333
- grado, 332
- hoja, 332
- nodo, 332
- notación, 336
- profundidad, 332
- recorrido, 336

Archivo, 167

- buffer, 167
- clave, 167, 195
- colisiones, 197
- conceptos generales, 167
- conversión de clave, 196
- de texto, 170
- directo, 195
- factor de bloqueo, 167
- fusión, 253
- operaciones básicas, 168
- ordenación, 253
- partición, 254-255

registro físico, 167

registro lógico, 167

secuencial, 169

sinónimo, 197

Array, 109

- bidimensional, 111
- como parámetro, 113
- de registros, 114
- matriz, 111, 128, 133-137
- paralelo, 114
- recorrido, 112
- tabla, 122
- tridimensional, 110
- unidimensional, 110
- vector, 110, 115, 118, 127

Arreglo, 109 (*véase* Array)

B

Böhm, 57

Búsqueda, 235

- binaria, 236
- externa, 253
- secuencial, 235
- transformación de claves, 236

C

Cadena, 157

- constante, 137
- posición, 159
- variable, 157

Clave, 167, 195, 236

Cola, 283
 aplicaciones, 284
 doble cola, 284

Colisiones, 238

Constante, 5
 cadena, 5
 carácter, 5
 entera, 5
 lógica, 5
 real, 5

Contador, 40

D

Datos, 4
 cadena, 4
 carácter, 4
 entero, 4
 estructurados, 109
 lógico, 4
 real, 4

desde, 60-61

Diagrama, 23
 de flujo, 23-24, 59, 61
 N-S, 23-24, 59, 61

Doble cola, 284

E

Estructura, 25
 anidada, 61
 básica, 57
 de control, 58
 de datos lineal, 277
 de datos no lineal, 331
 dinámica, 277

Expresión, 5
 alfanumérica, 6
 booleana, 6
 lógica, 6
 numérica, 6
 prioridad, 6

F

Fichero, 167 (*véase* Archivo)

Función, 7, 82
 abs, 7
 arctan, 7
 cos, 7
 cuadrado, 7
 declaración, 82
 ent, 7
 exp, 7
 ln, 7
 log10, 7
 raiz2, 7
 redondeo, 7
 sen, 7
 trunc, 7

Fusión, 254

G

Grafo, 337
 arco, 338
 arista, 338
 completo, 338
 conectado, 338
 desconectado, 338
 dirigido, 338
 nulo, 338
 representación, 338
 terminología, 338

H

hash, 238
hasta, 60

I

Identificador, 7, 27
Instrucciones, 40
 asignación, 40

bifurcación, 40
escritura, 40
fin, 40
inicio, 40
lectura, 40
Intercalación, 244
Interruptor, 40-41

J

Jacopini, 57

L

Lenguaje algorítmico, 371
 UPSAM, 371
LIFO, 282
lista, 278
 circular, 280
 contigua, 279
 doblemente enlazada, 281
 enlazada, 279

M

mientras, 60
Módulo, 81

N

Nassi-Schneiderman, 23

O

Ocho reinas, 107
Operador, 5
 prioridad, 6
Operando, 5
Ordenación, 235, 240
 burbuja, 241

de archivos, 253-256
 externa, 254
 inserción binaria, 242
 inserción directa, 241
 interna, 240
mezcla directa, 256
mezcla natural, 256
rápida, 87, 242
Shell, 242-244
selección, 240

P

Parámetro, 82, 84
 array como, 113
 entrada, 85
 paro de, 84
 salida, 85
Pila, 282
 aplicaciones, 28
Polaca, notación, 336
postfija, 336
prefija, 336
problema, 1
 análisis, 2
 fase de implementación, 1, 4
 fase de resolución, 1
 pseudocódigo, 25
 resolución, 23
procedimiento, 83
 declaración, 83
 invocación, 83
 paro de parámetros, 84
 variable global, 86
 variable local, 86
programa, 1, 39
 estructura, 39
Programación, 1
 descendente, 61
 estructurada, 57
 modular, 81
pseudocódigo, 23, 59, 61
puntero, 277

Q

quick-sort, 87, 242

R

Recursividad, 86

 algoritmos, 87

 funciones, 105, 251

Reina, 107

Resolución de problemas, 23

S

Subalgoritmo, 81

Subprograma, 81

T

Tabla, 122

Tabla de decisión, 359
 ampliada, 361

 entrada de condiciones, 359

 estructura, 363

 limitada, 361

 lista de acciones, 359

 lista de condiciones, 359

 regla de decisión, 362

 salida de acciones, 359

 tipos, 361

 transformación en diagrama de flujo, 363

Tabla de verdad, 6

Torre de Hanoi, 87

U

UPSAM, 371

V

Variable, 5, 31, 40, 86

 global, 86

 local, 86

 puntero, 277