EVALUACION DEL PERSONAL DOCENTE

CURSO: METODOLOGIA DE LA PROGRAMACION
FECHA: 6-11 FEBRERO

CONFERENCISTA	DOMINIO DEL TEMA	USO DE AYUDAS AUDIOVISUALES	COMUNICACION CON EL ASISTENTE	PUNTUALIDAD
ING. JAVIER VALENCIA FIGUEROA				

EVALUACION DE LA ENSEÑANZA

ORGANIZACION Y DESARROLLO DEL CURSO	
GRADO DE PROFUNDIDAD LOGRADO EN EL CURSO	
ACTUALIZACION DEL CURSO	
APLICACION PRACTICA DEL CURSO	

EVALUACION DEL CURSO

CONCEPTO	CALIF
CUMPLIMIENTO DE LOS OBJETIVOS DEL CURSO	
CONTINUIDAD EN LOS TEMAS	
CALIDAD DEL MATERIAL DIDACTICO UTILIZADO	

ESCALA DE EVALUACION. 1 A 10

1.- ¿LE AGRADO SU ESTANCIA EN LA DIVISION DE EDUCACION CONTINUA?

SI	NO

SI INDICA QUE "NO" DIGA PORQUE.

COORDINACION CURSOS DE COMPUTO
CENTRO DE INFORMACIÓN Y DOCUMENTACION

2.- MEDIO A TRAVES DEL CUAL SE ENTERO DEL CURSO:

PERIODICO EXCELSIOR		FOLLETO ANUAL		GACETA UNAM		OTRO MEDIO	
PERIODICO EL UNIVERSAL		FOLLETO DEL CURSO		REVISTAS TECNICAS			

3.- ¿QUE CAMBIOS SUGERIRIA AL CURSO PARA MEJORARLO?

_____

_____

4.- ¿RECOMENDARIA EL CURSO A OTRA(S) PERSONA(S)?

SI		**NO**	

5.- ¿QUE CURSOS LE SERVIRIA QUE PROGRAMARA LA DIVISION DE EDUCACION CONTINUA.?

_____

_____

6.- OTRAS SUGERENCIAS:

_____

_____

7.- ¿EN QUE HORARIO LE SERIA CONVENIENTE SE IMPARTIERAN LOS CURSOS DE LA DIVISION DE EDUCACION CONTINUA?
MARQUE EL HORARIO DE SU AGRADO

LUNES A VIERNES DE 16 A 20 HORAS	**MARTES Y JUEVES DE 17 A 21 HS SABADO DE 10 A 14 HS.**	**OTRO**
LUNES, MIERCOLES Y VIERNES DE 17 A 21 HORAS	**VIERNES DE 17 A 21 HS. SABADOS DE 10 A 14 HS**	

COORDINACION CURSOS DE COMPUTO
CENTRO DE INFORMACIÓN Y DOCUMENTACION

METODOLOGIA DE LA PROGRAMACION

Aplicaciones en COBOL y Pascal

Segunda edición

METODOLOGIA DE LA PROGRAMACION

Aplicaciones en COBOL y Pascal

Segunda edición

EDUARDO ALCALDE LANCHARRO
Profesor de Informática de Gestión
IFP de Alcobendas. Madrid
Profesor en E. U. de Informática
Universidad Pontificia Comillas. Madrid

MIGUEL GARCIA LOPEZ
Profesor de Informática de Gestión
IPFP «Palomeras-Vallecas». Madrid
Profesor Asociado en E. U. de Informática
Universidad Politécnica de Madrid

McGRAW-HILL

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

METODOLOGÍA DE LA PROGRAMACIÓN. Segunda edición

Edificio Oasis-A, 1.ª planta
Basauri, s/n
28023 Aravaca (Madrid)

ISBN 84-7615-913-7
Depósito legal: M.27.496-1992

Editor: Teodoro Bartolomé
Diseño cubierta: Juan García

Impreso en México Printed in Mexico

4567890123 IE-93 9087651234

Esta obra se terminó de imprimir en Septiembre de 1994 en Ávalos, S.A. de C.V. Temoluco Núm. 40 Col. Zona Residencial Acueducto de Guadalupe Delegación Gustavo A. Madero 07270 México, D.F.

Se tiraron 1200 ejemplares

Contenido

Prólogo xi

Prólogo a la Segunda Edición xii

Capítulo 1. La programación de computadoras 1

1.1. Introducción 1
1.2. Ciclo de vida de una aplicación informática 2
1.2.1. Diseño del programa 2
1.2.2. Instalación y explotación del programa 3
1.3. Errores 4
1.3.1. Errores de compilación 4
1.3.2. Errores de ejecución 5
1.3.3. Errores de lógica 5
1.3.4. Errores de especificación 5
1.4. Calidad de los programas 5
1.5. Metodología de la programación 6
1.5.1. Programación modular 6
1.5.2. Programación estructurada 6
1.6. Documentación de los programas 6
1.6.1. Documentación interna 6
1.6.2. Documentación externa 7
1.7. Objetos de un programa 7
1.8. Identificadores 7
1.9. Tipos de datos 8
1.9.1. Tipo numérico entero 8
1.9.2. Tipo numérico real 9
1.9.3. Tipo carácter 9
1.9.4. Tipo booleano 9
1.10. Constantes 10
1.11. Variables 10
1.12. Expresiones 10
1.12.1. Tipos de expresiones 10
1.12.2. Operadores 11
1.12.3. Tablas de verdad de los operadores lógicos 11
1.12.4. Orden de evaluación de los operadores 12
Ejercicios resueltos 13
Ejercicios propuestos 14

Capítulo 2. Diagramas de flujo 15

2.1. Introducción 15
2.2. Diagramas de flujo del sistema. Organigramas 16
2.2.1. Símbolos de soporte 17
2.2.2. Símbolos de proceso 19
2.2.3. Líneas de flujo 19
2.3. Diagramas de flujo del programa, Ordinogramas 21
2.3.1. Símbolos de operación 22
2.3.2. Símbolos de decisión 23
2.3.3. Líneas de flujo 23
2.3.4. Símbolos de conexión 23
2.3.5. Símbolo de comentarios 24
2.4. Plantillas y normalización 24
2.5. Proyecto de norma española PNE 71-001 25
Ejercicios resueltos 30
Ejercicios propuestos 37

Capítulo 3. Estructura general de un programa ... 39

3.1. Introducción ... 39
3.2. Partes principales de un programa ... 39
3.2.1. Entrada de datos ... 40
3.2.2. Proceso o algoritmo ... 40
3.2.3. Salida de resultados ... 40
3.3. Clasificación de las instrucciones ... 41
3.3.1. Instrucciones de declaración ... 41
3.3.2. Instrucciones primitivas ... 41
3.3.3. Instrucciones de control ... 43
3.3.4. Instrucciones compuestas ... 51
3.3.5. Comentarios ... 51
3.4. Variables auxiliares de un programa ... 51
3.4.1. Contadores ... 52
3.4.2. Acumuladores ... 53
3.4.3. Interruptores o conmutadores (switches) ... 54
3.5. Tipos de programas ... 56
3.6. Lenguajes de programación ... 57
Ejercicios resueltos ... 58
Ejercicios propuestos ... 63

Capítulo 4. Pseudocódigo ... 65

4.1. Introducción ... 65
4.2. Pseudocodificación de programas ... 65
4.2.1. Acciones simples ... 65
4.2.2. Sentencias de control ... 66
4.2.3. Acciones compuestas ... 69
4.2.4. Comentarios ... 70
4.2.5. Objetos del programa ... 70
4.2.6. Programa ... 71
4.3. Paso de pseudocódigo a diagrama de flujo ... 73
4.4. Paso de pseudocódigo a lenguaje de programación ... 77
Ejercicios resueltos ... 82
Ejercicios propuestos ... 91

Capítulo 5. Estructuras de datos internas (tablas) ... 93

5.1. Introducción ... 93
5.2. Conceptos y definiciones ... 93
5.3. Tipos de tablas ... 95
5.3.1. Tablas unidimensionales (vectores) ... 95
5.3.2. Tablas bidimensionales (matrices) ... 97
5.3.3. Tablas multidimensionales (poliedros) ... 102
5.4. Representación gráfica de las tablas ... 107
5.4.1. Tabla de una dimenión (vector) ... 107
5.4.2. Tabla de dos dimensiones (matriz) ... 108
5.4.3. Tabla de tres dimensiones (poliedro-3) ... 108
5.4.4. Tabla de cuatro dimensiones (poliedro-4) ... 109
5.5. Tratamiento secuencial de una tabla ... 110
5.5.1. Tratamiento secuencial de un vector ... 110
5.5.2. Tratamiento secuencial de una matriz ... 111
5.5.3. Tratamiento secuencial de un poliedro ... 113
Ejercicios resueltos ... 114
Ejercicios propuestos ... 131

Capítulo 6. Búsqueda y clasificación interna ... 132

6.1. Introducción ... 132
6.2. Búsqueda lineal ... 132
6.2.1. Búsqueda lineal en un vector ... 133

6.2.2. Búsqueda lineal en un vector ordenado ... 134
6.2.3. Búsqueda lineal en una matriz ... 135
6.3. Búsqueda binaria o dicotómica ... 137
6.4. Ordenación de tablas ... 138
6.4.1. Ordenación por inserción directa ... 139
6.4.2. Ordenación por selección directa ... 141
6.4.3. Ordenación por intercambio directo. Método de la burbuja ... 143
6.4.4. Ordenación por intercambio directo con test de comprobación (switch) ... 146
6.4.5. Ordenación por intercambio directo. Método de la sacudida ... 147
6.4.6. Ordenación por intercambio con incrementos decrecientes. Método Shell ... 151
Ejercicios resueltos ... 154
Ejercicios propuestos ... 172

Capítulo 7. Diseño descendente: Subprogramas ... 174

7.1. Introducción ... 174
7.2. Programa principal y subprogramas ... 175
7.3. Subprogramas internos ... 176
7.4. Subprogramas externos ... 177
7.5. Objetos globales y locales ... 179
7.6. Variables de enlace (parámetros) ... 180
7.7. Paso de parámetros ... 182
7.8. Recursividad ... 184
Ejercicios resueltos ... 186
Ejercicios propuestos ... 201

Capítulo 8. Técnicas de programación estructurada ... 203

8.1. Introducción ... 203
8.2. Teorema de la estructura ... 204
8.3. Herramientas de la programación estructurada ... 207
8.4. Método de Warnier ... 212
8.5. Método de Jackson ... 215
8.6. Método de Bertini ... 218
8.7. Método de Tabourier ... 221
8.8. Método de Chapin (Nassi/Shneiderman) ... 224
Ejercicios resueltos ... 227
Ejercicios propuestos ... 234

Capítulo 9. Estructuras de datos externas (archivos) ... 235

9.1. Introducción ... 235
9.2. Conceptos y definiciones ... 235
9.3. Características de los archivos ... 236
9.4. Clasificación de los archivos según su uso ... 236
9.5. Organización de archivos ... 237
9.5.1. Organización secuencial ... 237
9.5.2. Organización aleatoria o directa ... 238
9.5.3. Organización secuencial indexada ... 238
9.6. Operaciones sobre archivos ... 239
9.7. Instrucciones para manejo de archivos ... 239
9.7.1. Creación de archivos secuenciales ... 240
9.7.2. Lectura de archivos secuenciales ... 245
9.7.3. Lectura-escritura de archivos directos ... 250
9.7.4. Lectura-escritura de archivos indexados ... 255
Ejercicio resuelto ... 260
Ejercicios propuestos ... 269

Capítulo 10. Métodos de tratamiento de archivos ... 270

10.1. Introducción ... 270
10.2. Búsqueda en archivos secuenciales ... 270

10.2 1. Búsqueda en archivos desordenados ... 271
10.2 2. Búsqueda en archivos ordenados ... 271
10.3. Partición de archivos ... 272
10.3.1. Partición por contenido ... 272
10.3.2. Partición en secuencias ... 273
10.4. Mezcla de archivos ... 274
10.4.1. Mezcla con registro centinela ... 274
10.4 2. Mezcla controlada por valor de clave máximo ... 275
10.4.3. Mezcla controlada por fin de archivo ... 275
10.5. Clasificación de archivos ... 277
10.5 1. Clasificación por mezcla directa ... 277
10.5.2. Clasificación por mezcla equilibrada ... 278
10 5 3. Clasificación de raiz ... 279
10.6 Actualización de archivos ... 279
10.6.1. Actualización de archivos secuenciales ... 280
10 6 2. Actualización de archivos directos ... 283
10 6 3. Actualización de archivos indexados ... 285
10.6.4. Actualización interactiva ... 286
10.7. Rupturas de secuencia ... 289
10.8. Sincronización de archivos ... 292
Ejercicios resueltos ... 295
Ejercicios propuestos ... 302

Capitulo 11. Estructuras de datos dinámicas ... 304

11.1. Introducción ... 304
11.2. Punteros y variables dinámicas ... 304
11.3. Listas ... 306
11.3.1 Listas densas ... 307
11.3.2. Listas enlazadas ... 311
11.4. Pilas ... 314
11 4 1. Implementación mediante tablas ... 314
11.4.2. Implementación mediante punteros ... 317
11.5. Colas ... 318
11.5.1. Implementación mediante tablas circulares ... 318
11.5.2. Implementación mediante punteros ... 320
11.6. Arboles ... 321
11.6.1. Conceptos y definiciones ... 321
11.6.2. Arboles binarios ... 323
11.6.3. Representación mediante tablas ... 324
11.6.4 Representación mediante punteros ... 324
11.6.5. Transformación de árboles n-arios en binarios ... 324
11.6.6. Recorridos de un árbol binario ... 325
11.6.7. Arboles binarios de búsqueda ... 326
Ejercicio resuelto ... 326
Ejercicios propuestos ... 331

Capitulo 12. Tablas de decisión ... 333

12.1. Introducción ... 333
12.2. Estructura de una tabla de decisión ... 333
12.3. Tipos de reglas ... 335
12.4. Clasificación de las tablas de decisión ... 336
12.4.1. Tablas de decisión binarias ... 336
12.4.2. Tablas de decisión múltiples ... 337
12.4.3. Tablas de decisión mixtas ... 337
12.5. Proceso de resolución de una tabla de decisión ... 337
12.5.1. Redundancias ... 338
12.5.2. Completitud ... 338
12.5.3. Simplificación ... 339
12.5.4. Ordenación por importancias ... 341

12.6. Paso a ordinograma ... 342
Ejercicios resueltos ... 348
Ejercicios propuestos ... 356

Capítulo 13. Metodologías de desarrollo de programas ... 359

13.1. Introducción ... 359
13.2. Metodología Jackson ... 359
13.2.1. Introducción ... 359
13.2.2. Fundamentos básicos. Entidades ... 361
13.2.3. Fases de la Metodología Jackson ... 363
13.2.4. Errores e invalidez ... 377
13.2.5. Collating ... 378
13.2.6. Backtracking ... 380
13.2.7. Colisión de estructuras ... 380
13.2.8. Inversión de programas ... 381
13.3. Metodología Warnier ... 387
13.3.1. Introducción ... 387
13.3.2. Estructuras básicas ... 388
13.3.3. Fases de la metodología Warnier ... 391
13.3.4. Estudio de los datos de salida ... 391
13.3.5. Estudio de los datos de entrada ... 392
13.3.6. Cuadro de descomposición de secuencias ... 392
13.3.7. Organigrama de secuencias de Warnier ... 393
13.3.8. Lista de instrucciones y asignación de las mismas ... 394
13.4. Otras metodologías ... 401
Ejercicios resueltos ... 402
Ejercicios propuestos ... 409

Bibliografía ... 411

Indice analítico ... 413

Prólogo

Las necesidades actuales de profesionales en el campo de la Informática han sido abordadas por los estamentos docentes públicos y privados desde diferentes enfoques con diversos planes de estudios.

El área específica de la Programación de Computadoras es fundamental en todos ellos, y de su correcto aprendizaje dependerá, en gran medida, el que las computadoras cumplan la misión para la que han sido creadas de una forma eficaz.

Este trabajo trata de cubrir el vacío existente de un texto adecuado para la disciplina de la Programación de Computadoras en las enseñanzas medias, así como en otros niveles educativos.

Compendia los aspectos principales que debe conocer (e incluso llegar a dominar) una persona que quiera ser programador profesional, y es útil también para aquellos que, como complemento a otras actividades, desean usar la computadora como herramienta auxiliar de su trabajo.

Como punto de partida se ha utilizado el cuestionario oficial de la asignatura Metodología de la Programación, correspondiente a la especialidad de Informática de Gestión de los estudios de Formación Profesional de Segundo Grado. No obstante, se ha superado dicho cuestionario con aspectos importantes que no figuran en el mismo, lo cual hace posible su utilidad fuera del ámbito descrito.

Asimismo se ha comprobado que en esta obra queda contemplada la programación de esta asignatura en la reforma de las enseñanzas medias.

Se da gran importancia a la presentación teórica, de forma intuitiva, de los conceptos de Programación. Para su más fácil comprensión éstos se acompañan con ejemplos de aplicación y con programas resueltos y codificados en dos lenguajes de amplia difusión en el campo informático.

El libro va dirigido a:

— Alumnos de la especialidad de Informática de Gestión.
— Alumnos de enseñanzas de Informática no regladas.
— Alumnos de enseñanza universitaria con asignaturas de programación de computadoras.
— Profesionales y aficionados a la programación.

A lo largo del libro se presentan los elementos básicos de los lenguajes de programación, las principales estructuras de datos y los métodos y técnicas para el desarrollo de programas que cumplan unos ciertos requisitos de calidad, haciendo hincapié en la necesidad de imponer una serie de reglas o normas de diseño, principalmente las de la programación estructurada.

La utilización de un lenguaje de programación concreto para conseguir el objetivo fijado se ha demostrado inadecuada por las particularidades que presenta cada uno de los lenguajes. Por este motivo hemos optado por usar dos notaciones algorítmicas independientes de cualquier lenguaje de programación: el *pseudocódigo*, que surge de una forma natural al intentar describir la solución de un problema, y los *diagramas de flujo*, que presentan de forma gráfica la solución y la hacen fácil de comprender.

Por otra parte, los algoritmos, una vez diseñados, se codifican en los lenguajes COBOL y Pascal, con el fin de poder simultanear su estudio con el de la programación en general.

El estudiante que no esté interesado en alguno de ellos puede omitir su estudio sin que ello represente limitación alguna en cuanto al aprendizaje de esta disciplina.

Finalmente, no puede quedar sin aclarar la evidente ausencia a lo largo del libro de los términos «ordenador» y «fichero», los cuales han sido sustituidos por sus sinónimos «computadora» y «archivo», respectivamente. Esto es debido a nuestra aceptación de las recomendaciones de la Editorial, en el sentido de que los términos mencionados en primer lugar son extraños a la mayoría de los lectores hispanoamericanos. Esperamos que dicha sustitución tenga buena acogida entre ellos, y confiamos asimismo en la disculpa de aquellos otros lectores acostumbrados a su uso, en la convicción de que no les representará ninguna dificultad adicional.

Prólogo a la Segunda Edición

Desde la aparición de la primera edición del presente libro, en 1987, hasta la actualidad, la Programación de Computadoras ha continuado su rápida evolución en métodos y técnicas de diseño con el fin de facilitar el trabajo del programador y conseguir una mayor sencillez y claridad en el producto obtenido por él.

Aunque el libro ha gozado de una muy amplia aceptación por parte de profesores y estudiantes, no obstante hemos recibido algunas sugerencias sobre la utilidad de ampliar algunos puntos concretos que no podemos dejar de tomar en consideración.

Por otro lado se está procediendo a la implantación de un nuevo sistema educativo en todos los niveles y en particular en las enseñanzas en las que se ubica la asignatura Metodología de la Programación.

Teniendo en cuenta todo lo expuesto hemos considerado la conveniencia de sacar a la luz esta segunda edición para reflejar las últimas innovaciones sucedidas, dar cabida a las sugerencias recibidas y adaptarlo a los nuevos planes de estudio.

En esta edición se ha procurado mantener la mayor parte de su contenido original, por considerar que cumple los objetivos inicialmente previstos, y así lo han manifestado la mayoría de lectores.

Hemos añadido dos nuevos capítulos de Estructuras de Datos Dinámicas y Metodologías de Desarrollo de Programas; también hemos ampliado algunos otros capítulos, en particular el referente a Tratamiento de Archivos.

Al igual que en la primera edición, la solución a los ejercicios y programas propuestos se realiza en varias notaciones: diagramas de flujo, pseucodódigo, COBOL, Pascal, pero seleccionando sólo algunas de ellas en cada caso.

Los compiladores utilizados para la prueba de estos programas son RM/COBOL-85, de Ryan-McFarland Co., y Turbo Pascal, de Borland.

Asimismo hemos decidido suprimir el lenguaje BASIC en las soluciones de los programas por creer que prácticamente ha dejado de utilizarse en nuestro ámbito.

El libro queda estructurado de la siguiente manera:

En el capítulo primero se introducen las generalidades y aspectos básicos de la Programación. El segundo expone la notación de Diagramas de Flujo, incluyéndose la normalización española. El tercero incorpora las principales herramientas utilizadas en los programas. En el capítulo cuarto se presenta el Pseudocódigo como notación útil para diseñar y escribir programas y la traducción a los lenguajes COBOL y Pascal. Los capítulos quinto y sexto tratan de la estructura de datos Tabla y de su manejo. El séptimo introduce al Diseño Descendente de Programas y a la Programación Estructurada y Modular. En el octavo se presentan las bases teóricas de la Programación Estructurada y sus principales métodos de representación. Los capítulos noveno y décimo tratan de las Estructuras de Archivo y los principales métodos de tratamiento de los mismos. El undécimo presenta las Estructuras de Datos Dinámicas más importantes y su implementación. El duodécimo aborda extensamente las Tablas de Decisión y su transformación en programas. El decimotercero y último introduce las Metodologías de Desarrollo de Programas, presentando las dos más difundidas.

Finalmente, no podemos dejar de dar las gracias a tantos compañeros que con su estímulo han hecho posible este trabajo. Igualmente hemos de agradecer el gran interés y buen hacer de la editorial McGraw-Hill desde la aparición de la primera edición y con ella el comienzo de la serie «Informática de Gestión», especialmente a Wenceslao Ortega, su continuo ánimo y apoyo.

Un agradecimiento singular debemos a nuestro editor, amigo y compañero Teodoro Bartolomé, que ha compartido los buenos y malos momentos en el desarrollo de este trabajo. Gracias.

Abril de 1992.

LOS AUTORES

CAPITULO 1

La programación de computadoras

1.1. INTRODUCCION

Muchas personas piensan que una computadora puede realizar tareas o trabajos de complejidad superior a una inteligencia humana. La realidad es que una computadora no tiene ninguna inteligencia. No olvidemos que no es más que una máquina creada por el hombre y, por tanto, no podrá realizar una tarea que no haya sido previamente determinada por él.

Una **computadora (ordenador)** es una máquina de origen electrónico que puede realizar una gran variedad de trabajos, pero, en principio, sólo es capaz de hacer físicamente tres clases de operaciones básicas:

1. Sumar, restar, multiplicar y dividir dos valores numéricos, es decir, realizar operaciones aritméticas sencillas.
2. Comparar dos valores (comprobar si son iguales, si el primero es mayor que el segundo, etc.), es decir, realizar operaciones lógicas sencillas.
3. Almacenar o recuperar información.

Con estas pocas operaciones utilizadas y combinadas de forma adecuada, mediante lo que llamamos programa, se pueden llegar a realizar tareas increiblemente complejas que aporten la solución a un determinado problema, ya sea de gestión, técnico o de cualquier otro tipo.

La potencia de cálculo de una computadora se deriva de las características físicas que posee, entre las que se encuentran principalmente:

RAPIDEZ | PRECISION | MEMORIA

Las características citadas provienen de los componentes electrónicos que conforman una computadora:

- Velocidad de conmutación de circuitos electrónicos.
- Rapidez de transmisión de señales eléctricas.
- Fiabilidad de los circuitos.
- Manejo de señales digitales.
- Gran capacidad de almacenamiento de información.

Nuestro objetivo es, para un problema dado, diseñar una solución que pueda ser realizada por una computadora. Para ello necesitaremos, en primer lugar, un lenguaje o notación para expresar la solución obtenida. Tal solución debe estar adaptada a las particularidades de la computadora, si bien en una primera fase de su diseño podremos utilizar una notación intermedia entre el lenguaje natural y el de la computadora, posteriormente será preciso escribirla en un lenguaje comprensible por la máquina, como, por ejemplo, en COBOL o Pascal, y, finalmente para su ejecución, la máquina precisará realizar una traducción a su lenguaje nativo, el denominado lenguaje máquina.

1.2. CICLO DE VIDA DE UNA APLICACION INFORMATICA

Una **aplicación informática** se compone de uno o varios programas interrelacionados que tienen por objeto la realización de una determinada tarea de forma automática mediante el uso de un sistema informático.

El proceso que se sigue desde el planteamiento de un problema o tarea hasta que se tiene una solución instalada en la computadora, y en funcionamiento por los usuarios finales mientras sea de utilidad, se denomina **ciclo de vida de una aplicación informática**.

El citado proceso se compone de varias fases, agrupadas en dos bloques bien diferenciados, según la Figura 1.1, en la que se muestran las distintas fases y el resultado obtenido de cada una de ellas.

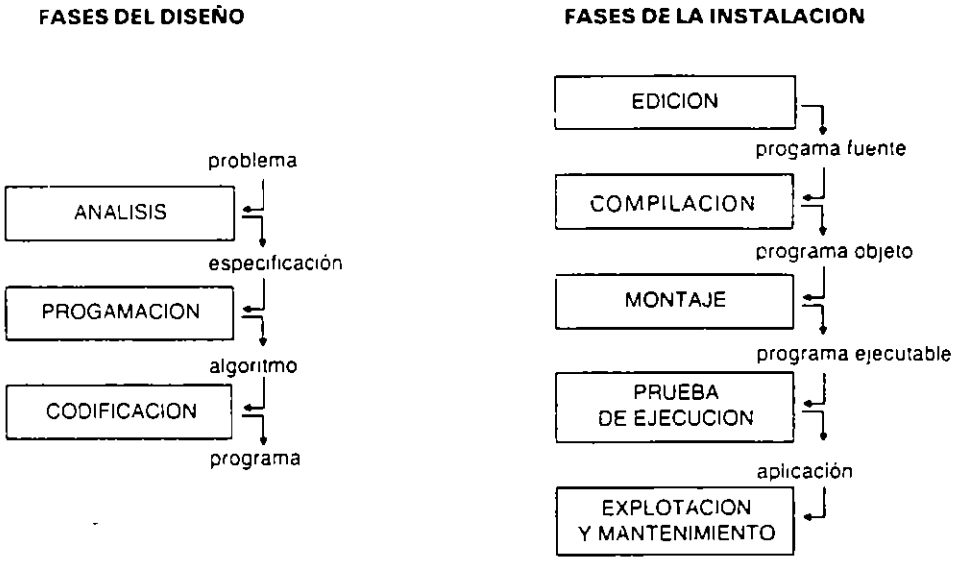


Figura 1.1. Ciclo de vida de una aplicación informática.

1.2.1. DISEÑO DEL PROGRAMA

Este apartado incluye las fases correspondientes a la creación del programa.

Es de destacar el hecho de que para la realización de esta labor no se necesita usar la computadora.

■ **Fase de análisis**

Consiste en el examen y descripción detallada de los siguientes aspectos relativos al problema:

- Equipo a utilizar (computadora, periféricos, soportes, material auxiliar, etc.).
- Personal informático.
- Estudio de los datos de entrada (INPUT).
- Estudio de los datos de salida o resultados (OUTPUT).
- Relación entre la salida y la entrada.
- Descomposición del problema en módulos.

El resultado de esta fase es lo que se denomina **especificación** del problema, formada por el conjunto de documentos elaborados para los aspectos citados.

■ **Fase de programación**

Consiste en el diseño de la solución al problema planteado en forma de algoritmo.

Se debe abordar de forma sistemática, esto es, aplicando explícitamente un conjunto de técnicas, como programación estructurada y diseño descendente.

El resultado es un **algoritmo** o descripción del conjunto de acciones que deberán ser realizadas por la computadora.

Para su representación se utilizan diferentes notaciones, como ordinogramas, pseudocódigos, etc.

■ **Fase de codificación**

En esta fase se transcribe el algoritmo resultante de la fase anterior a un lenguaje de programación concreto. En general es una tarea totalmente mecánica.

Al resultado se le denomina **programa**, que normalmente será escrito en **hojas de codificación** (Figura 1.2).

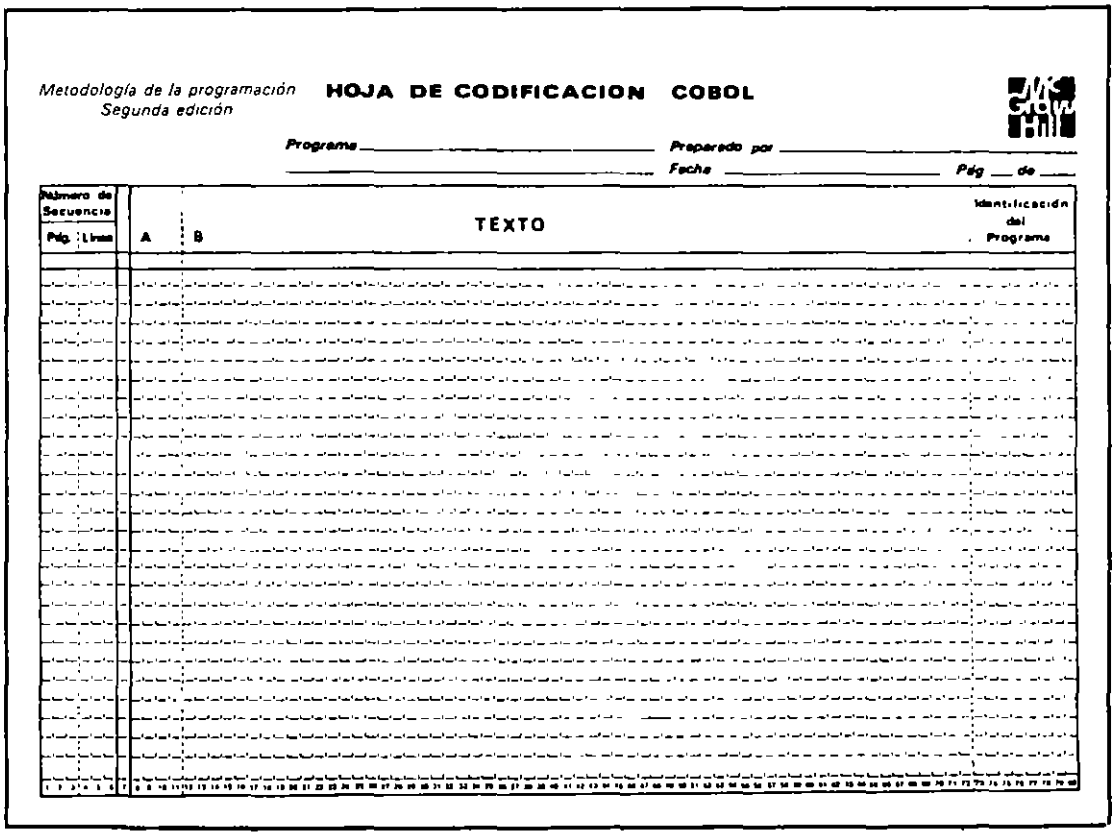


Figura 1.2. Hoja de codificación.

1.2.2. INSTALACION Y EXPLOTACION DEL PROGRAMA

Se engloban en este apartado las fases correspondientes a la instalación y puesta a punto del programa en la computadora.

■ **Fase de edición**

Escritura del **programa fuente** a partir de las hojas de codificación en la memoria de la computadora, grabándolo en algún soporte permanente. Se hace con la ayuda de un programa del sistema denominado **editor**.

■ **Fase de compilación**

Traducción del programa fuente a lenguaje máquina cuyo resultado es el **programa objeto**. Para ello se dispone de programas **compiladores** o **intérpretes**, que, además, comprueban la correcta sintaxis del programa

■ **Fase de montaje**

En los programas compilados es necesario añadir al programa objeto algunas rutinas del sistema o en algunos casos subprogramas externos que se hayan compilado separadamente. De ello se encarga el programa **montador** (linker).

■ **Fase de prueba de ejecución**

Consiste en ejecutar el programa sucesivas veces con diferentes **datos de prueba** para asegurar su correcto funcionamiento.

■ **Fase de explotación y mantenimiento**

La explotación consiste en el uso continuo y habitual por parte de los usuarios de la aplicación dentro de un entorno productivo determinado mientras tenga utilidad.

Paralelamente a la explotación de una aplicación se realiza el mantenimiento de la misma, consistente en la comprobación periódica de su buen funcionamiento y en la adaptación a cualquier nueva circunstancia que implique su actualización.

1.3. ERRORES

Durante el desarrollo de un programa o aplicación se ha de ser especialmente cuidadoso para evitar que el producto obtenido presente errores que lo hagan inservible. En muchos casos, sobre todo en programadores principiantes, se dedica más tiempo a la corrección de errores que al diseño del programa en sí.

Si se ha seguido un buen método de diseño, no ha de preocuparnos excesivamente la presencia de errores: lo importante será utilizar una técnica adecuada de depuración que nos permita eliminarlos con facilidad.

Según el momento o fase en que se detectan, los errores se clasifican de la siguiente manera:

1.3.1. ERRORES DE COMPILACION

Los errores en tiempo de compilación, o errores sintácticos, corresponden al incumplimiento de las reglas sintácticas del lenguaje, como, por ejemplo, una palabra reservada del lenguaje mal escrita, una instrucción incompleta, etc.

Estos errores son los más fáciles de corregir, ya que son detectados por el compilador, el cual dará información sobre el lugar donde está y la naturaleza de cada uno de ellos mediante un mensaje de error.

1.3.2. ERRORES DE EJECUCION

Se deben generalmente a operaciones no permitidas, como dividir por cero, leer un dato no numérico en una variable numérica, exceder un rango de valores permitidos, etc.

Se detectan porque se produce una parada anormal del programa durante su ejecución, y se dice entonces que el programa se ha «caído» o que ha sido «abortado» por el sistema.

Son más difíciles de detectar y corregir que los errores sintácticos, ya que ocurren o no, dependiendo de los datos de entrada que se utilicen.

1.3.3. ERRORES DE LOGICA

Corresponden a la obtención por el programa de resultados que no son correctos, y la única manera de detectarlos es realizando un número suficiente de ejecuciones de prueba con una gama lo más amplia posible de juegos de datos de ensayo, comparando los resultados producidos por el programa con los obtenidos «a mano» para esos mismos datos.

Son los más difíciles de corregir, no sólo por la dificultad para detectarlos, sino porque se deben a la propia concepción y diseño del programa.

1.3.4. ERRORES DE ESPECIFICACION

Es posiblemente el peor tipo de error y el más costoso de corregir. Se deben a la realización de unas especificaciones incorrectas motivadas por una mala comunicación entre el programador y quien plantea el problema (cliente, profesor, etc.).

Se detectan normalmente cuando ya ha concluido el diseño e instalación del programa, por lo que su corrección puede suponer la repetición de gran parte del trabajo realizado.

1.4. CALIDAD DE LOS PROGRAMAS

Para un determinado problema se pueden construir diferentes algoritmos de resolución o programas. La elección del más adecuado se debe basar en una serie de requisitos de calidad que adquieren gran importancia a la hora de evaluar el coste de su diseño y mantenimiento.

Las características generales que debe reunir un programa son las siguientes:

- **Legibilidad**

  Ha de ser claro y sencillo, de tal forma que facilite su lectura y comprensión.

- **Fiabilidad**

  Ha de ser «robusto», es decir, capaz de recuperarse frente a errores o usos inadecuados.

- **Portabilidad**

  Su diseño debe permitir la codificación en diferentes lenguajes de programación, así como su instalación en diferentes sistemas.

- **Modificabilidad**

  Ha de facilitar su mantenimiento, esto es, las modificaciones y actualizaciones necesarias para adaptarlo a una nueva situación.

- **Eficiencia**

  Se deben aprovechar al máximo los recursos de la computadora, minimizando la memoria utilizada y el tiempo de proceso o ejecución, siempre que no sea a costa de los requisitos anteriores.

1.5. METODOLOGIA DE LA PROGRAMACION

Es el conjunto de métodos y técnicas disciplinadas que ayudan al desarrollo de unos programas que cumplan los requisitos anteriormente expuestos.

Los métodos propuestos utilizados en este libro, tanto explícita como implícitamente, son el método de **programación modular** y el de **programación estructurada**.

1.5.1. PROGRAMACION MODULAR

El diseño modular, descendente o mediante refinamientos sucesivos (**top-down, stepwise refinement**) se basa en la realización de una serie de descomposiciones sucesivas del algoritmo inicial, que describen el refinamiento progresivo del repertorio de instrucciones que van a constituir el programa.

Un programa quedará formado por una serie de **módulos**, cada uno de los cuales realiza una parte concreta de la tarea total.

1.5.2. PROGRAMACION ESTRUCTURADA

Se basa en el uso exclusivo de las estructuras **secuencia**, **alternativa** e **iteración** para el control del flujo de ejecución de las instrucciones.

Los programas así diseñados serán fáciles de verificar, depurar y mantener.

1.6. DOCUMENTACION DE LOS PROGRAMAS

Con el fin de facilitar la explotación y el mantenimiento de un programa es fundamental que éste se acompañe de una documentación amplia, clara y precisa. En ella deben figurar desde las especificaciones obtenidas de la fase de análisis del problema hasta los detalles acerca de cómo sacar el máximo rendimiento del mismo.

Existen dos clases de documentación según su ubicación: interna y externa.

1.6.1. DOCUMENTACION INTERNA

Constituida básicamente por el listado del programa fuente, su principal objetivo es facilitar la lectura y comprensión del mismo.

Se consideran parte de la documentación interna los siguientes aspectos:

- **Comentarios**

Son frases explicativas que se insertan en cualquier lugar del programa fuente y que son ignoradas por el compilador (no son traducidas a lenguaje objeto y, por tanto, no ocupan memoria adicional).

Se deben incluir tantos comentarios como sean necesarios para aclarar el significado de las líneas de código que no sean obvias, especialmente respecto a:

- Las variables y estructuras de datos declaradas.
- Las estructuras de control (bucles, alternativas).
- Los subprogramas y sus parámetros.
- Las secciones confusas.

■ **Código autodocumentado**

Las palabras reservadas que emplean los lenguajes de alto nivel constituyen en sí mismas parte de la documentación, ya que corresponden a términos (en inglés) que expresan su cometido. No obstante se mejora la documentación interna de un programa si se siguen los criterios enumerados a continuación:

- Uso de identificadores adecuados para nombrar las variables, constantes, subprogramas, etc.
- Declaración de constantes para valores fijos. Por ejemplo, declarar la constante IVA para el valor 0,13.
- Sangrado, paginación e intercalación de líneas en blanco para dar un aspecto agradable al programa.

1.6.2. DOCUMENTACION EXTERNA

Es el conjunto de documentos que se acompañan con el programa pero sin formar parte de él.

La documentación externa debe incluir al menos los siguientes apartados:

- Especificaciones del análisis.
- Descripción del diseño del programa.
- Descripción de las versiones si las hubiere.
- Descripción del programa principal y subprogramas.
- Manual de usuario.
- Manual de mantenimiento.

1.7. OBJETOS DE UN PROGRAMA

Son objetos de un programa todos aquellos manipulados por las instrucciones. Mediante ellos, en un programa podremos realizar el almacenamiento de los datos y de los resultados de las distintas operaciones que intervienen en la solución del problema.

Todo objeto tiene tres atributos:

- **Nombre:** Es el identificador del mismo.
- **Tipo:** Conjunto de valores que puede tomar.
- **Valor:** Elemento del tipo que se le asigna.

1.8. IDENTIFICADORES

Son palabras creadas por el programador para dar nombre a los objetos y demás elementos que necesita declarar en un programa: variables, constantes, tipos, estructuras de datos, archivos, subprogramas, etc.

En general se utiliza una cadena de letras y dígitos que empiece por letra. En COBOL se permite utilizar guiones intermedios, y, en Pascal, el carácter de subrayado.

Ejemplo:

```
X
PI
CURSO91
ESTADO-CIVIL (en COBOL)
ESTADO_CIVIL (en Pascal)
```

1.9. TIPOS DE DATOS

Se denomina datos a las caracteristicas propias de cualquier entidad. Por ejemplo, la edad y el domicilio de una persona forman parte de sus datos.

Los programas procesan datos a fin de obtener resultados o informaciones útiles.

Cada variable, constante o expresión lleva asociado un tipo de datos que determina el conjunto de valores que puede tomar.

Los tipos de datos pueden ser **simples** o elementales y **estructuras de datos** o estructurados (Figura 1.3).

A continuación estudiaremos los tipos simples, dejando las estructuras de datos para capitulos posteriores.

En una primera clasificación, los tipos simples se dividen en numéricos y no numéricos. En los primeros se incluyen las cantidades o magnitudes, y, en los segundos, el resto de datos posibles.

Una clasificación más detallada es la expuesta a continuación.

1.9.1. TIPO NUMERICO ENTERO

Es un subconjunto de los números enteros cuyo rango o tamaño dependen del lenguaje y computadora utilizada.

Los datos de este tipo se expresan mediante una cadena de dígitos que puede ir precedida de signo (+ o −).

Ejemplos:

```
1987
-12
+3300
```

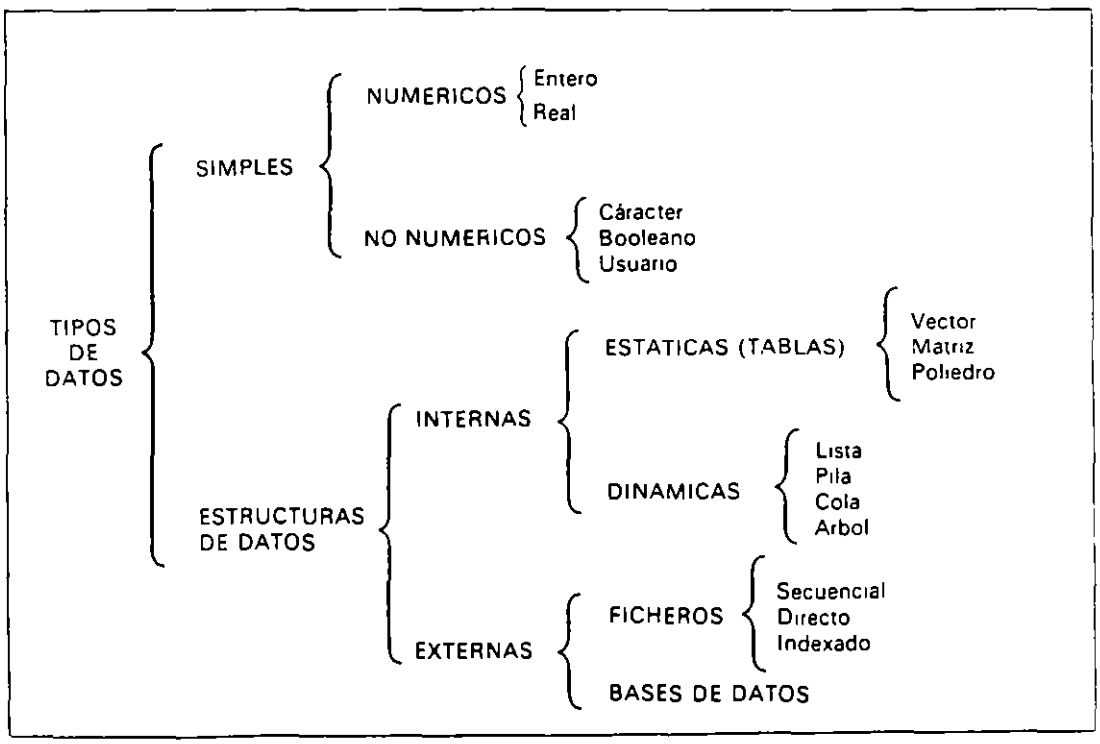


Figura 1.3. Tipos de datos.

1.9.2. TIPO NUMERICO REAL

Es un subconjunto de los números reales limitado no sólo en cuanto al tamaño, sino también en cuanto a la precisión.

Se expresan de dos maneras diferentes denominadas notación de punto fijo y notación exponencial. En la primera, un valor consiste en una cadena de dígitos que puede llevar signo y un punto decimal intermedio.

Ejemplos:

```
 97.84
-12.00
 +0.5
```

Un valor en notación exponencial tiene la forma «mantisa E exponente», donde «mantisa» es un número real y «exponente» un número entero y representa la cantidad «mantisa multiplicado por 10 elevado al exponente».

Ejemplos:

```
  0.9784E2  (=0.9784x10^2=97.84)
-120000E-4
+0.0005E+3
```

1.9.3. TIPO CARACTER

Es el conjunto formado por todos los caracteres o símbolos de que dispone la computadora.

Se expresan mediante el carácter colocado entre comillas o apóstrofes.

El conjunto de los caracteres está formado por:

- Los caracteres alfabéticos mayúsculas
  `"A", "B", "C", "D", ..., "Z"`
- Los caracteres alfabéticos minúsculas
  `"a", "b", "c", "d", ..., "z"`
- Los caracteres dígitos
  `"0", "1", "2", "3", ..., "9"`
- Los caracteres especiales
  `" "  (espacio o carácter blanco)`
  `"+", "-", "*", "/", "=", "<", ">", ...`
  `".", ",", ":", ";", "(", ")", "!", "?", ...`

En algunos lenguajes se considera también como tipo de datos simples el de los **literales** o cadenas de caracteres de longitud variable (**string** en inglés):

Ejemplos:

```
"ALCALA"
"28035"
"C/ Pez, núm. 12"
```

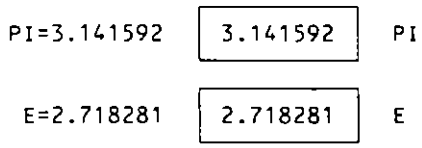
1.9.4. TIPO BOOLEANO

Es el conjunto formado por los valores FALSO y CIERTO. Se expresan con su nombre.

1.10. CONSTANTES

Son objetos cuyo valor permanece invariable a lo largo de la ejecución de un programa. Una constante es la denominación de un valor concreto, de tal forma que se utiliza su nombre cada vez que se necesita referenciarlo.

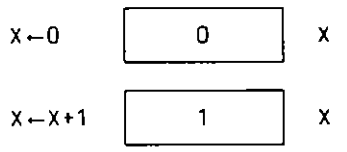
Ejemplos:



1.11. VARIABLES

Son objetos cuyo valor puede ser modificado a lo largo de la ejecución de un programa.

Ejemplos:



X es una variable de tipo numérico.

1.12. EXPRESIONES

Una expresión es la representación de un cálculo necesario para la obtención de un resultado.

Se define una expresión de la siguiente forma:

1. Un valor es una expresión.
   Ejemplos: `1.25 , "JUAN".`
2. Una constante o variable es una expresión.
   Ejemplos: `PI , E , X .`
3. Una función es una expresión.
   Ejemplos: `COS(X) , SQR(25) .`
4. Una combinación de valores, constantes, variables, funciones y operadores que cumplen determinadas reglas de construcción es una expresión.
   Ejemplos:
   ```
   COS(PI * X) + 1.25
   2 * PI * X
   N = "JUAN"
   ```

1.12.1. TIPOS DE EXPRESIONES

Las expresiones, según el resultado que producen, se clasifican en:

■ **Numéricas.**—Son las que producen resultados de tipo numérico. Se construyen mediante los operadores aritméticos.

Ejemplo: `PI * SQR(X)`

■ **Alfanuméricas.**—Son las que producen resultados de tipo alfanumérico. Se construyen mediante los operadores alfanuméricos.

Ejemplo: `"Don " + N`

■ **Booleanas.**—Son las que producen resultados CIERTO o FALSO. Se construyen mediante los operadores relacionales y lógicos.

Ejemplo: `A > 0 y B <= 5`

1.12.2. OPERADORES

Para la construcción de expresiones se pueden utilizar, de forma general, los siguientes operadores:

■ **Aritméticos:**

^	Potencia
*	Producto
/	División
DIV o \	División entera
MOD	Resto de la división entera
+	Suma o signo positivo
-	Resta o signo negativo

■ **Alfanuméricos:**

+	Concatenación

■ **Relacionales:**

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

■ **Lógicos:**

no	Negación
y	Conjunción
o	Disyunción

■ **Paréntesis:**

( )	Se utilizan para anidar expresiones

1.12.3. TABLAS DE VERDAD DE LOS OPERADORES LOGICOS

El resultado de las operaciones lógicas está determinado por las tablas de verdad correspondientes a cada una de ellas.

■ **Operador NO:**

A	no A
F	C
C	F

Siendo:
A una expresión booleana.
F valor FALSO.
C valor CIERTO.

■ **Operador Y:**

A	B	A y B
F	F	F
F	C	F
C	F	F
C	C	C

Siendo A y B expresiones booleanas

■ **Operador O:**

A	B	A o B
F	F	F
F	C	C
C	F	C
C	C	C

Siendo A y B expresiones booleanas

1.12.4. ORDEN DE EVALUACION DE LOS OPERADORES

Los operadores de una expresión se evalúan, en general, según el siguiente orden:

1.° Paréntesis (comenzando por los más internos).
2.° Signo.
3.° Potencias.
4.° Productos y divisiones.
5.° Sumas y restas.
6.° Concatenación.
7.° Relacionales.
8.° Negación.
9.° Conjunción.
10.° Disyunción.

La evaluación de operadores de igual orden se realiza siempre de izquierda a derecha
Este orden de evaluación tiene algunas modificaciones en determinados lenguajes de programación.

Ejemplos: Evaluar las siguientes expresiones:

A:

```
((3 + 2) ^ 2 - 15) / 2 * 5
--------
      (5 ^ 2 - 15) / 2 * 5
      ------
         (25 - 15) / 2 * 5
         ---------
                10 / 2 * 5
                ------
                     5 * 5
                     -----
                       25
```

B:

```
5 - 2 > 4 y no 0.5 = 1 / 2
                    -----
5 - 2 > 4 y no 0.5 = 0.5
-----
    3 > 4 y no 0.5 = 0.5
    -----
    FALSO y no 0.5 = 0.5
              ---------
    FALSO y no CIERTO
             ---------
    FALSO y FALSO
    -------------
    FALSO
```

EJERCICIOS RESUELTOS

1. *Dadas las siguientes variables y constantes:*

1	4	10	3.141592	2.718281
X	Y	Z	PI	E

Evaluar las expresiones:

1.
```
2 * X + 0.5 * Y - 1 / 5 * Z
2 + 0.5 * Y - 1 / 5 * Z
2 + 2 - 1 / 5 * Z
2 + - 0.2 * Z
2 + 2 - 2
4 - 2
2
```

2.
```
PI * X ^ 2 > Y o 2 * PI * X <= Z
PI * 1 > Y o 2 * PI * X <= Z
3.141592 > Y o 2 * PI * X <= Z
3.141592 > Y o 6.283184 * X <= Z
3.141592 > Y o 6.283184 <= Z
FALSO o 6.283184 <= Z
FALSO o CIERTO
CIERTO
```

3.
```
E ^ (X - 1) / (X * Z) / (X / Z)
E ^ 0 / (X * Z) / (X / Z)
E ^ 0 / 10 / (X / Z)
E ^ 0 / 10 / 0.1
1 / 10 / 0.1
0.1 / 0.1
1
```

4.
```
"DON " + "JUAN" = "DON JUAN" o "A" = "a"
"DON JUAN" = "DON JUAN" o "A" = "a"
CIERTO o "A" = "a"
CIERTO o FALSO
CIERTO
```

2. *Construir expresiones correctas para las siguientes:*

   1. $ax^2 + bx + c \geq 0$

      ```
      A * X ^ 2 + B * X + C >= 0
      ```

   2. $\dfrac{-b + \sqrt{b^2 - 4ac}}{2a}$

      ```
      (-B + (B ^ 2 - 4 * A * C) ^ 0.5) / (2 * A)
      ```

   3. $\dfrac{A - B}{X} + \dfrac{C - D}{Y} > \dfrac{Y}{X}$

      ```
      (A - B) / X + (C - D) / Y > Y / X
      ```

   4. *A* es igual a *B*, pero no es igual a *C*

      ```
      A = B y no (A = C)
      ```

   5. Con DN, MN, AN día, mes y año de nacimiento de una persona y DA, MA, AA días, mes y año actual. Expresar si tiene 18 años cumplidos.

      ```
      AA - AN > 18 o AA - AN = 18 y MA > MN o AA - AN = 18 y MA = MN y DA >= DN
      ```

EJERCICIOS PROPUESTOS

1. Dadas las variables X, Y, Z y las constantes PI, E anteriores, evaluar las expresiones:

   1. `X - Y + Z + PI - E + 2.576689`
   2. `- 3 * X + 2 * Y - 1 / 2 * Z`
   3. `2 * Y ^ 2 - 6 * Y + 12`
   4. `( 2 * Y ) ^ 2 - 6 * Y + 12`
   5. `( Y ^ ( 2 * X ) - 6 * ( Z / 10 ) ) / 2`
   6. `X > 3 y Y = 4 o X + Y < = Z`
   7. `X > 3 y ( Y = 4 o X + Y < = Z )`
   8. `no "METODO" + "LOGIA" = "LOGIA" + "METODO"`
   9. `( - X + ( Z ^ 2 - 4 * X * Y ) ^ 0.5 ) / 2 / X`
   10. `no ( 2 * X = Y / 2 o ( PI - E ) * Z > Y )`
       `= no Y / 2 = 2 * X y no Y < ( PI - E ) * Z`

2. Construir las expresiones correctas para las siguientes:

   1. $ax^2 - by^2 \leq 0$

   2. $\dfrac{3x - y}{z} - \dfrac{2xy^2}{z - 1} + \dfrac{x}{y}$

   3. $\dfrac{A}{B - \dfrac{C}{D - \dfrac{E}{F - G}}} + \dfrac{H + I}{J + K}$

   4. $A + X \leq B < C + Z$

   5. Con DN, MN, AN día, mes y año de nacimiento de una persona y DA, MA AA día, mes y año actual. Expresar si cumple hoy 20 años.

CAPITULO 2

Diagramas de flujo

2.1. INTRODUCCION

En general, durante el diseño de un programa, y en sus fases de análisis y programación, surge la necesidad de utilizar una herramienta de diseño gráfico para la representación de los flujos de datos manipulados por el mismo, asi como la secuencia lógica de las operaciones que constituyen el algoritmo de resolución del problema para el que ha sido creado.

Toda representación gráfica, de cualquier tipo que sea, debe cumplir las siguientes cualidades:

- **Sencillez.** Un método gráfico de diseño de algoritmos debe permitir la construcción de éstos de manera fácil y sencilla.
- **Claridad.** Cuando un algoritmo representado por algún método gráfico necesita ser interpretado por otra persona distinta de la que lo diseñó, debe estar suficientemente claro para un fácil reconocimiento de todos sus elementos.
- **Normalización.** Tanto los diseñadores de programas como los usuarios que necesitan interpretar la documentación de éstos deben utilizar las mismas normas de construcción.
- **Flexibilidad.** Todo método gráfico de representación debe permitir, sin grandes dificultades, posteriores modificaciones de alguna de las partes de un algoritmo y la inserción de alguna nueva.

En el presente capítulo vamos a estudiar el primer método de diseño gráfico que se ha utilizado en los sistemas de tratamiento de la información, aunque en la actualidad su uso ha quedado limitado por la aparición de otros métodos de diseño estructurado más eficaces para la representación y documentación de programas. No obstante, consideramos que todo programador debe conocer esta técnica por ser la más básica, por la importancia que ha tenido y porque en la actualidad se siguen utilizando los diagramas de flujo de datos.

Este primer método de diseño se denomina **diagramas de flujo** (*Flowchart*), y engloba tanto a la representación gráfica de la circulación de datos e informaciones dentro de un programa (**organigrama**) como a la representación gráfica de la secuencia de operaciones que se han de realizar en el mismo (**ordinograma**).

Estas representaciones se corresponden con distintas fases del diseño de un programa. Aunque utilizan símbolos comunes, su significado en cada tipo de representación es distinto.

El orden de utilización de los distintos diagramas de flujo está representado en la Figura 2.1. En ella puede verse cómo los organigramas se utilizan en la fase de análisis (en el denominado análisis orgánico), y los ordinogramas son utilizados en la fase de programación para facilitar su posterior codificación en el lenguaje correspondiente.

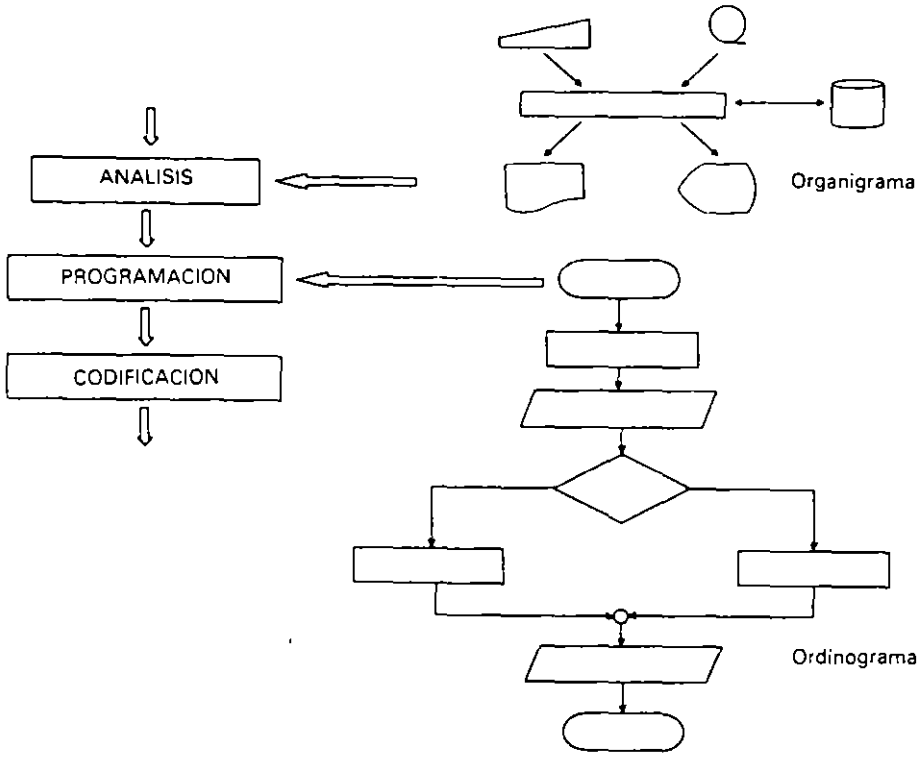


Figura 2.1. Correspondencia entre diagramas de flujo y fases del diseño.

2.2. DIAGRAMAS DE FLUJO DEL SISTEMA. ORGANIGRAMAS

Son representaciones gráficas del flujo de datos e informaciones que maneja un programa. En general, una aplicación se compone de más de un programa y es necesario realizar un organigrama por cada uno de ellos, siendo recomendable en estos casos la existencia de uno general que represente todo el movimiento de datos de la aplicación.

El conjunto del organigrama permitirá con facilidad la identificación de los siguientes elementos:

- Los soportes en que se encuentran los datos (símbolos de soporte).
- El programa y su identificación (rectángulo central).
- Los soportes donde se encontrarán los resultados (símbolos de soporte).
- El flujo de los datos (líneas de flujo).

Para la representación de un organigrama se deben seguir las siguientes reglas (Figura 2.2):

1. En el centro figurará el símbolo de proceso que representa al programa (rectángulo con el identificador del programa).
2. En la parte superior aparecerán los soportes que suministran los datos de entrada (símbolos con algún identificador de los datos).
3. En la parte inferior aparecerán los soportes que reciben los datos de salida (símbolos con algún identificador de los datos).
4. En las zonas de la derecha e izquierda aparecerán los soportes de los datos de entrada y salida (símbolos con algún identificador de los datos o archivos).

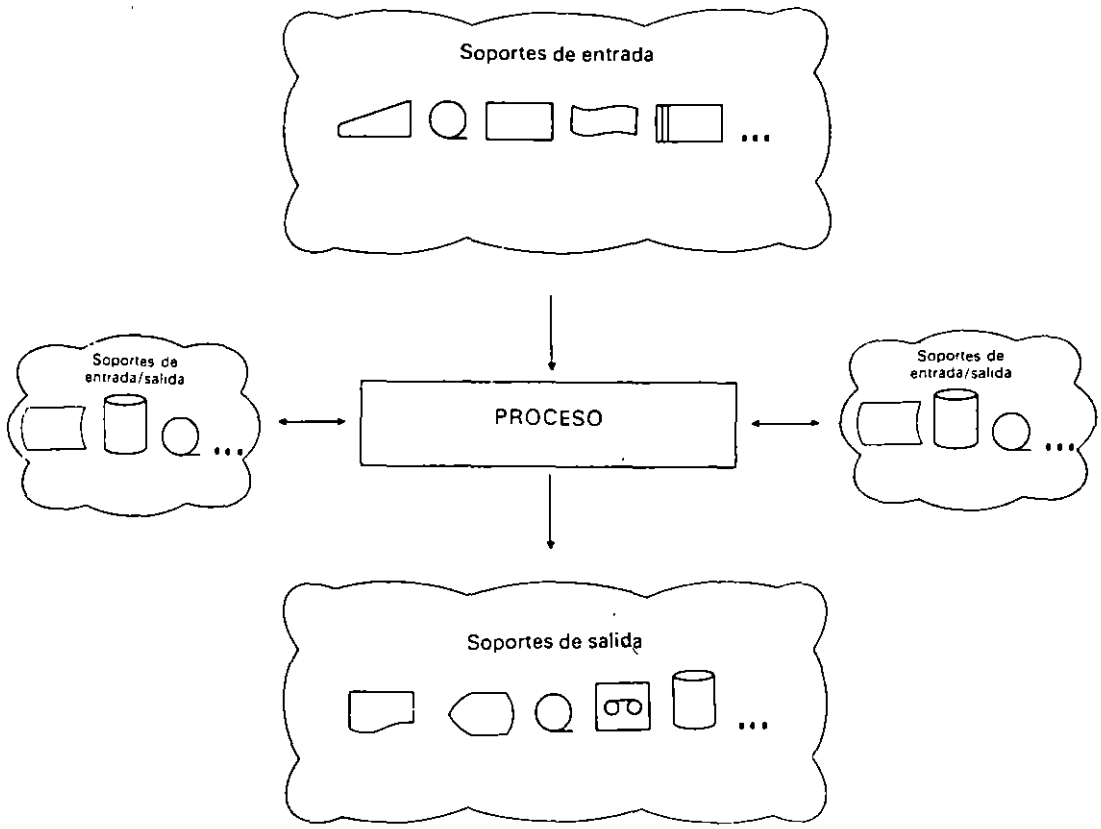


Figura 2.2. Diagrama de flujo del sistema (organigrama).

Estas reglas estarán a su vez supeditadas a una representación sencilla, clara y precisa.

Los símbolos que se utilizan en la confección de organigramas se agrupan en tres bloques:

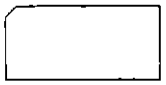
- Símbolos de soporte.
- Símbolos de proceso.
- Líneas de flujo.

2.2.1. SIMBOLOS DE SOPORTE

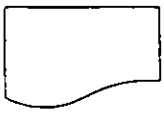
Representan los soportes físicos donde se encuentran los datos de entrada y donde van a ser registrados los resultados. Suelen tener una configuración que por sí sola indica características del soporte.

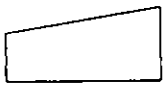
E = Soporte de entrada.
S = Soporte de salida.
E/S = Soporte de entrada y salida.

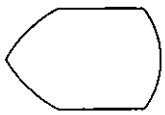
Existe una gran variedad de dispositivos que por su actualidad y diversidad aún no han sido normalizados ni existe acuerdo sobre la forma de representarlos: Reconocedores de voz y de gráficos (scanners), tabletas digitalizadoras, lápices ópticos, ratones, trazadores gráficos (plotters), pantallas digitalizadoras, dispositivos analógicos, etc.


TARJETA PERFORADA (E/S).

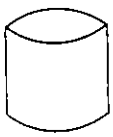

CINTA DE PAPEL (E/S).


IMPRESORA (S).

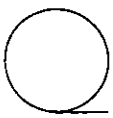

TECLADO (E).

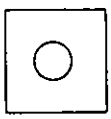

PANTALLA (S).

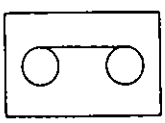

TAMBOR MAGNETICO (E/S).

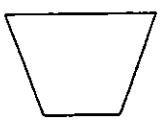

DISCO MAGNETICO (E/S).


SOPORTE MAGNETICO (E/S)


CINTA MAGNETICA (E/S).


DISCO FLEXIBLE (E/S).

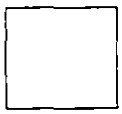

CINTA ENCAPSULADA (E/S).


SOPORTE GENERICO (E).

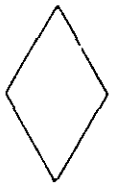
2.2.2. SIMBOLOS DE PROCESO

Representan el programa o un conjunto de operaciones que realizan un determinado trabajo completo. Para la representación de procesos típicos se utilizan símbolos específicos, como puede ser el de la clasificación.

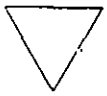
PROCESO



OPERACION AUXILIAR



CLASIFICACION DE ARCHIVOS



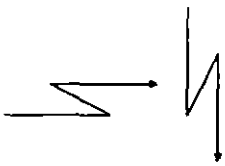
FUSION DE ARCHIVOS

PARTICION DE ARCHIVOS

2.2.3. LINEAS DE FLUJO

Indican el sentido del movimiento de los datos e informaciones y si se realiza dicho movimiento a corta o larga distancia.

DIRECCION DEL FLUJO

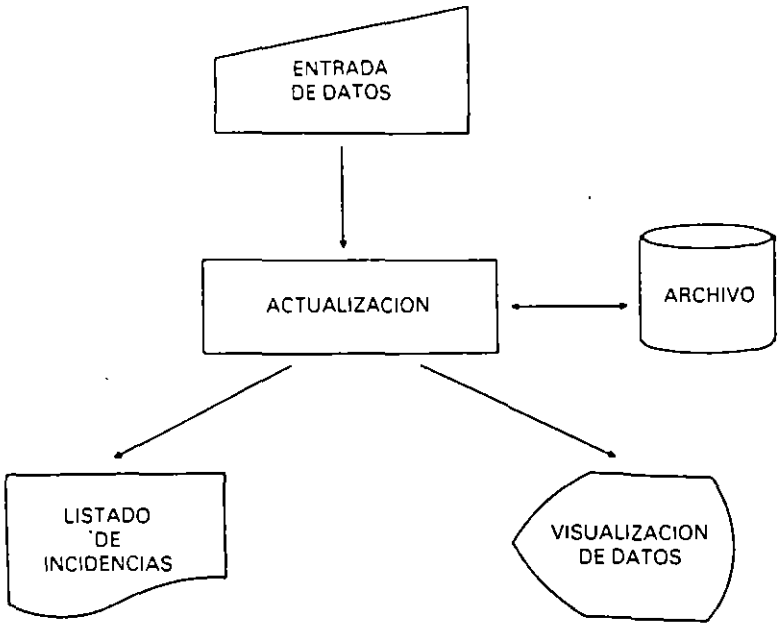


LINEAS DE TELEPROCESO

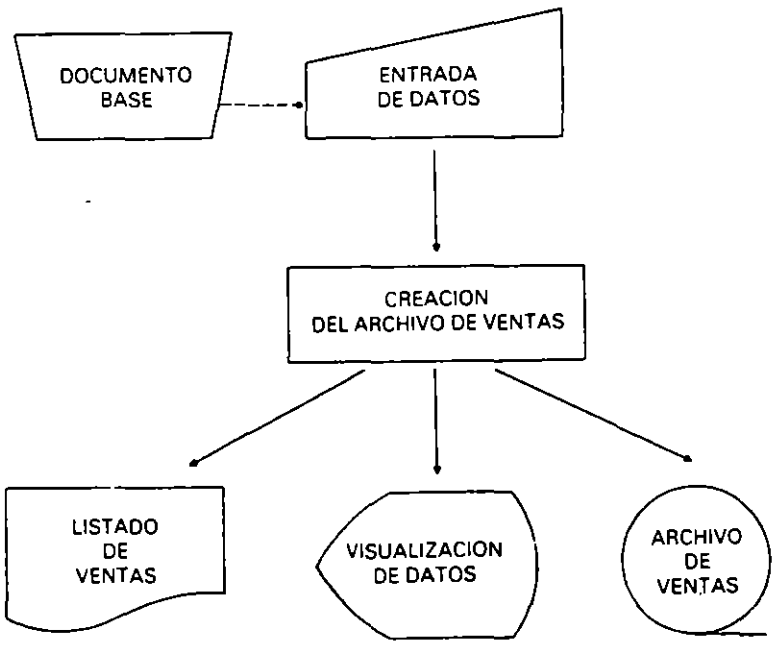
El objetivo del presente capítulo no es el aprendizaje de la técnica de diseño de organigramas, puesto que se utilizan en análisis de aplicaciones más que en diseño de programas; no obstante, presentamos algunos ejemplos genéricos.

Ejemplos:

El organigrama de una aplicación de actualización (puesta al dia) de un archivo soportado en disco, con entrada de datos por teclado, consulta de datos por pantalla y confección de un listado en papel de todas las incidencias ocurridas, es el siguiente:



El organigrama de una aplicación para crear un archivo de ventas realizadas en un determinado establecimiento, en cinta magnética, extraidas del documento en papel (factura) donde figuran los datos de las mismas, con entrada de datos por teclado, consulta y peticiones por pantalla y confección de un listado final de dichas ventas en papel, es el siguiente:



2.3. DIAGRAMAS DE FLUJO DEL PROGRAMA. ORDINOGRAMAS

Son representaciones gráficas de la secuencia lógica de las operaciones que se han de realizar para la resolución de un problema por medio de un programa dirigido a una computadora.

En la fase de programación, el programador crea para cada programa un ordinograma, a partir del cual realiza la codificación en el correspondiente lenguaje de programación. Es necesario indicar que existen otras técnicas más modernas para realizar la misma función que la de un ordinograma. Por ello, en la actualidad, ha decaído su uso, aunque existen aún muchas aplicaciones que cuentan con ellos en su documentación y muchas publicaciones sobre temas de programación que los utilizan.

La mayor utilidad que poseen las técnicas de diseño de programas, y en este caso también la técnica de ordinogramas, es la de realizar el diseño con cierta independencia de las características particulares de los lenguajes de programación. Con ello se consigue que un algoritmo de resolución de un problema pueda codificarse en cualquier lenguaje de cualquier máquina con las ventajas que supone la portabilidad de un diseño.

Tras el diseño de un programa utilizando esta técnica, y sabiendo cuál va a ser el lenguaje de programación en el que se va a codificar, sólo necesitaremos conocer la técnica de paso del ordinograma al lenguaje teniendo en cuenta sus características particulares. Esta técnica de traducción es, generalmente, muy sencilla.

Un ordinograma que representa un programa debe reflejar con claridad algunos de los elementos esenciales del mismo (Figura 2.3):

- El comienzo del programa (1).
- Las operaciones (2).
- La secuencia en que se realizan (3).
- El final del programa (4).

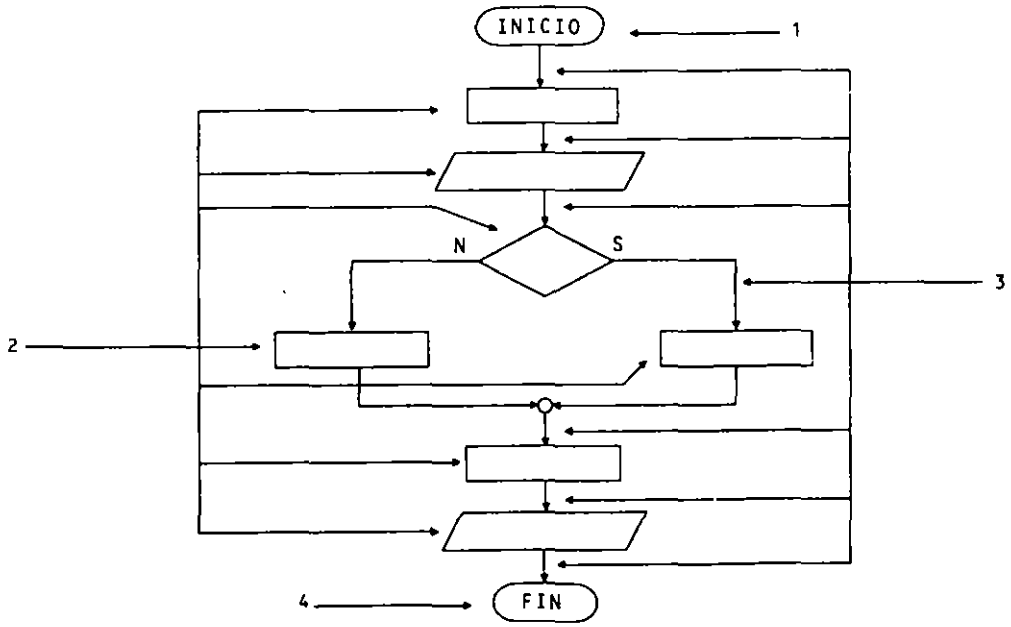


Figura 2.3. Elementos esenciales de un ordinograma.

En esta técnica de representación de programas es conveniente fijar una serie de reglas que nos permitirán actuar con unos mismos criterios a todos los que la utilicemos. Estas reglas son simplemente una recopilación de recomendaciones de las pocas normas que

existen y de aquellos que después de varios años de utilizar esta técnica suelen aconsejar.

- El comienzo del programa figurará en la parte superior.
- El flujo de las operaciones irá, siempre que sea posible, de arriba abajo y de izquierda a derecha, en cuyo caso se pueden omitir las puntas de flecha.
- El final del programa figurará en la parte inferior, aunque siempre tenderá a estar desplazado hacia la derecha.
- Los símbolos de INICIO y FIN deben aparecer una única vez, utilizando el símbolo de PARADA para representar cualquier otro tipo de interrupción o finalización.
- Se debe guardar simetría y equilibrio en la composición del conjunto del ordinograma.
- Aunque se permiten, se evitarán siempre los cruces de las líneas de flujo utilizando conectores.
- El uso de comentarios se restringirá al mínimo imprescindible, al contrario que en la codificación, en la que son mucho más recomendables.
- A un reagrupamiento de líneas de flujo pueden llegar varias de ellas, pero sólo puede salir una.

A continuación exponemos los símbolos utilizados en la confección de ordinogramas que se agrupan en cinco bloques:

- Símbolos de operación.
- Símbolos de decisión.
- Líneas de flujo.
- Símbolos de conexión.
- Símbolo de comentarios.

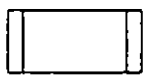
2.3.1. SIMBOLOS DE OPERACION

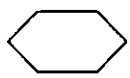
Son los símbolos que representan acciones u operaciones que se van realizando en la secuencia lógica correspondiente, consiguiendo con ello la resolución del problema objeto del programa.


TERMINAL (INICIO, FIN Y PARADA)


OPERACION EN GENERAL.


OPERACION DE E/S EN GENERAL.


SUBPROGRAMA.

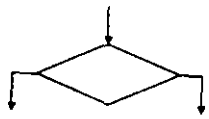

MODIFICACION DE INSTRUCCION O INICIALIZACIONES.


OPERACION MANUAL.

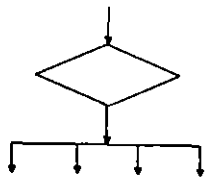
Además, para la representación de operaciones de entrada/salida se utilizan los símbolos de soporte ya indicados en los símbolos de organigramas en aquellos casos en que se quieren expresar explícitamente características de los soportes o las unidades que se van a utilizar. Cuando la entrada o salida de datos se realiza desde los dispositivos denominados estándar (teclado, pantalla e impresora) o cuando simplemente no es necesario en el diseño indicar desde o hacia qué dispositivo se dirigen los datos, se utiliza el símbolo de operación de entrada/salida en general (romboide).

2.3.2. SIMBOLOS DE DECISION

Los símbolos de decisión se utilizan para el establecimiento de bifurcaciones o la construcción de estructuras en las que se evalúa una expresión lógica o múltiple, derivándose la secuencia lógica de ejecución de las operaciones entre varios caminos posibles.



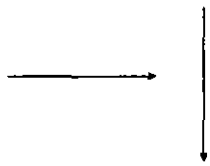
DECISION.



DECISION MULTIPLE.

2.3.3. LINEAS DE FLUJO

Indican la secuencia lógica de ejecución de las operaciones desde el comienzo (INICIO) hasta llegar al final (FIN o PARADA).



DIRECCION DEL FLUJO.

2.3.4. SIMBOLOS DE CONEXION

Se utilizan para la unión de líneas de flujo en los casos de reagrupamiento y de conexión o continuación en otra parte por cualquier motivo (cambio de hoja de papel, cruce de líneas, etc.).

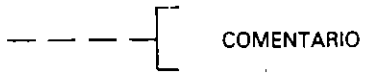
REAGRUPAMIENTO.

CONECTOR.

CONECTOR A DISTINTA PAGINA.

2.3.5. SIMBOLO DE COMENTARIOS

Se utilizan para aclarar o documentar el diseño del algoritmo con algún comentario que se considere necesario.

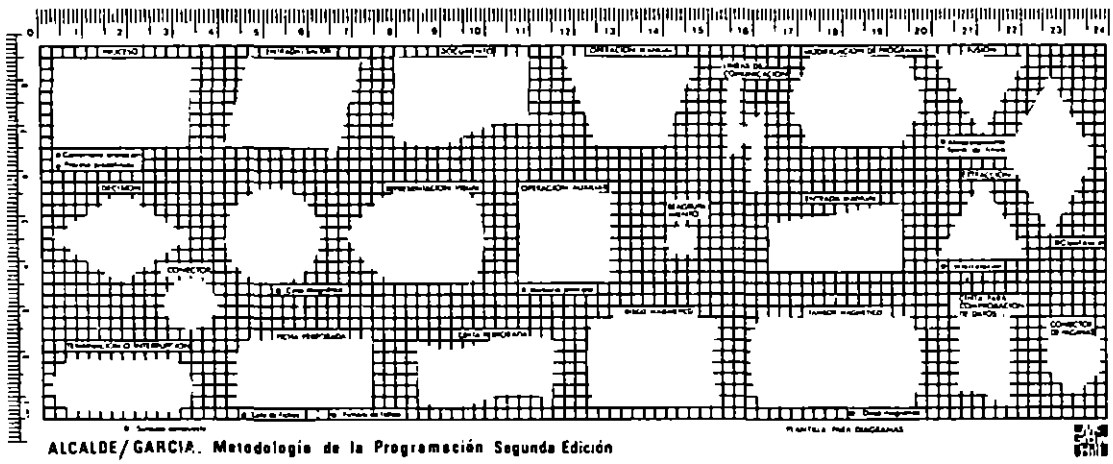


Es conveniente indicar que, generalmente, en el diseño de algoritmos, mediante la técnica de los ordinogramas, existen definiciones y normas referentes a los símbolos utilizados, pero no en cuanto al contenido de los mismos; es decir, existe cierta libertad para expresar con palabras y signos los detalles de una operación. En la presente edición utilizamos el estilo propio de los autores, que coincide con el de la mayoría.

2.4. PLANTILLAS Y NORMALIZACION

Como hemos indicado anteriormente es necesaria la utilización de símbolos similares en el diseño de algoritmos. Para facilitar la confección de los diagramas de flujo existe en el mercado una herramienta que consiste en una regla, generalmente de plástico troquelado o fundido, en la que aparecen los símbolos normalizados. Esta regla se denomina comúnmente Plantilla de Ordinogramas/Organigramas (*Flowcharting*). En la Figura 2.4 pueden verse varias de estas plantillas.

(a)



(b)

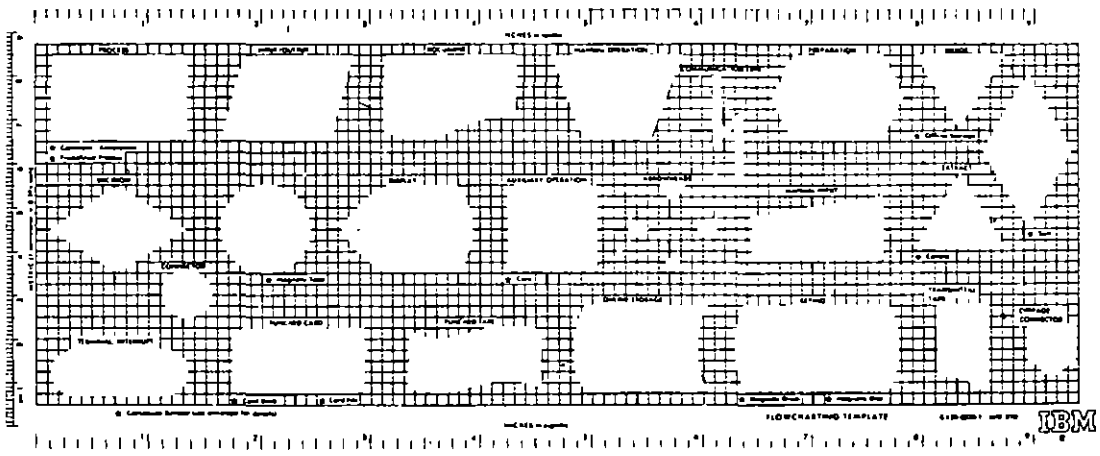


Figura 2.4. Plantillas de Ordinogramas/Organigramas.

En cada país se han establecido una serie de normas para la utilización de los diagramas de flujo con el objeto de normalizar su uso y obtener con ello mejoras en el intercambio de documentaciones y métodos.

En este sentido, en Estados Unidos existen las normas X3.5-1970 Flowchart Symbols and Their Usage de la American National Standards Institute (ANSI), y la norma ISO 1028-1973, dictada por la International Standard Organization (ISO); en Francia, la normativa Z 670 10 dictada por la Association Française de Normalisation (AFNOR), y en España existe el Proyecto de Norma Española PNE 71-001, publicado en 1977 por el Instituto Nacional de Racionalización y Normalización (IRANOR), que reproducimos textualmente a continuación.

2.5. PROYECTO DE NORMA ESPAÑOLA PNE 71-001 (Publicado el 16-05-1977)

Informática. Símbolos de organigrama. Para el tratamiento de la información

1. OBJETO

Esta norma tiene por objeto establecer los símbolos gráficos que se han de utilizar en los organigramas para los sistemas de tratamiento de la información, incluidos los sistemas de proceso automático de datos.

2. CAMPO DE APLICACION

Se entiende que los símbolos gráficos incluidos en la presente norma representan, en los organigramas, tanto

— la secuencia de las operaciones como
— la circulación de datos y documentos

en los sistemas de tratamiento de la información. Esta norma no comprende las informaciones que se escriben en el interior, o al lado de un símbolo, para asegurar su identificación, su descripción o explicación, ni los organigramas de tipo esquemático que utilizan dibujos o esquemas para describir un sistema.

3. CONVENCIONES

3.1. La dirección general de las líneas debe ser:

— de izquierda a derecha y
— de arriba abajo.

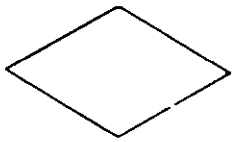
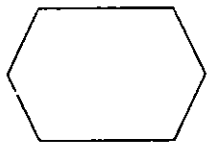
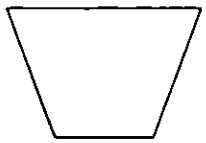
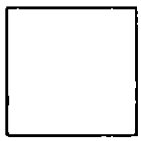
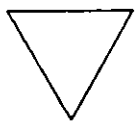
Se utilizarán flechas cuando no se respete ésta y siempre que faciliten una mejor comprensión.

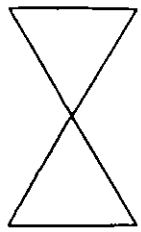
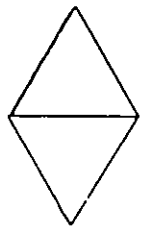
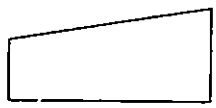
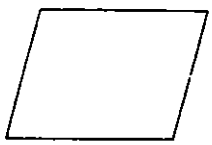
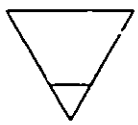
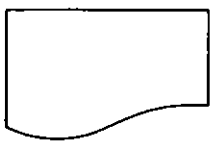
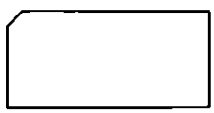
3.2. Los trazos de unión pueden cruzarse, lo que significa que no existe relación lógica entre ellas.

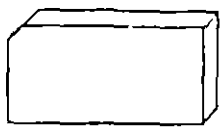
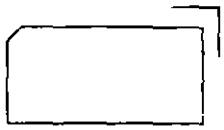
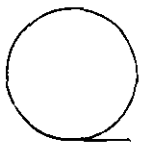
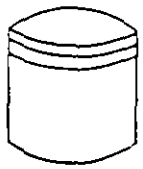
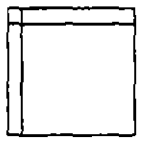
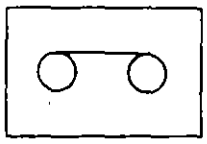
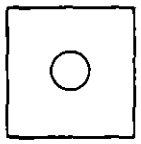
3.3. Pueden unirse a una línea de salida varios trazos de unión afluyentes.

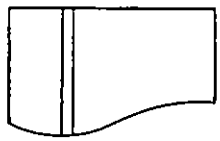
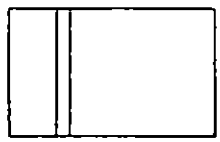
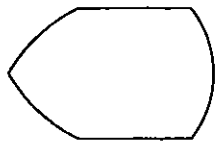
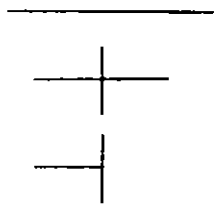
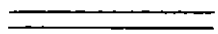
3.4. Aunque la presente norma no proporcione especificaciones exactas respecto a la relación altura/anchura se pide al usuario que no modifique estas relaciones, dé tal manera que impida el reconocimiento inmediato del símbolo.

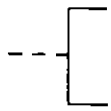
4. SIMBOLOS DE ORGANIGRAMA

1	**Tratamiento** Este símbolo representa todas las variedades de funciones de tratamiento, por ejemplo: ejecución de una operación particular o de un grupo de operaciones determinado que tenga por resultado una modificación en el valor, en la forma o en la situación de una información, o en la determinación del camino que haya de seguirse entre varios.	
2	**Enlace** Este símbolo representa una operación de decisión o bifurcación que determine el camino a seguir entre los varios posibles.	
3	**Preparación** Este símbolo representa la modificación de una instrucción o de un grupo de instrucciones que alteren el programa en sí mismo, por ejemplo: puesta en posición de una bifurcación, modificación de un registro de índice y puesta de un programa en su estado inicial.	
4	**Subprograma** Este símbolo representa un tratamiento con referencia, compuesto de una o varias operaciones o secuencias de programa que se definen complementariamente, como, por ejemplo, un subprograma.	
5	**Operación manual** Este símbolo representa cualquier tratamiento exterior al sistema que depende de la actuación de un ser humano, sin que se utilice ninguna ayuda mecánica.	
6	**Operación auxiliar** Este símbolo representa una operación periférica al sistema efectuado con un equipo que no se encuentre bajo el control directo de la unidad central de tratamiento.	
7	**Fusión** Este símbolo representa la combinación de dos o más series en una única serie.	
8	**Separación** Este símbolo representa la extracción de una o varias series particulares a partir de una sola serie.	

9	**Interclasificación/reunión** Este símbolo representa una fusión con separación, es decir, la formación de dos o más series a partir de otras dos o más series.	
10	**Clasificación** Este símbolo representa la ordenación de un grupo de artículos con arreglo a una secuencia particular.	
11	**Entrada manual** Este símbolo representa una función de entrada en la que se introduce manualmente la información en el momento del tratamiento, por ejemplo: por medio de una máquina de teclado conectada, de posicionamientos de palancas, de pulsadores.	
12	**Entrada/salida** Este símbolo representa una función de entrada/salida, por ejemplo: puesta a disposición de una información para el tratamiento (entrada) o registro de una información tratada (salida).	
13	**Memoria conectada (al sistema)** Este símbolo representa una función de entrada/salida que utiliza un tipo cualquiera de memoria interior al sistema, por ejemplo: una cinta magnética, un tambor magnético, un disco magnético.	
14	**Memoria exterior al sistema** Este símbolo representa la función de conservación de una información en el exterior del sistema, sin tener en cuenta el soporte sobre el que se registra esta información.	
15	**Documento** Este símbolo representa una función de entrada/salida para la cual el soporte es un documento.	
16	**Tarjeta** Este símbolo representa una función de entrada/salida para la cual el soporte es una tarjeta perforada, con marcas de matriz, o de foto-lectura.	

17	**Paquete de tarjetas** Este símbolo representa un conjunto de tarjetas.	
18	**Fichero de tarjetas** Este símbolo representa un conjunto de registros reunidos sobre tarjetas.	
19	**Cinta perforada** Este símbolo representa una función de entrada salida para la cual el soporte es una cinta perforada	
20	**Cinta magnética** Este símbolo representa una función de entrada/salida para la cual el soporte es una cinta magnetica.	
21	**Tambor magnético** Este símbolo representa una función de entrada, salida para la cual el soporte es un tambor magnético.	
22	**Disco magnético** Este símbolo representa una función de entrada/salida para la cual el soporte es un disco magnético.	
23	**Memoria de ferrita** Este símbolo representa una función de entrada/salida para la cual el soporte es una memoria de ferrita magnética.	
24	**Casete** Este símbolo representa una función de entrada/salida o almacenamiento de una información en una casete.	
25	**Disquete** Este símbolo representa una función de entrada/salida o almacenamiento de información en disquete.	

26	**Ficha de banda magnética** Este símbolo representa una función de entrada/salida o almacenamiento para la cual el soporte es un documento con información legible directamente por el hombre y además con banda magnética.	
27	**Tarjeta magnética** Este símbolo representa una función de entrada/salida o almacenamiento de información en una tarjeta magnética.	
28	**Salida visualizada** Este símbolo representa una función de entrada/salida, gracias a lo cual la información es extraída en el momento del tratamiento de una forma visual utilizable por el operador, mediante indicadores, pantallas de televisión, máquinas de escribir, pupitres, trazadores de curvas, etc., interiores al sistema.	
29	**Línea de unión** (Véase convención 3.1) Este símbolo representa la función que consiste en relacionar dos símbolos entre sí. Cruce de líneas de unión (Véase convención 3.2) Conexión de líneas de unión (Véase convención 3.3)	
30	**Modo síncrono/paralelo/asíncrono** (No se representa ninguna línea de unión; véase convención 3.1). Este símbolo representa el comienzo o el fin de dos o más operaciones simultáneas.	
31	**Transmisión** (Véase convención 3.1) Este símbolo representa una función para la cual se transmite una información mediante una telecomunicación.	
32	**Conector** Este símbolo representa una salida hacia o una entrada en otra parte del organigrama.	

33	**Comienzo, fin, interrupción** Este símbolo representa una etapa en un organigrama, por ejemplo: un arranque, una parada, un momento de parada, una espera o una interrupción.	
34	**Comentario** Este símbolo representa la función de anotación, es decir, la adición de comentarios descriptivos o de notas explicativas destinadas a hacer aclaraciones.	

5. CORRESPONDENCIA CON OTRAS NORMAS

Esta norma concuerda con la norma internacional ISO 1.028 de 1973.

EJERCICIOS RESUELTOS

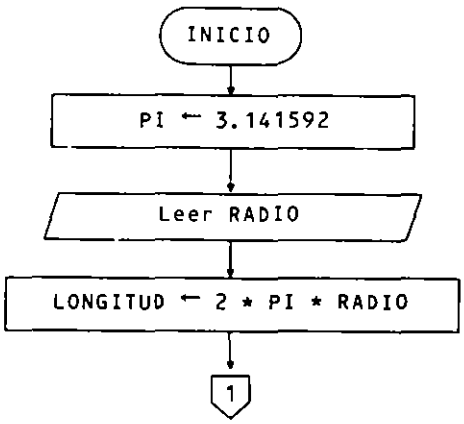
1. *Algoritmo que toma como dato de entrada un número que corresponde a la longitud de un radio y nos calcula y escribe la longitud de la circunferencia, el área del círculo y el volumen de la esfera que se corresponden con dicho radio.*

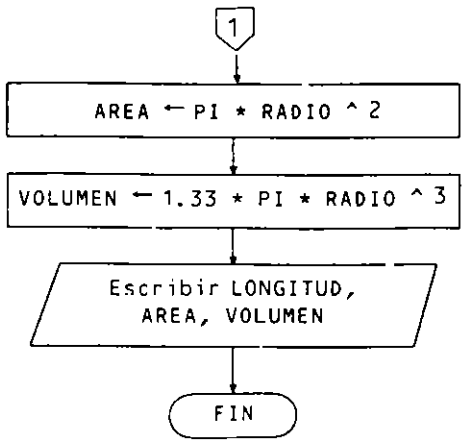
Objetos (constantes y variables):

PI	Constante para almacenar 3.141592.
RADIO	Variable para tomar el dato de entrada.
LONGITUD	Variable para calcular la longitud de la circunferencia.
AREA	Variable para calcular el área del círculo.
VOLUMEN	Variable para calcular el volumen de la esfera.

Las fórmulas matemáticas que nos dan los cálculos son:

$$l = 2\pi r \qquad a = \pi r^2 \qquad v = \frac{4}{3}\pi r^3$$





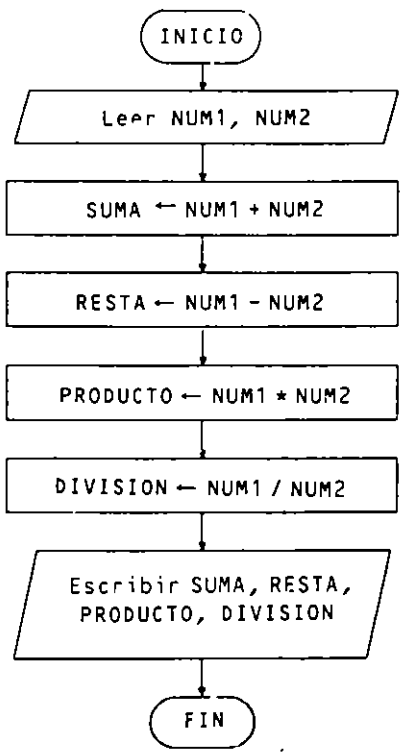
2. *Algoritmo que lee dos números, calculando y escribiendo el valor de su suma, resta, producto y división.*

Objetos (variables):

NUM1 y NUM2	Variables para tomar los datos de entrada.
SUMA	Variable para calcular la suma.
RESTA	Variable para calcular la resta.
PRODUCTO	Variable para calcular el producto.
DIVISION	Variable para calcular la división.

Las fórmulas matemáticas para realizar los cálculos son las propias de cada operación:

$$s = n_1 + n_2 \qquad r = n_1 - n_2 \qquad p = n_1 \cdot n_2 \qquad d = \frac{n_1}{n_2}$$

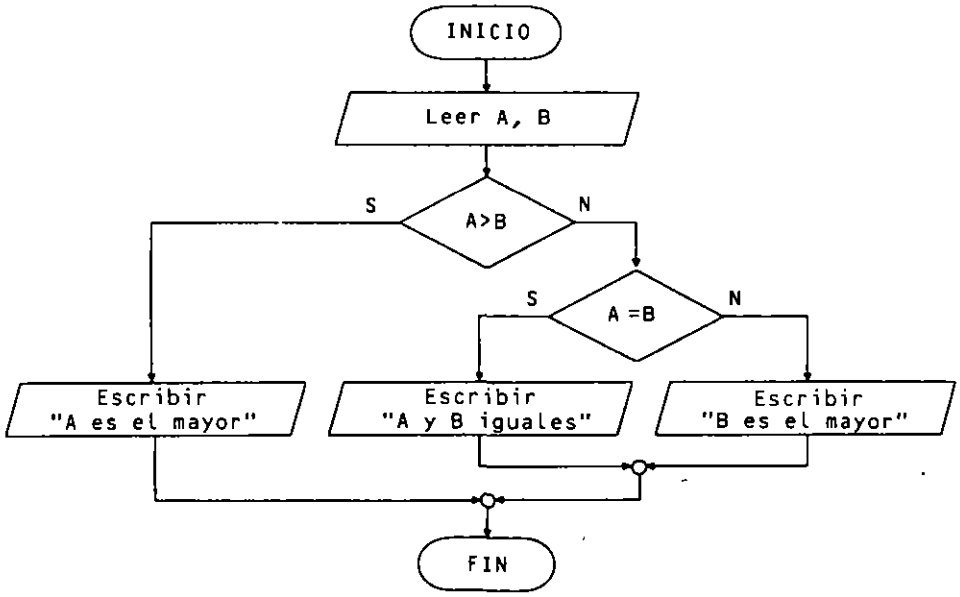


3. *Algoritmo que lee dos números y nos dice cuál es el mayor o si son iguales.*

Objetos (variables):

A y B Variables para tomar los datos de entrada.

El algoritmo se resuelve relacionando A y B con los operadores $>$ e $=$.

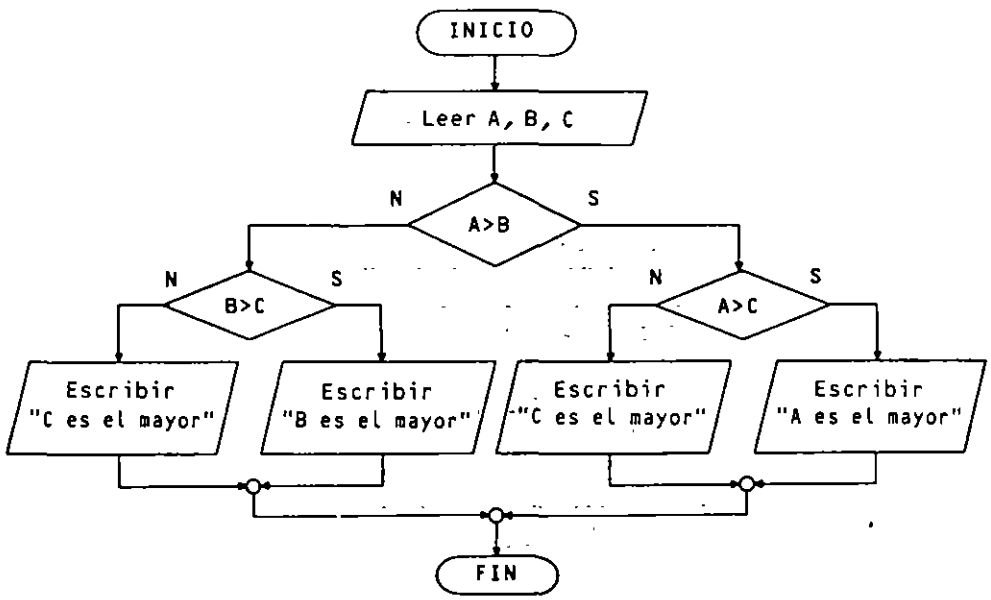


4. *Algoritmo que lee tres números distintos y nos dice cuál de ellos es el mayor.*

Objetos (variables):

A, B y C Variables para tomar los datos de entrada.

El algoritmo se resuelve relacionando los números con el operador $>$.



5. *Algoritmo que lee una temperatura en la escala centígrada y nos calcula y escribe su valor en las escalas Reamur, Farenheit y Kelvin.*

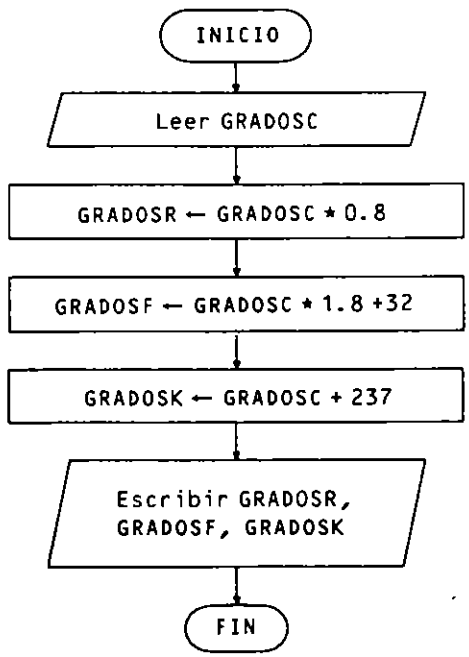
Objetos (variables):

GRADOSC	Variable para tomar el dato de entrada.
GRADOSR	Variable para calcular el valor en grados Reamur.
GRADOSF	Variable para calcular el valor en grados Farenheit.
GRADOSK	Variable para calcular el valor en grados Kelvin.

Las equivalencias entre las diferentes escalas termométricas están reflejadas por las siguientes ecuaciones:

	°C	°R	°F	°K
Punto ebullición del agua	100	80	212	373
Punto fusión del agua	0	0	32	273

$$\frac{°C}{100} = \frac{°R}{80} = \frac{°F - 32}{180} = \frac{°K - 273}{100}$$

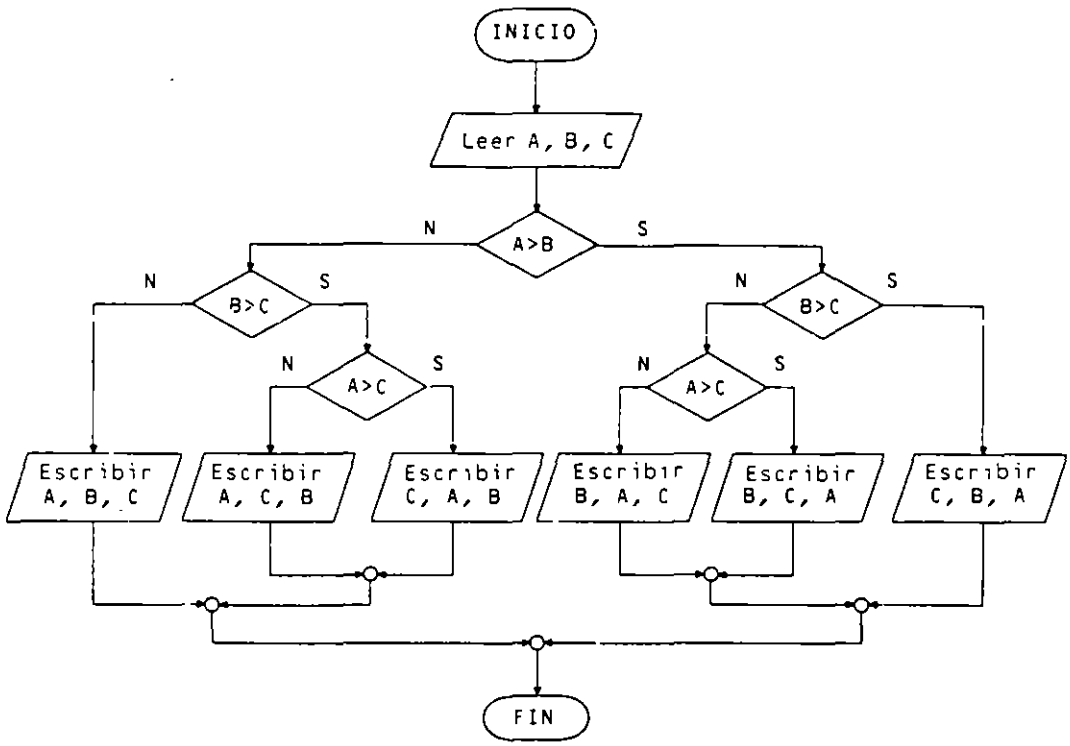


6. *Algoritmo que lee tres números cualesquiera y los escribe ordenados de forma ascendente.*

Objetos (variables):

A, B y C	Variables para tomar los datos de entrada.

El algoritmo se puede resolver relacionando los números con el operador >.



7. *Algoritmo que lee una calificación numérica entre 0 y 10 y la transforma en calificación alfabética, escribiendo el resultado.*

Objeto (variable):

NOTA Variable para tomar el dato de entrada

El algoritmo se puede resolver utilizando el operador < para relacionar la nota con los límites que existen entre las calificaciones alfabéticas y que se dan en la siguiente tabla:

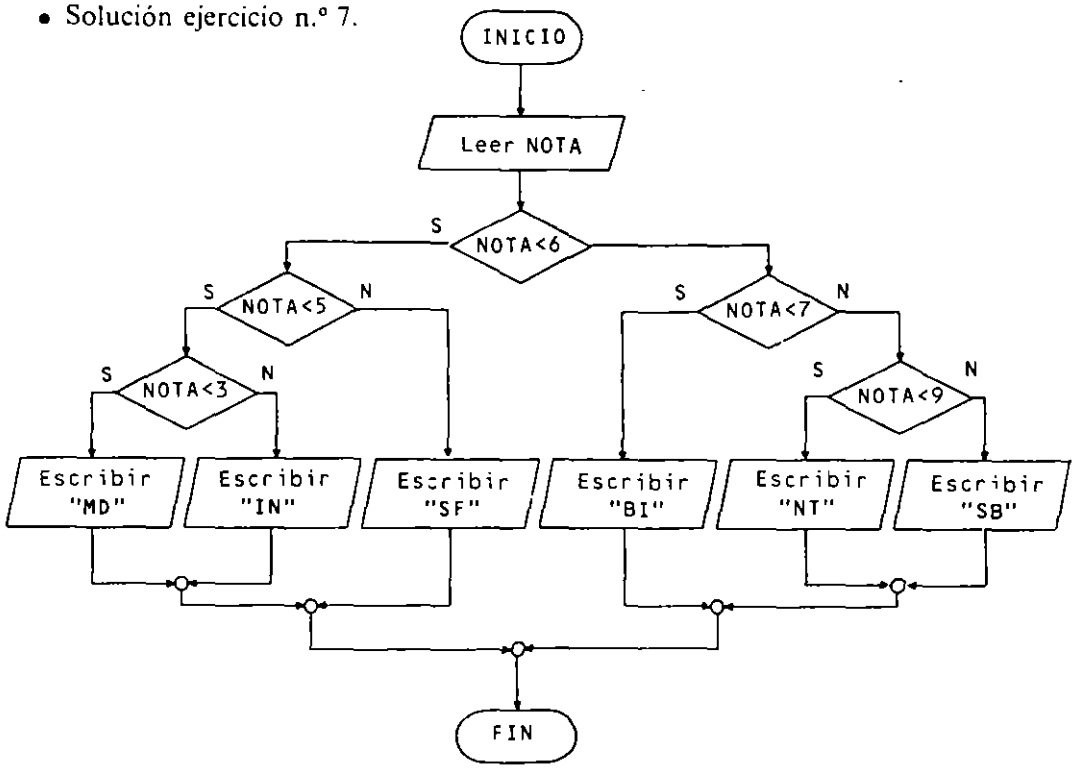
Nota numérica	*Nota alfabética*
$0 \leqslant NOTA < 3$	Muy deficiente
$3 \leqslant NOTA < 5$	Insuficiente
$5 \leqslant NOTA < 6$	Suficiente
$6 \leqslant NOTA < 7$	Bien
$7 \leqslant NOTA < 9$	Notable
$9 \leqslant NOTA \leqslant 10$	Sobresaliente

8. *Algoritmo que lee tres números cualesquiera y nos indica todas sus relaciones de igualdad.*

Objetos (variables):

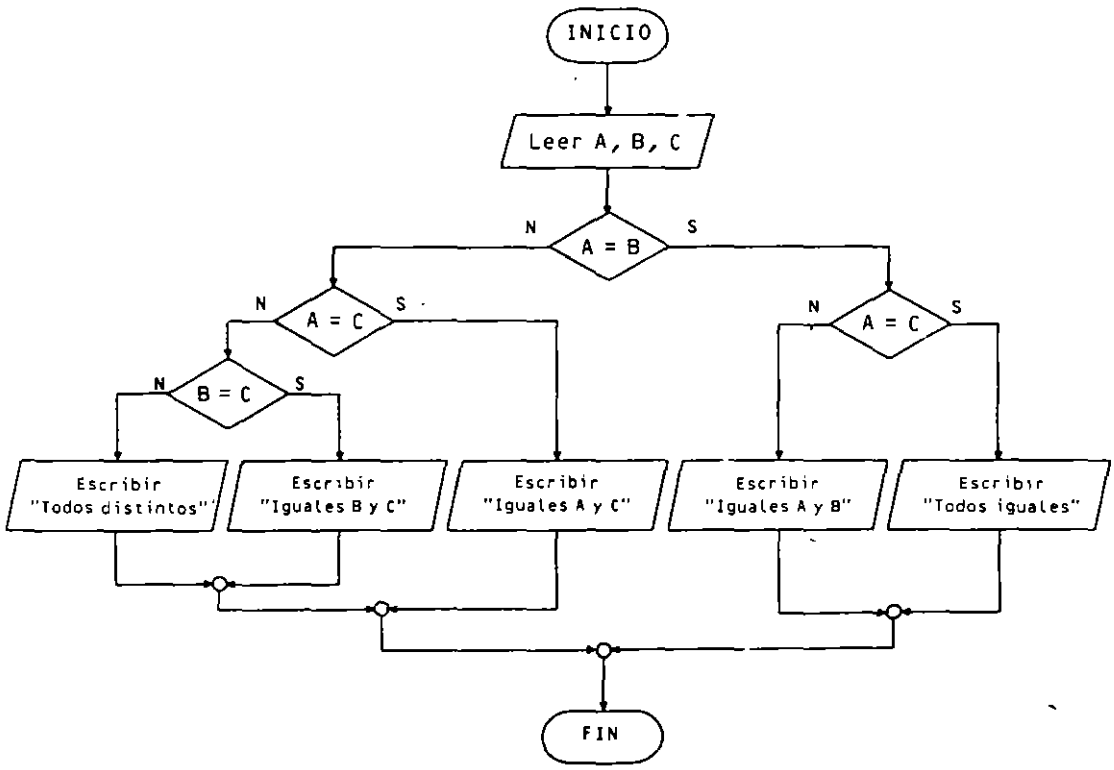
A, B y C Variables para tomar los datos de entrada.

- Solución ejercicio n.º 7.



- Solución ejercicio n.º 8.

El algoritmo se resuelve relacionando los números con el operador = hasta obtener información sobre todas sus relaciones.

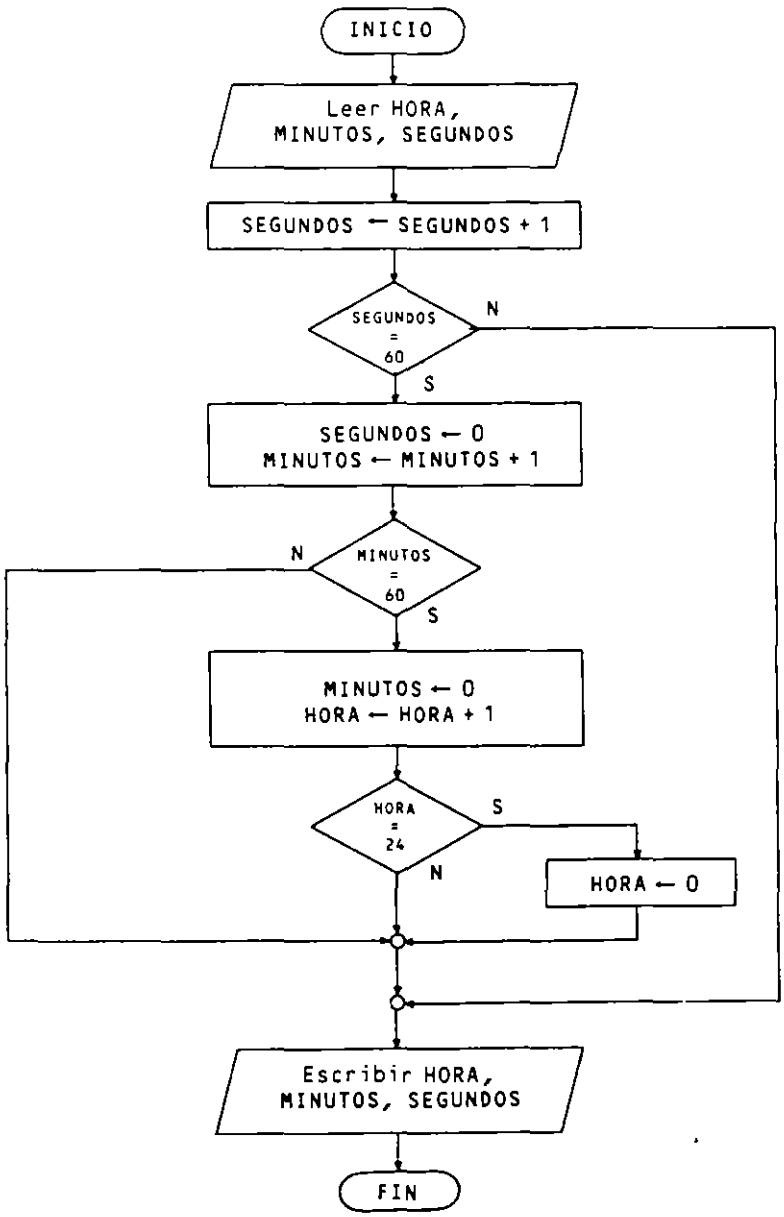


9. *Algoritmo que recibe como datos de entrada una hora expresada en horas, minutos y segundos que nos calcula y escribe la hora, minutos y segundos que serán, transcurrido un segundo.*

Objetos (variables):

HORA	Variable para tomar la hora inicial y calcular la hora final
MINUTOS	Variable para tomar los minutos iniciales y calcular los minutos finales
SEGUNDOS	Variable para tomar los segundos iniciales y calcular los segundos finales

El algoritmo se resuelve sumando a los datos de entrada un segundo, pero si sobrepasamos los 59 segundos, 59 minutos o 23 horas, realizaremos los ajustes necesarios. Se supone un reloj de 24 horas.



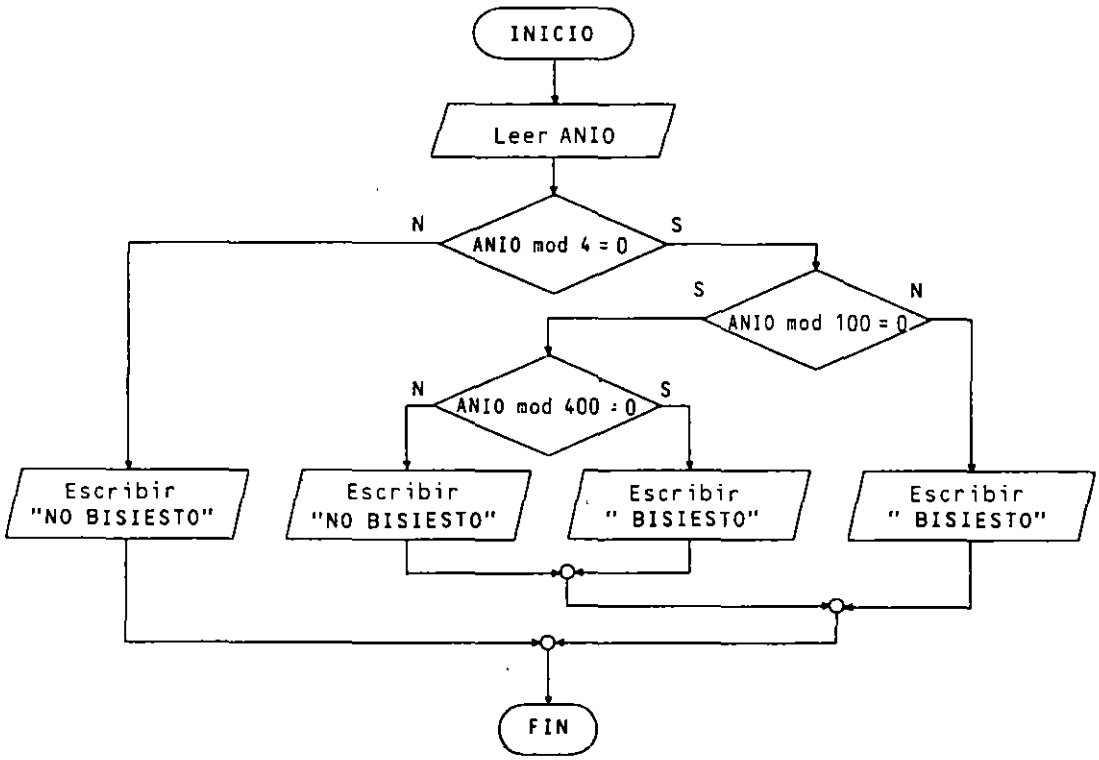
10. *Algoritmo que lee como dato de entrada un año y nos dice si se trata de un año bisiesto o no. Se sabe que son bisiestos todos los años múltiplos de 4, excepto los que sean múltiplos de 100 sin ser múltiplos de 400.*

Objeto (variable):

ANIO Variable para tomar el dato de entrada.

Nota: En los identificadores de variables no es conveniente usar la letra Ñ por no existir en el alfabeto inglés y, consiguientemente, no estar permitida en la mayoría de los lenguajes de programación.

El algoritmo se puede resolver investigando las relaciones existentes entre el dato de entrada y los números 4, 100 y 400 en cuanto a divisibilidad. Se sabe que entre dos números enteros y positivos existe divisibilidad cuando coinciden su división real / (con decimales) y su división entera \ (sin decimales), es decir, cuando al dividirlos el resto es 0.



EJERCICIOS PROPUESTOS

1. En un determinado comercio se realiza un descuento dependiendo del precio de cada producto. Si el precio es inferior a 1.000 pesetas, no se hace descuento; si es mayor o igual a 1.000 pesetas y menor que 10.000 pesetas, se hace un 5 por 100 de descuento, y si es mayor o igual a 10.000 pesetas, se hace un 10 por 100 de descuento.

Algoritmo que lee el precio de un producto y nos calcula y escribe su precio final.

2. Algoritmo que lee el precio final pagado por un producto y su precio de tarifa y nos calcula el porcentaje de descuento que le ha sido aplicado.

3. Algoritmo que toma como datos de entrada el capital C en pesetas depositado en un banco, el interés R en tanto por ciento y el tiempo T en años que estará el mencionado capital depositado y nos calcula y escribe los intereses en pesetas que recibiremos transcurridos los T años. El cálculo se realizará por medio de la fórmula del interés simple.

$$I = \frac{C \cdot R \cdot T}{100}$$

4. Algoritmo que lee tres números distintos y nos dice cuál de ellos es el menor.

5. Algoritmo que lee tres números cualesquiera y los escribe ordenados de forma descendente.

6. Algoritmo que lee una calificación alfabética en una variable tipo carácter y nos escribe su valor numérico que será el establecido por la tabla de conversión de medias ponderadas.

	Nota alfabética	*Media ponderada numérica*
Muy deficiente	"MD"	1.5
Insuficiente	"IN"	4.0
Suficiente	"SF"	5.5
Bien	"BI"	6.5
Notable	"NT"	8.0
Sobresaliente	"SB"	9.5

7. Algoritmo que lee cuatro número A, B, C y D cualesquiera y nos indica todas sus relaciones de igualdad (son los cuatro iguales, son iguales A y C, son iguales B, C y D, son iguales A y C al tiempo que B y D, etc.).

8. Algoritmo que lee como datos de entrada una fecha expresada en día (del 1 al 31), mes (del 1 al 12) y año (en número) y nos dice la fecha que será al día siguiente. Se supondrá que febrero tiene siempre 28 días.

9. Algoritmo que lee dos números enteros, positivos y distintos y nos dice si el mayor es múltiplo del menor o, lo que es lo mismo, si el menor es divisor del mayor.

10. Algoritmo que calcula la edad de una persona recibiendo como datos de entrada la fecha de nacimiento y la fecha actual, ambas en tres variables que recibirán el día (del 1 al 31), el mes (del 1 al 12) y el año en número entero.

CAPITULO 3

Estructura general de un programa

3.1. INTRODUCCION

Un programa puede considerarse como una secuencia lógica de acciones (instrucciones) que manipulan un conjunto de objetos (datos) para obtener unos resultados que serán la solución al problema que resuelve dicho programa.

Todo programa, en general, contiene dos bloques bien diferenciados para la descripción de los dos aspectos anteriormente citados:

- **Bloque de declaraciones.** En él se especifican todos los objetos que utiliza el programa (constantes, variables, tablas, registros, archivos, etc.) indicando sus características. Este bloque se encuentra localizado siempre por delante del comienzo de las acciones.
- **Bloque de instrucciones.** Constituido por el conjunto de operaciones que se han de realizar para la obtención de los resultados deseados.

En algunos lenguajes de programación puede figurar explícita o implícitamente el bloque de declaraciones (FORTRAN, BASIC), pero, en general, es necesario declarar y definir todos los objetos que se van a utilizar en un programa (COBOL, PASCAL, PL/I, C, ADA).

La ejecución de un programa consiste en la realización secuencial del conjunto de instrucciones de que se compone, desde la primera a la última, de una en una. Este orden de realización únicamente será alterado mediante instrucciones denominadas de ruptura de secuencia, que en la actualidad han caído en desuso y mediante sentencias de control.

Las instrucciones de un programa consisten, en general, en modificaciones sobre los objetos del programa, que constituyen su entorno, desde un estado inicial hasta otro final que contendrá los resultados del proceso.

El entorno de un programa u objetos del mismo es el conjunto de elementos capaces de almacenar unidades de información, necesarios para contener tanto los datos de entrada como los datos resultantes de todos los procesos que se lleven a cabo.

Los procesos que se pueden llevar a cabo en un programa pueden ser de tipo aritmético o lógico, incluyéndose algunas operaciones de manejo de caracteres y operaciones de entrada/salida.

3.2. PARTES PRINCIPALES DE UN PROGRAMA

Las partes principales de un programa están relacionadas con sus dos bloques ya mencionados. Dentro del bloque de instrucciones podemos diferenciar tres partes fundamentales, como se muestra en la Figura 3.1.

En algunos programas (generamente los orientados al proceso en cola o BATCH), las tres partes del bloque de instrucciones están perfectamente delimitadas, pero en la mayoría de los programas (principalmente los que utilizan proceso interactivo) sus instrucciones quedan entremezcladas a lo largo del programa, si bien mantienen una

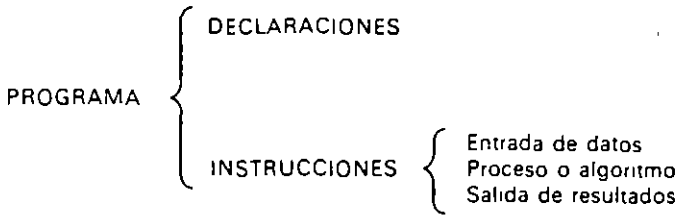


Figura 3.1. Partes principales de un programa.

cierta localización impuesta por la propia naturaleza de las mismas; es decir, la entrada de datos se encuentra desplazada al comienzo del programa, el proceso se encuentra en-medio y la salida de resultados se encuentra desplazada hacia el final.

3.2.1. ENTRADA DE DATOS

La constituyen todas las instrucciones que toman los datos de entrada desde un dispositivo externo y los almacenan en la memoria central para que puedan ser procesados (Figura 3.2).

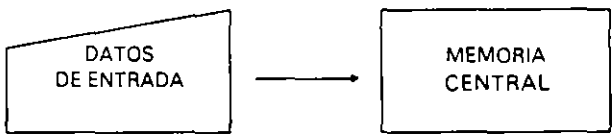


Figura 3.2. Entrada de datos.

También se consideran dentro de esta parte las instrucciones de depuración de los datos de entrada, es decir, las que se encargan de comprobar la corrección de los mismos

3.2.2. PROCESO O ALGORITMO

Está formado por las instrucciones que modifican los objetos a partir de su estado inicial (datos de entrada) hasta el estado final (resultados), dejando los objetos que lo contienen disponibles en la memoria central (Figura 3.3).

Figura 3.3. Proceso o algoritmo.

3.2.3. SALIDA DE RESULTADOS

Conjunto de instrucciones que toman los datos finales (resultados) de la memoria central y los envían a los dispositivos externos (Figura 3.4).

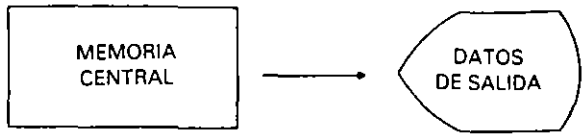


Figura 3.4. Salida de resultados.

Se incluyen también todas las órdenes e instrucciones para dar formato a los resultados y controlar el dispositivo (saltos de página, borrar pantalla, etc.).

3.3. CLASIFICACION DE LAS INSTRUCCIONES

Una instrucción se caracteriza por un estado inicial (estado de los objetos que maneja el programa antes de la ejecución de la instrucción) y otro final (estado en que quedan los objetos después de la ejecución de la instrucción). El estado final de una instrucción siempre coincide con el estado inicial de la siguiente.

Ahora bien, no siempre una instrucción modifica el entorno (conjunto de objetos), pues su cometido puede limitarse a una mera observación del mismo o a un control en el orden de ejecución de otras instrucciones.

Según la función que desempeñan dentro de un programa, las instrucciones se clasifican de la siguiente manera.

3.3.1. INSTRUCCIONES DE DECLARACION

Su misión es anunciar la utilización de objetos en un programa indicando qué identificador, tipo y otras características corresponde a cada uno de ellos. Existen lenguajes que tienen ya establecidas declaraciones por defecto, que constituyen las denominadas declaraciones implícitas.

Ejemplo: *La declaración de las variables numéricas DIA, MES y ANIO será:*

- **COBOL:**

```
DATA DIVISION.
WORKING-STORAGE SECTION.
01  VARIABLES.
    05 DIA  PIC 99.
    05 MES  PIC 99.
    05 ANIO PIC 9999.
```

- **Pascal:**

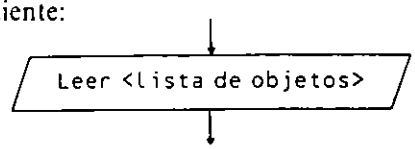
3.3.2. INSTRUCCIONES PRIMITIVAS

Son aquellas que ejecuta el procesador de modo inmediato. Es decir, no dependen de otra cosa que de su propia aparición en el programa para ser ejecuadas. Las instrucciones primitivas pueden ser de entrada, de asignación o de salida.

■ **Instrucción de entrada**

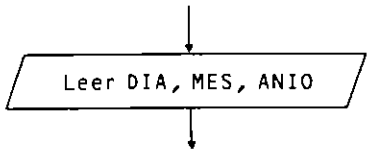
Su misión consiste en tomar uno o varios datos desde un dispositivo de entrada y almacenarlos en la memoria central en los objetos cuyos identificadores aparecen en la propia instrucción. Si estos objetos tuviesen algún valor previo, éste se perdería.

En un ordinograma aparece con el símbolo de operación de entrada/salida (romboide) y con la fórmula literal siguiente:



donde <lista de objetos> es el conjunto de elementos donde se van a depositar en memoria central los datos leídos. Cuando la lista se compone de más de un elemento, éstos se separan por comas.

Ejemplo: *Entrada de los datos DIA, MES y ANIO desde un dispositivo estándar (teclado).*

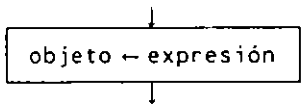


- **COBOL:** `ACCEPT DIA, ACCEPT MES, ACCEPT ANIO`
- **Pascal:** `READ(DIA,MES,ANIO)`

■ Instrucción de asignación

Es la instrucción que nos permite realizar cálculos evaluando una expresión y depositando su valor final en un objeto o realizar movimientos de datos de un objeto a otro.

En un ordinograma aparece bajo el símbolo de operación en general (rectángulo horizontal) con el siguiente formato.

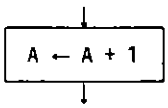


Esta instrucción se realiza en dos tiempos:

1. Se evalúa la expresión convirtiéndose en su valor final.
2. El valor final se asigna al objeto, borrándose cualquier otro valor previo que éste pudiera tener.

El objeto y la expresión deben coincidir en tipo y se admite que el propio objeto que recibe el valor final de la expresión pueda intervenir en la misma, pero entendiéndose que lo hace con su valor anterior.

Ejemplo: *Instrucción de asignación para incrementar en una unidad el valor de una variable A.*



- **COBOL:** `ADD 1 TO A`
  `COMPUTE A = A + 1` (Equivalente a la anterior)
- **Pascal:** `A := A + 1`

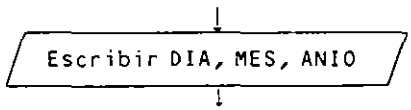
■ Instrucción de salida

Su misión consiste en enviar datos a un dispositivo externo, bien tomándolos de objetos depositados en la memoria central o definidos de alguna forma en la propia instrucción.

En un ordinograma se representa por medio del símbolo de operaciones de entrada/salida (romboide) y su fórmula literal es la siguiente:

donde <lista de expresiones> es el conjunto de informaciones y datos que queremos exteriorizar y que puede estar constituida por objetos, valores o expresiones que serán evaluados, siendo su valor final el que resultará exteriorizado. En los casos en que aparezca más de una expresión, éstas se separarán por comas.

Ejemplo: *Escritura en un dispositivo estándar (pantalla) de una fecha situada en la memoria en las variables DIA, MES y ANIO*



- **COBOL:** `DISPLAY "LA FECHA ES ", DIA, "-", MES, "-", ANIO`
- **Pascal:** `WRITE('LA FECHA ES ',DIA,'-',MES,'-',ANIO)`

3.3.3. INSTRUCCIONES DE CONTROL

Son instrucciones que no realizan trabajo efectivo alguno salvo la evaluación de expresiones, generalmente lógicas, con el objetivo de controlar la ejecución de otras instrucciones o alterar el orden de ejecución normal de las instrucciones de un programa.

Existen tres grandes grupos de instrucciones de control, que veremos a continuación.

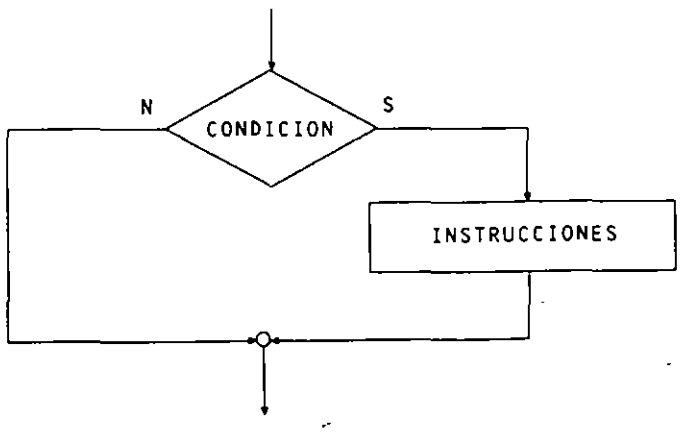
■ Instrucciones alternativas

Son aquellas que controlan la ejecución de uno o varios bloques de instrucciones, dependiendo del cumplimiento o no de alguna condición o del valor final de una expresión. Existen tres modelos típicos de instrucciones alternativas:

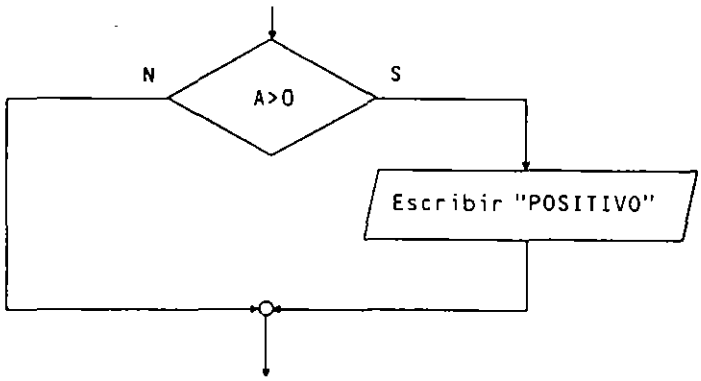
A) **Alternativa simple**

Controla la ejecución de un conjunto de instrucciones por el cumplimiento o no de una condición, de tal forma que, si se cumple, se ejecutan, si no se cumple, no se ejecutan.

Su representación en ordinograma es la siguiente:



Ejemplo: *Instrucción alternativa simple que escribe la palabra POSITIVO si el contenido de la variable A es mayor que* 0.



- **COBOL:**

```
IF A > 0
    THEN DISPLAY "POSITIVO"
END-IF
```

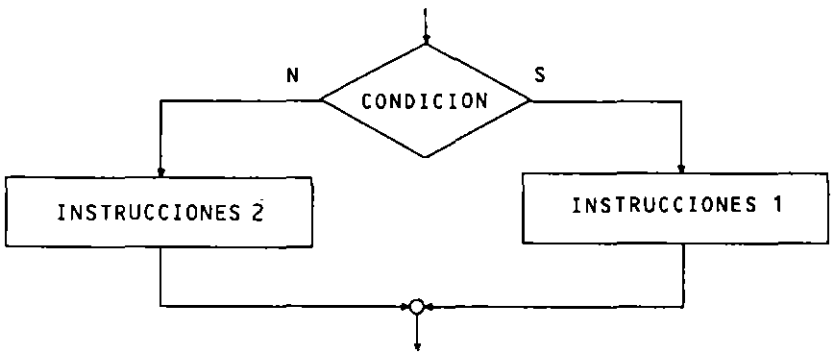
- **Pascal:**

```
IF A > 0
    THEN WRITE('POSITIVO')
```

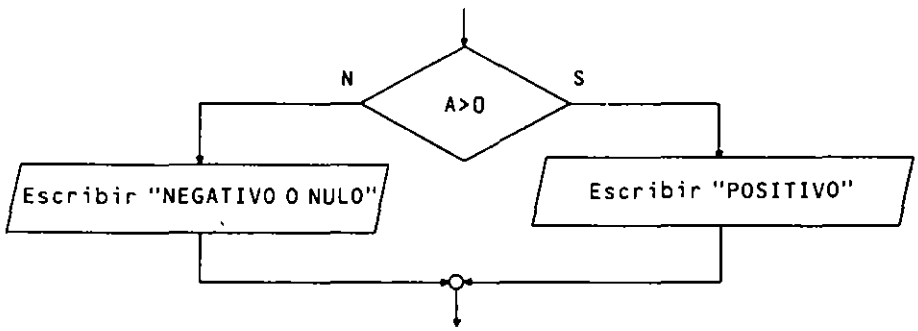
B) **Alternativa doble**

Controla la ejecución de dos conjuntos de instrucciones por el cumplimiento o no de una condición, de tal forma que, si se cumple, se ejecutan las instrucciones del primer bloque: si no se cumple, se ejecutan las instrucciones del segundo.

Su representación en ordinograma es la siguiente



Ejemplo: *Instrucción alternativa doble que escribe la palabra POSITIVO si el contenido de la variable A es mayor que 0 y NEGATIVO o NULO si no lo es.*



- **COBOL:**

```
IF A > 0
     THEN DISPLAY "POSITIVO"
     ELSE DISPLAY "NEGATIVO O NULO"
END-IF
```

- **Pascal:**

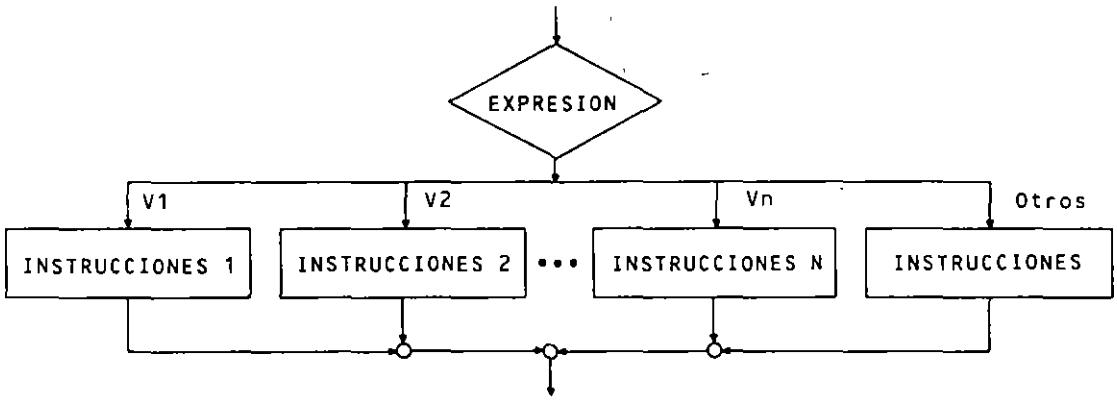
```
IF A>0
     THEN WRITE('POSITIVO')
     ELSE WRITE('NEGATIVO O NULO')
```

C) **Alternativa múltiple**

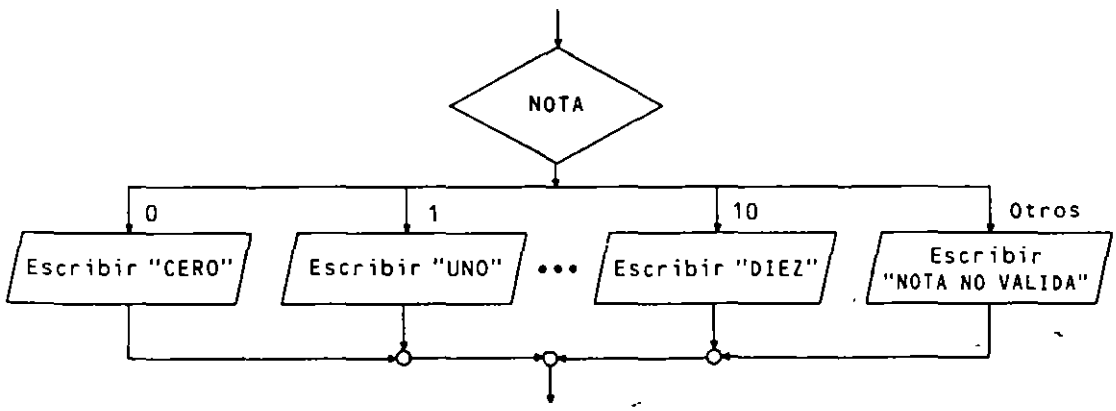
Controla la ejecución de varios conjuntos de instrucciones por el valor final de una expresión, de tal forma que cada conjunto de instrucciones está ligado a un posible valor de la expresión, existiendo un bloque al final que engloba otros posibles valores no definidos. Se ejecutará el conjunto que se encuentre relacionado con el valor que resulte de la evaluación de la expresión, de tal forma que si éste no aparece se ejecutará el último.

Las distintas opciones tienen que ser disjuntas, es decir, sólo puede cumplirse una de ellas.

Su representación en ordinograma es la siguiente.



Ejemplo: *Instrucción alternativa múltiple que escribe una nota numérica entera de 0 a 10 con el nombre de la propia nota en letras.*



- **COBOL:**

```
EVALUATE NOTA
    WHEN  0 DISPLAY "CERO   "
    WHEN  1 DISPLAY "UNO    "
    WHEN  2 DISPLAY "DOS    "
    WHEN  3 DISPLAY "TRES   "
    WHEN  4 DISPLAY "CUATRO"
    WHEN  5 DISPLAY "CINCO  "
    WHEN  6 DISPLAY "SEIS   "
    WHEN  7 DISPLAY "SIETE  "
    WHEN  8 DISPLAY "OCHO   "
    WHEN  9 DISPLAY "NUEVE  "
    WHEN 10 DISPLAY "DIEZ   "
    WHEN OTHER DISPLAY "NOTA NO VALIDA"
END-EVALUATE
```

- **Pascal:**

```
CASE NOTA OF
     0: WRITE('CERO   ') ;
     1: WRITE('UNO    ') ;
     2: WRITE('DOS    ') ;
     3: WRITE('TRES   ') ;
     4: WRITE('CUATRO') ;
     5: WRITE('CINCO  ') ;
     6: WRITE('SEIS   ') ;
     7: WRITE('SIETE  ') ;
     8: WRITE('OCHO   ') ;
     9: WRITE('NUEVE  ') ;
    10: WRITE('DIEZ   ')
    ELSE WRITE('NOTA NO VALIDA')
END;
```

■ Instrucciones repetitivas

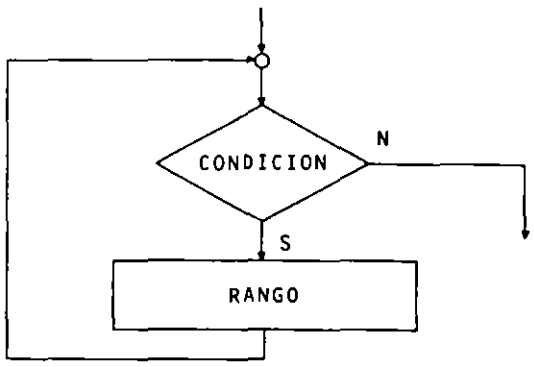
Son aquellas que controlan la repetición de un conjunto de instrucciones denominado rango mediante la evaluación de una condición que se realiza cada nueva repetición o por medio de un contador asociado.

Existen cuatro tipos de instrucciones repetitivas que dependen de su configuración:

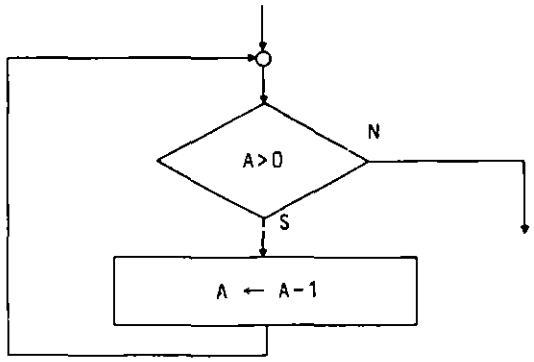
A) Instrución MIENTRAS (WHILE)

Controla la ejecución del conjunto de instrucciones que configuran su rango, de tal forma que éstas se ejecutan mientras se cumpla la condición, que será evaluada siempre antes de cada repetición, es decir, mientras la condición sea CIERTA.

Su configuración en ordinograma es la siguiente:



Ejemplo: *Repetición de la asignación $A \leftarrow A - 1$ mientras el valor de A sea positivo.*



- **COBOL:**

```
PERFORM UNTIL A <= 0
    COMPUTE A = A - 1
END-PERFORM
```

- **Pascal:**

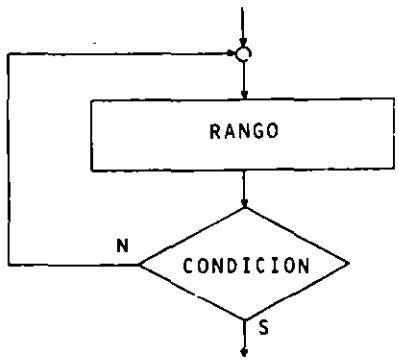
```
WHILE A>0 DO A:=A-1
```

B) Instrucción REPETIR (REPEAT-UNTIL)

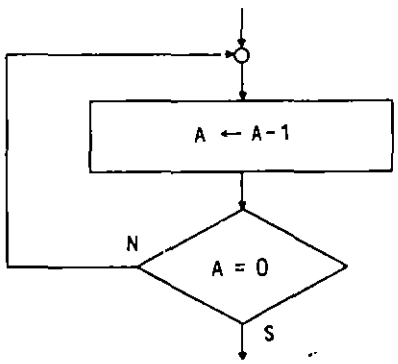
Controla la ejecución del conjunto de instrucciones que configuran su rango, de tal forma que éstas se ejecutan hasta que se cumpla la condición, que será evaluada siempre después de cada repetición, es decir, hasta que la condición sea CIERTA.

Una diferencia esencial entre este bucle y el anterior es, además de la posición de la condición, que este bucle siempre se ejecuta al menos una vez.

Su configuración en ordinograma es la siguiente:



Ejemplo: *La repetición de la asignación $A \leftarrow A - 1$ suponiendo A previamente positivo hasta que se haga 0 es.*



- **COBOL:**

```
PERFORM TEST AFTER UNTIL A = 0
    COMPUTE A = A - 1
END-PERFORM
```

- **Pascal:**

```
REPEAT
   A:=A-1
UNTIL A=0
```

C) Instrucción PARA (FOR)

Controla la ejecución del conjunto de instrucciones que configuran su rango, de tal forma que éstas se ejecutan un número determinado de veces que queda definido en lo que se denomina la cabecera del bucle. En ella se define un identificador de variable que va a actuar como contador asociado y que se denomina variable de control del bucle (V_c), definiéndose al mismo tiempo su valor inicial (V_i), su valor final (V_f) y el incremento (I_n) que esta variable de control va a adquirir en cada repetición.

La definición del bucle será, por tanto:

$$V_c = V_i, V_f [, I_n]$$

La variable de control toma el valor inicial y va incrementándose en I_n cada nueva repetición, de tal forma que el proceso termina cuando la variable de control supera el valor final (en caso de incremento negativo se invierten los términos).

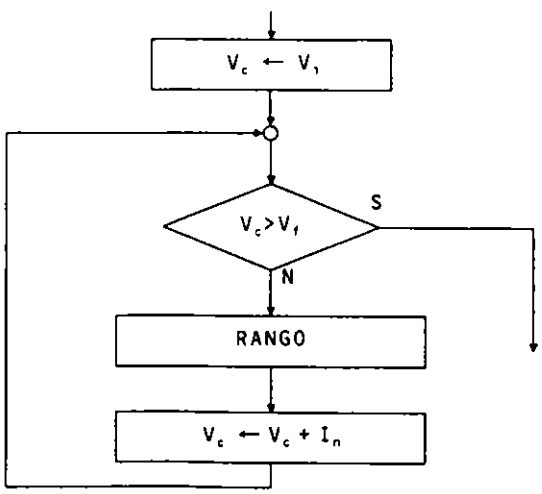
Los valores V_i, V_f e I_n pueden estar expresados por un valor, una variable o una expresión, entendiéndose que si las variables que intervienen son modificadas en el rango del bucle, esta modificación no afecta a la definición inicial del mismo.

Asimismo, la V_c puede ser utilizada como dato en el rango del bucle, no siendo correcto alterar su valor en ningún momento.

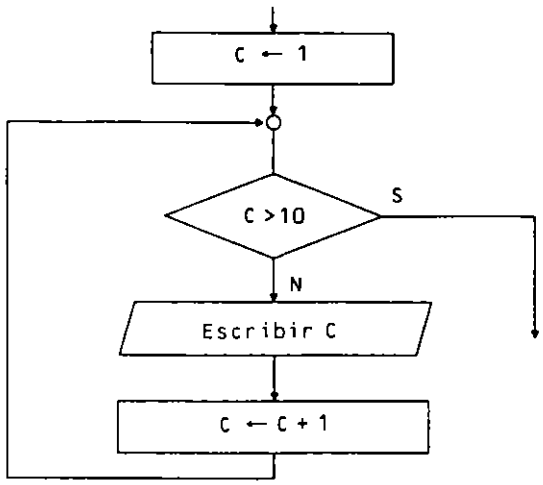
El número de repeticiones de una instrucción *para* viene definido por la fórmula:

$$\text{Número de repeticiones} = \text{parte entera de } \left(\frac{V_f - V_i}{I_n}\right) + 1$$

Su configuración en ordinograma es la siguiente:



Ejemplo *Repetición para escribir la serie de los 10 primeros números naturales.*



- **COBOL:**

```
PERFORM VARYING C FROM 1 BY 1 UNTIL C > 10
    DISPLAY C
END-PERFORM
```

- **Pascal:**

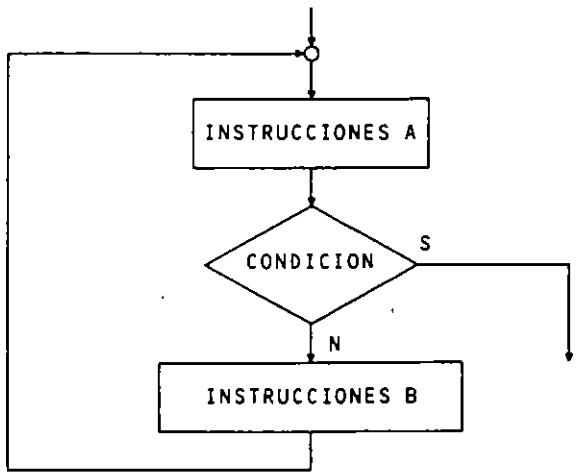
```
FOR C:= 1 TO 10 DO WRITE(C)
```

D) Instrucción ITERAR (LOOP)

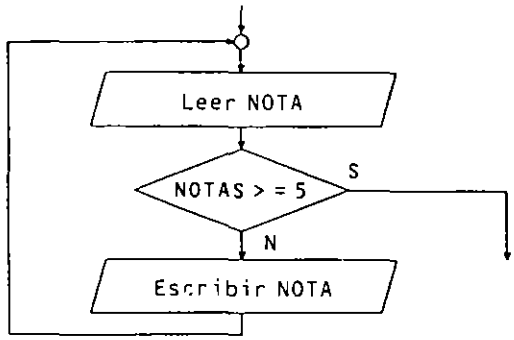
Controla la ejecución de dos conjuntos de instrucciones de manera alternativa, de tal forma que éstos se ejecutan hasta que se cumpla la condición, que será evaluada siempre entre ambos conjuntos de instrucciones.

Uno de los dos conjuntos de instrucciones puede ser el vacio; es decir, puede no tener instrucciones activas, con lo cual este bucle puede realizar funciones similares a los anteriores.

Su configuración en ordinograma es la siguiente:



Ejemplo: *Lectura de una serie de notas para seleccionar la primera que sea aprobada escribiendo las anteriormente leídas.*



No existe en COBOL ni en Pascal. Se simula de la siguiente forma:

- **COBOL:**

```
DISPLAY "INTRODUCIR NOTA"
ACCEPT NOTA
PERFORM UNTIL NOTA > = 5
   DISPLAY NOTA
   DISPLAY "INTRODUCIR NOTA"
   ACCEPT NOTA
END-PERFORM
```

- **Pascal:**

```
WRITE ('INTRODUCIR NOTA');
READ (NOTA);
WHILE NOTA > 5 DO
  BEGIN
    WRITE (NOTA);
    WRITE ('INTRODUCIR NOTA');
    READ (NOTA)
  END
```

□ Instrucciones de ruptura de secuencia

Alteran la secuencia normal de ejecución de instrucciones en un programa que, como se sabe, es desde la primera hasta la última, y de una en una, según aparecen escritas. La alteración de esta secuencia hace que continúe en otro lugar definido en la propia instrucción utilizando lo que se denomina etiqueta.

En la actualidad estas instrucciones han caído en desuso debido a los nuevos métodos de diseño de programas que tratan de eliminarlas. Por ello, solamente nos limitaremos a mencionarlas sin dar pie a su utilización en los ordinogramas, aun sabiendo que en este método han sido muy utilizadas hasta hace poco tiempo.

A) Instrucción de salto incondicional

Altera la secuencia normal de ejecución de las instrucciones de un programa, continuando la misma en la instrucción referenciada por medio de la etiqueta que figura en la propia instrucción.

ir a ⟨etiqueta⟩

B) **Instrucción de salto condicional**

Altera la secuencia normal de ejecución de las instrucciones de un programa si se cumple una condición asociada a la propia instrucción, continuando la misma en la instrucción referenciada por una etiqueta que figura también en la instrucción.

si CONDICION entonces ir a ⟨etiqueta⟩

Estas instrucciones no las representamos en ordinograma, ni en COBOL ni en Pascal intencionadamente.

3.3.4. INSTRUCCIONES COMPUESTAS

Una instrucción compuesta es aquella que representa a un conjunto de instrucciones que están definidas en otra parte. En general son llamadas a subprogramas (funciones, subrutinas, parrafos, etc.).

Su representación en ordinograma utiliza el símbolo de subprograma acompañado de un identificador del subprograma o conjunto de instrucciones representadas.

- **COBOL:**

```
PERFORM NOMBRE-PARRAFO
CALL NOMBRE-SUBPROGRAMA
```

El párrafo o el subprograma estarán en otra parte.

- **Pascal:**

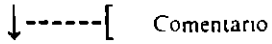
```
NOMBRE_PROCEDIMIENTO
```

El subprograma (PROCEDURE) se encontrará definido en otra parte.

3.3.5. COMENTARIOS

Son frases que se incluyen de forma moderada en el diseño de un algoritmo (en la codificación suele ser más recomendable su uso) con intención de aclarar el cometido o función de un objeto o conjunto de instrucciones.

En ordinograma se representan con el símbolo:



- **COBOL:**

```
Columna 7
   ↓
   *  Comentario
```

- **Pascal:**

```
(* Comentario *)
{ Comentario }
```

3.4. VARIABLES AUXILIARES DE UN PROGRAMA

Son objetos que utiliza un programa y por la función que realizan dentro del mismo toman un nombre especial, modelizando su funcionamiento debido a su frecuente utilización.

3.4.1. CONTADORES

Un contador es un objeto que se utiliza para contar cualquier evento que pueda ocurrir dentro de un programa. En general suelen contar de forma natural desde 0 y de 1 en 1, aunque se pueden realizar otros tipos de cuenta necesarios en algunos procesos.

Se utilizan realizando sobre ellos dos operaciones básicas:

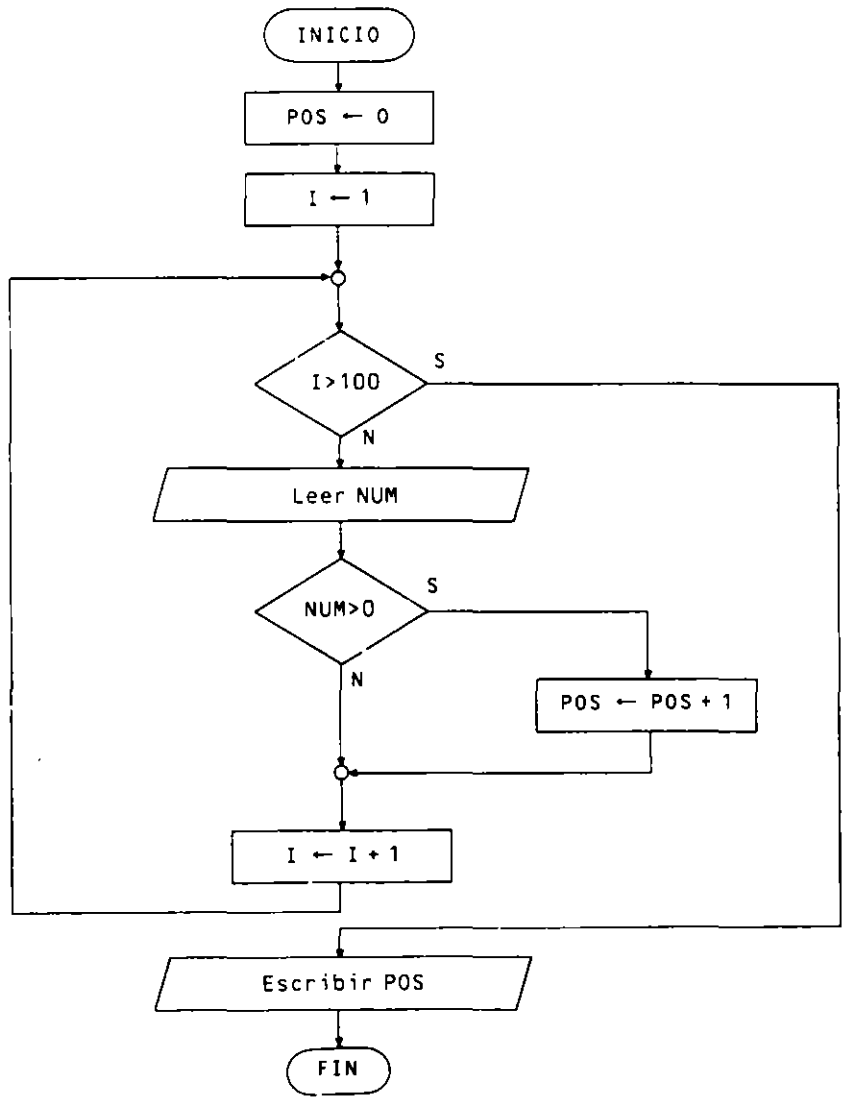
1. **Inicialización.** Todo contador se inicializa a 0 si realiza cuenta natural o a I_i (I_i = Valor inicial) si se desea realizar otro tipo de cuenta.

```
CONTADOR — 0
```

2. **Contabilización o incremento.** Cada vez que aparece el evento a contar se ha de incrementar el contador en 1 si se realiza cuenta natural o en la I_n (Incremento) si se realiza otro tipo de cuenta.

```
CONTADOR ← CONTADOR + 1
```

Ejemplo: *Algoritmo que lee 100 números y cuenta cuántos de ellos son positivos (mayores que 0).*



- **Objetos:**

NUM variable para leer los 100 números

Se utilizan dos contadores: el primero, I, contabiliza la cantidad de números leídos, y el segundo, POS, cuenta el resultado pedido

3.4.2. ACUMULADORES

Son objetos que se utilizan en un programa para acumular elementos sucesivos con una misma operación. En general se utilizan para calcular sumas y productos, sin descartar otros posibles tipos de acumulación.

Al igual que los contadores, para utilizarlos hay que realizar sobre ellos dos operaciones básicas:

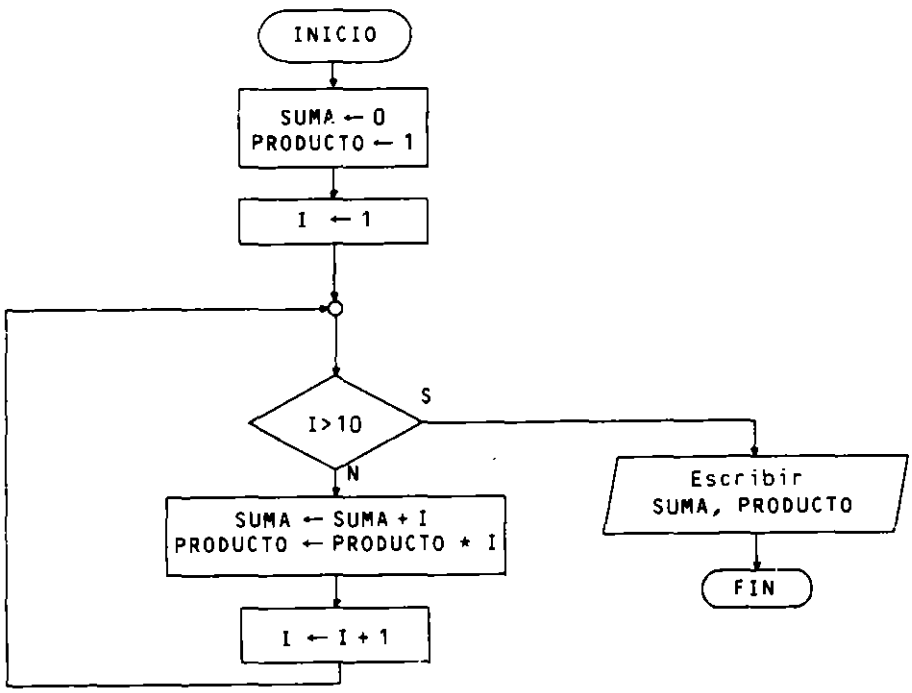
1. **Inicialización.** Todo acumulador necesita ser inicializado con el valor neutro de la operación que va a acumular, que en el caso de la suma es 0 y en el del producto es 1.

```
SUMA ← 0
PRODUCTO ← 1
```

2. **Acumulación.** Cuando se hace presente en la memoria el elemento a acumular por la realización de una lectura o un cálculo, se efectúa la acumulación del mismo por medio de la asignación:

```
SUMA ← SUMA + elemento
PRODUCTO ← PRODUCTO * elemento
```

Ejemplo: *Algoritmo que calcula y escribe la suma y el producto de los 10 primeros números naturales.*



- **Objetos:**

I	contador de 1 a 10.
SUMA	acumulador para calcular la suma.
PRODUCTO	acumulador para calcular el producto.

3.4.3. INTERRUPTORES O CONMUTADORES (SWITCHES)

Los interruptores son objetos que se utilizan en un programa y sólo pueden tomar dos valores (CIERTO y FALSO, 0 y 1), realizando la función de transmitir información de un punto a otro dentro del programa. Podemos decir que actúan como recordatorios manteniendo características de objetos o cálculos que estuvieron presentes en un momento anterior de la ejecución de un programa.

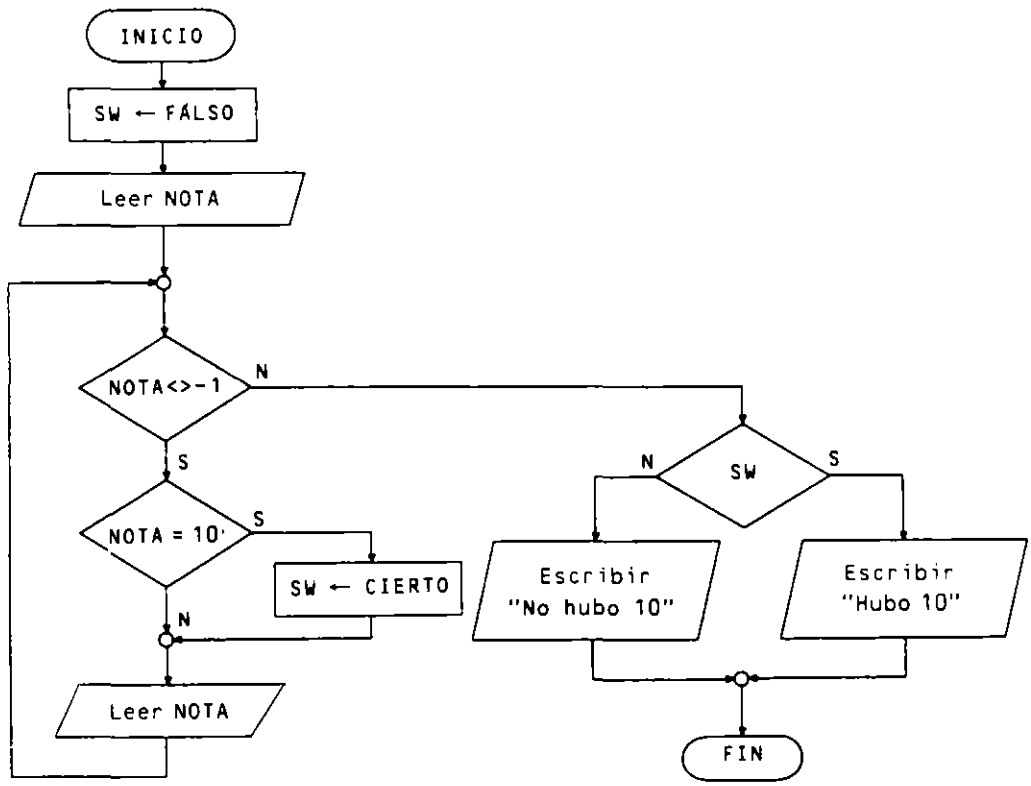
Se utilizan inicializándolos con un valor y en los puntos en que corresponda se cambian al valor contrario, de tal forma que examinando su valor posteriormente podemos realizar la transmisión de información que deseábamos.

Ejemplo: *Algoritmo que lee una secuencia de notas (con valores que van de 0 a 10) que termina con el valor −1 y nos dice si hubo o no alguna nota con valor 10.*

- **Objetos:**

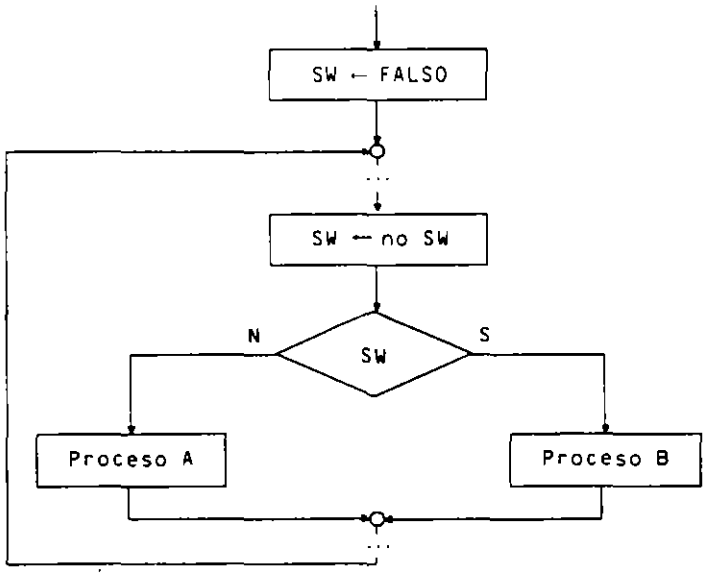
NOTA	variable para leer la secuencia.
SW	switch para controlar la aparición de notas 10 con los siguientes significados:

FALSO . . . . . No hay notas 10.
CIERTO . . . . . Sí hay notas 10.



Un conmutador es un objeto que sólo puede tomar dos valores opuestos (CIERTO y FALSO, 1 y −1) que se utiliza para conmutar entre dos procesos distintos de forma alternativa.

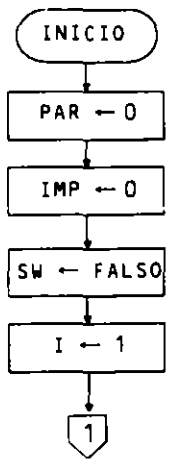
Se utilizan dándoles un valor inicial y en los puntos donde corresponda se niegan (lógica o numéricamente), de tal forma que si después de negarlos examinamos el resultado, será en cada ocasión contrario al anterior

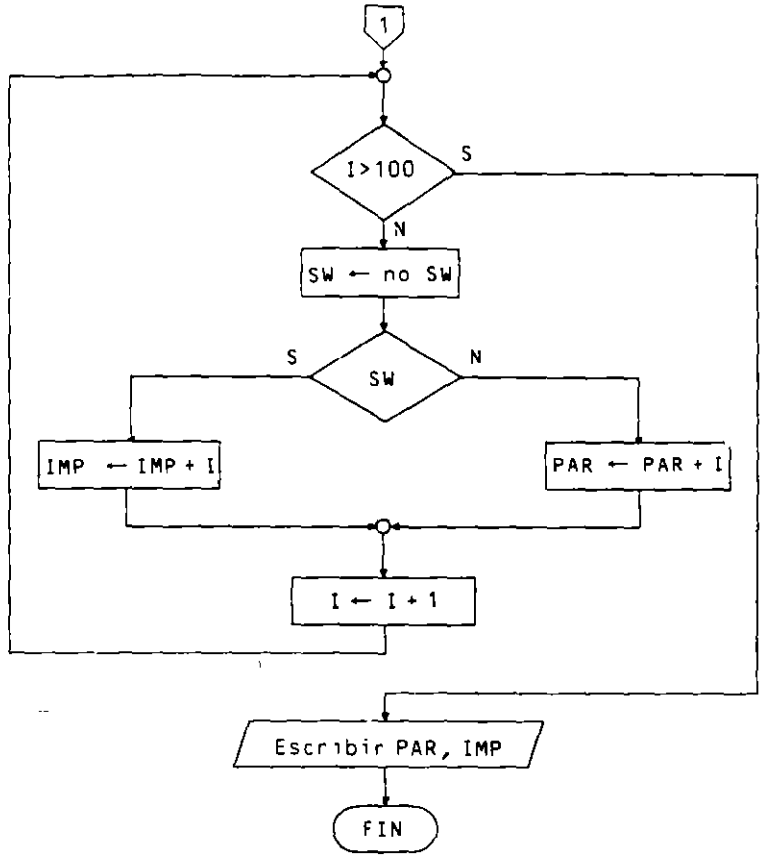


Ejemplo: *Algoritmo que suma independientemente los pares y los impares de los números comprendidos entre 1 y 100.*

- **Objetos:**

I	contador de 1 a 100 que genera la serie.
PAR, IMP	acumuladores para calcular la suma de pares e impares respectivamente.
SW	conmutador significando:
	FALSO . . . . . Par
	CIERTO . . . . . Impar.





3.5. TIPOS DE PROGRAMAS

Un programa, por lo general, estará compuesto por una secuencia de acciones, algunas de las cuales serán alternativas o repetitivas. En determinados programas sencillos no se da esta mezcla de acciones; en este caso los podemos clasificar como sigue:

- **Programas lineales.** Consisten en una secuencia de acciones primitivas (su ejecución es lineal en el orden en que han sido escritas).
- **Programas alternativos.** Consisten en el anidamiento de acciones alternativas entremezcladas con instrucciones primitivas.
- **Programas cíclicos.** Son aquellos en los que un conjunto de instrucciones de las existentes en el programa se repite un número determinado o indeterminado de veces.

Otra clasificación relativa a la aplicación desarrollada por el programa, y que está relacionada con la naturaleza de las operaciones que se realizan en la misma, es:

- **Programas de gestión.** Se caracterizan por el manejo de una gran cantidad de datos con pocos cálculos (resuelven problemas de gestión).
- **Programas técnico-científicos.** Al contrario que los anteriores, realizan una gran cantidad de cálculos con pocos datos (resuelven problemas matemáticos, físicos, etcétera).
- **Programas de diseño (CAD).** Se caracterizan por la utilización de técnicas gráficas para resolver problemas de diseño gráfico.
- **Programas de simulación.** Intentan reflejar una situación real para facilitar su estudio y analizar los problemas que se pueden plantear.
- **Programas educativos (EAO).** Utilizan las ventajas de la computadora para la docencia, convirtiéndola en un profesor para autodidactas.

- **Programas de Inteligencia Artificial.** Se utilizan para simular el razonamiento humano.
- **Etcétera.**

3.6. LENGUAJES DE PROGRAMACION

Un lenguaje de programación es una notación para escribir programas, es decir, para describir algoritmos dirigidos a la computadora.

Un lenguaje viene dado por una gramática o conjunto de reglas que se aplican a un alfabeto.

El primer lenguaje de programación que se utilizó fue el **lenguaje máquina**, que es el único que entiende la computadora y que se diferencia de unas a otras dependiendo del procesador que posean. Su alfabeto se compone exclusivamente de unos y ceros.

El **lenguaje ensamblador** resultó de la simplificación del lenguaje máquina cambiando las cadenas de unos y ceros por símbolos nemotécnicos, existiendo una correspondencia de instrucciones de 1 a 1, con respecto al lenguaje máquina.

Posteriormente surgieron los **lenguajes de alto nivel**, que se alejaban notablemente del lenguaje binario formando instrucciones con frases relativamente parecidas al lenguaje utilizado por las personas y cuya evolución (de los más importantes) se encuentra reflejada en la Figura 3.5

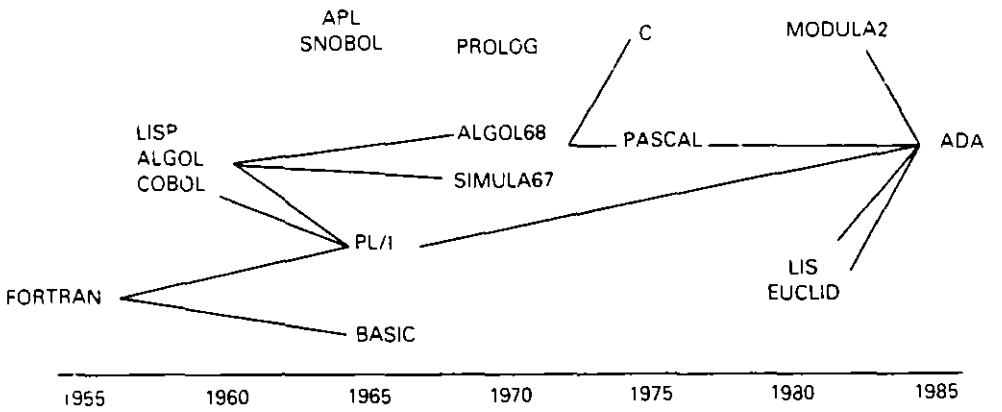


Figura 3.5. Evolución de los lenguajes de alto nivel.

Los lenguajes de programación pueden clasificarse de la siguiente manera:

a) Según su parecido con el lenguaje natural.
- **Bajo nivel:** Lenguajes máquina y ensambladores.
- **Alto nivel:** Todos los demás (próximos al lenguaje natural).

b) Según la estructura de los programas.
- **Convencionales o línea a línea:** Ensambladores, FORTRAN, BASIC, COBOL, etc.
- **Estructurados:** Algol, PL/I, Pascal, C, Ada, COBOL 85, etc.

c) Según la realización de los programas
- **Funcionales:** Lisp, Prolog, APL, etc.
- **Imperativos:** La mayoría.

d) Según el tipo de proceso.
- **Interactivos o conversacionales:** BASIC, COBOL 85, Pascal, etc.
- **Orientados al proceso por lotes (batch):** COBOL, FORTRAN, etc.

Estas clasificaciones no son rigurosas puesto que algunos lenguajes se solapan en una o varias denominaciones.

EJERCICIOS RESUELTOS

1. *Algoritmo que lee una secuencia de 100 números y obtiene e imprime cuántos hay positivos, negativos y nulos.*

Objetos:

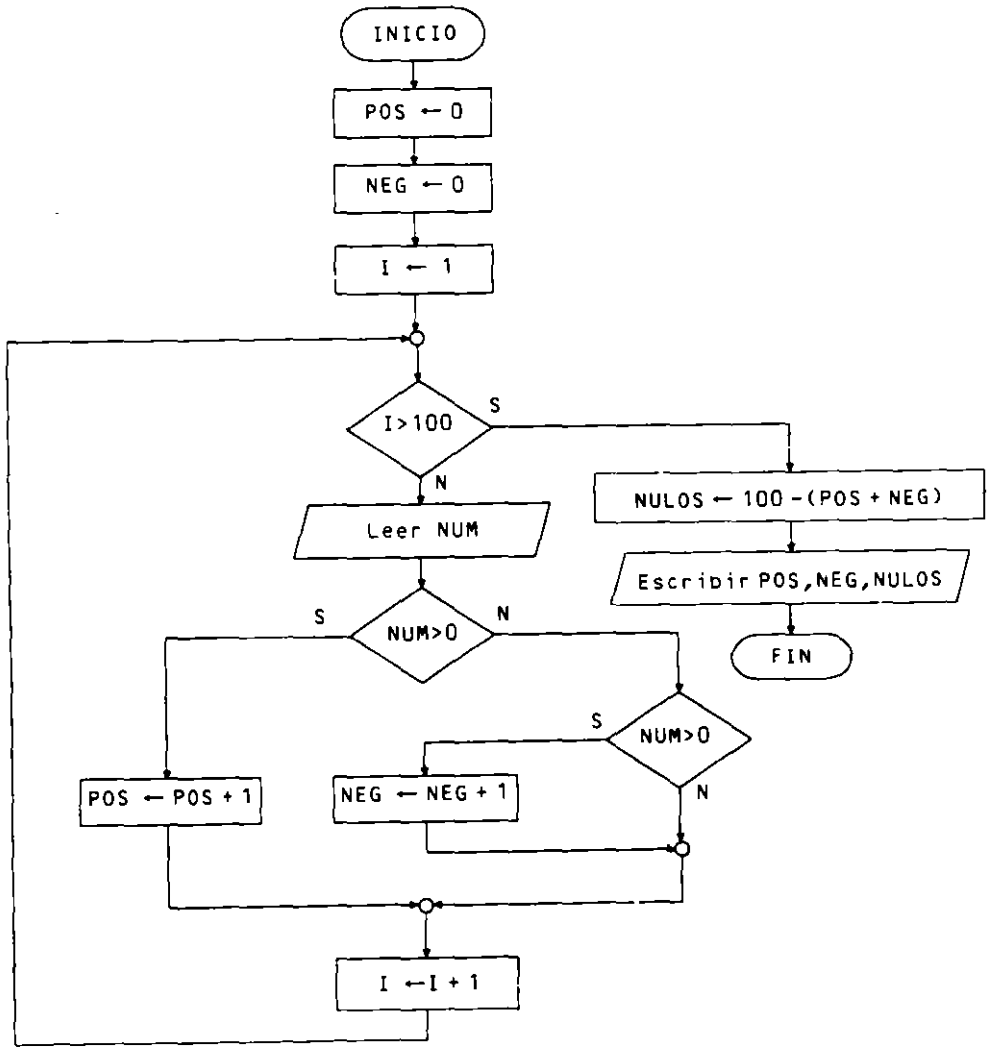
Se utilizan tres contadores:

I para contar el número de lecturas.
POS para contar los mayores que 0.
NEG para contar los menores que 0.

El número de NULOS no es necesario contarlo, pues se calcula al final mediante la expresión:

$$NULOS = 100 - (POS + NEG)$$

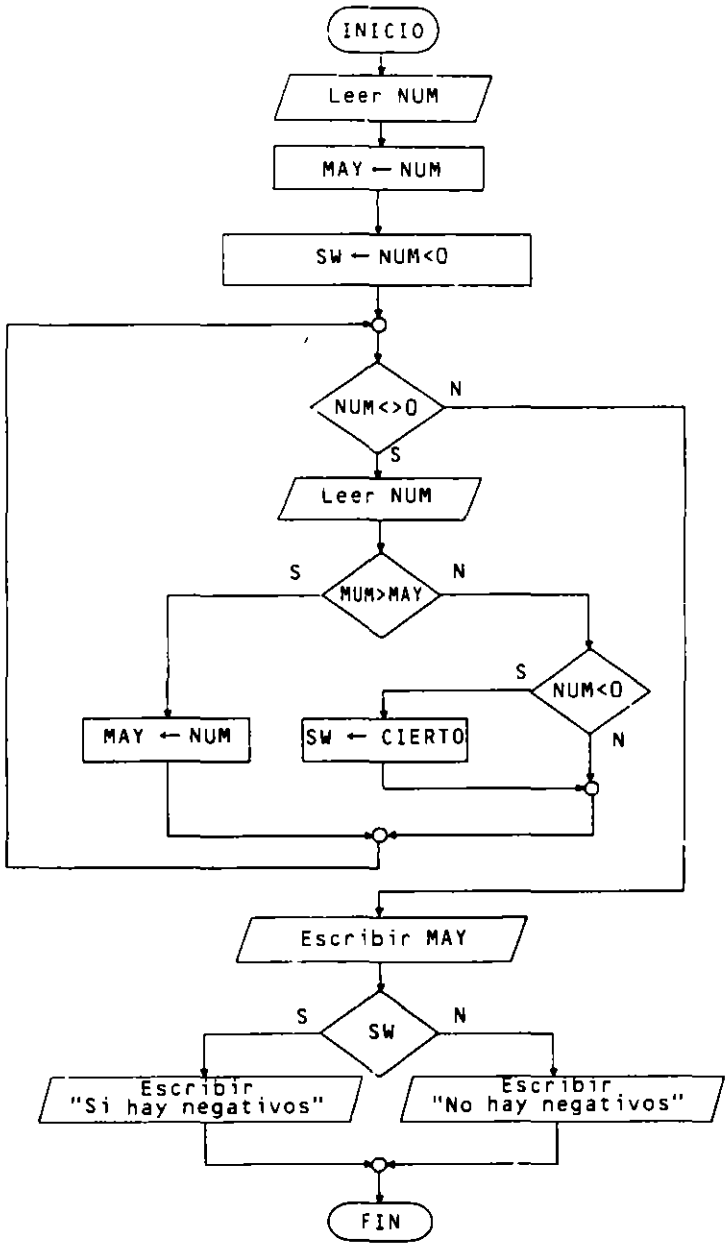
NUM variable para leer los 100 números.



2. *Algoritmo que lee una secuencia de números no nulos, terminada con la introducción de un 0, y obtiene e imprime el mayor, visualizando un mensaje de si se ha leído algún número negativo.*

El programa consiste en un bucle de lectura cuya condición de terminación es la lectura del número 0. Utiliza un interruptor que cambia de valor al detectarse el primer número negativo.

- **Objetos:** NUM variable para leer la secuencia de números.
  MAY variable para retener el número mayor.
  switch para controlar la aparición de números negativos:
  FALSO . . . . . No hay negativos.
  CIERTO . . . . . Si hay negativos.

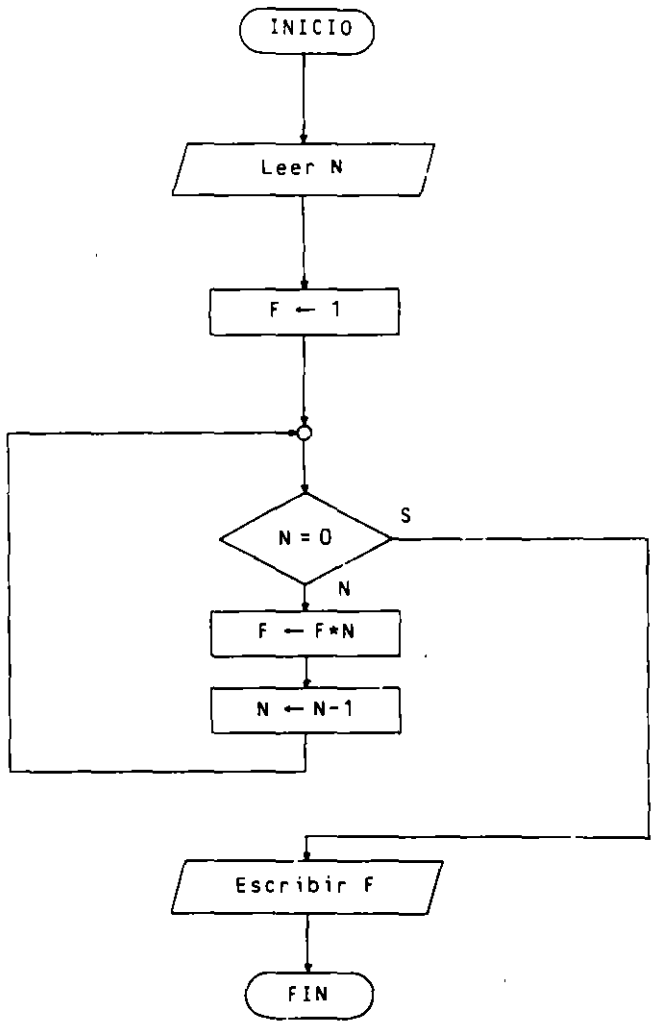


3. *Algoritmo que lee un número entero positivo N y calcula e imprime su factorial N!*

El factorial de un número se calcula:

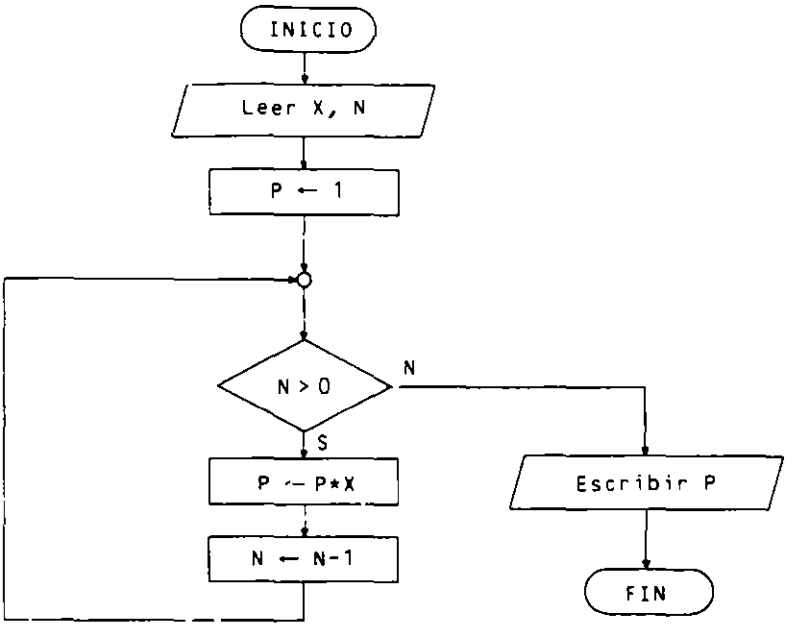
```
0! = 1
1! = 1
2! = 2 * 1
3! = 3 * 2 * 1
...
N! = N * (N - 1) * (N - 2) * ... * 3 * 2 * 1
```

- **Objetos:** N variable para leer el dato de entrada.
  F acumulador para calcular el factorial de N.



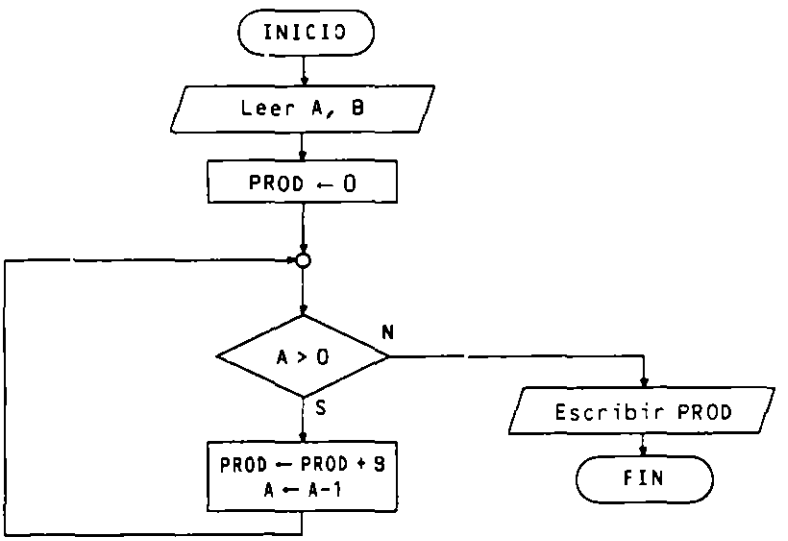
4. *Algoritmo que lee un número X y otro entero positivo N y calcula la N-ésima potencia de X.*

- **Objetos:** X, N variables para leer los datos de entrada (N entero).
  P acumulador para calcular la potencia.



5. *Algoritmo que obtenga el producto de dos números enteros positivos mediante sumas sucesivas.*

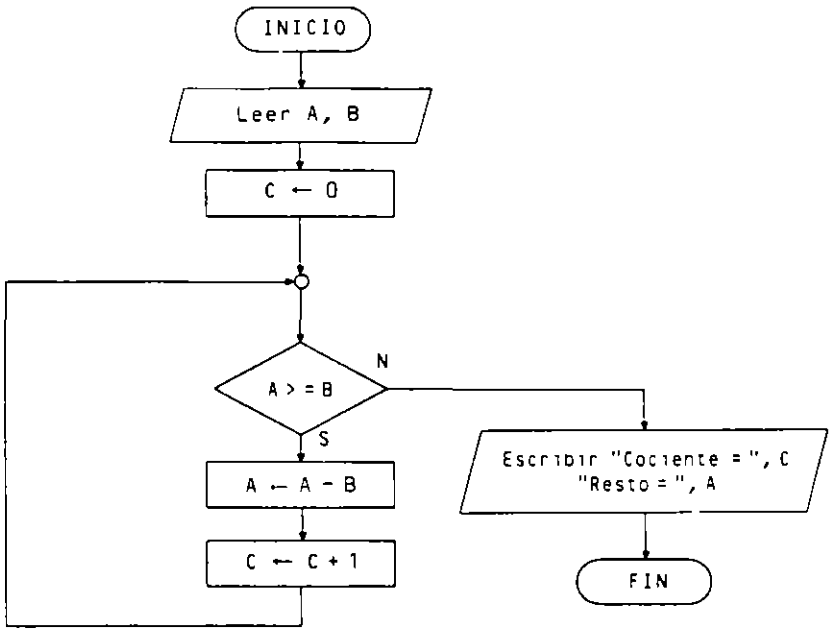
- **Objetos:** A, B variables para leer los datos de entrada.
  PROD acumulador para calcular el producto.



6. *Algoritmo que obtenga el cociente y el resto de dos números enteros positivos mediante restas.*

- **Objetos:** A, B variable para leer los datos de entrada.
  C variable para calcular el cociente entero de la división.

  El resto se calcula en A.

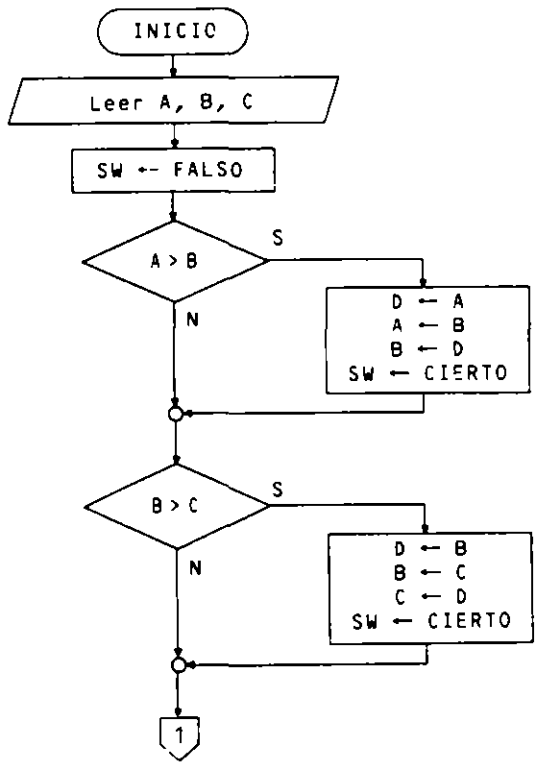


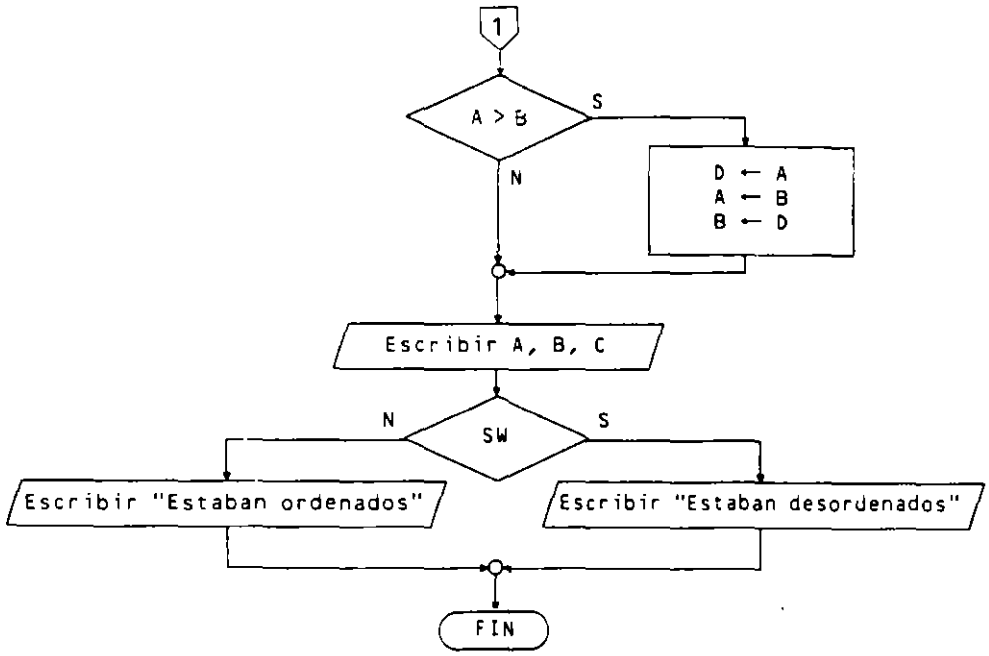
7. *Algoritmo que lee tres números A, B y C, los imprime en orden creciente e indica si fueron introducidos en ese orden.*

- **Objetos:**

A, B, C	variables para tomar los datos de entrada.
D	variable auxiliar para hacer intercambios.
SW	interruptor (switch) para saber si los números se introdujeron ordenados o no:

FALSO . . . . . Ordenados.
CIERTO . . . . . Desordenados.





EJERCICIOS PROPUESTOS

1. Algoritmo que genera la lista de los N primeros números primos, siendo N el dato de entrada.

2. Algoritmo que calcula e imprime el valor del número e como suma de la serie:

$$\Sigma\ 1\ /\ i! = 1\ /\ 0! + 1\ /\ 1! + 1\ /\ 2! + \ldots + 1\ /\ N!$$

   La precisión del resultado será mayor cuanto mayor sea el dato de entrada N (entero positivo).

3. Algoritmo que calcula e imprime los números perfectos menores que 1000. (Un número es perfecto si la suma de sus divisores, excepto él mismo, es igual al propio número.)

4. Algoritmo que calcula el máximo común divisor de dos números enteros positivos por el algoritmo de Euclides.

5. Algoritmo que evalúa un polinomio de grado N. Los datos de entrada son el valor de la variable y de los coeficientes.

6. Algoritmo que determina si dos números enteros positivos son amigos. (Dos números son amigos si la suma de los divisores del primero, excepto él mismo, es igual al segundo, y viceversa.)

7. Algoritmo que lee un número entero positivo N e imprime su tabla de multiplicar.

8. Algoritmo que lee un número entero y positivo N y escribe la lista de sus divisores.

9. Algoritmo que lee un número entero y positivo N y calcula y escribe la suma de sus divisores.

10. Algoritmo que lee un número entero y positivo N y escribe los N primeros términos de la sucesión de Fibonacci. La sucesión se caracteriza porque cada término es igual a la suma de sus dos anteriores, dándose por definición el primero (0) y el segundo (1).

$$a_1 = 0 \qquad a_2 = 1 \qquad a_n = a_{n-1} + a_{n-2}$$

CAPITULO 4

Pseudocódigo

4.1. INTRODUCCION

La solución de un problema, para su ejecución por parte de una computadora, requiere el uso de una notación que sea entendida por ella, es decir, un lenguaje de programación. No obstante, durante la fase de diseño del programa, la utilización de un lenguaje así no es aconsejable debido a su rigidez.

Además de la utilización de las representaciones gráficas (ordinogramas), un programa puede describirse mediante un lenguaje intermedio entre el lenguaje natural y el lenguaje de programación, de tal manera que permita flexibilidad para expresar las acciones que se han de realizar y, sin embargo, imponga algunas limitaciones, importantes desde el punto de vista de su posterior codificación en un lenguaje de programación determinado.

Al igual que las otras técnicas, la utilización de una notación intermedia permite el diseño del programa sin depender de ningún lenguaje de programación, y, es después del diseño, cuando se codificará el algoritmo obtenido en aquel lenguaje que nos interese.

4.2. PSEUDOCODIFICACION DE PROGRAMAS

Diremos que una notación es un **pseudocódigo** si mediante ella podemos describir la solución de un problema en forma de algoritmo dirigido a la computadora, utilizando palabras y frases del lenguaje natural sujetas a unas determinadas reglas.

El pseudocódigo se ha de considerar más bien una herramienta para el diseño de programas que una notación para la descripción de los mismos. Debido a su flexibilidad permite obtener la solución a un problema mediante aproximaciones sucesivas, es decir, mediante lo que se denomina diseño descendente.

Todo pseudocódigo debe posibilitar la descripción de:

- Instrucciones de entrada/salida.
- Instrucciones de proceso.
- Sentencias de control del flujo de ejecución.
- Acciones compuestas que hay que refinar posteriormente.

Asimismo tendrá la posibilidad de describir datos, tipos de datos, constantes, variables, expresiones, archivos y cualquier otro objeto que sea manipulado por el programa.

4.2.1. ACCIONES SIMPLES

Las acciones simples, también denominadas instrucciones primitivas, son aquellas que el procesador ejecuta de forma inmediata.

- **Asignación**

```
VARIABLE ← EXPRESION
```

Almacena en una variable el resultado de evaluar una expresión.

Ejemplo:

```
MEDIA ← SUMA / 6
```

- **Entrada**

```
leer VARIABLE
```

Toma un dato del dispositivo estándar de entrada y lo almacena en una variable. Si se leen varias variables se pueden colocar éstas en una misma instrucción separándolas por comas.

Ejemplo:

```
leer ALUMNO, CALIFICACION
```

- **Salida**

```
escribir EXPRESION
```

Imprime en el dispositivo estándar de salida el resultado de evaluar una expresión. Al igual que en la lectura se pueden imprimir varias expresiones en una sola instrucción de escritura.

Ejemplo:

```
escribir SUMAMEDIAS / NUMALUMNOS
```

Las instrucciones leer y escribir no son en realidad acciones primitivas en ningún lenguaje de programación, pero conviene considerarlas así desde el punto de vista del diseño del programa.

4.2.2. SENTENCIAS DE CONTROL

También se denominan sentencias estructuradas y controlan el flujo de ejecución de otras instrucciones.

- **Secuencia**

```
I1; I2; I3; ... In
```

o bien

```
I1
I2
...
In
```

Se ejecutan las instrucciones I1, I2, ..., In en el mismo orden en que aparecen escritas. Utilizamos el punto y coma como separador de instrucciones que están en la misma línea.

Ejemplo:

```
leer NOTA
SUMA ← SUMA + NOTA
MEDIA ← SUMA / 6
escribir MEDIA
```

- **Alternativa**

— Alternativa simple:

```
si CONDICION
  entonces I1; I2; ... In
finsi
```

En esta instrucción, la condición es una expresión booleana. Las instrucciones I1, I2, ... In se ejecutan solamente si la evaluación de la condición produce el resultado CIERTO; en otro caso no se hace nada.

Ejemplo:

```
si NOTA > 0
  entonces SUMA ← SUMA + NOTA
finsi
```

— Alternativa doble:

```
si CONDICION
  entonces I1; I2; ... In
  sino     J1; J2; ... Jk
finsi
```

Si la evaluación de la condición produce el resultado CIERTO se ejecutarán las instrucciones I1, I2, ... In; en caso contrario, las J1, J2, ... Jk.

Ejemplo:

```
si NOTA < 5
  entonces escribir "SUSPENSO"
  sino     escribir "APROBADO"
finsi
```

— Alternativa múltiple:

```
opción EXPRESION de
  V1   hacer I1; I2; ... Ip
  V2   hacer J1; J2; ... Jq
  ...
  Vn   hacer K1; K2; ... Kr
  otro hacer L1; L2; ... Ls
finopción
```

La expresión puede ser alfanumérica o numérica entera. Si su valor es V1 se ejecutarán las instrucciones I1, I2, ... Ip; si es V2, las J1, J2, ... Jq, etc. Si el valor de la expresión no es ninguno de los indicados explícitamente, V1, V2, ... Vn, entonces se ejecutarán las instrucciones L1, L2, ... Ls.

Ejemplo:

```
opción ESTADO-CIVIL de
  "S" hacer escribir "SOLTERO"
  "C" hacer escribir "CASADO"
  "V" hacer escribir "VIUDO"
  "D" hacer escribir "DIVORCIADO"
  otro hacer escribir "Error: Datos incorrectos"
finopción
```

- **Repeticiones o bucles**

En todo bucle hay una o varias acciones que se han de repetir y una condición que determina el número de repeticiones de las mismas. Es fundamental que el valor de la condición sea afectado por las acciones para asegurar la terminación del bucle en algún momento.

Según que la evaluación de la condición se realice al comienzo, al final o dentro del bucle, se tienen las siguientes sentencias:

— Mientras (**while**):

```
mientras CONDICION hacer
  I1; I2; ... In
finmientras
```

Se evalúa la condición antes de iniciar el bucle, y se repiten sucesivamente las instrucciones I1, I2, ... In, mientras siga siendo CIERTA.

Ejemplo:

```
mientras IN = 0 hacer
  leer NOTA
  si NOTA > 0
    entonces SUMA ← SUMA + NOTA
    sino     IN ← 1
  finsi
finmientras
```

— Repetir (**repeat**):

```
repetir
  I1; I2; ... In
hasta CONDICION
```

Se evalúa la condición después de cada ejecución de las instrucciones I1, I2, ... In y se termina el bucle si es CIERTA.

Ejemplo:

```
repetir
  leer NOTA
  SUMA ← SUMA + NOTA
hasta NOTA = 0
```

— Para (**for**):

```
para Vc de Vi a Vf con incremento In hacer
  I1; I2; ... In
finpara
```

Se repiten las instrucciones I1, I2, ... In, un número fijo de veces tantas como sucesivos valores toma la variable de control del bucle Vc desde inicialmente Vi, incrementándose a cada repetición en In, hasta que el valor de Vc supera Vf.

Si el incremento es +1, que es el caso más usual, el bucle se expresa:

```
para Vc de Vi a Vf hacer
  I1; I2; ... In
finpara
```

Ejemplo:

```
para I de 1 a 6 hacer
  leer NOTA
  SUMA ← SUMA + NOTA
finpara
```

— Iterar (**loop**):

```
iterar
  I1; I2; ... In
  salir si CONDICION
  J1; J2; ... Jk
finiterar
```

Se ejectuan las instrucciones I1, I2, ... In y a continuación se evalúa la condición de salida del bucle: si no es cierta se ejecutan J1, J2, ... Jk, repitiéndose de nuevo el proceso hasta que la condición sea cierta.

Ejemplo:

```
iterar
  leer NOMBRE
  salir si NOMBRE = "FIN"
  leer NOTA1, NOTA2
  MEDIA ← (NOTA1 + NOTA2) / 2
  escribir NOMBRE, MEDIA
finiterar
```

Los bucles **mientras, hasta** y **para** son casos particulares del anterior: es decir, siempre se puede utilizar un bucle **iterar** en lugar de cualquiera de los otros, aunque cada uno de ellos se adapta mejor a una determinada situación.

4.2.3. ACCIONES COMPUESTAS

Un acción compuesta es aquella que ha de ser relizada dentro del algoritmo, pero que aún no está resuelta en términos de acciones simples y sentencias de control.

En el diseño del programa se incluirán los nombres de las acciones compuestas en el algoritmo y posteriormente habrá que refinarlas, sustituyendo cada nombre por las

instrucciones correspondientes o colocándolas aparte, mediante lo que se denomina subprograma, de la siguiente manera.

```
...
I1
...
ACCION-COMPUESTA
...
In
...
Subprograma ACCION-COMPUESTA
  J1
  J2
  ...
  Jk
Finsubprograma
```

La utilización de subprogramas se verá con más detalle en el Capítulo 7.

4.2.4. COMENTARIOS

Son líneas explicativas cuyo objetivo es facilitar la comprensión del programa a quien lo lea. Estas líneas serán ignoradas por el procesador cuando ejecute el programa.

Un programa, en cualquier lenguaje, debe estar ampliamente documentado mediante comentarios intercalados a lo largo de todo su listado. Esto facilitará las posibles y necesarias modificaciones del mismo al simplificar su comprensión al programador que lo diseñó o a otro diferente encargado de su mantenimiento.

Los comentarios se utilizan para aclarar:

- El significado o cometido de un objeto del programa.
- El objetivo de un bloque de instrucciones.
- La utilización de una determinada instrucción.
- Cualquier aspecto del programa que sea necesario aclarar.

En la fase de diseño del programa no es necesario excederse en los comentarios pero sí se han de incluir aquellos que consideremos necesarios.

Se escribirán en cualquier línea a continuación del símbolo **.

```
** Comentario de aclaración
```

4.2.5. OBJETOS DEL PROGRAMA

Podemos considerarlos como los recipientes de los datos que manipula el programa. Será necesario indicar cuáles son sus nombres y sus tipos, lo que se hará previamente a la descripción del conjunto de instrucciones que forman el algoritmo.

El conjunto de objetos de un programa se denomina **entorno**.

Ejemplo:

```
Entorno:
  I es numérica entera
  NOTA es numérica real
  NOMBRE es alfanumérica
```

En la descripción anterior se declaran tres variables: I, que admitirá valores numéricos enteros; NOTA, que admitirá valores numéricos reales, y NOMBRE, que admitirá cadenas de caracteres.

4.2.6. PROGRAMA

Un programa es la solución final de un problema. Esta notación consiste en la descripción de los objetos (**entorno**) y de las instrucciones (**algoritmo**).

Tendrá una cabecera con el nombre del programa y dos bloques precedidos por las palabras «Entorno:» y «Algoritmo:».

En caso de utilizar subprogramas, éstos figurarán a continuación del programa con una sintaxis similar.

```
Programa NOMBRE DEL PROGRAMA
Entorno:
  ** descripción de los objetos
  ...
Algoritmo:
  ** descripción de las acciones
  ...
Finprograma
Subprograma NOMBRE DEL SUBPROGRAMA
Entorno:
  ** descripción de los objetos locales
  ...
Algoritmo:
  ** descripción de las acciones locales
  ...
Finsubprograma
...
```

Ejemplo:

Generación de actas. Se introduce por teclado una secuencia de informaciones, cada una de ellas compuesta por un nombre y seis números correspondientes al nombre de un alumno y las calificaciones que ha obtenido en sus seis asignaturas. La secuencia termina al introducir el nombre «FIN».

Se desea un programa que imprima un listado de calificaciones en el que ha de figurar el nombre del alumno seguido de su nota media. Finalmente se imprimirá la nota media del grupo.

La entrada de datos será de la forma:

```
EMILIO PEREZ GARCIA
5, 8, 7, 5, 6, 6
...
ANA CASAS ORTIZ
5, 3, 6, 2, 7, 4
FIN
```

El listado proporcionado será:

```
                LISTADO DE CALIFICACIONES
                -------------------------

NOMBRE DEL ALUMNO           NOTA MEDIA
-----------------           ----------
EMILIO PEREZ GARCIA             6.1
...                             ...
ANA CASAS ORTIZ                 4.5

          NOTA MEDIA DEL GRUPO: 5.7
```

Diseño del programa:

```
Algoritmo:
  imprimir cabeceras del listado
  inicializar acumuladores
  iterar
     leer NOMBRE
     salir si NOMBRE = "FIN"
     contabilizar alumno
     leer notas del alumno
     calcular su nota media
     imprimir línea con alumno y nota media
     acumular nota media
  finiterar
  obtener nota media del grupo
  imprimir nota media del grupo
Finalgoritmo
```

Mediante refinamientos del algoritmo anterior obtenemos el siguiente programa:

```
Programa GENERACION DE ACTAS
Entorno:
  NOMBRE es alfanumérica
  I, NUMALUMNOS son numéricas enteras
  NOTA, SUMA, MEDIA, SUMAMEDIA, MEDIAGRUPO son numéricas reales
Algoritmo:
  escribir "       LISTADO DE CALIFICACIONES"
  escribir "       -------------------------"
  escribir "NOMBRE DEL ALUMNO          NOTA MEDIA"
  escribir "-----------------          ----------"
  NUMALUMNOS ← 0
  SUMAMEDIA ← 0
  iterar
    leer NOMBRE
    salir si NOMBRE = "FIN"
    NUMALUMNOS ← NUMALUMNOS + 1
    SUMA ← 0
    para I de 1 a 6 hacer
       leer NOTA
       SUMA ← SUMA + NOTA
    finpara
    MEDIA ← SUMA / 6
    escribir NOMBRE, MEDIA
    SUMAMEDIA ← SUMAMEDIA + MEDIA
  finiterar
  MEDIAGRUPO ← SUMAMEDIA / NUMALUMNOS
  escribir " NOTA MEDIA DEL GRUPO:", MEDIAGRUPO
Finprograma
```

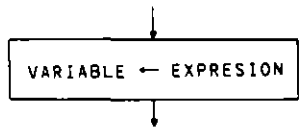
4.3. PASO DE PSEUDOCODIGO A DIAGRAMA DE FLUJO

La traducción es totalmente mecánica y no presenta ningún problema. La traducción inversa, de diagrama de flujo a pseudocódigo, no siempre es posible por ser los diagramas de flujo una notación de más bajo nivel que permite la creación de estructuras de control que quedan fuera del ámbito de la programación estructurada y que no tienen equivalencia en pseudocódigo.

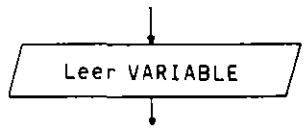
A continuación figuran las estructuras equivalentes en ambas notaciones.

- **Pseudocódigo:**
- **Ordinograma:**

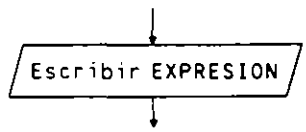
```
VARIABLE ← EXPRESION
```



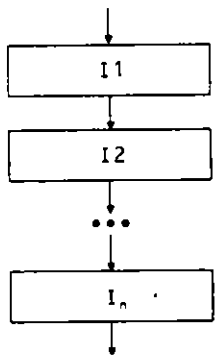

```
leer VARIABLE
```



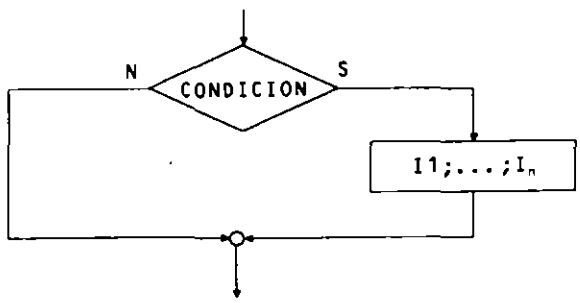

```
escribir EXPRESION
```




```
I1; I2; ... In
```



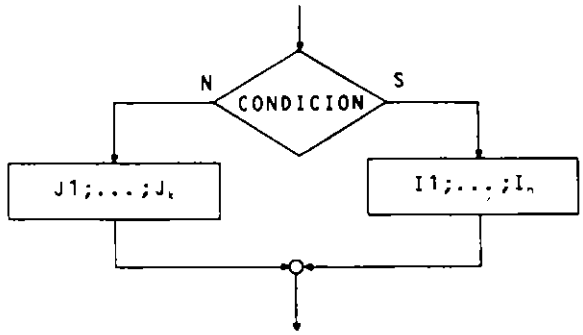

```
si CONDICION
  entonces I1; ... In
finsi
```



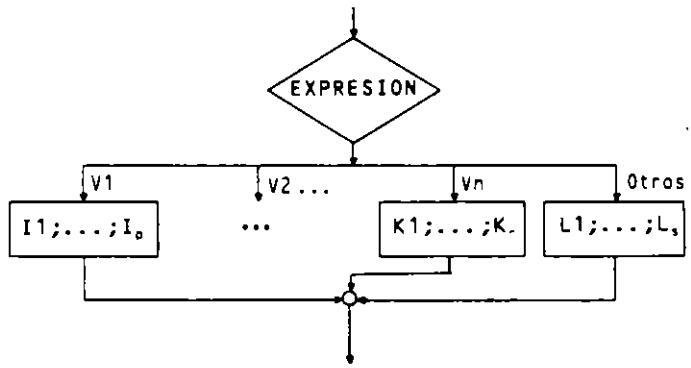
- **Pseudocódigo:**

```
si CONDICION
  entonces I1; ... In
  sino     J1; ... Jk
finsi
```

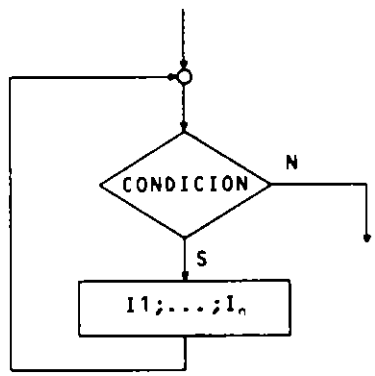
- **Ordinograma**



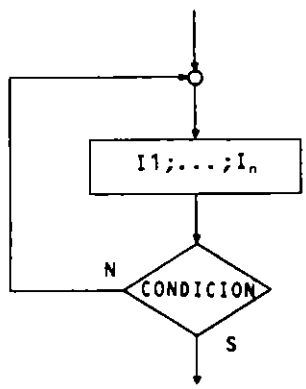

```
opción EXPRESION de
    V1 hacer I1; I2; ... Ip
    V2 hacer J1; J2; ... Jq
    ...
    Vn hacer K1; K2; ... Kr
  otro hacer L1; L2; ... Ls
finopción
```




```
mientras CONDICION hacer
  I1; ... In
finmientras
```



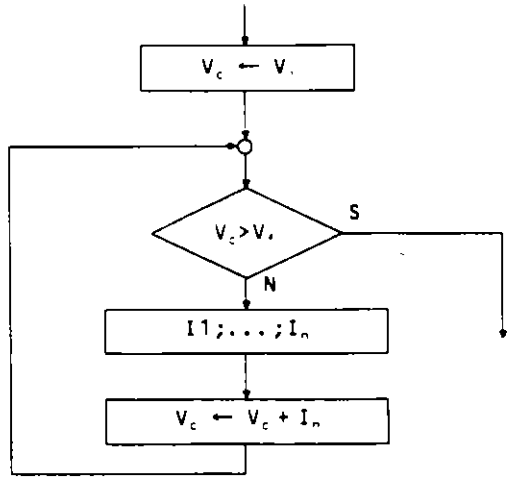

```
repetir
  I1; ... In
hasta CONDICION
```



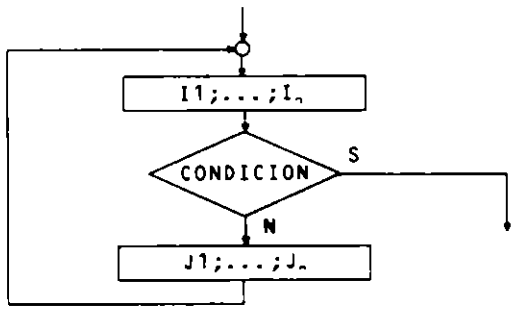
- **Pseudocódigo:**

```
para Vc de Vi a Vf con
    incremento In hacer
  I1; ... In
finpara
```

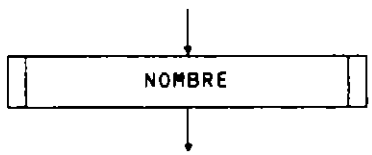
- **Ordinograma**



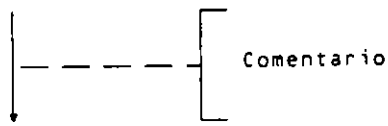

```
iterar
  I1; ... In
  salir si CONDICION
  J1; ... Jk
finiterar
```



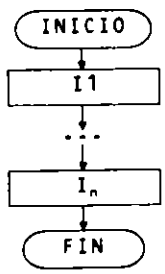

```
ACCION COMPUESTA
```




```
** Comentario
```

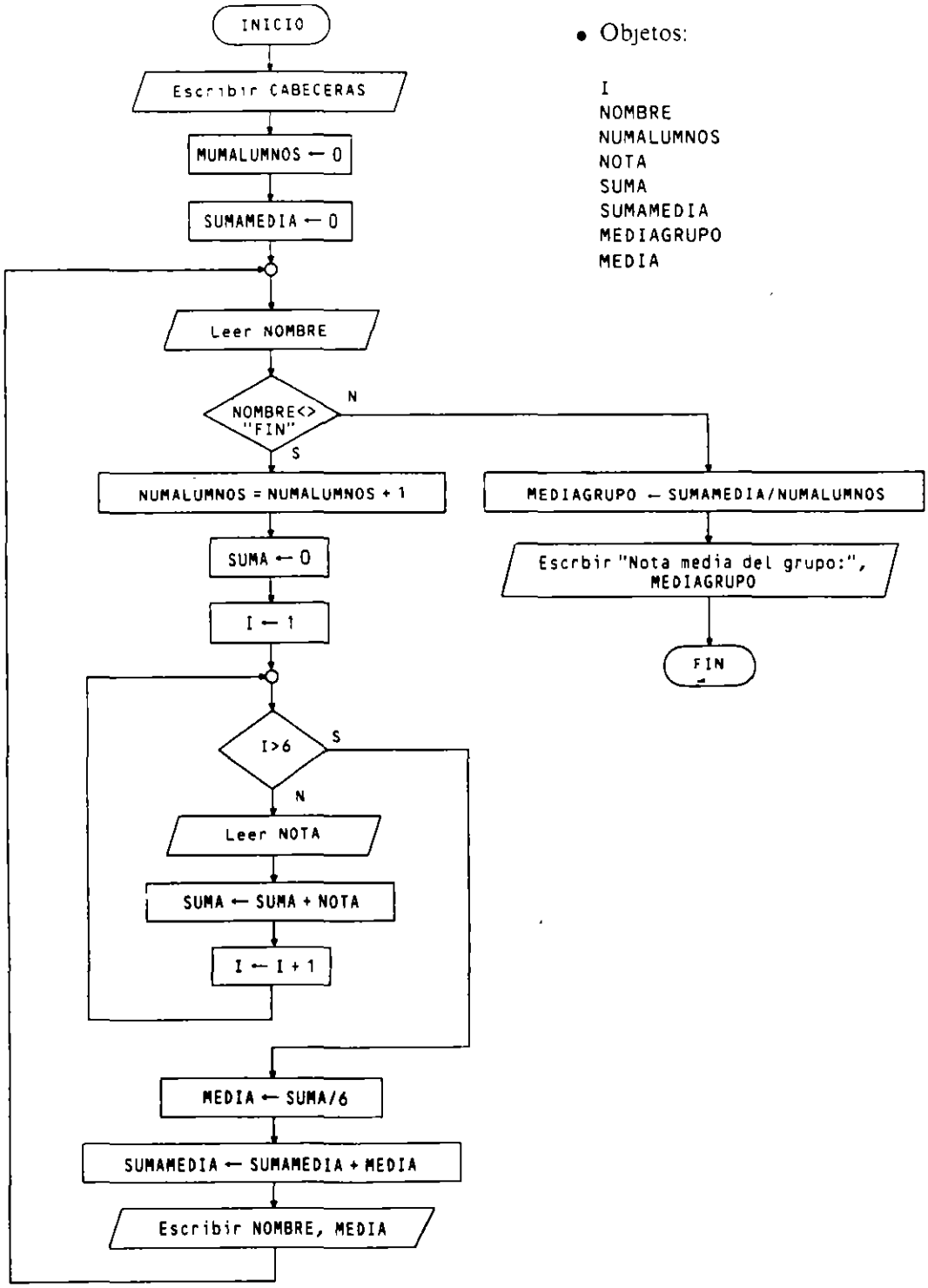



```
Programa NOMBRE-PROGRAMA
Entorno:
  ** Objetos

Algoritmo:
  I1
  ...
  In
Finprograma
```



Ejemplo: *Ordinograma correspondiente al programa anterior.*

GENERACION DE ACTAS



- Objetos:

I
NOMBRE
NUMALUMNOS
NOTA
SUMA
SUMAMEDIA
MEDIAGRUPO
MEDIA

4.4. PASO DE PSEUDOCODIGO A LENGUAJE DE PROGRAMACION

La codificación en un lenguaje de programación de un programa escrito mediante pseudocódigo es, al igual que el paso a ordinograma, bastante sencilla

Algunas sentencias no tienen explícitamente una instrucción equivalente en algún lenguaje, por lo que será preciso implementarlas mediante los recursos disponibles en dicho lenguaje.

A continuación se da una traducción de las sentencias del pseudocódigo a los lenguajes **COBOL** y **Pascal**.

Para ampliar el conocimiento de cada uno de los lenguajes se recomienda leer el manual correspondiente o algunos de los libros que figuran en la bibliografía.

■ **Asignación:** `VARIABLE ← EXPRESION`

- **COBOL:** `COMPUTE VARIABLE = EXPRESION`

Para asignar un valor o el contenido de un campo elemental se utiliza también la siguiente instrucción:

```
MOVE VALOR TO VARIABLE
```

Existen además instrucciones particulares para los casos concretos en que la expresión es una suma (ADD), resta (SUBTRACT), producto (MULTIPLY) o división (DIVIDE).

- **Pascal:** `VARIABLE := EXPRESION`

■ **Entrada:** `Leer VARIABLE`

- **COBOL** `ACCEPT VARIABLE`
- **Pascal:** `READ(VARIABLE)`

■ **Salida:** `Escribir EXPRESION`

- **COBOL:** `DISPLAY VALOR`

Si se quiere escribir el resultado de una expresión que implique operaciones es necesario realizarlas previamente con una sentencia de asignación y a continuación escribir el resultado:

```
COMPUTE VARIABLE = EXPRESION
DISPLAY VARIABLE
```

- **Pascal:** `WRITE(EXPRESION)`

■ **Secuencia:** `I1; I2; ... In`

- **COBOL:** La secuencia está dada por el orden de escritura de las instrucciones, respetando las reglas de formato del lenguaje.

```
I1
I2        o también        I1 I2 ... In
...
In
```

- **Pascal:** Las instrucciones se escriben con formato libre, separándolas con punto y coma.

```
I1;
I2;        o también        I1; I2; ... In
...
In
```

■ **Alternativa simple:**

```
si CONDICION
   entonces I1; ... In
finsi
```

- **COBOL:**

```
IF CONDICION
  THEN I1 ... In
END-IF
```

- **Pascal:**

```
IF CONDICION
  THEN BEGIN
         I1; ... In
       END
```

■ **Alternativa doble:**

```
si CONDICION
  entonces I1; ... In
  sino     J1; ... Jk
finsi
```

- **COBOL:**

```
IF CONDICION
  THEN I1 ... In
  ELSE J1 ... Jk
END-IF
```

- **Pascal:**

```
IF CONDICION
  THEN BEGIN
         I1; ... In
       END
  ELSE BEGIN
         J1; ... Jk
       END
```

■ **Alternativa múltiple:**

```
opción EXPRESION de
   V1 hacer I1; ... Ip
   V2 hacer J1; ... Jq
   ...
   Vn hacer K1; ... Kr
  otro hacer L1; ... Ls
finopción
```

- **COBOL:**

```
EVALUATE EXPRESION
  WHEN V1     I1 ... Ip
  WHEN V2     J1 ... Jq
  ...
  WHEN Vn     K1 ... Kr
  WHEN OTHER L1 ... Ls
END-EVALUATE
```

- **Pascal:**

```
CASE EXPRESION OF
  V1:  BEGIN I1; ... In END;
  V2:  BEGIN J1; ... Jq END;
  ...
  Vn:  BEGIN K1; ... Kr END;
  ELSE BEGIN L1; ... Ls END
END
```

■ **Bucle mientras:**

```
mientras CONDICION hacer
  I1; ... In
finmientras
```

- **COBOL:**

```
PERFOM UNTIL NOT CONDICION
  I1 ... In
END-PERFORM
```

- **Pascal:**

```
WHILE CONDICION DO
  BEGIN
    I1; ... In
  END
```

■ **Bucle repetir:**

```
repetir
  I1; ... In
hasta CONDICION
```

- **COBOL:**

```
PERFORM TEST AFTER UNTIL CONDICION
  I1 ... In
END-PERFORM
```

- **Pascal:**

```
REPEAT
  I1; ... In
UNTIL CONDICION
```

■ **Bucle para:**

```
para Vc de Vi a Vf con incremento In hacer
  I1; ... In
finpara
```

- **COBOL:**

```
PERFORM VARYING Vc FROM Vi BY In UNTIL Vc > Vf
  I1 ... In
END-PERFORM
```

- **Pascal:** Tiene dos sentencias para incrementos $+1$ y -1; para incrementos diferentes es necesario implementarlo a partir de un bucle WHILE.

```
FOR Vc := Vi TO Vf DO (* incremento +1 *)
  BEGIN
    I1; ... In
  END

FOR Vc := Vi DOWNTO Vf DO (* incremento -1 *)
  BEGIN
    I1; ... In
  END
```

■ **Bucle iterar:**

```
iterar
  I1; ... In
  salir si CONDICION
  J1; ... Jk
finiterar
```

No existe explícitamente en ninguno de los dos lenguajes, por lo que se realizará como sigue:

• **COBOL:**

```
I1 ... In
PERFORM UNTIL CONDICION
  J1 ... Jk
  I1 ... In
END-PERFORM
```

• **Pascal:**

```
I1; ... In;
WHILE NOT CONDICION DO
  BEGIN
    J1; ... Jk;
    I1; ... In
  END
```

■ **Comentarios:**

```
** comentario
```

• **COBOL:** Ocupa una línea en la cual se ha escrito el carácter «*» en la columna 7.

```
*  comentario
```

• **Pascal:** Se coloca en cualquier lugar, encerrado entre «(*» y «*)», o bien entre «{» y «}».

```
(* comentario *)
```

■ **Programa:**

```
Programa NOMBRE-DE-PROGRAMA
  ...
Finprograma
```

• **COBOL:** Su forma puede variar según los casos, siendo la más generalizada:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. NOMBRE-DE-PROGRAMA.
...
PROCEDURE DIVISION.
NOMBRE-DE-PARRAFO.
    ...
    STOP RUN.
```

• **Pascal:** Consta de un encabezamiento con el nombre del programa y los archivos utilizados, un bloque de declaraciones (etiquetas, constantes, tipos, variable y subprogramas) y un bloque de instrucciones entre «BEGIN» y «END.».

```
PROGRAM NOMBRE_DE_PROGRAMA (ficheros);
  ...
  BEGIN (* NOMBRE_DE_PROGRAMA *)
    ...
  END.  (* NOMBRE_DE_PROGRAMA *)
```

Ejemplo: *Codificación del programa GENERACION DE ACTAS.*

- **Codificación COBOL:**

```
IDENTIFICATION DIVISION
PROGRAM-ID. GENERACION-DE-ACTAS.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  VARIABLES.
    05  NOMBRE      PIC X(25).
    05  I           PIC 9.
    05  NUMALUMNOS PIC 999.
    05  NOTA        PIC 99V99.
    05  SUMA        PIC 99V99.
    05  MEDIA       PIC 99V99.
    05  SUMAMEDIA   PIC 9999V99.
    05  MEDIAGRUPO PIC 99V99.
    05  NOTAEDI     PIC Z9.99.
PROCEDURE DIVISION.
PROCESO.
    DISPLAY "       LISTADO DE CALIFICACIONES" UPON PRINTER
    DISPLAY "       ------------------------" UPON PRINTER
    DISPLAY " " UPON PRINTER
    DISPLAY "NOMBRE DEL ALUMNO          NOTA MEDIA" UPON PRINTER
    DISPLAY "-----------------          ----------" UPON PRINTER
    MOVE 0 TO NUMALUMNOS
    MOVE 0 TO SUMAMEDIA
    DISPLAY "Escriba nombre del alumno o FIN "
    ACCEPT NOMBRE
    PERFORM UNTIL NOMBRE = "FIN"
      ADD 1 TO NUMALUMNOS
      MOVE 0 TO SUMA
      PERFORM VARYING I FROM 1 BY  1 UNTIL I > 6
        DISPLAY "Escriba nota número ", I
        ACCEPT NOTA
        ADD NOTA TO SUMA
      END-PERFORM
      DIVIDE SUMA BY 6 GIVING MEDIA
      ADD MEDIA TO SUMAMEDIA
      MOVE MEDIA TO NOTAEDI
      DISPLAY NOMBE, NOTAEDI UPON PRINTER
      DISPLAY "Escriba nombre del alumno o FIN "
      ACCEPT NOMBRE
    END-PERFORM
    DIVIDE SUMAMEDIA BY NUMALUMNOS GIVING MEDIAGRUPO
    MOVE MEDIAGRUPO TO NOTAEDI
    DISPLAY " " UPON PRINTER
    DISPLAY "  NOTA MEDIA DEL GRUPO: ", NOTAEDI UPON PRINTER
    STOP RUN.
```

- **Codificación Pascal:**

```
PROGRAM GENERACION_DE_ACTAS (INPUT, OUTPUT, LST);
VAR NOMBRE : STRING[25];
    I, NUMALUMNOS : INTEGER;
    NOTA, SUMA, MEDIA, SUMAMEDIA, MEDIAGRUPO : REAL;
BEGIN (*GENERACION_DE_ACTAS*)
  WRITELN(LST, '       LISTADO DE CALIFICACIONES');
  WRITELN(LST, '       ------------------------');
  WRITELN(LST);
```

```
  WRITELN(LST, 'NOMBRE DEL ALUMNO          NOTA MEDIA');
  WRITELN(LST, '-----------------          ----------');
  NUMALUMNOS := 0;
  SUMAMEDIA := 0;
  WRITE('Escriba nombre del alumno o FIN ');
  READLN(NOMBRE);
  WHILE NOMBRE <> 'FIN' DO
  BEGIN (*1*)
    NUMALUMNOS := NUMALUMNOS + 1;
    SUMA := 0;
    FOR I := 1 TO 6 DO
    BEGIN (*2*)
      WRITE('Escriba nota número ',I);
      READLN(NOTA);
      SUMA := SUMA + NOTA
    END; (*2*)
    MEDIA := SUMA / 6;
    SUMAMEDIA := SUMAMEDIA + MEDIA;
    WRITELN(LST, NOMBRE, MEDIA);
    WRITE('Escriba nombre del alumno o FIN ');
    READLN(NOMBRE);
  END; (*1*)
  MEDIAGRUPO := SUMAMEDIA / NUMALUMNOS;
  WRITELN(LST);
  WRITELN(LST, ' NOTA MEDIA DEL GRUPO: ', MEDIAGRUPO)
END. (*GENERACION_DE_ACTAS*)
```

EJERCICIOS RESUELTOS

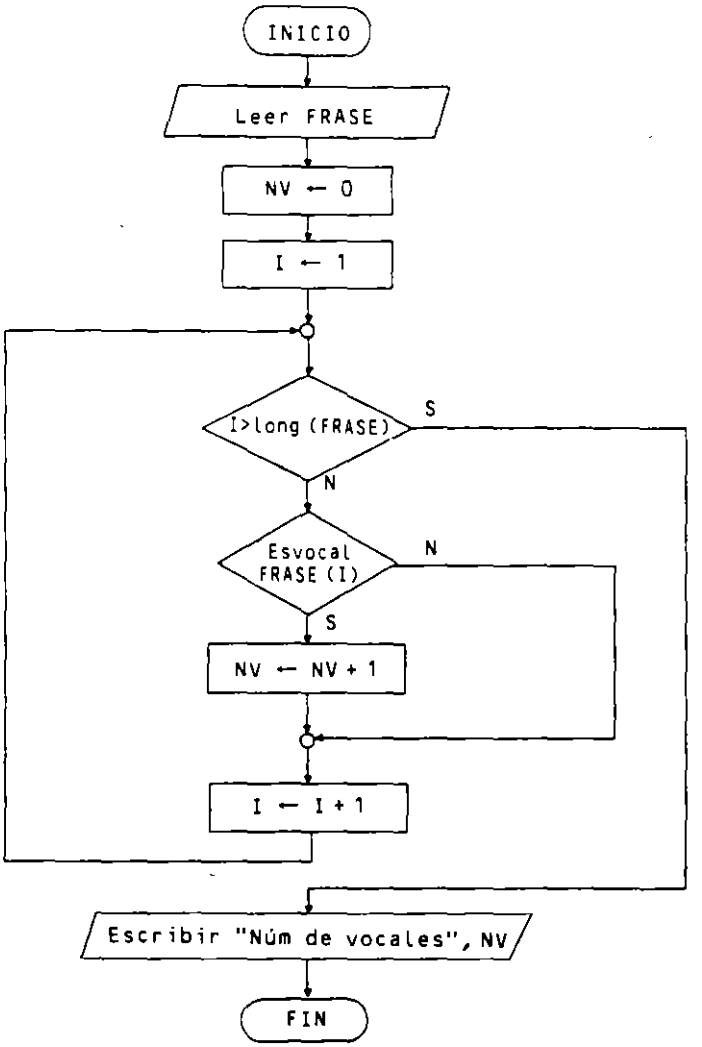
1. *Programa que lee una frase en una línea y cuenta su número de vocales.*

La frase se lee en una variable alfanumérica y mediante un índice se recorren y examinan todos sus caracteres.

- **Pseudocódigo:**

```
Programa CONTAR VOCALES
Entorno:
  FRASE es alfanumérica
  NV, I son numéricas enteras
Algoritmo:
  leer FRASE
  NV ← 0
  para I de 1 a longitud(FRASE) hacer
    si es vocal el carácter I de FRASE
      entonces NV ← NV + 1
    finsi
  finpara
  escribir "Número de vocales de la frase: ", NV
Finprograma
```

- **Ordinograma:**



- **Codificación COBOL:**

El acceso a cada uno de los caracteres de la frase, que se almacena en una variable alfanumérica, se realiza mediante un método denominado «modificación por referencia» que permite seleccionar un trozo de la misma, desde una posición y con una longitud determinadas:

```
nombre-de-variable (posición-izquierda : longitud)
```

```
IDENTIFICATION DIVISION.
PROGRAM-ID. CONTAR-VOCALES.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  VARIABLES.
    05  FRASE          PIC X(255).
    05  NUMERO-VOCALES PIC 999.
    05  I              PIC 999.
PROCEDURE DIVISION.
```

```
PROCESO.
    DISPLAY "Escriba una frase en una linea"
    ACCEPT FRASE
    MOVE 0 TO NUMERO-VOCALES
    PERFORM VARYING I FROM 1 BY 1 UNTIL I > 255
      IF FRASE (I:1) = "a" OR = "e" OR = "i" OR = "o" OR = "u"
                     OR = "A" OR = "E" OR = "I" OR = "O" OR = "U"
        THEN ADD 1 TO NUMERO-VOCALES
      END-IF
    END-PERFORM
    DISPLAY "Número de vocales de la frase: ", NUMERO-VOCALES
    STOP RUN.
```

Otra solución diferente utiliza la instrucción «INSPECT», que cuenta las repeticiones de un determinado carácter en una cadena alfanumérica. En algunas versiones de COBOL esta instrucción se denomina «EXAMINE».

```
IDENTIFICATION DIVISION.
PROGRAM-ID. CONTAR-VOCALES.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  VARIABLES.
    05  FRASE           PIC X(255).
    05  NUMERO-VOCALES PIC 999.
PROCEDURE DIVISION.
PROCESO.
    DISPLAY "Escriba una frase en una linea"
    ACCEPT FRASE
    MOVE 0 TO NUMERO-VOCALES
    INSPECT FRASE TALLYING NUMERO-VOCALES
      FOR ALL "a", "e", "i", "o", "u", "A", "E", "I", "O", "U"
    DISPLAY "Número de vocales de la frase: ", NUMERO-VOCALES
    STOP RUN.
```

- **Codificación Pascal:**

```
PROGRAM CONTAR_VOCALES (INPUT, OUTPUT);
VAR  FRASE : STRING[255];
     NUMERO_VOCALES, I : INTEGER;
     C : CHAR;
BEGIN (*CONTAR_VOCALES*)
  WRITELN('Escriba una frase');
  READLN(FRASE);
  NUMERO_VOCALES := 0;
  FOR I := 1 TO LENGTH(FRASE) DO
  BEGIN (*1*)
    C := FRASE[I];
    IF (C='a') OR (C='e') OR (C='i') OR (C='o') OR (C='u') OR
       (C='A') OR (C='E') OR (C='I') OR (C='O') OR (C='U')
      THEN NUMERO_VOCALES := NUMERO_VOCALES + 1
  END; (*1*)
  WRITELN('Número de vocales de la frase: ', NUMERO_VOCALES)
END. (*CONTAR_VOCALES*)
```

Otra solución en **Pascal** lee la frase carácter a carácter hasta el fin de línea examinando su valor.

```
PROGRAM CONTAR_VOCALES (INPUT, OUTPUT);
VAR     C : CHAR;
        NUMERO_VOCALES, I : INTEGER;
```

```
BEGIN (*CONTAR_VOCALES*)
  WRITELN('Escriba una frase');
  NUMERO_VOCALES := 0;
  WHILE NOT EOLN DO
  BEGIN (*1*)
    READ(C);
    IF (C='a') OR (C='e') OR (C='i') OR (C='o') OR (C='u') OR
       (C='A') OR (C='E') OR (C='I') OR (C='O') OR (C='U')
      THEN NUMERO_VOCALES := NUMERO_VOCALES + 1
  END; (*1*)
  WRITELN('Número de vocales de la frase: ', NUMERO_VOCALES)
END. (*CONTAR-VOCALES*)
```

2. *Programa que obtiene e imprime la lista de interés producido y capital acumulado anualmente, por un capital inicial* C, *impuesto con un rédito* R *durante* N *años a interés compuesto.*

El interés anual obtenido se calcula mediante la fórmula:

$$I = \frac{C \cdot R}{100}$$

El capital se incrementa cada año con los intereses producidos en el mismo

- **Pseudocódigo:**

```
Programa INTERES-COMPUESTO
Entorno:
  C, R, I son numéricas reales
  N, A son numéricas enteras
Algoritmo:
  leer C, R, N
  escribir "Capital inicial: ", C, "Rédito: ", R, "%"
  escribir "Año     Intereses     Capital acumulado"
  escribir "---     ---------     -----------------"
  para A de 1 a N hacer
    I ← C * R / 100
    C ← C + I
    escribir A, I, C
  finpara
Finprograma
```

- **Codificación COBOL:**

```
IDENTIFICATION DIVISION.
PROGRAM-ID. INTERES-COMPUESTO.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  VARIABLES.
    05  C        PIC 9(7)V99.
    05  R        PIC 99V99.
    05  I        PIC 9(7)V99.
    05  N        PIC 99.
    05  A        PIC 99.
01  CAMPOS-EDITADOS.
    05  SALIDAC PIC Z(8)9.99.
    05  SALIDAR PIC Z9.99.
    05  SALIDAI PIC Z(8)9.99
    05  SALIDA  PIC Z9.
```

```
PROCEDURE DIVISION.
PROCESO.
    DISPLAY "Capital impuesto: ", ACCEPT C
    DISPLAY "Rédito en %: ", ACCEPT R
    DISPLAY "Años de imposición: ", ACCEPT N
    MOVE C TO SALIDAC, MOVE R TO SALIDAR
    DISPLAY "Capital inicial: ", SALIDAC, " Rédito: ", SALIDAR, "%"
    DISPLAY "Año    Intereses    Capital acumulado"
    DISPLAY "---    ---------    ------------------ "
    PERFORM VARYING A FROM 1 BY 1 UNTIL A > N
      COMPUTE I = C * R / 100
      ADD I TO C
      MOVE A TO SALIDAA
      MOVE I TO SALIDAI
      MOVE C TO SALIDAC
      DISPLAY SALIDAA, "    ", SALIDAI , "    ", SALIDAC
    END-PERFORM
    STOP RUN.
```

- **Codificación Pascal:**

```
PROGRAM INTERES_COMPUESTO (INPUT, OUTPUT);
VAR C, R, I : REAL;
    N, A : INTEGER;
BEGIN (*INTERES_COMPUESTO*)
  WRITE('Capital impuesto: '); READLN(C);
  WRITE('Rédito en %: '); READLN(R);
  WRITE('Años de imposición: '); READLN(N);
  WRITELN('Capital inicial: ', C, ' Rédito: ', R , '%');
  WRITELN('Año    Intereses    Capital acumulado');
  WRITELN('---    ---------    ------------------');
  FOR A := 1 TO N DO
  BEGIN (*1*)
    I := C * R / 100;
    C := C + I;
    WRITELN(A, '  ', I, '     ', C)
  END (*1*)
END. (*INTERES_COMPUESTO*)
```

3. *Programa que calcula e imprime el valor del número* e *como suma de la serie:*

$$\Sigma\ 1\ /\ i! = 1\ /\ 0! + 1\ /\ 1! + 1\ /\ 2! + \ldots + 1\ /\ N!$$

El resultado se obtendrá con una precisión superior a un valor *P*, que será el dato de entrada.

El programa ha de obtener y acumular sumandos de la serie hasta el primero que sea inferior a *P*.

- **Pseudocódigo:**

```
Programa NUMERO-E
Entorno:
  P, NUME, SUMANDO son numéricas reales
  N, FAC son numéricas enteras
Algoritmo:
  escribir "Introduzca la precisión: "
  leer P
  N ← 0
  FAC ← 1
```

```
  NUME ← 1; ** Se inicializa con el primer sumando
  repetir
    N ← N + 1
    FAC ← FAC * N
    SUMANDO ← 1 / FAC
    NUME ← NUME + SUMANDO
  hasta SUMANDO < P
  escribir "Valor del número e = ", NUME
finprograma
```

- **Codificación COBOL:**

```
IDENTIFICATION DIVISION.
PROGRAM-ID. NUMERO-E.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  VARIABLES.
    05  P          PIC 9V9(8).
    05  NUM-E      PIC 9V9(8).
    05  SUMANDO    PIC 9V9(8).
    05  N          PIC 99.
    05  FAC        PIC 9(8).
01  CAMPOS-EDITADOS.
    05  SALIDA-E PIC 9.9(8).
PROCEDURE DIVISION.
PROCESO.
    DISPLAY "Introduzca precisión: "
    ACCEPT P
    MOVE 0 TO N
    MOVE 1 TO FAC
    MOVE 1 TO NUM-E
    PERFORM TEST AFTER UNTIL SUMANDO < P
      ADD 1 TO N
      MULTIPLY N BY FAC
      DIVIDE 1 BY FAC GIVING SUMANDO
      ADD SUMANDO TO NUM-E
    END-PERFORM
    MOVE NUME TO SALIDA-E
    DISPLAY "Valor del número e = ", SALIDA-E
    STOP RUN.
```

- **Codificación Pascal:**

```
PROGRAM NUMERO_E (INPUT, OUTPUT);
VAR  P, NUM_E, SUMANDO : REAL;
     N, FAC : INTEGER;
BEGIN (*NUMERO_E*)
  WRITE('Introduzca la precisión: ');
  READLN(P);
  N := 0;
  FAC := 1;
  NUM_E := 1;
  REPEAT
    N := N + 1;
    FAC := FAC * N;
    SUMANDO := 1 / FAC;
    NUM_E := NUM_E + SUMANDO
  UNTIL SUMANDO < P;
  WRITELN('Valor del número e = ', NUM_E)
END. (*NUMERO_E*)
```

4. *Programa que calcula el producto de dos números enteros positivos mediante el denominado «algoritmo ruso del producto».*

El algoritmo consiste en duplicar el primer factor y dividir (cociente entero) por 2 el segundo, obteniendo un producto equivalente, salvo si el segundo factor es impar: en este caso es necesario acumular previamente el primero en donde se va a obtener el resultado. El proceso termina cuando el segundo factor se hace 0.

Ejemplo: *Multiplicación de 25 por 6.*

Primer factor	Segundo factor	Acumulador
25	6	0
50	3	50
100	1	150
200	0	150

- **Pseudocódigo:**

```
Programa ALGORITMO-RUSO
Entorno:
  PF, SF, AC son numéricas enteras
Algoritmo:
  leer PF, SF
  AC ← 0
  mientras SF <> 0 hacer
    si es impar SF
      entonces AC ← AC + PF
    finsi
    PF ← PF * 2
    SF ← parte entera(SF / 2)
  finmientras
  escribir "El producto es: ", AC
Finprograma
```

- **Codificación COBOL:**

```
IDENTIFICATION DIVISION.
PROGRAM-ID. ALGORITMO-RUSO.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  VARIABLES.
    05  PF     PIC 9(6).
    05  SF     PIC 9(6).
    05  AC     PIC 9(12).
    05  COCI   PIC 9(6).
    05  RESTO  PIC 9.
01  CAMPOS-EDITADOS.
    05  SALIDA PIC Z(11)9.
PROCEDURE DIVISION.
PROCESO.
    DISPLAY "Introduzca primer factor: ", ACCEPT PF
    DISPLAY "Introduzca segundo factor: ", ACCEPT SF
    MOVE 0 TO AC
    PERFORM UNTIL SF = 0
      DIVIDE SF BY 2 GIVING COCI REMAINDER RESTO
      IF RESTO = 1
        THEN ADD PF TO AC
      END-IF
```

```
        MULTIPLY 2 BY PF
        MOVE COCI TO SF
      END-PERFORM
      MOVE AC TO SALIDA
      DISPLAY "El producto es: ", SALIDA
      STOP RUN.
```

Codificación Pascal:

```
PROGRAM ALGORITMO_RUSO (INPUT, OUTPUT);
VAR PF, SF, AC : INTEGER;
BEGIN (*ALGORITMO_RUSO*)
  WRITE('Introduzca primer factor: ');
  READLN(PF);
  WRITE('Introduzca segundo factor: ');
  READLN(SF);
  AC := 0;
  WHILE SF <> 0 DO
  BEGIN (*1*)
    IF ODD(SF)
      THEN AC := AC + PF;
    PF := PF * 2;
    SF := SF DIV 2
  END; (*1*)
  WRITELN('El producto es: ', AC)
END. (*ALGORITMO_RUSO*)
```

5. *Programa que lee 100 datos, compuesto cada uno de ellos por un nombre de persona y su sueldo neto mensual, y obtiene e imprime el nombre y sueldo de la persona que más cobra y de la que menos. Si hay varias se imprime la primera que aparezca en la secuencia de entrada.*

- **Pseudocódigo:**

```
Programa SUELDOS
Entorno:
  NOMBRE, NMAX, NMIN son alfanuméricas
  SUELDO, SMAX, SMIN son numéricas reales
  I es numérica entera
Algoritmo:
  leer NOMBRE, SUELDO
  NMAX ← NOMBRE
  SMAX ← SUELDO
  NMIN ← NOMBRE
  SMIN ← SUELDO
  para I de 2 a 100 hacer
    leer NOMBRE, SUELDO
    si SUELDO > SMAX
      entonces NMAX ← NOMBRE
               SMAX ← SUELDO
      sino si SUELDO < SMIN
             entonces NMIN ← NOMBRE
                      SMIN ← SUELDO
           finsi
    finsi
  finpara
  escribir NMAX, SMAX, NMIN, SMIN
finprograma
```

- **Codificación COBOL:**

```
IDENTIFICATION DIVISION.
PROGRAM-ID. SUELDOS.
DATA DIVISION.
WORKING STORAGE SECTION.
01  VARIABLES.
    05  NOMBRE PIC X(30).
    05  NMAX   PIC X(30).
    05  NMIN   PIC X(30).
    05  SUELDO PIC 9(6).
    05  SMAX   PIC 9(6).
    05  SMIN   PIC 9(6).
01  CAMPOS-EDITADOS.
    05  SL     PIC Z(5)9.
PROCEDURE DIVISION.
PROCESO.
    DISPLAY "Nombre: "
    ACCEPT NOMBRE
    DISPLAY "Sueldo: "
    ACCEPT SUELDO
    MOVE NOMBRE TO NMAX, NMIN
    MOVE SUELDO TO SMAX, SMIN
    PERFORM 99 TIMES
      DISPLAY "Nombre: "
      ACCEPT NOMBRE
      DISPLAY "Sueldo: "
      ACCEPT SUELDO
      IF SUELDO > SMAX
        THEN MOVE NOMBRE TO NMAX
             MOVE SUELDO TO SMAX
        ELSE IF SUELDO < SMIN
               THEN MOVE NOMBRE TO NMIN
                    MOVE SUELDO TO SMIN
             END-IF
      END-IF
    END-PERFORM
    MOVE SMAX TO SL
    DISPLAY "Cobra más: ", NMAX, " con ", SL, " pts."
    MOVE SMIN TO SL
    DISPLAY "Cobra menos: ", MMIN, " con ", SL, " pts."
    STOP RUN.
```

- **Codificación Pascal:**

```
PROGRAM SUELDOS (INPUT, OUTPUT);
VAR NOMBRE, NMAX, NMIN : STRING[30];
    SUELDO, SMAX, SMIN : REAL;
    I : INTEGER;
BEGIN (*SUELDOS*)
  WRITE('Nombre: ');
  READLN(NOMBRE);
  WRITE('Sueldo: ');
  READLN(SUELDO);
  NMAX := NOMBRE;
  SMAX := SUELDO;
  NMIN := NOMBRE;
  SMIN := SUELDO;
  FOR I := 2 TO 100 DO
  BEGIN (*1*)
```

```
      WRITE('Nombre: ');
      READLN(NOMBRE);
      WRITE('Sueldo: ');
      READLN(SUELDO);
      IF SUELDO > SMAX
        THEN BEGIN (*2*)
               NMAX := NOMBRE;
               SMAX := SUELDO
             END (*2*)
        ELSE IF SUELDO < SMIN
               THEN BEGIN (*3*)
                      NMIN := NOMBRE;
                      SMIN := SUELDO
                    END (*3*)
  END; (*1*)
  WRITELN('Cobra más: ', NMAX, ' con ', SMAX, ' pts.');
  WRITELN('Cobra menos: ', NMIN, ' con ', SMIN, ' pts.')
END. (*SUELDOS*)
```

EJERCICIOS PROPUESTOS

1. Escribir mediante pseudocódigo los ejercicios resueltos y propuestos del Capitulo 2.

2. Escribir mediante pseudocódigo los ejercicios resueltos y propuestos del Capitulo 3.

3. Programa que recibe como datos de entrada dos números enteros positivos N y M, y calcula e imprime los múltiplos de N, inferiores a M, que sean capicúas.

4. Programa que lee una fecha en formato dia (1 a 31), mes (1 a 12) y año (en número) y obtiene el número de orden del día en el total del año.

   Ejemplo: Si se lee 01 03 1992, se obtiene 61° de 1992.

5. Programa que lee el número de orden de un día y su año e indica de qué fecha se trata.

   Ejemplos: Si lee 61 1992 se obtiene 01 03 1992.

6. Programa que lee un número N entero positivo y obtiene su descomposición factorial (sus factores primos).

7. Programa que lee dos números $N1$ y $N2$ enteros positivos y obtiene su mínimo común múltiplo. (Se sabe que el mínimo común múltiplo de dos números es igual a su producto $N1 \cdot N2$ dividido entre su MCD.)

8. Programa que lee una secuencia de calificaciones numéricas (entre 0 y 10) que termina con el valor -1 y calcula e imprime la media aritmética, el número y porcentaje de aprobados y el número y porcentaje de suspensos.

9. Programa que lee una secuencia de 50 nombres y escribe el número de veces que se repite el primero de ellos.

10. Programa que calcula el valor de e^X según la fórmula:

$$e^X = \sum_{i=0}^{N} \frac{X^i}{i!} = \frac{X^0}{0!} + \frac{X^1}{1!} + \frac{X^2}{2!} + \cdots + \frac{X^N}{N!}$$

Siendo los datos de entrada X (real) y N (entero positivo).

CAPITULO 5

Estructuras de datos internas (tablas)

5.1. INTRODUCCION

En los capítulos anteriores, los datos manejados en los programas han sido los denominados datos simples (numéricos o alfanuméricos).

En un gran numero de problemas es necesario manejar un conjunto de datos, más o menos grande, que están relacionados entre sí, de tal forma que constituyen una unidad para su tratamiento. Por ejemplo, si se quiere manipular una lista de 100 nombres de personas, es conveniente tratar este conjunto de datos de forma unitaria en lugar de utilizar 100 variables, una para cada dato simple.

Un conjunto de datos homogéneos que se tratan como una sola unidad se denomina **estructura de datos**.

Si una estructura de datos reside en la memoria central de la computadora se denomina **estructura de datos interna**. Recíprocamente, si reside en un soporte externo, se denomina **estructura de datos externa**.

Las estructuras de datos internas pueden ser de dos tipos:

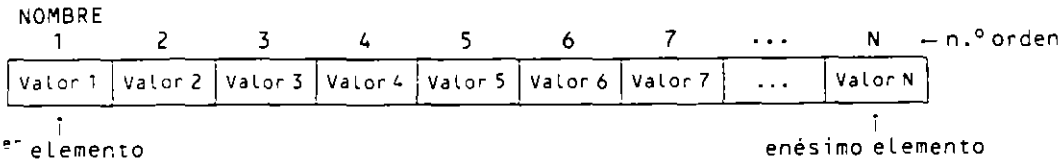
- **Estáticas.** Tienen un número fijo de elementos que queda determinado desde la declaración de la estructura en el comienzo del programa.
- **Dinámicas.** Tienen un número de elementos que varía a lo largo de la ejecución del programa. Estas estructuras se estudian en el Capítulo 11.

La estructura de datos interna más importante, desde el punto de vista de utilización, es la **tabla**, que existe en casi todos los lenguajes de programación.

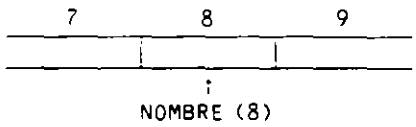
Esta estructura se corresponde con los conceptos matemáticos de vector, matriz y poliedro.

5.2. CONCEPTOS Y DEFINICIONES

Una **tabla** consiste en un número fijo, finito y ordenado de elementos, todos del mismo tipo y bajo un **nombre** común para todos ellos.



El nombre de una tabla es un identificador común para todos sus elementos, distinguiéndose cada uno por una lista de índices que complementan a dicho nombre para referenciarlos.



Se denomina **componentes** a los elementos de una tabla.

Una tabla se puede estructurar en una, dos o más dimensiones, según el número de índices necesarios para acceder a sus elementos. Por tanto, la **dimensión** de una tabla es el número de índices que utiliza.

Longitud o **tamaño** de una tabla es el número de componentes que contiene.

El **tipo** de una tabla es el tipo de sus componentes (numéricos, alfanuméricos, etc.).

La posición de cada componente dentro de la tabla está determinada por uno o varios **índices.** A cada componente se puede acceder de forma directa indicando el nombre de la tabla y sus índices.

Un índice puede estar expresado, en una referencia a un elemento de una tabla, de tres formas:

- Por un valor numérico entero (por ejemplo 8).
- Por una variable numérica entera (por ejemplo I).
- Por una expresión numérica entera (por ejemplo I + 1).

Las componentes de una tabla se utilizan de la misma forma que cualquier otra variable de un programa, pudiendo por tanto intervenir en instrucciones de asignación, entrada/salida, etc.

Ejemplo: *Tabla que contiene ocho nombres de cuatro caracteres.*

ALUMNOS

1	2	3	4	5	6	7	8
LUIS	JOSE	ROSA	JUAN	TERE	RAFA	JAVI	LOLA

TABLA: La estructura de datos representada.

NOMBRE DE LA TABLA: ALUMNOS.

COMPONENTES: ALUMNOS(1), ALUMNOS(2), ... ALUMNOS(8).

INDICE: Los números del 1 al 8 que direccionan cada componente.

DIMENSION: Una.

LONGITUD: Ocho.

TIPO: Alfanumérica.

En los diferentes lenguajes de programación hay que declarar las tablas antes de su utilización.

La tabla anterior se declara de la siguiente manera:

```
COBOL:  01 TABLA-ALUMNOS.
           02 ALUMNOS OCCURS 8 TIMES PIC 2(4).
Pascal: ALUMNOS : ARRAY[1..8] OF DTRING[4];
```

5.3. TIPOS DE TABLAS

Las tablas se clasifican según su dimensión en:

— **Unidimensionales.**
— **Bidimensionales.**
— **Multidimensionales.**

5.3.1. TABLAS UNIDIMENSIONALES (VECTORES)

Son tablas de una dimensión. También se denominan **vectores**.

Tienen un solo índice. Cada componente del vector se direcciona mediante su **nombre** seguido del número correspondiente al **índice** entre paréntesis.

El ejemplo anterior nos muestra una tabla de este tipo, en la cual la componente tercera que contiene el valor «ROSA» se denota por:

```
ALUMNOS(3)
```

Si queremos intercambiar los contenidos de las componentes primera y segunda, lo haremos utilizando una variable alfanumérica auxiliar AUX de la siguiente manera:

```
AUX ← ALUMNOS(1)
ALUMNOS(1) ← ALUMNOS(2)
ALUMNOS(2) ← AUX
```

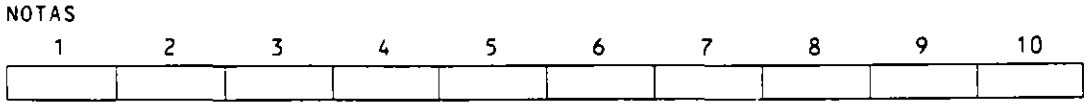
Los elementos de un vector se almacenan en la memoria interna de la computadora de forma consecutiva, desde el primer elemento hasta el último.

```
ALUMNOS(1),ALUMNOS(2),...,ALUMNOS(8)
```

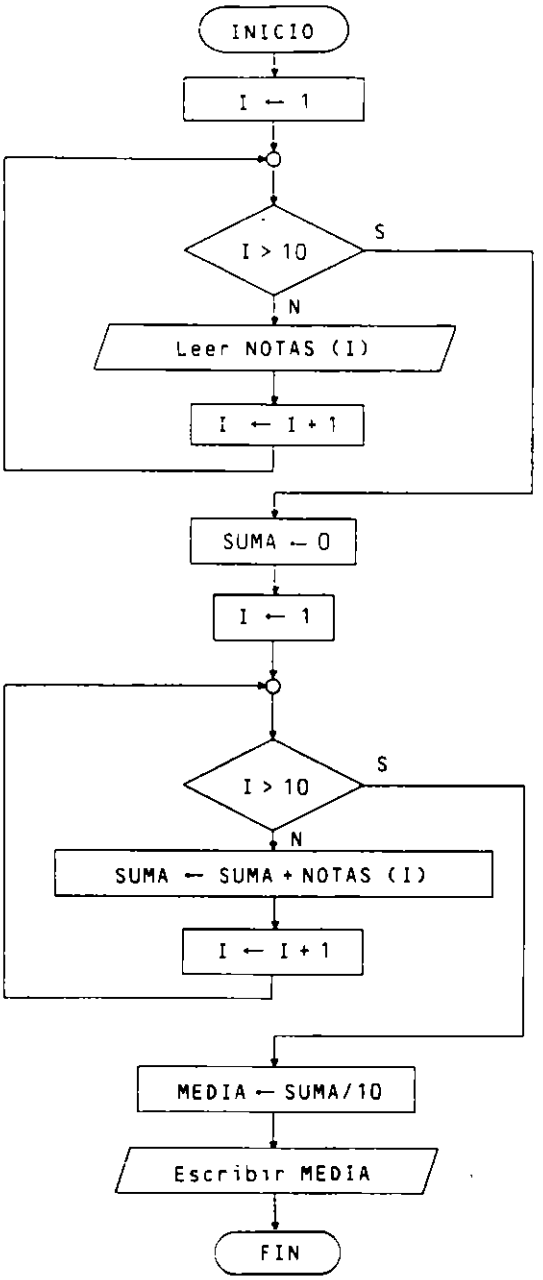
Ejercicio: *Programa que lee las calificaciones de un alumno en 10 asignaturas, las almacena en un vector y calcula e imprime su media.*

- **Objetos:**

I es el contador asociado a los bucles e índice de la tabla.
SUMA es un acumulador de notas.
MEDIA es una variable para el cálculo final.



Vector numérico de 10 elementos para retener las notas.



- **Pseudocódigo:**

```
Programa NOTA MEDIA
Entorno:
  NOTAS es tabla(10) numérica real
  SUMA, MEDIA son numéricas reales
  I es numérica entera
Algoritmo:
  para I de 1 a 10 hacer
    escribir "Escriba nota ", I, "ª:"
    leer NOTAS(I)
  finpara
  SUMA ← 0
```

```
  para I de 1 a 10 hacer
    SUMA ← SUMA + NOTAS(I)
  finpara
  MEDIA ← SUMA / 10
  escribir "Nota media: ", MEDIA
Finprograma
```

- **Codificación COBOL:**

```
IDENTIFICATION DIVISION.
PROGRAM-ID. NOTA-MEDIA.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  TABLA.
    02 NOTAS OCCURS 10 TIMES PIC 99V99.
01  VARIABLES.
    05 SUMA   PIC 999V99.
    05 MEDIA  PIC 99V99.
    05 I      PIC 99.
    05 SALIDA PIC Z9.99.
PROCEDURE DIVISION.
PROCESO.
    PERFORM VARYING I FROM 1 BY 1 UNTIL I > 10
       DISPLAY "Nota nº ", I NO ADVANCING
       ACCEPT NOTAS(I)
    END-PERFORM
    MOVE 0 TO SUMA
    PERFORM VARYING I FROM 1 BY 1 UNTIL I > 10
       ADD NOTAS(I) TO SUMA
    END-PERFORM
    DIVIDE SUMA BY 10 GIVING MEDIA
    MOVE MEDIA TO SALIDA
    DISPLAY "Nota media: ", SALIDA
    STOP RUN.
```

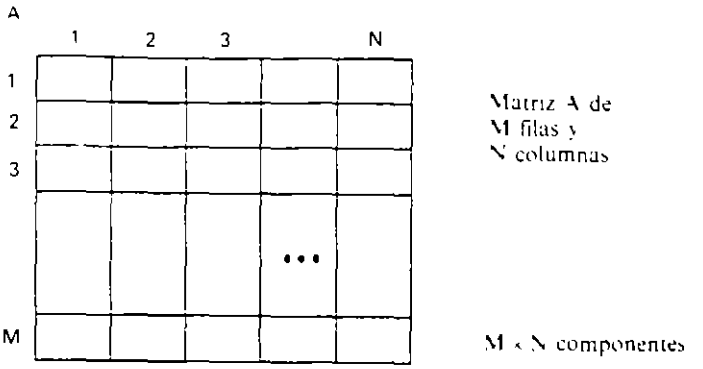
- **Codificación Pascal:**

```
PROGRAM NOTAMEDIA (INPUT,OUTPUT);
VAR NOTAS : ARRAY[1..10] OF REAL;
    SUMA, MEDIA : REAL;
    I : INTEGER;
BEGIN (*NOTAMEDIA*)
  FOR I := 1 TO 10 DO
  BEGIN (*1*)
    WRITE('Nota nº ', I);
    READLN(NOTAS[I])
  END; (*1*)
  SUMA := 0;
  FOR I := 1 TO 10 DO
    SUMA := SUMA + NOTAS[I];
  MEDIA := SUMA / 10;
  WRITE('Nota media: ', MEDIA)
END.(*NOTAMEDIA*)
```

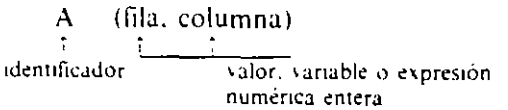
5.3.2. TABLAS BIDIMENSIONALES (MATRICES)

Son tablas de dos dimensiones. También se denominan **matrices**.

Tienen dos índices. por lo cual cada componente de la matriz se direcciona mediante su **nombre** seguido de los dos **índices** separados por coma y entre paréntesis.

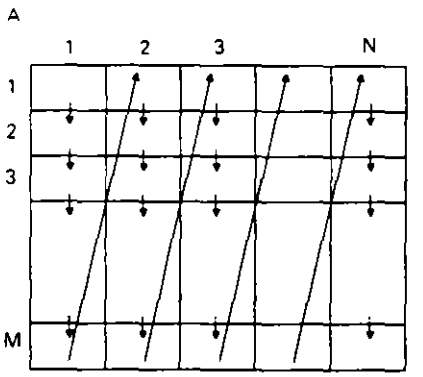


Direccionamiento de un elemento:



En general, los elementos de una matriz se almacenan en la memoria interna de la computadora por columnas, es decir:

A(1,1), A(2,1), A(3,1), ..., A(M,1), A(1,2), A(2,2), A(3,2), ...



Ejemplo: *Matriz de seis filas y ocho columnas que contiene el número de alumnos matriculados en cada grupo de un centro docente por asignatura. Las filas corresponden a los grupos y las columnas a las asignaturas.*

MATRICULA

	1	2	3	4	5	6	7	8
1	35	30	32	32	34	35	34	28
2	40	33	40	37	36	39	40	29
3	25	23	26	21	24	24	25	15
4	33	33	33	32	34	30	32	20
5	45	44	45	44	43	40	44	33
6	24	20	22	22	24	25	24	12

NOMBRE DE LA TABLA: MATRICULA.

COMPONENTES: MATRICULA(1,1), MATRICULA(1,2), ..., MATRICULA(6,7), MATRICULA(6,8).

INDICES: Los números del 1 al 6 para las filas y los números 1 al 8 para las columnas.

DIMENSION: Dos.

LONGITUD: 6 * 8 = 48

TIPO: Numérica entera.

La componente MATRICULA(4,6) almacena el número de alumnos matriculados en el grupo número 4 en la asignatura número 6, que en el ejemplo representado son 30 alumnos.

La tabla anterior se declara:

COBOL:
```
01 TABLA-MATRICULA.
   02 GRUPO OCCURS 6 TIMES.
      03 MATRICULA PIC 99 OCCURS 8 TIMES.
```

Pascal:
```
MATRICULA : ARRAY[1..6,1..8] OF INTEGER;
```

Ejercicio: *Programa que carga la tabla del ejemplo anterior y a continuación calcula e imprime el total de alumnos matriculados por asignatura.*

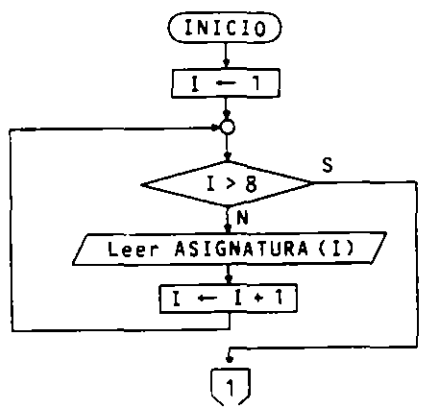
El nombre de las asignaturas es el siguiente:

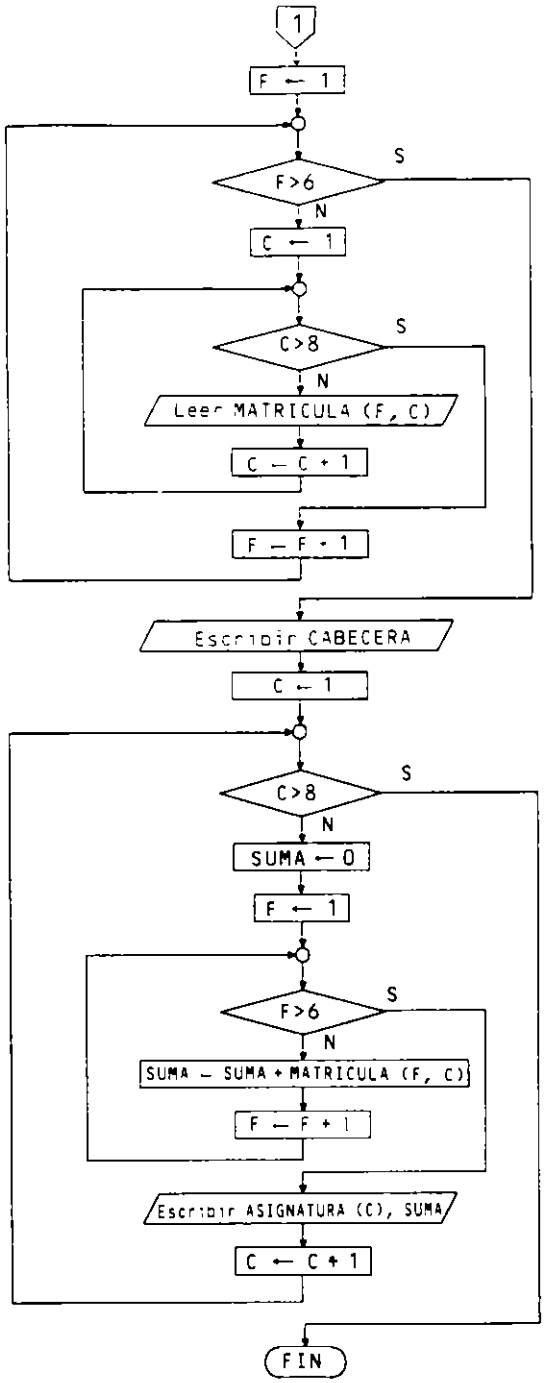
Asignatura n.º 1. Matemáticas.
Asignatura n.º 2. Lengua Española.
Asignatura n.º 3. Formación Humanística.
Asignatura n.º 4. Ciencias Naturales.
Asignatura n.º 5. Inglés.
Asignatura n.º 6. Informática Básica.
Asignatura n.º 7. Estructura de la Información.
Asignatura n.º 8. Metodología de la Programación.

Suponemos que los seis grupos son del mismo nivel y tienen las mismas asignaturas.

- **Objetos:**

I, F, C son contadores asociados a los bucles e índices.
SUMA es un acumulador del número de alumnos por asignatura.
ASIGNATURA es un vector de 8 elementos para almacenar los nombres de las asignaturas.
MATRICULA es la matriz del ejemplo anterior.





- **Pseudocódigo:**

```
Programa MATRICULACION
Entorno:
  MATRICULA es tabla(6,8) numérica entera
  SUMA, I, F, C son numéricas enteras
  ASIGNATURA es tabla(8) alfanumérica
Algoritmo:
  ASIGNATURA(1) ← "Matemáticas"
  ASIGNATURA(2) ← "Lengua Esp."
```

```
  ASIGNATURA(3) ← "Form. Humana"
  ASIGNATURA(4) ← "Ciencias N."
  ASIGNATURA(5) ← "Inglés"
  ASIGNATURA(6) ← "Inf. Basica"
  ASIGNATURA(7) ← "Estr. Infor."
  ASIGNATURA(8) ← "Metod. Prog."
  para F de 1 a 6 hacer
    para C de 1 a 8 hacer
      escribir "Grupo", F, "Asignatura", ASIGNATURA(C)
      leer MATRICULA(F,C)
    finpara
  finpara
  escribir "Alumnos matriculados"
  escribir "ASIGNATURA", "NUM. ALUMNOS"
  para C de 1 a 8 hacer
    SUMA ← 0
    para F de 1 a 6 hacer
      SUMA ← SUMA + MATRICULA(F,C)
    finpara
    escribir ASIGNATURA(C), SUMA
  finpara
Finprograma
```

- **Codificación COBOL:**

```
IDENTIFICATION DIVISION.
PROGRAM-ID. MATRICULACION.
DATA DIVISION.
WORKING-STORAGE SECTION.
01   TABLA-MATRICULA.
     02 GRUPO OCCURS 6 TIMES.
       03 MATRICULA PIC 99 OCCURS 8 TIMES.
01   TABLA-ASIGNATURAS.
     02 FILLER PIC X(11) VALUE "Matemáticas".
     02 FILLER PIC X(11) VALUE "Lengua Esp.".
     02 FILLER PIC X(11) VALUE "Form.Human.".
     02 FILLER PIC X(11) VALUE "Ciencias N.".
     02 FILLER PIC X(11) VALUE "Inglés      ".
     02 FILLER PIC X(11) VALUE "Inf. Básica".
     02 FILLER PIC X(11) VALUE "Estr.Infor.".
     02 FILLER PIC X(11) VALUE "Metod.Prog.".
01   TABLA-ASIG REDEFINES TABLA-ASIGNATURAS.
     02 ASIGNATURA PIC X(11) OCCURS 8 TIMES.
01   VARIABLES.
     05 SUMA PIC 999.
     05 I PIC 9.
     05 F PIC 9.
     05 C PIC 9.
PROCEDURE DIVISION.
PROCESO.
     PERFORM VARYING F FROM 1 BY 1 UNTIL F > 6
       AFTER VARYING C FROM 1 BY 1 UNTIL C > 8
         DISPLAY "Grupo: ", F, " Asignatura: ", ASIGNATURA(C)
         ACCEPT MATRICULA(F, C)
     END-PERFORM
     DISPLAY "Alumnos matriculados"
     DISPLAY "ASIGNATURA ", "NUM. ALUMNOS"
     DISPLAY "========== ", "============"
```

```
PERFORM VARYING C FROM 1 BY 1 UNTIL C > 8
  MOVE 0 TO SUMA
  PERFORM VARYING F FROM 1 BY 1 UNTIL F > 6
    ADD MATRICULA(F, C) TO SUMA
  END-PERFORM
  DISPLAY ASIGNATURA(C), SUMA
END-PERFORM
STOP RUN.
```

- **Codificación Pascal:**

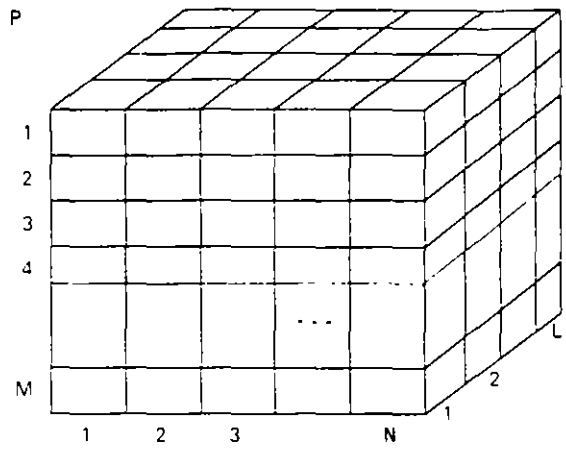
```
PROGRAM MATRICULACION (INPUT,OUTPUT);
TYPE STRING = PACKED ARRAY[1..11] OF CHAR;
VAR MATRICULA : ARRAY[1..6,1..8] OF INTEGER;
    ASIGNATURA : ARRAY[1..8] OF STRING;
    SUMA, I, F, C : INTEGER;
BEGIN (*MATRICULACION*)
  ASIGNATURA[1] := 'Matemáticas';
  ASIGNATURA[2] := 'Lengua Esp.';
  ASIGNATURA[3] := 'Form.Human.';
  ASIGNATURA[4] := 'Ciencias N.';
  ASIGNATURA[5] := 'Inglés      ';
  ASIGNATURA[6] := 'Inf. Básica';
  ASIGNATURA[7] := 'Estr.Infor.';
  ASIGNATURA[8] := 'Metod.Prog.';
  FOR F := 1 TO 6 DO
    FOR C := 1 TO 8 DO
    BEGIN (*1*)
      WRITE ('Grupo: ',F, 'Asignatura: ',ASIGNATURA[C]);
      READLN(MATRICULA[F,C])
    END; (*1*)
  WRITELN('Alumnos matriculados');
  WRITELN('ASIGNATURA    NUM. ALUMNOS');
  WRITELN('==========    ============');
  FOR C := 1 TO 8 DO
    BEGIN (*2*)
      SUMA := 0;
      FOR F := 1 TO 6 DO
        SUMA := SUMA + MATRICULA[F,C];
      WRITELN(ASIGNATURA[C], '       ', SUMA)
    END (*2*)
END. (*MATRICULACION*)
```

5.3.3. TABLAS MULTIDIMENSIONALES (POLIEDROS)

Son tablas de tres o más dimensiones. También se les denomina **poliedros**.

Este tipo de tablas no son de uso frecuente, no obstante son una herramienta útil para un determinado número de problemas.

La mayoría de los lenguajes de programación admiten estas estructuras, aunque cada uno de ellos tiene una limitación con respecto al número máximo de dimensiones permitidas.



Poliedro P de
M filas,
N columnas
y L capas

M × N × L componentes

Direccionamiento de un elemento:

P (fila, columna, capa)

identificador — valor, variable o expresión numérica entera

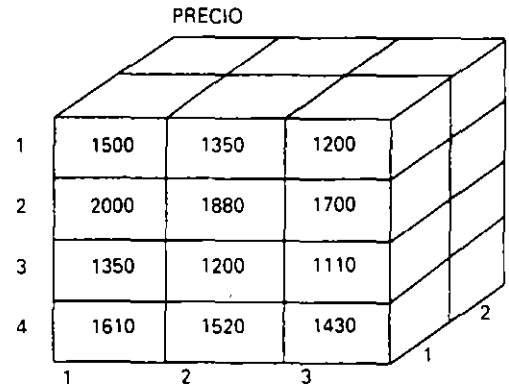
Los elementos de un poliedro se almacenan, en general, por capas, y en cada una de ellas por columnas:

P(1,1,1), P(2,1,1), ..., P(M,1,1), P(1,2,1), P(2,2,1), ..., P(M,2,1), P(1,3,1), P(2,3,1), ..., P(M,3,1), ..., P(1,N,1), P(2,N,1), ..., P(M,N,1), P(1,1,2), P(2,1,2), ..., P(M,1,2), P(1,2,2), P(2,2,2), ..., P(M,2,2), P(1,3,2), P(2,3,2), ..., P(M,3,2), ..., P(1,N,2), P(2,N,2), ..., P(M,N,2), ..., P(M,N,L)

Ejemplo: *Poliedro de tres dimensiones que consta de 24 componentes agrupados de la siguiente forma*

— Primera dimensión, que indica el número de un ARTICULO de 1 a 4.
— Segunda dimensión, que indica el grado de CALIDAD de 1 a 3.
— Tercera dimensión, que indica el tipo de VENTA de 1 a 2 (1 para venta al por mayor y 2 para venta al detalle).

Cada componente almacena el precio para los artículos según la calidad y el tipo de venta.



NOMBRE DE LA TABLA: PRECIO

COMPONENTES: PRECIO(1,1,1), PRECIO(2,1,1), . . PRECIO(1,2,1), PRECIO(2,2,1), ... PRECIO(3,3,2), PRECIO(4,3,2).

INDICES: Los números 1 al 4 para la primera dimensión, del 1 al 3 para la segunda y del 1 al 2 para la tercera

DIMENSION: Tres

LONGITUD: 4 * 3 * 2 = 24

TIPO: Numerica entera.

La componente PRECIO(4,3,1) almacena el precio de un ARTICULO del número 4, de CALIDAD 3, que se vende al por mayor. En el ejemplo representado su valor es 1430

Las declaraciones son en este caso:

COBOL:
```
01 TABLA-PRECIO.
   02 ARTICULO OCCURS 4 TIMES.
      03 CALIDAD OCCURS 3 TIMES.
         04 PRECIO PIC 9999 OCCURS 2 TIMES.
```

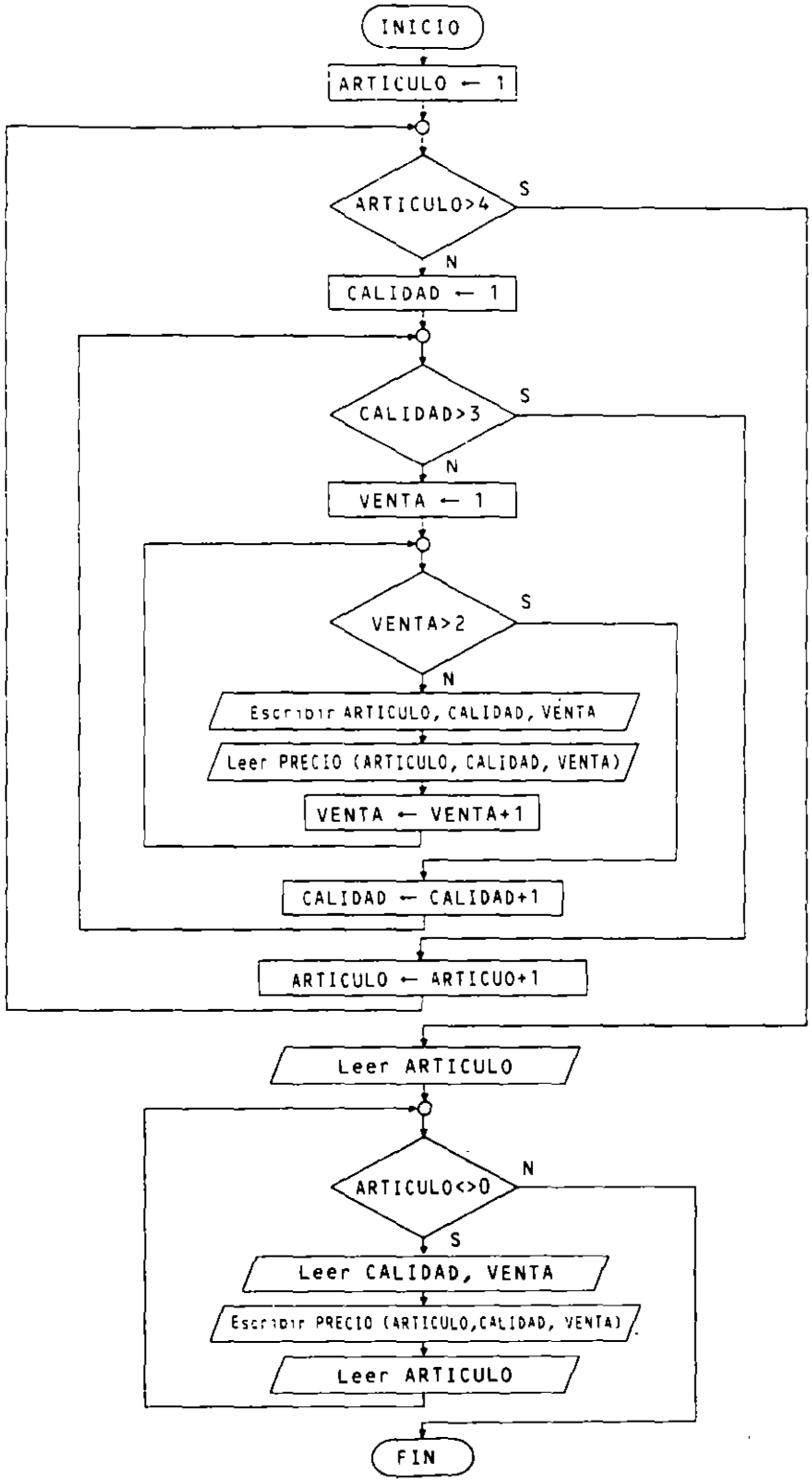
Pascal:
```
PRECIO : ARRAY[1..4,1..3,1..2] OF INTEGER;
```

Ejercicio: *Programa que carga la tabla del ejemplo anterior y posibilita sucesivas consultas de precios introduciendo como datos de entrada el número de artículo, su calidad y el tipo de venta. Para terminar la serie de consultas se introducirá un 0 en el número de artículo.*

El programa consta de dos bloques: el primero consiste en la carga de precios en la tabla y el segundo en el acceso sucesivo a las distintas componentes de la tabla solicitadas.

- **Objetos:**

ARTICULO	Variable para direccionar el artículo.
CALIDAD	Variable para direccionar la calidad.
VENTA	Variable para direccionar el tipo de venta.
PRECIO	Es el poliedro del ejemplo anterior.



- **Pseudocódigo:**

```
Programa CONSULTAS
Entorno:
  PRECIO es tabla(4,3,2) numérica entera
```

```
  ARTICULO, CALIDAD, VENTA son numéricas enteras
Algoritmo:
  para ARTICULO de 1 a 4 hacer
    para CALIDAD de 1 a 3 hacer
      para VENTA de 1 a 2 hacer
        escribir ARTICULO,CALIDAD,VENTA
        leer PRECIO(ARTICULO,CALIDAD,VENTA)
      finpara
    finpara
  finpara
  escribir "ARTICULO 1-4 (para terminar 0)"
  leer ARTICULO
  mientras ARTICULO <> 0 hacer
    escribir "CALIDAD 1-3"
    leer CALIDAD
    escribir "VENTA 1-2"
    leer VENTA
    escribir "PRECIO = ", PRECIO(ARTICULO,CALIDAD,VENTA), " Pts."
    escribir "ARTICULO 1-4 (para terminar 0)"
    leer ARTICULO
  finmientras
finprograma
```

- **Codificación COBOL:**

```
IDENTIFICATION DIVISION.
PROGRAM-ID. CONSULTAS.
DATA DIVISION.
WORKING-STORAGE SECTION.
01   TABLA-PRECIO.
     02 ARTICULO OCCURS 4 TIMES.
        03 CALIDAD OCCURS 3 TIMES.
           04 PRECIO PIC 9999 OCCURS 2 TIMES.
01   VARIABLES.
     05 ARTICULO PIC 9.
     05 CALIDAD  PIC 9.
     05 VENTA    PIC 9.
PROCEDURE DIVISION.
PROCESO.
     PERFORM VARYING ARTICULO FROM 1 BY 1 UNTIL ARTICULO > 4
       AFTER VARYING CALIDAD FROM 1 BY 1 UNTIL CALIDAD > 3
         AFTER VARYING VENTA FROM 1 BY 1 UNTIL VENTA > 2
           DISPLAY "Articulo ", "Calidad ", "Venta"
           DISPLAY ARTICULO, CALIDAD, VENTA
           ACCEPT PRECIO(ARTICULO, CALIDAD, VENTA)
     END-PERFORM
     DISPLAY "ARTICULO 1-4 (para terminar 0)"
     ACCEPT ARTICULO
     PERFORM UNTIL ARTICULO = 0
       DISPLAY "CALIDAD 1-3 "
       ACCEPT CALIDAD
       DISPLAY "VENTA 1-2 "
       ACCEPT VENTA
       DISPLAY "PRECIO = ", PRECIO(ARTICULO, CALIDAD, VENTA), " Pts."
       DISPLAY "ARTICULO 1-4 (para terminar 0)"
       ACCEPT ARTICULO
     END-PERFORM
     STOP RUN.
```

- **Codificación Pascal:**

```
PROGRAM CONSULTAS (INPUT,OUTPUT);
VAR PRECIO : ARRAY[1..4,1..3,1..2] OF INTEGER;
    ARTICULO, CALIDAD, VENTA : INTEGER;
BEGIN (*CONSULTAS*)
  FOR ARTICULO := 1 TO 4 DO
    FOR CALIDAD := 1 TO 3 DO
      FOR VENTA := 1 TO 2 DO
        BEGIN (*1*)
          WRITELN(ARTICULO, CALIDAD, VENTA);
          READLN(PRECIO[ARTICULO, CALIDAD, VENTA])
        END; (*1*)
  WRITE('ARTICULO 1-4 (para terminar 0)');
  READLN(ARTICULO);
  WHILE ARTICULO <> 0 DO
    BEGIN (*2*)
      WRITE('CALIDAD 1-3 ');
      READLN(CALIDAD);
      WRITE('VENTA 1-2 ');
      READLN(VENTA);
      WRITELN('PRECIO = ', PRECIO[ARTICULO,CALIDAD,VENTA], ' Pts.');
      WRITE('ARTICULO 1-4 (para terminar 0)');
      READLN(ARTICULO)
    END (*2*)
END. (*CONSULTAS*)
```

5.4. REPRESENTACION GRAFICA DE LAS TABLAS

La computadora almacena una tabla de cualquier dimensión en la memoria central de una forma lineal. Es conveniente, al utilizarlas, hacer una representación gráfica de las mismas para facilitar su comprensión, lo cual agiliza el diseño del programa que las maneja.

Recomendamos una forma general de representación que permite visualizar los contenidos de todas las componentes para tablas de cualquier dimensión.

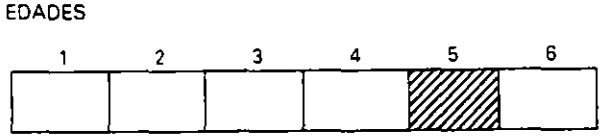
La representación consiste en descomponer aquellas tablas que por su dimensión no permiten una fácil presentación o no dejan ver todas sus componentes.

Veremos la representación gráfica a partir de ejemplos concretos para dimensiones de 1 a 4. Esto se puede generalizar sin dificultad para tablas de cualquier otra dimensión.

5.4.1. TABLA DE UNA DIMENSION (VECTOR)

Se representa de forma lineal, indicando el valor del índice asociado a cada componente en la parte superior y el nombre de la tabla.

Sea un vector EDADES de seis componentes.



La componente EDADES(5) es la que figura sombreada.

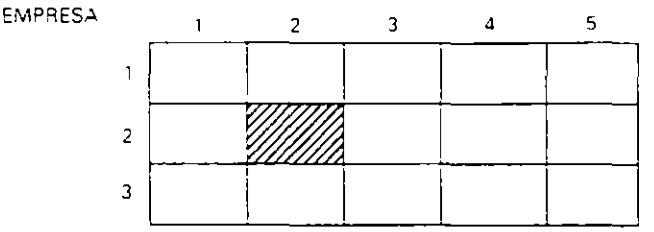
5.4.2. TABLA DE DOS DIMENSIONES (MATRIZ)

Se representa de forma tabular, de igual forma a la utilizada en Matemáticas, situando las filas horizontalmente y las columnas en vertical

Matemáticamente, una matriz A de M filas y N columnas se representa:

$$A = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1N} \\ a_{21} & a_{22} & \cdots & a_{2N} \\ \cdots & \cdots & \cdots & \cdots \\ a_{M1} & a_{M2} & \cdots & a_{MN} \end{pmatrix}$$

Sea una matriz EMPRESA con tres filas que indican los tres departamentos existentes y con cinco columnas, una para cada categoría laboral. Cada componente almacena el número de empleados para un departamento y categoría determinados.

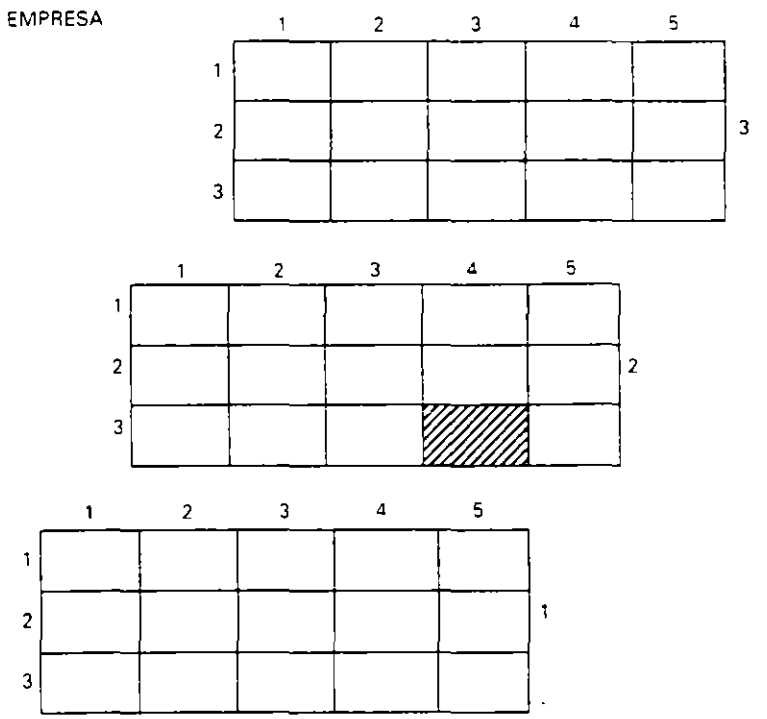


La componente EMPRESA(2,2) es la que figura sombreada.

5.4.3. TABLA DE TRES DIMENSIONES (POLIEDRO-3)

Se representa como un conjunto de tantas matrices como indique la tercera dimensión. Cada matriz tiene tantas filas y columnas como indiquen la primera y segunda dimensiones, respectivamente.

Sea un poliedro-3 de dimensiones 3, 5, 3 de nombre EMPRESA y con el mismo significado del ejemplo anterior considerando que la tercera dimensión corresponde a las tres sucursales de la empresa.



La tercera dimensión correspondiente al número de la sucursal se sitúa a la derecha de cada una de las matrices.

La componente EMPRESA(3,4,2) es la que figura sombreada.

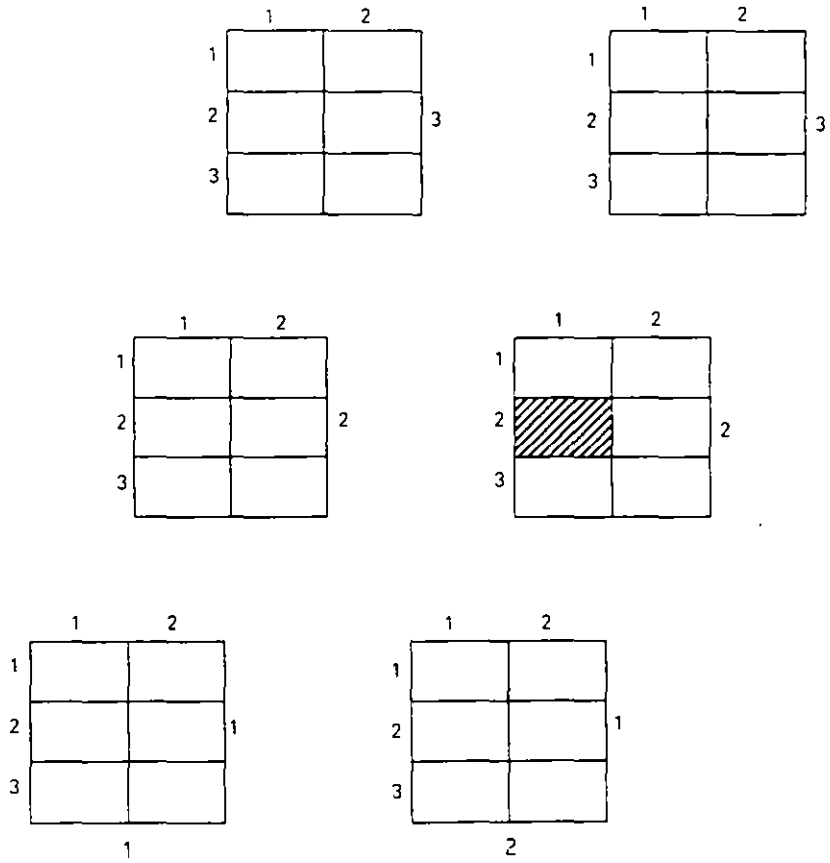
5.4.4. TABLA DE CUATRO DIMENSIONES (POLIEDRO-4)

Se representa como un conjunto de tantos poliedros-3 como indique la cuarta dimensión.

Sea un poliedro-4 de dimensiones 3, 2, 3, 2 de nombre EMPRESA y con el mismo significado del ejemplo anterior, considerando que la cuarta dimensión diferencia a los hombres de las mujeres. Es decir, cada dimensión indica:

— Primera dimensión = 3 departamentos.
— Segunda dimensión = 2 categorías laborales.
— Tercera dimensión = 3 sucursales.
— Cuarta dimensión = hombres y mujeres.

EMPRESA



La cuarta dimensión correspondiente a la distinción entre hombres y mujeres se indica en la parte inferior de ambos poliedros-3.

La componente EMPRESA(2,1,2,2) es la figura sombreada.

5.5. TRATAMIENTO SECUENCIAL DE UNA TABLA

Se define un tratamiento secuencial de una tabla como la acción de realizar una determinada operación o conjunto de operaciones sobre todos sus elementos.

Este tratamiento es diferente para cada tipo de tablas con respecto a su dimensión

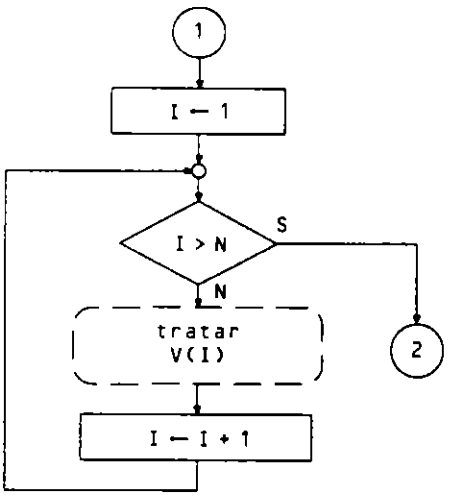
5.5.1. TRATAMIENTO SECUENCIAL DE UN VECTOR

Se realiza mediante un bucle PARA (FOR) en el que mediante el índice del mismo se accede sucesivamente a las componentes del vector.

Las componentes del vector se recorren de izquierda a derecha (de principio a fin), por lo cual el índice del bucle varía entre 1 y el número de elementos del vector.

Sea el vector V de N componentes:

- **Ordinograma:**



- **Pseudocódigo:**

```
...
para I de 1 a N hacer
   tratar V(I)
finpara
...
```

- **COBOL:**

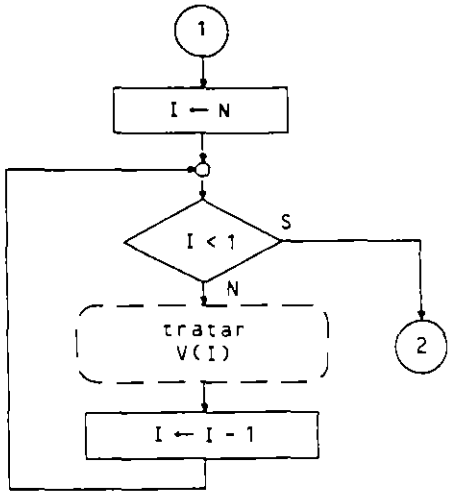
```
...
PERFORM VARYING I FROM 1 BY 1 UNTIL I > N
   tratar V(I)
END-PERFORM
...
```

- **Pascal:**

```
...
FOR I:= 1 TO N DO
     tratar V[I]
...
```

En algunos casos se hace el recorrido de derecha a izquierda.

- **Ordinograma:**



- **Pseudocódigo:**

```
...
para I de N a 1 con incremento -1 hacer
  tratar V(I)
finpara
...
```

- **COBOL:**

```
...
PERFORM VARYING I FROM N BY -1 UNTIL I < 1
   tratar V(I)
END-PERFORM
...
```

- **Pascal:**

```
...
FOR I:= N DOWNTO 1 DO
  tratar V[I]
...
```

5.5.2. TRATAMIENTO SECUENCIAL DE UNA MATRIZ

Se realiza mediante el anidamiento de dos bucles PARA. El bucle externo recorre cada una de las filas y el interno todas las componentes de una fila.

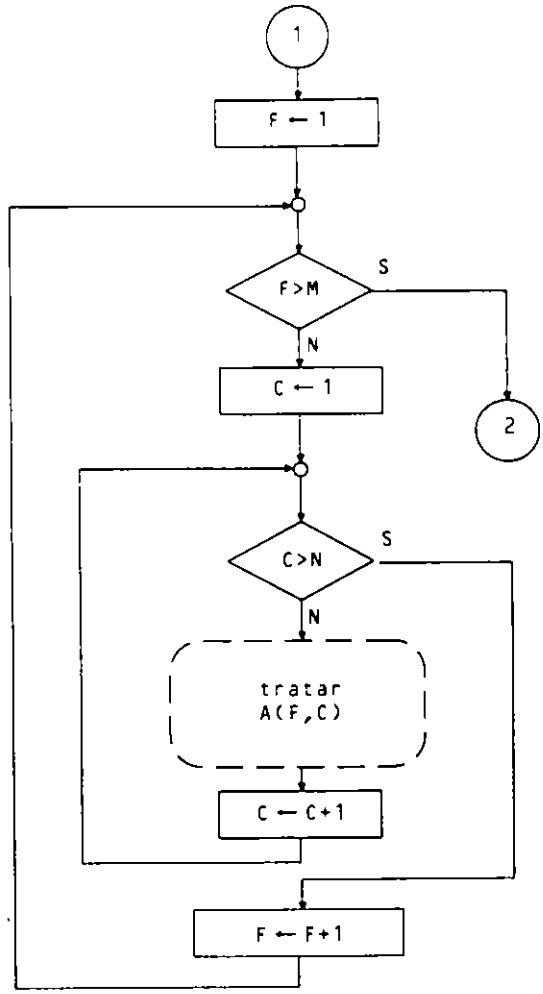
Este tratamiento también se puede realizar por columnas en los casos que interese, y además en ambos casos en orden creciente o decreciente para las filas y columnas.

Sea una matriz A de M filas y N columnas. Su tratamiento secuencial por filas en orden creciente de filas y columnas es:

- **Pseudocódigo:**

```
...
para F de 1 a M hacer
  para C de 1 a N hacer
    tratar A(F,C)
  finpara
finpara
...
```

- **Ordinograma:**



- **COBOL:**

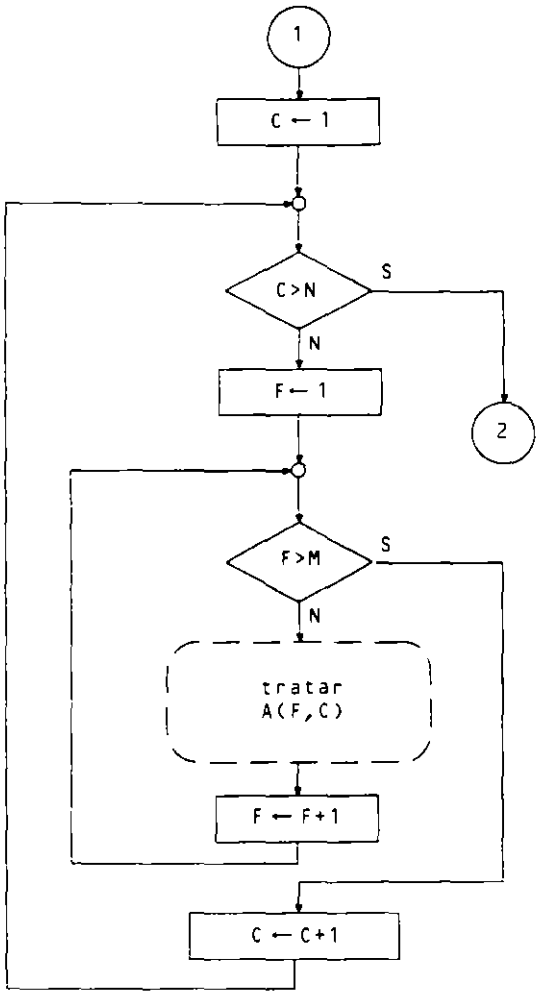
```
...
PERFORM VARYING F FROM 1 BY 1 UNTIL F > M
   AFTER VARYING C FROM 1 BY 1 UNTIL C > N
      tratar A(F, C)
END-PERFORM
...
```

- **Pascal:**

```
...
FOR F:=1 TO M DO
  FOR C:=1 TO N DO
     tratar A[F,C]
...
```

El tratamiento secuencial por columnas en orden creciente es:

- **Ordinograma:**



- **Pseudocódigo:**

```
...
para C de 1 a N hacer
  para F de 1 a M hacer
    tratar A(F,C)
  finpara
finpara
...
```

- **COBOL:**

```
...
PERFORM VARYING C FROM 1 BY 1 UNTIL C > N
   AFTER VARYING F FROM 1 BY 1 UNTIL F > M
      tratar A(F, C)
END-PERFORM
...
```

- **Pascal:**

```
...
FOR C:=1 TO N DO
  FOR F:=1 TO M DO
    tratar A[F,C]
...
```

5.5.3. TRATAMIENTO SECUENCIAL DE UN POLIEDRO

Requiere tantos bucles PARA anidados como dimensiones tenga el poliedro.

Cada índice de un bucle recorre una dimensión y puede hacerse en cualquier orden de anidamiento.

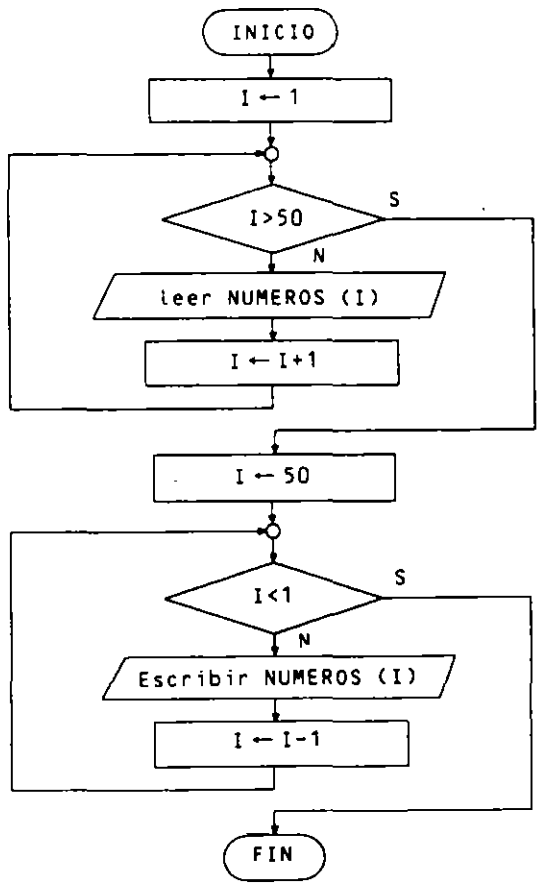
EJERCICIOS RESUELTOS

1. *Programa que lee una secuencia de 50 números almacenándolos en un vector NUMEROS y los imprime en orden inverso al de entrada.*

   • **Objetos:**

   NUMEROS vector de 50 elementos numéricos reales.
   I contador asociado a los bucles e índice del vector.

   • **Ordinograma:**



   • **Pseudocódigo:**

```
Programa INVERSION
Entorno:
  NUMEROS es tabla(50) numérica real
  I es numérica entera
Algoritmo:
  para I de 1 a 50 hacer
    escribir "Introducir componente número", I
    leer NUMEROS(I)
  finpara
  para I de 50 a 1 con incremento -1 hacer
    escribir NUMEROS(I)
  finpara
Finprograma
```

- **Codificación COBOL:**

```
IDENTIFICATION DIVISION.
PROGRAM-ID. INVERSION.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  TABLA-NUMEROS.
    02 NUMEROS PIC 999V99 OCCURS 50 TIMES.
01  VARIABLES.
    05 NUM PIC ZZ9.99.
    05 I   PIC 99.
PROCEDURE DIVISION.
PROCESO.
    PERFORM VARYING I FROM 1 BY 1 UNTIL I > 50
      DISPLAY "Introducir componente número ", I NO ADVANCING
      ACCEPT NUMEROS(I)
    END-PERFORM
    PERFORM VARYING I FROM 50 BY -1 UNTIL I < 1
      MOVE NUMEROS(I) TO NUM
      DISPLAY NUM, " " NO ADVANCING
    END-PERFORM
    STOP RUN.
```

- **Codificación Pascal:**

```
PROGRAM INVERSION (INPUT,OUTPUT);
VAR NUMEROS : ARRAY[1..50] OF REAL;
    I : INTEGER;
BEGIN (*INVERSION*)
  FOR I := 1 TO 50 DO
    BEGIN (*1*)
      WRITE('Introducir componente número ', I);
      READLN(NUMEROS[I])
    END; (*1*)
  FOR I := 50 DOWNTO 1 DO
    WRITE(NUMEROS[I], ' ')
END. (*INVERSION*)
```

2. *Frecuencias de calificaciones. Se introduce por teclado una secuencia de calificaciones (números enteros entre 0 y 10). La secuencia se termina cuando se introduce un número menor que 0 o mayor que 10.*

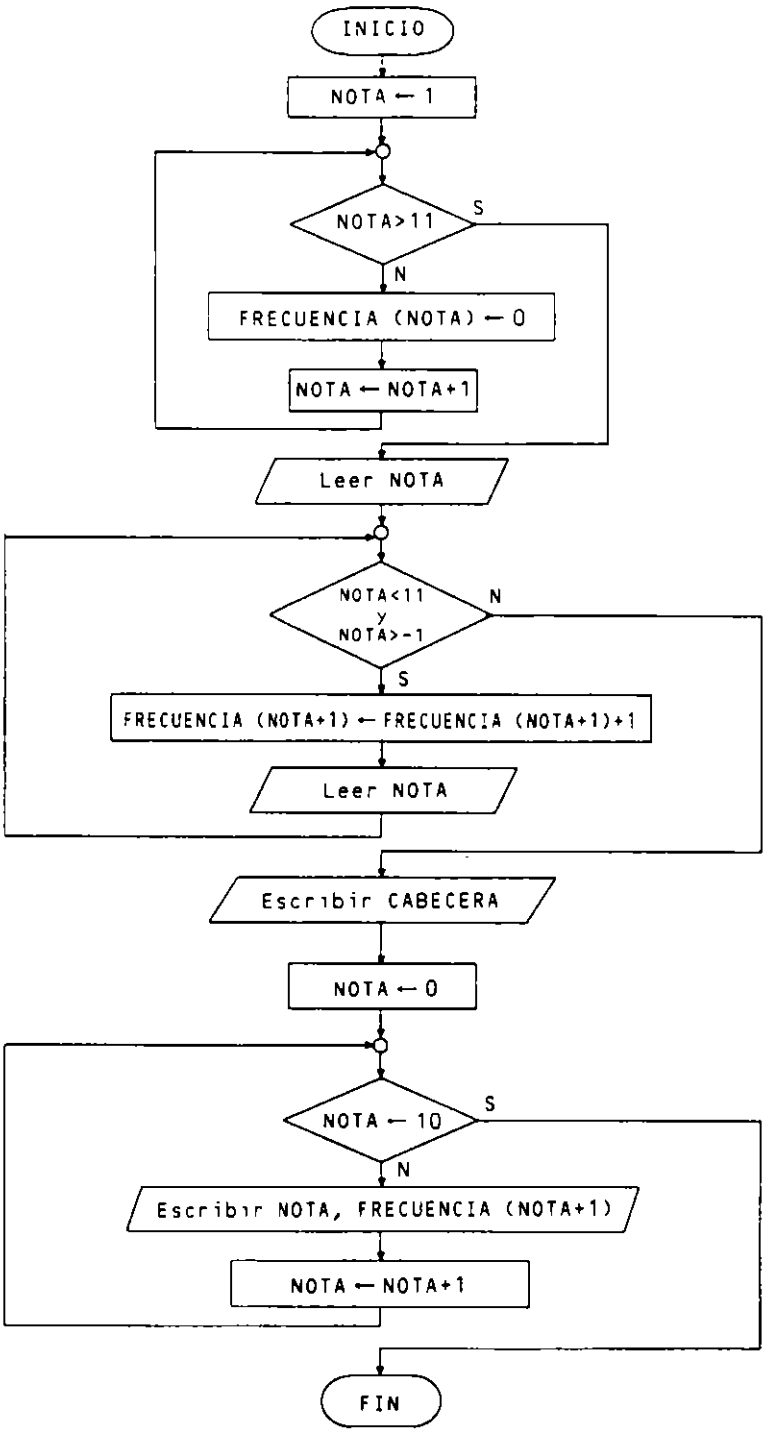
Programa que obtiene e imprime la lista de frecuencias (número de repeticiones) de cada una de las notas.

Como estructura para almacenar los datos de entrada se utilizará un vector FRECUENCIA de 11 componentes, cuyos índices 1, 2, ..., 11 son las posibles calificaciones + 1. Por tanto, el programa se reduce a leer cada calificación e incrementar en 1 la componente cuyo índice es la calificación leída + 1. Al final se imprimirá el vector FRECUENCIA resultante.

- **Objetos:**

NOTA variable para leer la secuencia de notas y contador asociado a los bucles.
FRECUENCIA vector de 11 elementos que actúan como contadores

- **Ordinograma:**



- **Pseudocódigo:**

```
Programa FRECUENCIAS
Entorno:
  FRECUENCIA es tabla(11) numérica entera
  NOTA es numérica entera
```

```
Algoritmo:
  para NOTA de 1 a 11 hacer
    FRECUENCIA(NOTA) ← 0
  finpara
  escribir " Introduzca nota "
  leer NOTA
  mientras NOTA<11 y NOTA>-1 hacer
    FRECUENCIA(NOTA+1) ← FRECUENCIA(NOTA+1) + 1
    escribir " Introduzca nota "
    leer NOTA
  finmientras
  escribir " Frecuencia de las calificaciones"
  escribir " Calificacion", " Frecuencia"
  escribir " ------------", " ----------"
  para NOTA de 0 a 10 hacer
    escribir NOTA, FRECUENCIA(NOTA+1)
  finpara
Finprograma
```

- **Codificación COBOL:**

```
IDENTIFICATION DIVISION.
PROGRAM-ID. FRECUENCIAS.
DATA DIVISION.
WORKING-STORAGE SECTION.
01   TABLA-FRECUENCIA.
     02 FRECUENCIA PIC 99 OCCURS 11 TIMES.
01   VARIABLES.
     05 NOTA PIC 99.
PROCEDURE DIVISION.
PROCESO.
     PERFORM VARYING NOTA FROM 1 BY 1 UNTIL NOTA > 11
       MOVE 0 TO FRECUENCIA(NOTA)
     END-PERFORM
     DISPLAY " Introduzca nota "
     ACCEPT NOTA
     PERFORM UNTIL NOTA > 10 OR < 0
       ADD 1 TO FRECUENCIA(NOTA + 1)
       DISPLAY " Introduzca nota "
       ACCEPT NOTA
     END-PERFORM
     DISPLAY " Frecuencia de las calificaciones"
     DISPLAY " Calificación", " Frecuencia"
     DISPLAY " ------------", " ----------"
     PERFORM VARYING NOTA FROM 0 BY 1 UNTIL NOTA > 10
       DISPLAY NOTA, FRECUENCIA(NOTA+1)
     END-PERFORM
     STOP RUN.
```

- **Codificación Pascal:**

```
PROGRAM FRECUENCIAS (INPUT,OUTPUT);
VAR FRECUENCIA : ARRAY[0..10] OF INTEGER;
    NOTA : INTEGER;
BEGIN (*FRECUENCIAS*)
  FOR NOTA := 0 TO 10 DO
      FRECUENCIA[NOTA] := 0;
  WRITE(' Introduzca nota: ');
  READLN(NOTA);
  WHILE (NOTA<11) AND (NOTA>-1) DO
```

```
    BEGIN (*1*)
      FRECUENCIA[NOTA] := FRECUENCIA[NOTA] + 1;
      WRITE(' Introduzca nota: ');
      READLN(NOTA)
    END; (*1*)
  WRITELN(' Frecuencia de las calificaciones');
  WRITELN(' Calificación', ' Frecuencia');
  WRITELN(' ------------', ' ----------');
  FOR NOTA := 0 TO 10 DO
    WRITELN(NOTA:10, FRECUENCIA[NOTA]:10)
END. (*FRECUENCIAS*)
```

En lenguaje Pascal pueden numerarse los elementos de un vector a partir de 0 utilizando la declaración

```
FRECUENCIA : ARRAY[0..10] OF INTEGER
```

en lugar de

```
FRECUENCIA : ARRAY[1..11] OF INTEGER
```

3. *Con la misma tabla del ejercicio anterior, programa que obtiene e imprime los siguientes datos estadísticos media aritmética, moda, mediana y desviación típica.*

Media aritmética.—Es el cociente entre la suma de valores por su frecuencia y la suma de frecuencias.

$$\text{MEDIA} = \frac{\Sigma\ \text{NOTA} * \text{FRECUENCIA(NOTA)}}{\Sigma\ \text{FRECUENCIA(NOTA)}}$$

Moda.—Valor de máxima frecuencia. (Puede haber más de una moda.)

Mediana.—Valor que tiene igual número de valores superiores que inferiores.

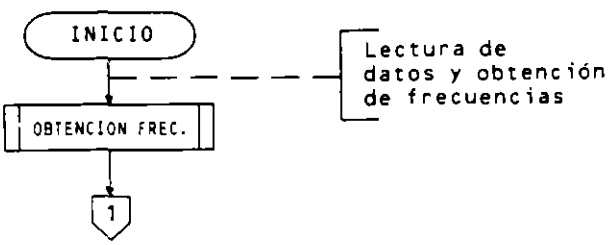
Desviación típica.—Es la variación del conjunto de valores respecto al valor medio.

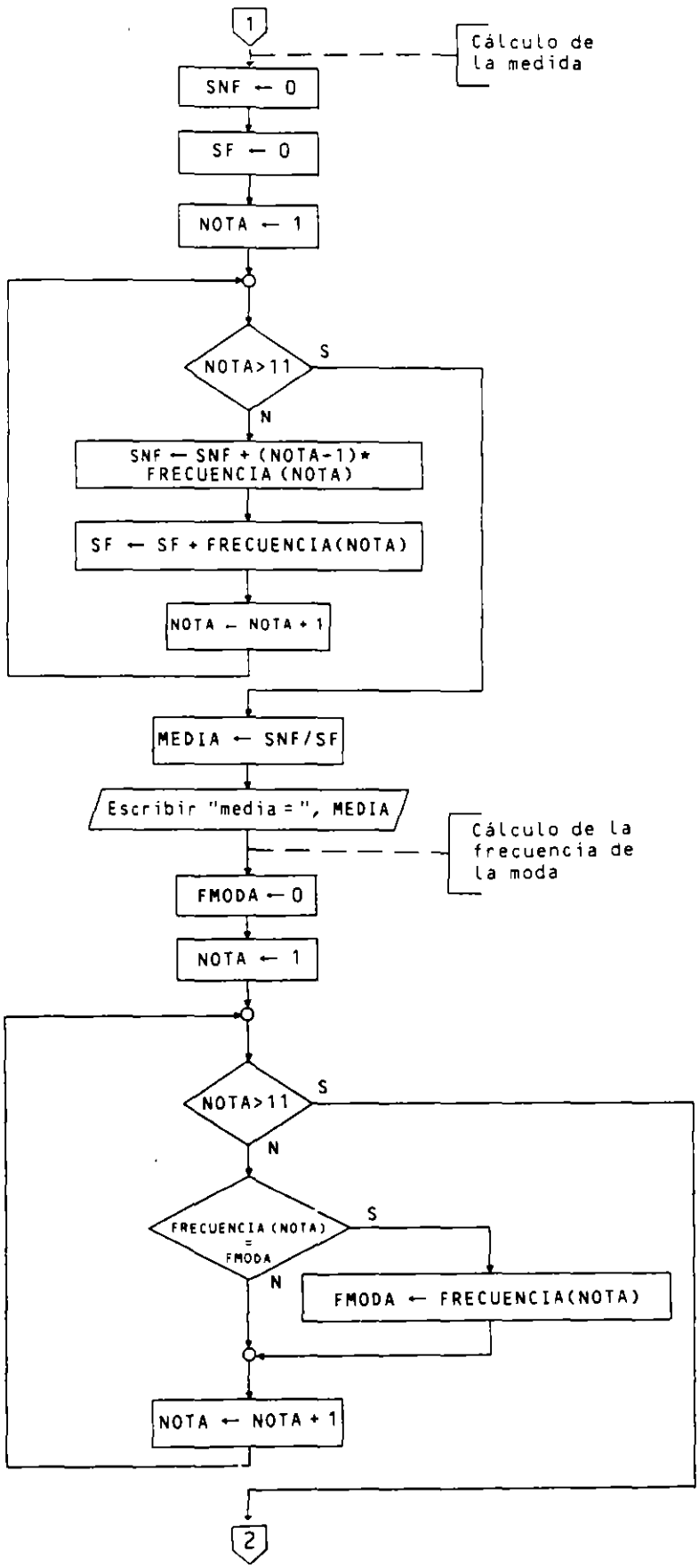
$$\text{DES} = \sqrt{\frac{\Sigma\ (\text{NOTA} - \text{MEDIA})^2 * \text{FRECUENCIA(NOTA)}^2}{\Sigma\ \text{FRECUENCIA(NOTA)}^2}}$$

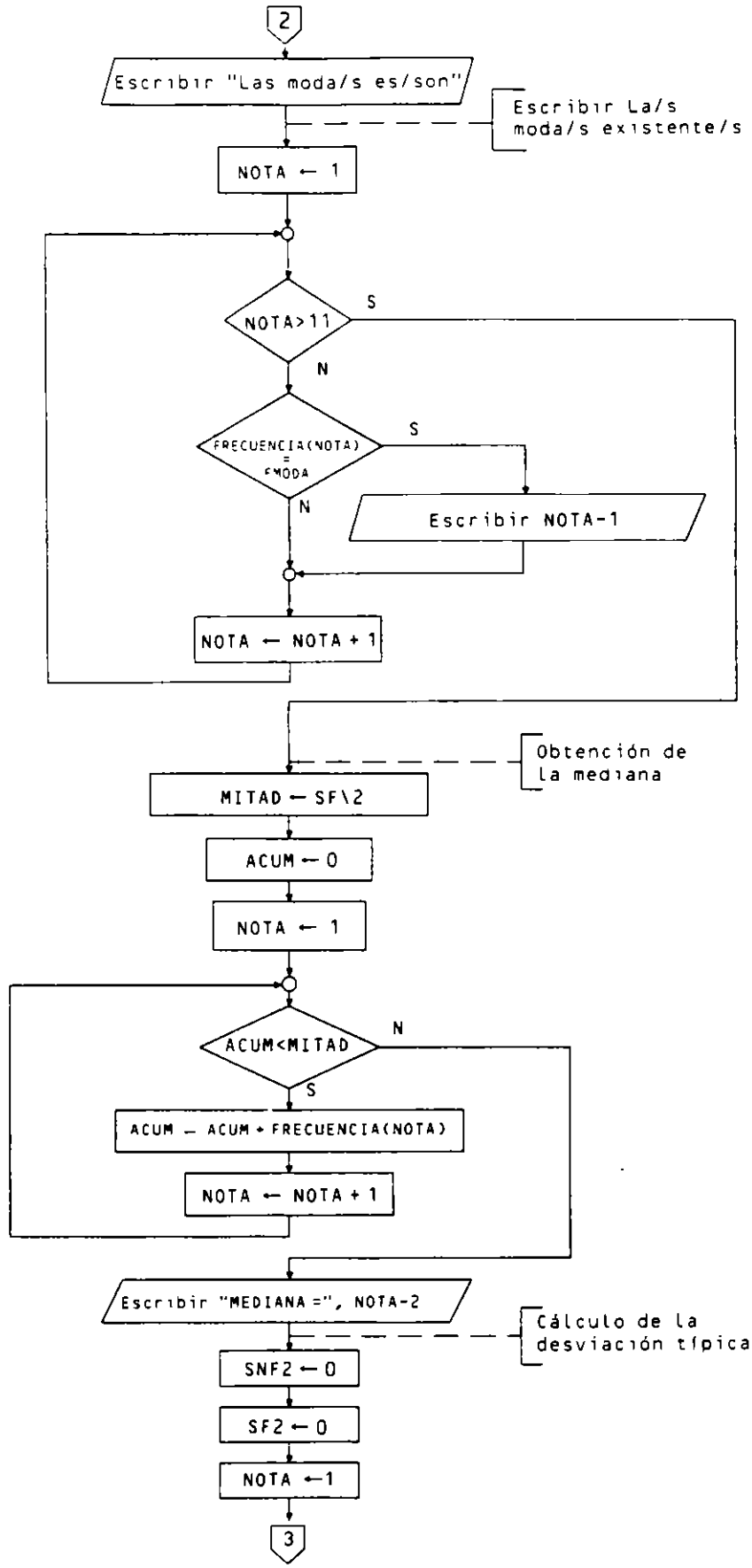
- **Objetos:**

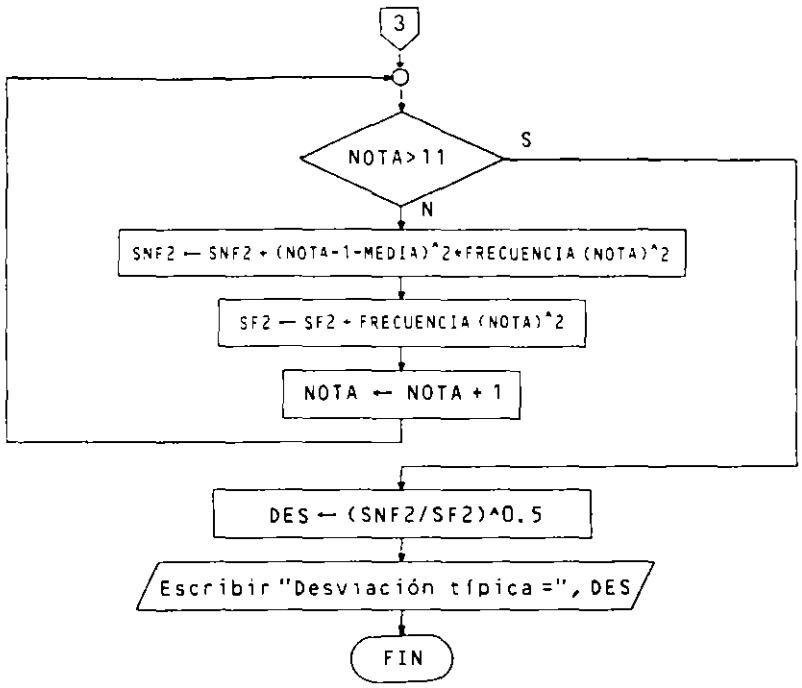
NOTA. Variable para leer la secuencia de notas y contador asociado a los bucles.
FRECUENCIA. Vector de 11 elementos que actúan como contadores.
SNF. Acumulador para sumar las notas.
SF. Contador de notas.
MEDIA. Variable para el cálculo de la media.
FMODA. Variable para encontrar la frecuencia de la moda.
MITAD. Variable para calcular la mitad de notas leidas.
ACUM. Acumulador para el cálculo de la mediana.
SNF2, SF2 y DES. Variables para el cálculo de la desviación típica.

- **Ordinograma:**




1
Cálculo de la medida
SNF ← 0
SF ← 0
NOTA ← 1
NOTA>11
S
N
SNF ← SNF + (NOTA-1)* FRECUENCIA (NOTA)
SF ← SF + FRECUENCIA(NOTA)
NOTA ← NOTA + 1
MEDIA ← SNF/SF
Escribir "media = ", MEDIA
Cálculo de la frecuencia de la moda
FMODA ← 0
NOTA ← 1
NOTA>11
S
N
FRECUENCIA (NOTA) = FMODA
S
N
FMODA ← FRECUENCIA(NOTA)
NOTA ← NOTA + 1
2


2
Escribir "Las moda/s es/son"
Escribir La/s moda/s existente/s
NOTA ← 1
NOTA>11
S
N
FRECUENCIA(NOTA) = FMODA
S
N
Escribir NOTA-1
NOTA ← NOTA + 1
Obtención de la mediana
MITAD ← SF\2
ACUM ← 0
NOTA ← 1
ACUM<MITAD
N
S
ACUM ← ACUM + FRECUENCIA(NOTA)
NOTA ← NOTA + 1
Escribir "MEDIANA =", NOTA-2
Cálculo de la desviación típica
SNF2 ← 0
SF2 ← 0
NOTA ←1
3



- **Pseudocódigo:**

```
Programa ESTADISTICA
Entorno:
  FRECUENCIA es tabla(11) numérica entera
  NOTA es numérica entera
  SNF, SF, FMODA, MITAD, ACUM son numéricas enteras
  MEDIA, SNF2, SF2, DES son numéricas reales
Algoritmo:
  ** No repetimos la parte de lectura de datos y
  ** obtención de frecuencias realizadas en el
  ** ejercicio anterior
  ** Cálculo de la media aritmética **********
  SNF ← 0
  SF ← 0
  para NOTA de 1 a 11 hacer
    SNF ← SNF + (NOTA-1) * FRECUENCIA(NOTA)
    SF ← SF + FRECUENCIA(NOTA)
  finpara
  MEDIA ← SNF / SF
  escribir "Media aritmética: ", MEDIA
  ** Obtención de la moda **********
  FMODA ← 0
  para NOTA de 1 a 11 hacer
    si FRECUENCIA(NOTA) > FMODA
       entonces FMODA ← FRECUENCIA(NOTA)
    finsi
  finpara
  escribir "La/s moda/s es/son:"
  para NOTA de 1 a 11 hacer
    si FRECUENCIA(NOTA) = FMODA
       entonces escribir NOTA-1
    finsi
  finpara
```

```
    ** Obtención de la mediana *****************
    MITAD ← SF\2
    ACUM ← 0
    NOTA ← 1
    mientras ACUM < MITAD hacer
      ACUM ← ACUM + FRECUENCIA(NOTA)
      NOTA ← NOTA + 1
    finmientras
    escribir "Mediana: ", NOTA-1
    ** Cálculo de la desviación típica *********
    SNF2 ← 0
    SF2 ← 0
    para NOTA de 1 a 11 hacer
      SNF2 ← SNF2 + (NOTA-1-MEDIA)^2 * FRECUENCIA(NOTA)^2
      SF2 ← SF2 + FRECUENCIA(NOTA)^2
    finpara
    DES ← (SNF2 / SF2)^0.5
    escribir "Desviación típica: ", DES
Finprograma
```

- **Codificación COBOL:**

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID. ESTADISTICA.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 01   TABLA-FRECUENCIA.
      02 FRECUENCIA PIC 99 OCCURS 11 TIMES.
* La componente I de la tabla almacenará las
* repeticiones de la nota I-1
 01   VARIABLES.
      05  NOTA  PIC 99.
      05  SNF   PIC 9999.
      05  SF    PIC 9999.
      05  FMODA PIC 999.
      05  MITAD PIC 9999.
      05  ACUM  PIC 9999.
      05  MEDIA PIC 99V99.
      05  SFN2  PIC 9999V99.
      05  SF2   PIC 9999V99.
      05  DES   PIC 9999V99.
      05  NOTA1 PIC 99.
 PROCEDURE DIVISION.
 PROCESO.
      DISPLAY " " ERASE
      ...
* Obtención de notas y sus frecuencias
      ...
      MOVE 0 TO SNF
      MOVE 0 TO SF
      PERFORM VARYING NOTA FROM 1 BY 1 UNTIL NOTA > 11
        COMPUTE SNF = SNF + (NOTA - 1) * FRECUENCIA(NOTA)
        COMPUTE SF = SF + FRECUENCIA(NOTA)
      END-PERFORM
      COMPUTE MEDIA = SNF / SF
      DISPLAY " MEDIA ARITMETICA: ", MEDIA
* Obtención de la moda
      MOVE 0 TO FMODA
      PERFORM VARYING NOTA FROM 1 BY 1 UNTIL NOTA > 11
```

```
           IF FRECUENCIA(NOTA) > FMODA
              MOVE FRECUENCIA(NOTA) TO FMODA
           END-IF
         END-PERFORM
         DISPLAY "La/s moda/s es/son: " NO ADVANCING
         PERFORM VARYING NOTA FROM 1 BY 1 UNTIL NOTA > 11
           IF FRECUENCIA(NOTA) = FMODA
              COMPUTE NOTA1 = NOTA - 1
              DISPLAY NOTA1 NO ADVANCING
           END-IF
         END-PERFORM
         DISPLAY " "
* Obtención de la mediana
         COMPUTE MITAD = SF / 2
         MOVE 0 TO ACUM
         MOVE 1 TO NOTA
         PERFORM UNTIL ACUM >= MITAD
            COMPUTE ACUM = ACUM + FRECUENCIA(NOTA)
            ADD 1 TO NOTA
         END-PERFORM
         COMPUTE NOTA1 = NOTA - 1
         DISPLAY " MEDIANA: ", NOTA1
* Cálculo de la desviación típica
         MOVE 0 TO SNF2
         MOVE 0 TO SF2
         PERFORM VARYING NOTA FROM 1 BY 1 UNTIL NOTA > 11
            COMPUTE SNF2 = SNF2 + (NOTA - 1 - MEDIA) ** 2 *
                     FRECUENCIA(NOTA) ** 2
            COMPUTE SF2 = SF2 + FRECUENCIA(NOTA) ** 2
         END-PERFORM
         COMPUTE DES = (SNF2 / SF2) ** 0.5
         DISPLAY "Desviación típica: ", DES
         STOP RUN.
```

- **Codificación Pascal:**

```
PROGRAM ESTADISTICA (INPUT,OUTPUT);
VAR FRECUENCIA : ARRAY[0..10] OF INTEGER;
    NOTA, SNF, SF, FMODA, MITAD, ACUM : INTEGER;
    MEDIA, SNF2, SF2, DES : REAL;
BEGIN (*ESTADISTICA*)
  ...
  (* Obtención de notas y sus frecuencias *)
  ...
  (* Cálculo de la media aritmética *)
  SNF : = 0;
  SF : = 0;
  FOR NOTA := 0 TO 10 DO
    BEGIN (*1*)
      SNF := SNF + NOTA * FRECUENCIA[NOTA];
      SF := SF + FRECUENCIA[NOTA]
    END; (*1*)
  MEDIA := SNF / SF;
  WRITELN('Media aritmética: ', MEDIA);
  (* Obtención de la moda *)
  FMODA := 0;
  FOR NOTA := 0 TO 10 DO
    IF FRECUENCIA[NOTA] > FMODA THEN
        FMODA := FRECUENCIA[NOTA];
```

```
    WRITE('La/s moda/s es/son ');
    FOR NOTA := 0 TO 10 DO
      IF FRECUENCIA[NOTA] = FMODA THEN
        WRITE (NOTA:4);
    WRITELN;
    (* Obtención de la mediana *)
    MITAD := SF DIV 2;
    ACUM := 0;
    NOTA := 0;
    WHILE ACUM < MITAD DO
      BEGIN (*2*)
        ACUM := ACUM + FRECUENCIA[NOTA];
        NOTA := NOTA + 1
      END; (*2*)
    WRITELN('Mediana: ', NOTA);
    (* Cálculo de la desviacion tipica *)
    SNF2 := 0;
    SF2 := 0;
    FOR NOTA := 0 TO 10 DO
      BEGIN (*3*)
        SNF2 := SNF2 + SQR(NOTA - MEDIA) * SQR(FRECUENCIA[NOTA]);
        SF2 := SF2 + SQR(FRECUENCIA[NOTA])
      END; (*3*)
    DES := SQRT(SNF2 / SF2);
    WRITELN('Desviacion tipica: ', DES)
  END. (* ESTADISTICA *)
```

4. *Programa que genera e imprime un cuadrado mágico de dimensión* N *(siendo* N *un número entero, positivo e impar).*

Un cuadrado mágico de dimensión N es una matriz cuadrada de orden N, conteniendo los números naturales de 1 a N^2, tal que coinciden la suma de los números de una cualquiera de las filas, columnas o diagonales principales.

El cuadrado se construye mediante las siguientes reglas:

— El número 1 se coloca en la casilla central de la primera fila.
— Cada número siguiente se coloca en la casilla correspondiente a la fila anterior y columna posterior.
— Si el número sigue a un múltiplo de N, no se aplica la regla anterior, sino que se coloca en la casilla de la fila posterior e igual columna.
— Se considera que la fila anterior a la primera es la última, y la columna posterior a la última es la primera.

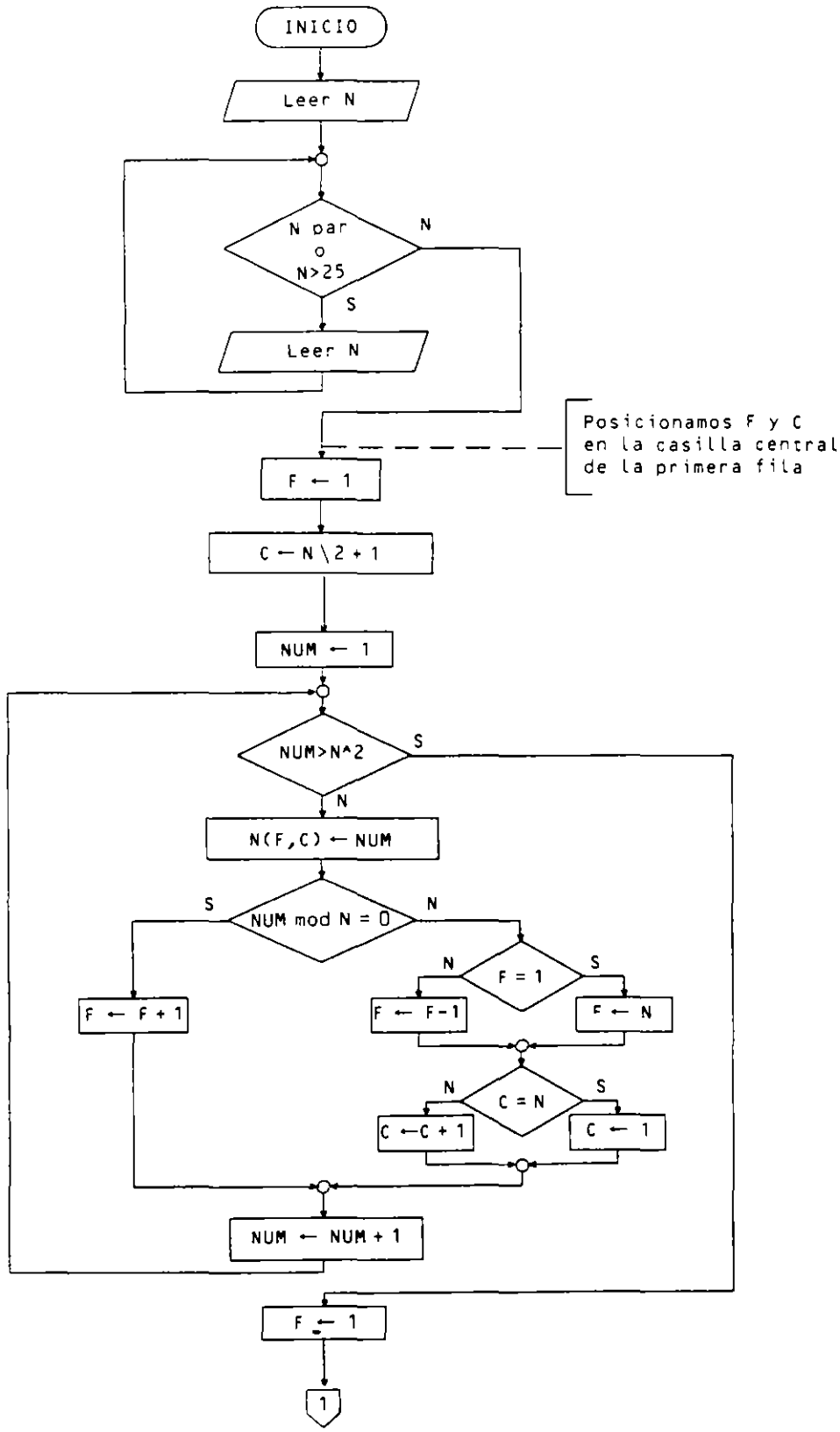
Por ejemplo, el cuadrado mágico de dimensión 5 es:

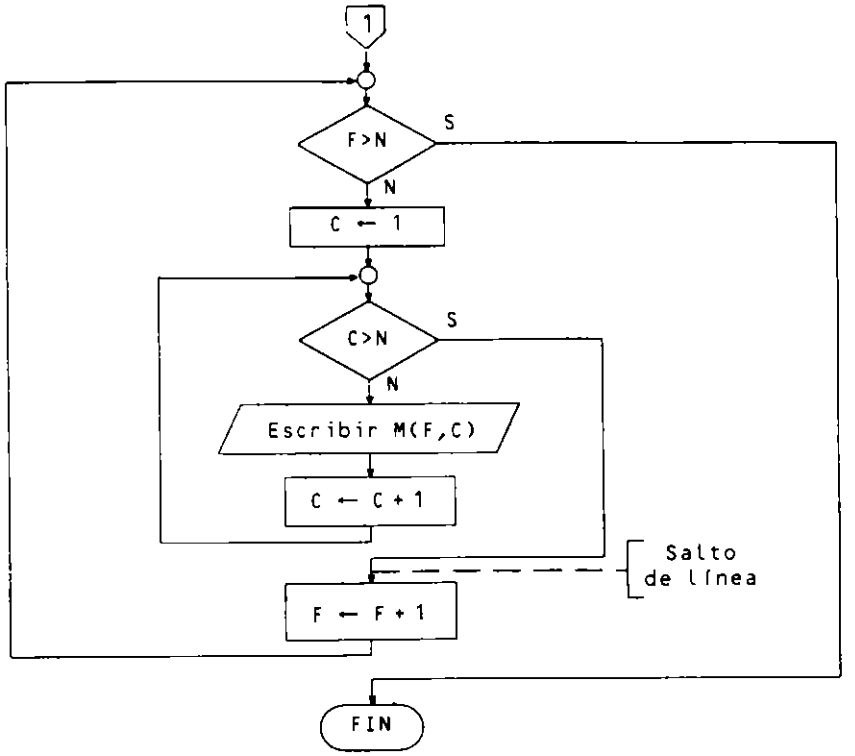
17	24	1	8	15
23	5	7	14	16
4	6	13	20	22
10	12	19	21	3
11	18	25	2	9

- **Objetos:**

Suponemos una matriz M cuadrada de 25 filas y 25 columnas.
N es la dimensión del cuadrado (impar y ≤ 25).
F y C son variables para direccionar filas y columnas.
NUM es un contador de 1 a N^2.

- **Ordinograma:**





- **Pseudocódigo:**

```
Programa CUADRADO MAGICO
Entorno: ** La dimensión máxima admitida será 25
  M es tabla(25,25) numérica entera
  N, NUM, F, C son numéricas enteras
Algoritmo:
  escribir "Dimensión "
  leer N
  mientras N mod 2 = 0 o N > 25 o N < 1 hacer
    escribir "Dimensión "
    leer N
  finmientras
  ** Posicionamos F y C en la casilla
  ** central de la primera fila.
  F ← 1
  C ← N\2 + 1
  para NUM de 1 a N^2 hacer
    M(F,C) ← NUM
    si NUM mod N = 0
      entonces F ← F + 1
      sino si F = 1 entonces F ← N
                    sino F ← F - 1
           finsi
           si C = N entonces C ← 1
                    sino C ← C + 1
           finsi
    finsi
  finpara
  para F de 1 a N hacer
```

```
    para C de 1 a N hacer
      escribir M(F,C)
    finpara
    ** Salto de linea
  finpara
Finprograma
```

- **Codificación COBOL:**

```
IDENTIFICATION DIVISION.
PROGRAM-ID. CUADRADO-MAGICO.
DATA DIVISION.
WORKING-STORAGE SECTION.
01   TABLA-M.
     02 FILA-M OCCURS 25 TIMES.
        03 M PIC 999 OCCURS 25 TIMES.
01   VARIABLES.
     05 N   PIC 99.
     05 NUM PIC 999.
     05 F   PIC 99.
     05 C   PIC 99.
     05 R   PIC 99.
     05 COC PIC 99.
PROCEDURE DIVISION.
PROCESO.
     DISPLAY "Dimensión (Impar entre 1 y 25):"
     ACCEPT N
     DIVIDE N BY 2 GIVING C REMAINDER R
     PERFORM UNTIL R NOT = 0 AND N >= 1 AND <= 25
       DISPLAY "Dimensión incorrecta"
       DISPLAY "Dimensión (Impar entre 1 y 25):"
       ACCEPT N
       DIVIDE N BY 2 GIVING C REMANDER R
     END-PERFORM
     DISPLAY " " ERASE
     MOVE 1 TO F
     ADD 1 TO C
     PERFOM VARYING NUM FROM 1 BY 1 UNTIL NUM > N ** 2
        MOVE NUM TO M(F, C)
        DIVIDE NUM BY N GIVING COC REMAINDER R
        IF R = 0
          THEN ADD 1 TO F
          ELSE IF F = 1
                 THEN MOVE N TO F
                 ELSE SUBTRACT 1 FROM F
               END-IF
               IF C = N
                 THEN MOVE 1 TO C
                 ELSE ADD 1 TO C
               END-IF
        END-IF
     END-PERFORM
     PERFORM VARYING F FROM 1 BY 1 UNTIL F > N
        PERFORM VARYING C FROM 1 BY 1 UNTIL C > N
           DISPLAY M(F, C) NO ADVANCING
        END-PERFORM
        DISPLAY " "
     END-PERFORM
     STOP RUN.
```

- **Codificación Pascal:**

```
PROGRAM CUADRADOMAGICO (INPUT, OUTPUT);
VAR M : ARRAY[1..25,1..25] OF INTEGER;
    N, NUM, F, C : INTEGER;
BEGIN (*CUADRADO MAGICO*)
  WRITE('Dimensión: ');
  READLN(N);
  WHILE (N MOD 2 = 0) OR (1 > N) OR (N > 25) DO
    BEGIN (*1*)
      WRITELN('Dimensión incorrecta');
      WRITE('Dimension: ');
      READLN(N)
    END; (*1*)
  F := 1;
  C := N DIV 2 + 1;
  FOR NUM := 1 TO N * N DO
    BEGIN (*2*)
      M[F,C] := NUM;
      IF NUM MOD N = 0
        THEN F := F + 1
        ELSE
          BEGIN (*3*)
            IF F = 1 THEN F := N
                     ELSE F := F -1;
            IF C = N THEN C := 1
                     ELSE C := C + 1
          END (*3*)
    END; (*2*)
  FOR F := 1 TO N DO
    BEGIN (*4*)
      FOR C := 1 TO N DO WRITE(M[F,C]:4);
      WRITELN
    END (*4*)
END. (*CUADRADO MAGICO*)
```

5. *Programa que recibe como dato un número entero positivo, correspondiente a una cantidad de dinero, y calcula e imprime el mejor desglose de moneda (mínimo número de unidades monetarias).*

Las unidades monetarias existentes son:

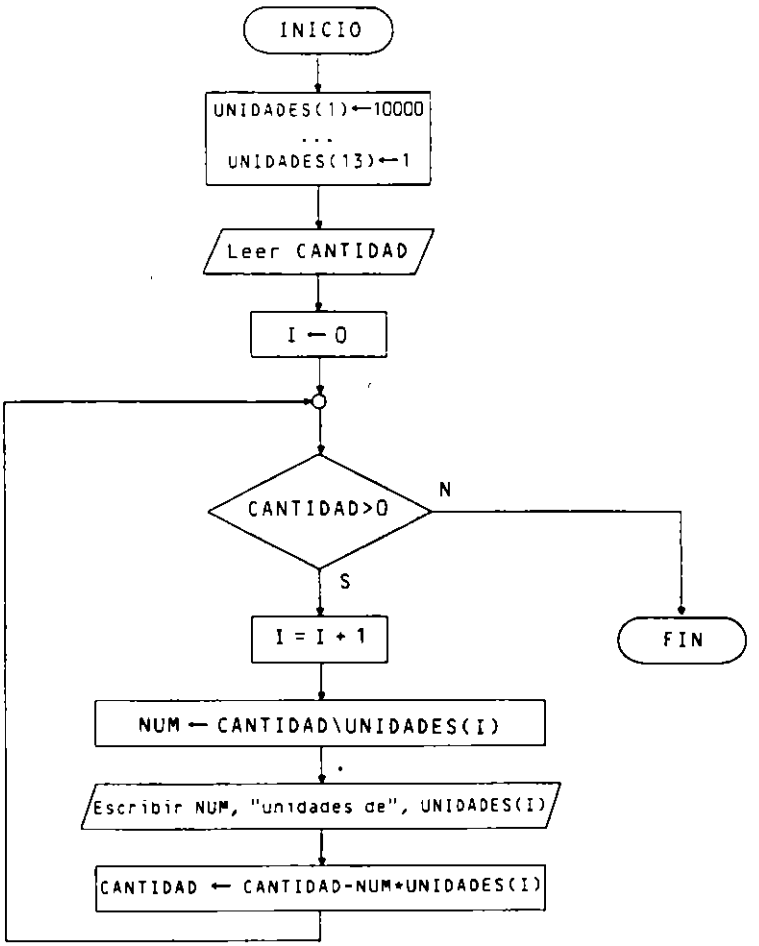
10000, 5000, 2000, 1000, 500, 200, 100, 50, 25, 10, 5, 2, 1.

El programa almacenará estas cantidades ordenadas en un vector y desglosará la cantidad en orden decreciente de las componentes del vector.

- **Objetos:**

UNIDADES. Vector de 13 elementos para almacenar las unidades monetarias.
CANTIDAD. Variable para mantener la cantidad de dinero a desglosar.
I. Contador.
NUM. Variable para calcular el número de unidades de cada unidad monetaria.

- **Ordinograma:**



- **Pseudocódigo:**

```
Programa DESGLOSE DE MONEDA
Entorno:
  UNIDADES es tabla(13) numérica entera
  CANTIDAD, NUM, I son numéricas enteras
Algoritmo:
  UNIDADES(1) ← 10000
  UNIDADES(2) ← 5000
  ...
  UNIDADES(12) ← 2
  UNIDADES(13) ← 1
  escribir " Escriba la cantidad a desglosar "
  leer CANTIDAD
  I ← 0
  mientras CANTIDAD > 0 hacer
    I ← I + 1
    NUM ← CANTIDAD \ UNIDADES(I)
    escribir NUM, " unidades de ", UNIDADES(I)
    CANTIDAD ← CANTIDAD - NUM * UNIDADES(I)
  finmientras
Finprograma
```

- **Codificación COBOL:**

```
IDENTIFICATION DIVISION.
PROGRAM-ID. DESGLOSE-MONEDAS.
DATA DIVISION.
WORKING-STORAGE SECTION.
01   TABLA-DATOS.
     02 FILLER PIC 9(5) VALUE 10000.
     02 FILLER PIC 9(5) VALUE  5000.
     02 FILLER PIC 9(5) VALUE  2000.
     02 FILLER PIC 9(5) VALUE  1000.
     02 FILLER PIC 9(5) VALUE   500.
     02 FILLER PIC 9(5) VALUE   200.
     02 FILLER PIC 9(5) VALUE   100.
     02 FILLER PIC 9(5) VALUE    50.
     02 FILLER PIC 9(5) VALUE    25.
     02 FILLER PIC 9(5) VALUE    10.
     02 FILLER PIC 9(5) VALUE     5.
     02 FILLER PIC 9(5) VALUE     2.
     02 FILLER PIC 9(5) VALUE     1.
01   TABLA-UNIDADES REDEFINES TABLA-DATOS.
     02 UNIDADES PIC 9(5) OCCURS 13 TIMES.
01   VARIABLES.
     05 CANTIDAD PIC 9(7).
     05 RESTO    PIC 9(7).
     05 NUM      PIC 9(3).
     05 I        PIC 99.
PROCEDURE DIVISION.
PROCESO.
   DISPLAY "Escriba la cantidad a desglosar:"
   ACCEPT CANTIDAD
   MOVE 0 TO I
   PERFORM UNTIL CANTIDAD = 0
      ADD 1 TO I
      DIVIDE CANTIDAD BY UNIDADES(I) GIVING NUM REMAINDER RESTO
      DISPLAY NUM, " unidades de ", UNIDADES(I)
      MOVE RESTO TO CANTIDAD
   END-PERFORM
   STOP RUN.
```

- **Codificación Pascal:**

```
PROGRAM DESGLOSEMONEDA (INPUT, OUTPUT);
VAR UNIDADES : ARRAY[1..13] OF INTEGER;
    CANTIDAD, NUM, I : INTEGER;
BEGIN (* DESGLOSEMONEDA *)
  UNIDADES[1]  := 10000; UNIDADES[2]  := 5000;
  UNIDADES[3]  := 2000;  UNIDADES[4]  := 1000;
  UNIDADES[5]  := 500;   UNIDADES[6]  := 200;
  UNIDADES[7]  := 100;   UNIDADES[8]  := 50;
  UNIDADES[9]  := 25;    UNIDADES[10] := 10;
  UNIDADES[11] := 5;     UNIDADES[12] := 2;
  UNIDADES[13] := 1;
  WRITE('Escriba la cantidad a desglosar: ');
  READLN(CANTIDAD);
  I := 0;
  WHILE CANTIDAD > 0 DO
    BEGIN (*1*)
      I := I + 1;
```

```
        NUM := CANTIDAD DIV UNIDADES[I];
        WRITELN(NUM, 'unidades de ', UNIDADES[I]);
        CANTIDAD := CANTIDAD MOD UNIDADES[I]
      END (*1*)
END. (* DESGLOSEMONEDA *)
```

EJERCICIOS PROPUESTOS

1. Programa que lee 50 números enteros sobre un vector y obtiene e imprime cuáles son el mayor y el menor número almacenados y cuántas veces se repiten ambos.

2. Con el mismo vector del ejercicio anterior. Programa que calcula e imprime las sumas de las componentes de índice par y las de índice impar.

3. Con el mismo vector del ejercicio anterior. Programa que invierte dicho vector imprimiéndolo.

4. Dado un número entero positivo de 10 cifras. Programa que compruebe si el número es capicúa utilizando un vector de 10 componentes.

5. Programa que carga una matriz de cinco filas y 10 columnas con números enteros, imprimiendo los valores máximo y mínimo y sus posiciones dentro de la tabla.

6. Con la misma matriz del ejercicio anterior. Programa que obtiene e imprime su matriz traspuesta.

7. Programa que genera e imprime una matriz unitaria de orden N. Una matriz unitaria de orden N es la que tiene N filas y N columnas con todas sus componentes a 0, excepto las de su diagonal principal, que están a 1.

8. Programa que lee una matriz cuadrada de orden 3 y calcula e imprime su potencia N-ésima, siendo N un dato de entrada.

9. Programa que lee un vector de N elementos y rota todas sus componentes un lugar hacia su derecha. Teniendo en cuenta que la última componente se ha de desplazar al primer lugar.

10. Programa que imprime un cuadrado latino de orden N. Un cuadrado latino de orden N es una matriz cuadrada en la que su primera fila contiene los N primeros números naturales y cada una de las siguientes N-1 filas contiene la rotación de la fila anterior un lugar a la derecha.

Ejemplo: *Cuadrado latino de orden 4.*

1	2	3	4
4	1	2	3
3	4	1	2
2	3	4	1

CAPITULO 6

Búsqueda y clasificación interna

6.1. INTRODUCCION

Existen multitud de algoritmos para manejar las estructuras de datos internas. De entre ellos destacan por su importancia y frecuencia de utilización los de búsqueda y clasificación.

Estos algoritmos son clásicos y su conocimiento es fundamental para cualquier programador.

A lo largo de la historia de la programación se han estudiado y propuesto diversos métodos. De todos ellos propondremos los más importantes y también los más sencillos.

La elección final de uno u otro método dependerá de las características particulares de un problema concreto y de sus datos.

Los algoritmos que vamos a estudiar a continuación son los siguientes:

■ **Algoritmos de búsqueda:**

- Para datos no ordenados.
  - — Búsqueda lineal.
- Para datos ordenados.
  - — Búsqueda lineal.
  - — Búsqueda dicotómica.

■ **Algoritmos de clasificación:**

- — Método de inserción directa.
- — Método de selección directa.
- — Método de intercambio directo (burbuja).
- — Método de intercambio directo con test (switch).
- — Método de intercambio directo (sacudida).
- — Método Shell.

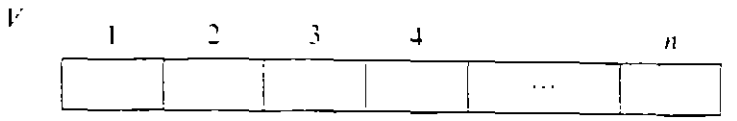
6.2. BUSQUEDA LINEAL

Dada una tabla y un valor del mismo tipo que sus componentes, la búsqueda consiste en determinar si ese valor está en la tabla y qué posición ocupa.

Los algoritmos pueden considerar los datos contenidos en la tabla repetidos o no. En principio basaremos el estudio suponiendo que no se repiten, dejando como ejercicio al lector las pequeñas modificaciones que en cada caso serían necesarias para la búsqueda de un dato que puede aparecer más de una vez, determinando los distintos lugares que ocupa.

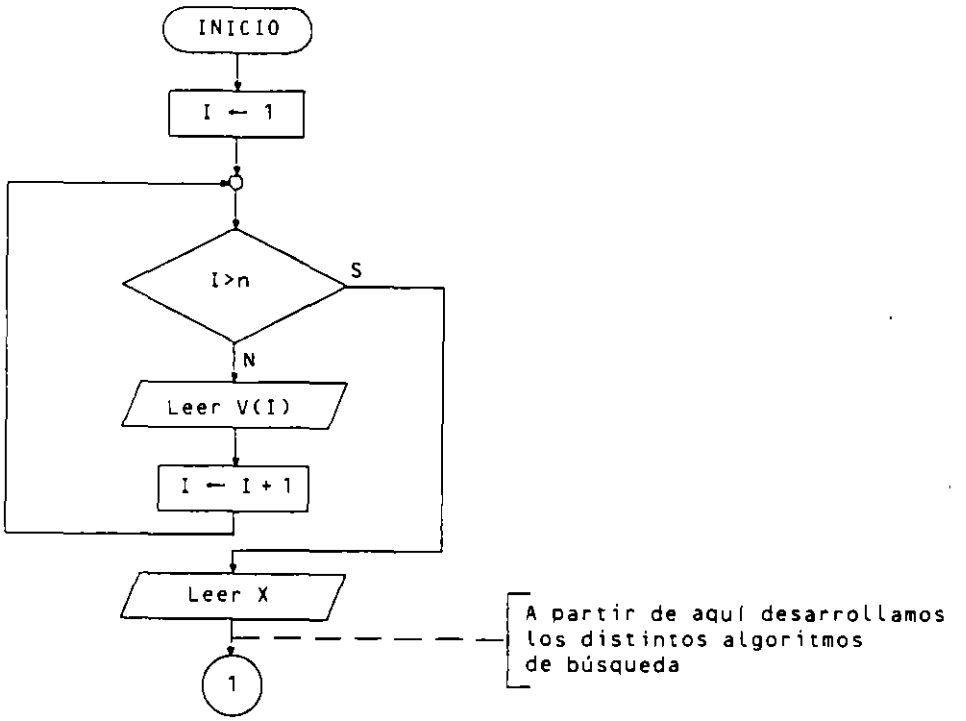
Aplicaremos los algoritmos a la búsqueda de un valor contenido en una variable X (leída por teclado) sobre un vector V de n elementos (siendo n un valor numérico entero

y positivo) que supondremos en todos los casos ya en memoria después de ejecutadas las siguientes instrucciones:



X Variable que contiene el valor a buscar en el vector V.

I Es el contador asociado a los bucles.



Los datos cargados en el vector V y el valor X pueden proceder de un proceso anterior diferente del expuesto.

6.2.1. BUSQUEDA LINEAL EN UN VECTOR

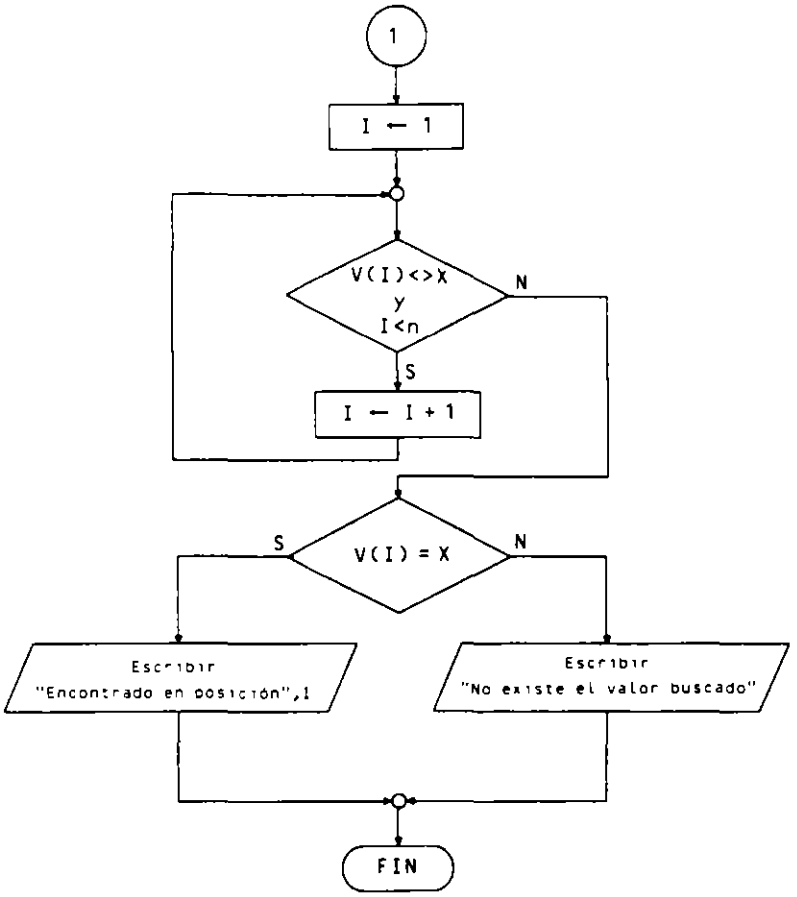
Se recorre el vector de izquierda a derecha hasta encontrar una componente cuyo valor coincida con el buscado o hasta que se acabe el vector. En este último caso, el algoritmo debe indicar la no existencia de dicho valor.

En los casos en que pueda aparecer el valor repetido, el algoritmo indicará el de índice menor. Con una pequeña modificación podemos obtener las distintas posiciones que ocupa el valor.

Este método es válido tanto para vectores desordenados como ordenados, aunque para vectores ordenados veremos otros métodos más eficientes.

Sea un vector V de n componentes y un valor X a buscar en V.

- **Ordinograma:**



- **Pseudocódigo:**

```
...
I ← 1
mientras V(I) <> X y I < n hacer
  I ← I + 1
finmientras
si V(I) = X
  entonces escribir "Encontrado en posición ", I
  sino escribir "No existe el valor buscado"
finsi
Finprograma
```

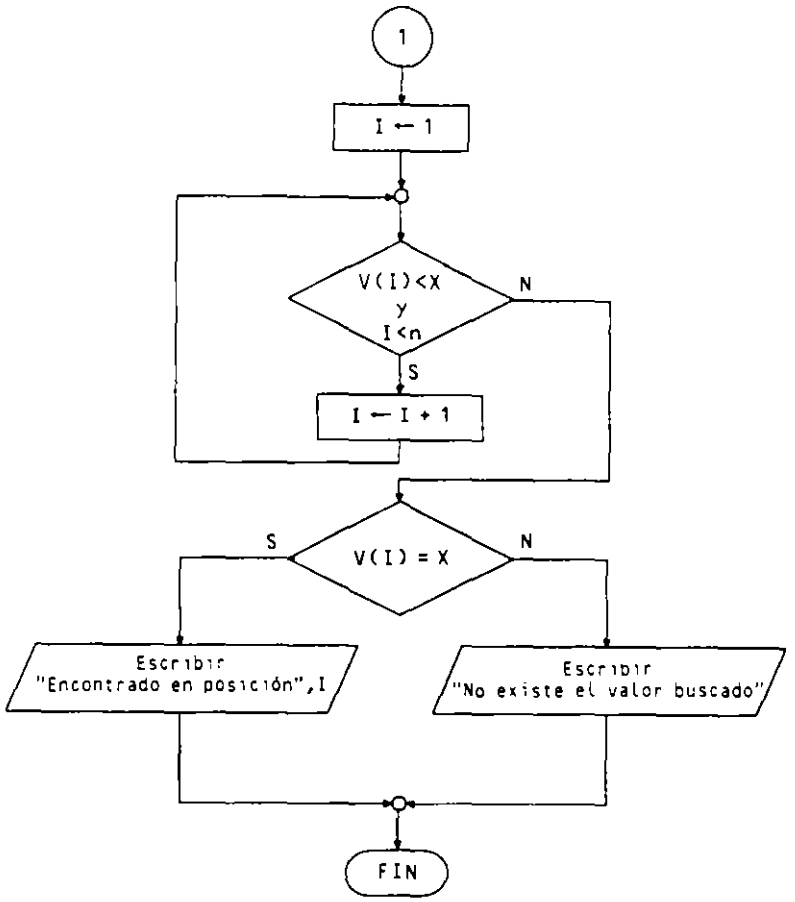
6.2.2. BUSQUEDA LINEAL EN UN VECTOR ORDENADO

Cuando el vector de búsqueda está ordenado se consigue un algoritmo más eficiente con sólo modificar la condición de terminación en el algoritmo anterior.

La ventaja que se obtiene es que, una vez sobrepasado el valor buscado, no es necesario recorrer el resto del vector para saber que el valor no existe.

Sea un vector V de n componentes clasificado en orden ascendente y un valor X a buscar:

- **Ordinograma:**



- **Pseudocódigo:**

```
...
I ← 1
mientras V(I) < X y I < n hacer
  I ← I + 1
finmientras
si V(I) = X
  entonces escribir "Encontrado en posición ", I
  sino escribir "No existe el valor buscado"
finsi
Finprograma
```

6.2.3. BUSQUEDA LINEAL EN UNA MATRIZ

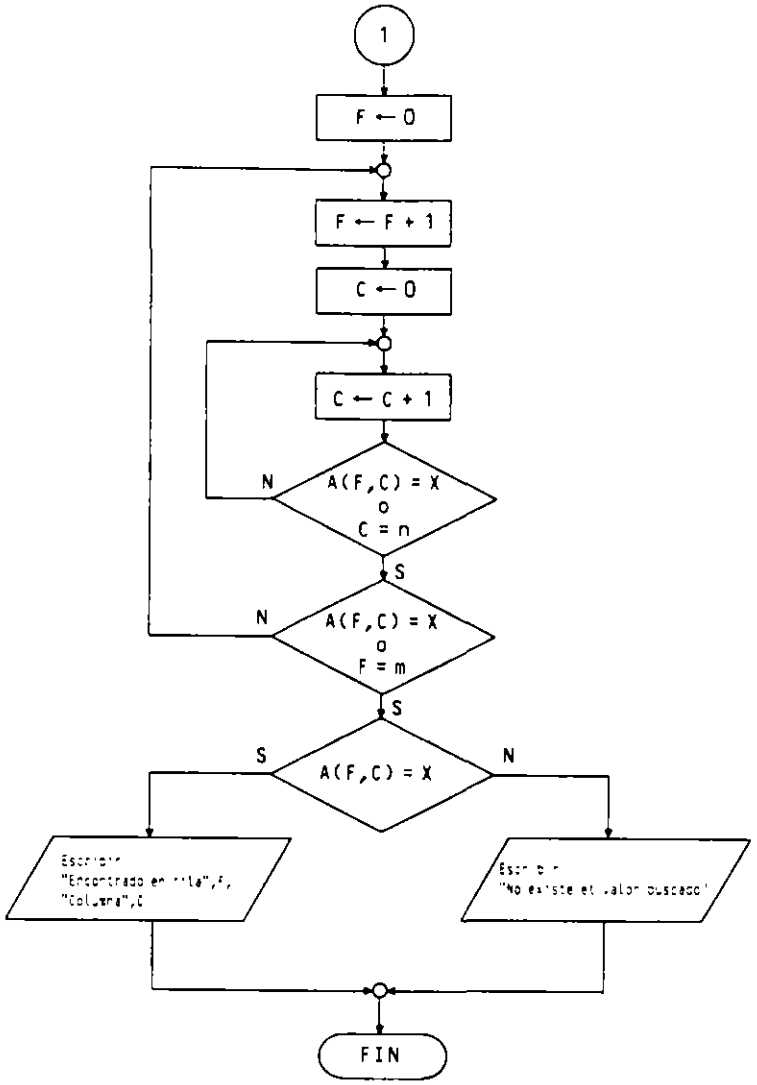
Se realiza mediante el anidamiento de dos bucles tipo hasta. cuya finalización vendrá dada por la aparición del valor buscado o la terminación de la matriz.

Normalmente se comienza recorriendo la matriz por filas: aunque cambiando de posición los indices con sus correspondientes límites se puede hacer igualmente por columnas.

Sea una matriz A de m filas y n columnas y un valor X a buscar en A:

Suponemos la matriz A y la variable X ya cargadas previamente. F y C son dos contadores para el direccionamiento de las filas y las columnas.

- **Ordinograma:**



- **Pseudocódigo:**

```
...
F ← 0
repetir
  F ← F + 1
  C ← 0
  repetir
     C ← C + 1
  hasta A(F,C) = X o C = n
hasta A(F,C) = X o F = m
si A(F,C) = X
  entonces escribir "Encontrado en fila ", F, " Columna ", C
  sino escribir "No existe el valor buscado"
finsi
Finprograma
```

6.3. BUSQUEDA BINARIA O DICOTOMICA

Este algoritmo es válido exclusivamente para vectores ordenados y consiste en comparar en primer lugar con la componente central del vector, y si no es igual al valor buscado se reduce el intervalo de búsqueda a la mitad derecha o izquierda, según donde pueda encontrarse el valor a buscar. El algoritmo termina si se encuentra el valor buscado o si el tamaño del intervalo de búsqueda queda anulado.

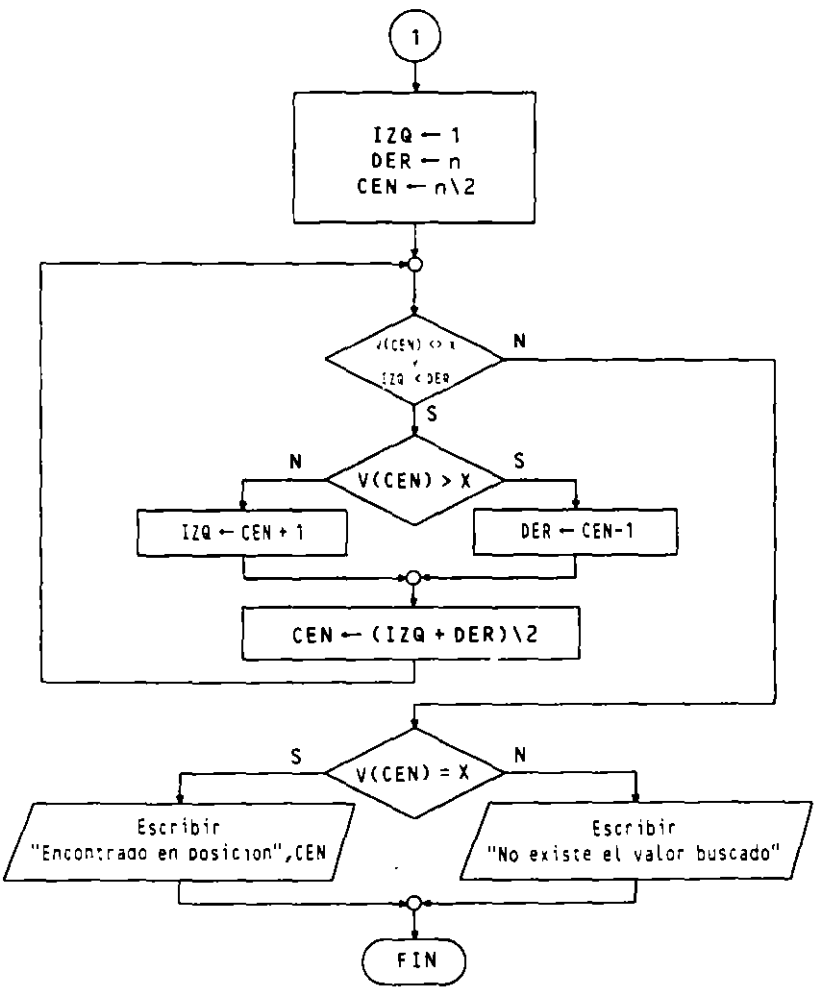
En los casos en que existan repeticiones en el vector, del valor buscado, este algoritmo obtendrá uno de ellos aleatoriamente según los lugares que ocupen, los cuales necesariamente son consecutivos.

Se trata de una búsqueda similar a la de una palabra en un diccionario, donde aprovechamos el orden alfabético para dirigirnos con rapidez al lugar donde puede estar la palabra buscada.

Para su realización utilizamos tres variables que nos indican en qué zona del vector nos encontramos. Son estas variables IZQ, DER y CEN.

El vector V se supone ordenado ascendentemente.

- **Ordinograma:**



- **Pseudocódigo:**

```
...
IZQ ← 1
DER ← n
CEN ← (n + 1)\2
mientras V(CEN) <> X y IZQ < DER hacer
   si V(CEN) > X
      entonces
        DER ← CEN - 1
      sino
        IZQ ← CEN + 1
   finsi
   CEN ← (IZQ + DER) \ 2
finmientras
si V(CEN) = X
   entonces
     escribir "Encontrado en posición ", CEN
   sino
     escribir "No existe el valor buscado"
finsi
Finprograma
```

6.4. ORDENACION DE TABLAS

Muchos algoritmos necesitan utilizar tablas ordenadas, por lo cual es preciso ordenarlas o clasificarlas previamente.

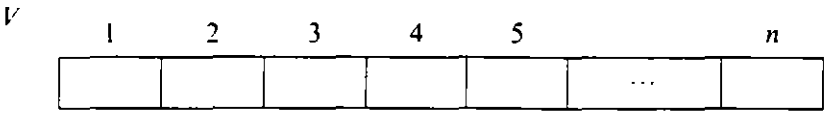
Podemos definir una **ordenación** como la reagrupación de un conjunto de elementos en una secuencia específica.

Los tipos de ordenación que realizaremos son:

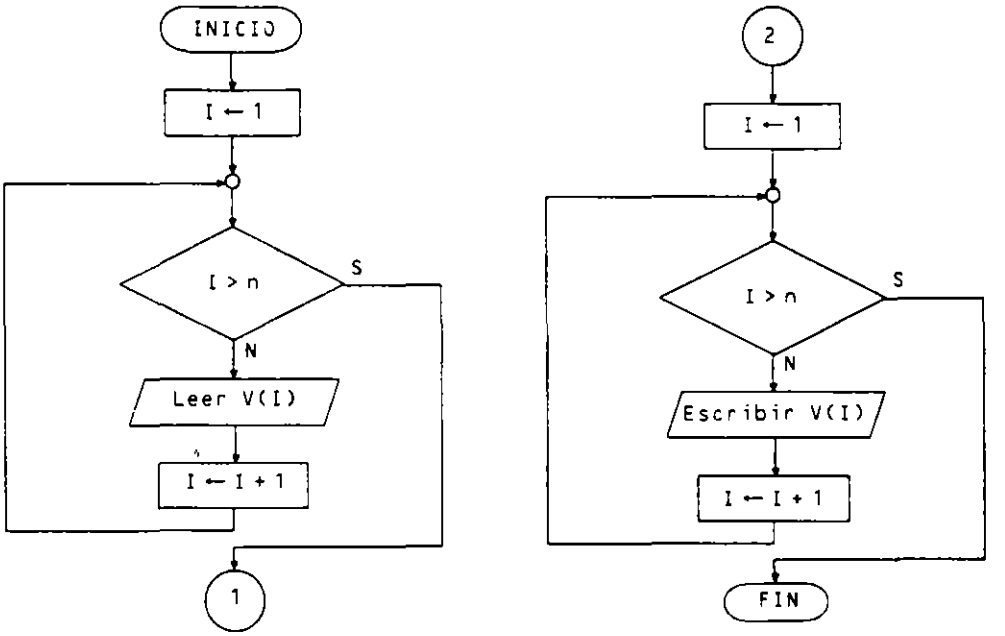
— Ordenación ascendente o creciente.
  Consiste en situar los valores mayores a la derecha y los menores a la izquierda. Los valores repetidos quedarán en posiciones consecutivas. En casos de valores alfanuméricos, el orden utilizado es el lexicográfico.
— Ordenación descendente o decreciente.
  Es la ordenación inversa a la anterior.

Estudiaremos varios métodos de ordenación aplicados a vectores, teniendo en cuenta que se pueden generalizar a matrices con respecto a una de sus filas o columnas y a poliedros con respecto a un valor de una de sus dimensiones.

Todos los algoritmos los realizaremos sobre un vector V de n elementos y podemos suponer realizadas las siguientes operaciones de carga previa de sus elementos y escritura posterior de los mismos.



I es el contador asociado a los bucles.



Se entiende que entre los conectores 1 y 2 se encuentran los algoritmos que estudiaremos a continuación.

6.4.1. ORDENACION POR INSERCION DIRECTA

También se denomina **método de la baraja**.

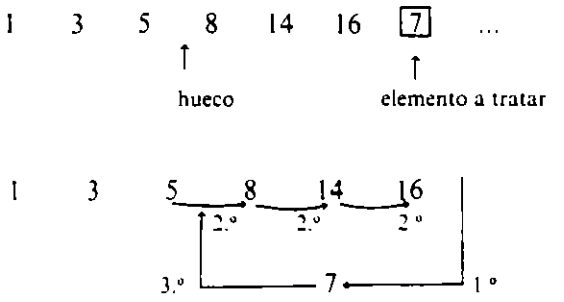
Supongamos el vector V de n componentes numéricas o alfanuméricas que deseamos clasificar en orden ascendente.

El método consiste en tomar los elementos del vector desde el segundo hasta el último y con cada uno de ellos repetir el siguiente conjunto de operaciones:

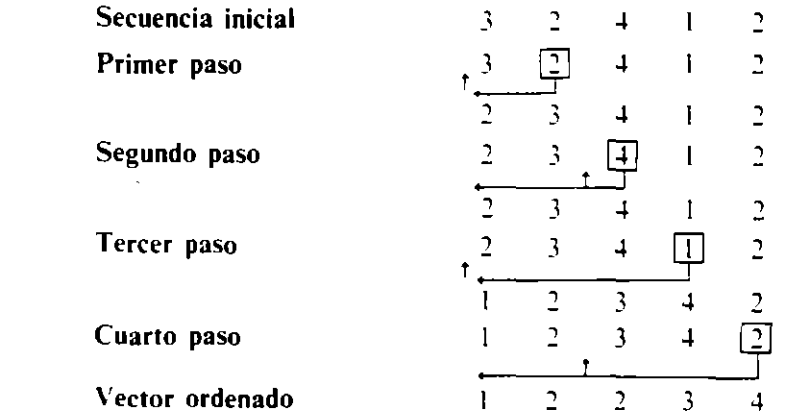
1.° Se saca del vector el elemento $V(I)$ (AUX es una variable que lo recibe).
2.° Desde el anterior al que estamos tratando y hasta el primero, desplazamos un lugar a la derecha todos los que sean mayores para buscar su hueco.
3.° Encontrado el hueco del elemento, lo insertamos en él.

Conviene observar que, cuando tratamos un elemento, todos los anteriores se encuentran ordenados.

Ejemplo:



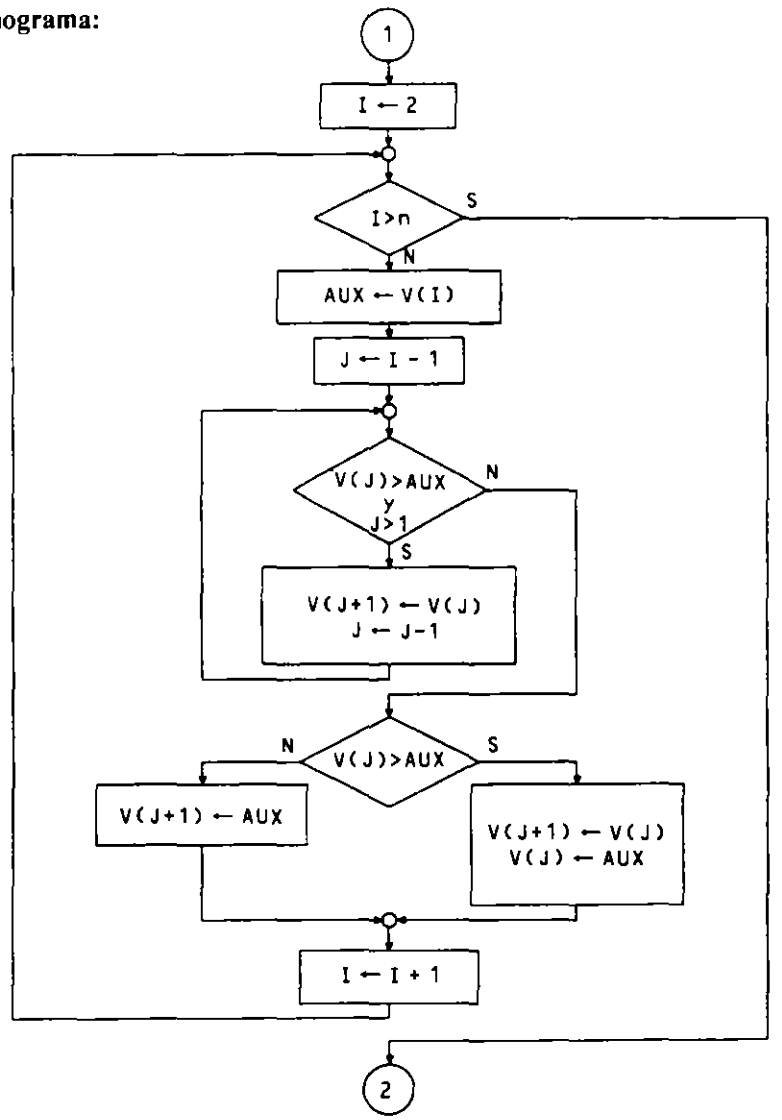
Veamos la evolución de una secuencia de números mediante este método:



- **Objetos:**

AUX es una variable auxiliar del mismo tipo que los elementos del vector.
I direcciona cada elemento a insertar.
J direcciona los anteriores.

- **Ordinograma:**



- **Pseudocódigo:**

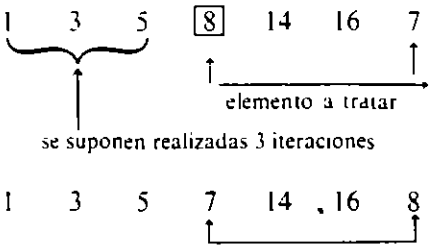
```
...
para I de 2 a n hacer
  AUX ← V(I)
  J ← I-1
  mientras V(J) > AUX y J > 1 hacer
    V(J + 1) ← V(J)
    J ← J-1
  finmientras
  si V(J) > AUX
    entonces
      V(J + 1) ← V(J)
      V(J) ← AUX
    sino
      V(J + 1) ← AUX
  finsi
finpara
...
```

6.4.2. ORDENACION POR SELECCION DIRECTA

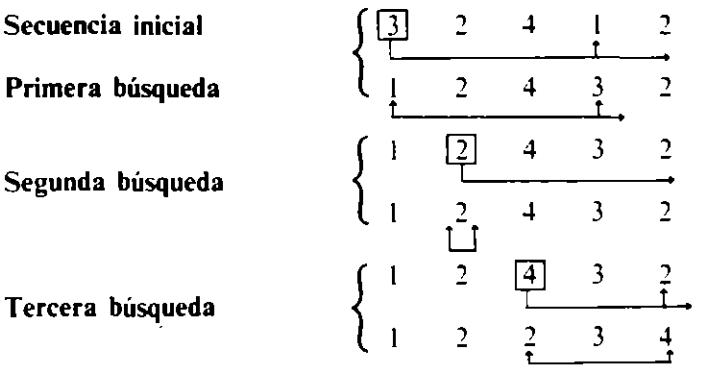
El método consiste en repetir el siguiente proceso desde el primer elemento hasta el penúltimo: se selecciona la componente de menor valor de todas las situadas a la derecha de la tratada y se intercambia con ésta.

En realidad se trata de sucesivas búsquedas del menor de los elementos que quedan por ordenar. En la primera iteración se busca el menor de todos.

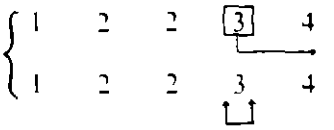
Ejemplo:



Para la realización del intercambio se utiliza una variable auxiliar AUX.
Veamos la evolución de una secuencia de números mediante este método:



Cuarta búsqueda

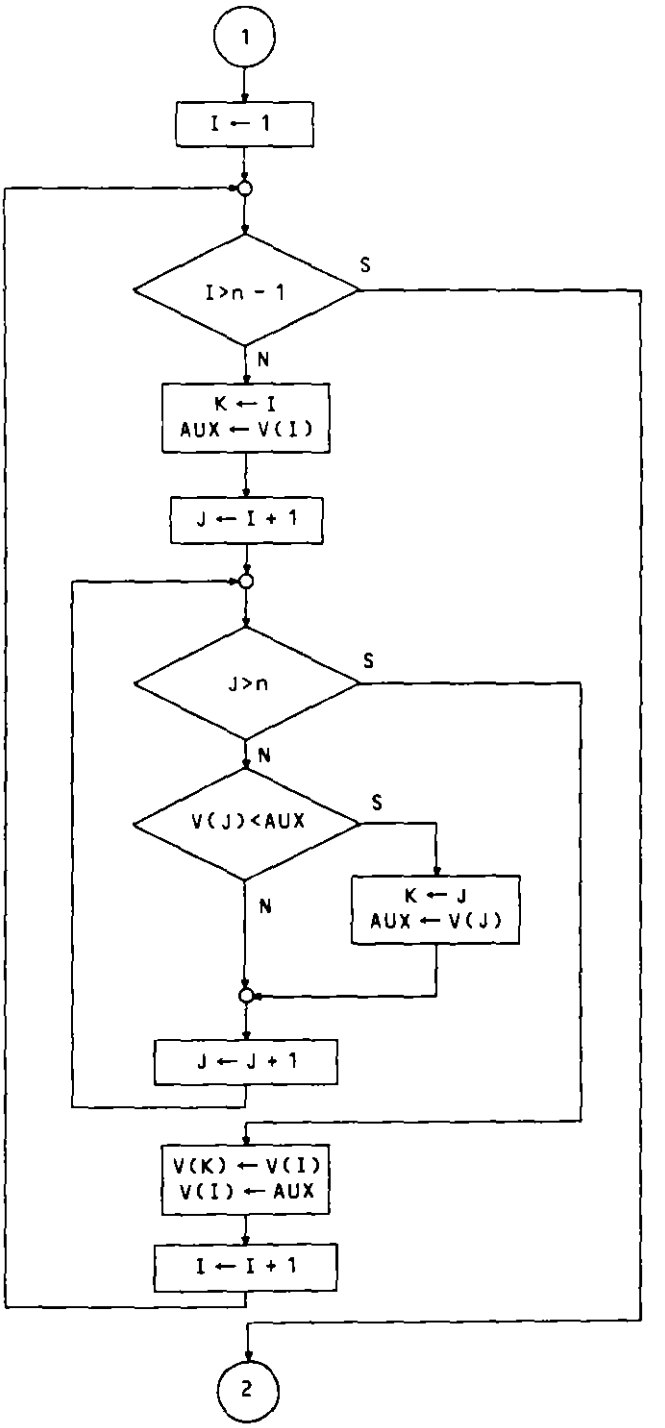


Vector ordenado 1 2 2 3 4

- **Objetos:**

K, AUX. Guardan la posición y valor del menor en cada búsqueda.

- **Ordinograma:**



- **Pseudocódigo:**

```
...
para I de 1 a n - 1 hacer
  K ← I
  AUX ← V(I)
  para J de I + 1 a n hacer
    si V(J) < AUX
      entonces
        K ← J
        AUX ← V(J)
    finsi
  finpara
  V(K) ← V(I)
  V(I) ← AUX
finpara
...
```

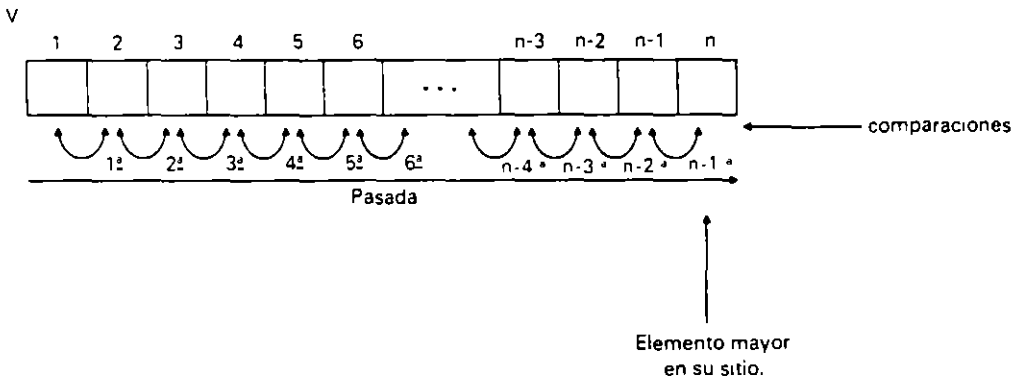
6.4.3. ORDENACION POR INTERCAMBIO DIRECTO. METODO DE LA BURBUJA

Este método tiene dos versiones basadas en la misma idea, que consiste en recorrer sucesivamente el vector comparando los elementos consecutivos e intercambiándolos cuando estén descolocados. El recorrido del vector se puede hacer de izquierda a derecha (desplazando los valores mayores hacia su derecha) o de derecha a izquierda (desplazando los valores menores hacia su izquierda), ambos para la clasificación en orden ascendente.

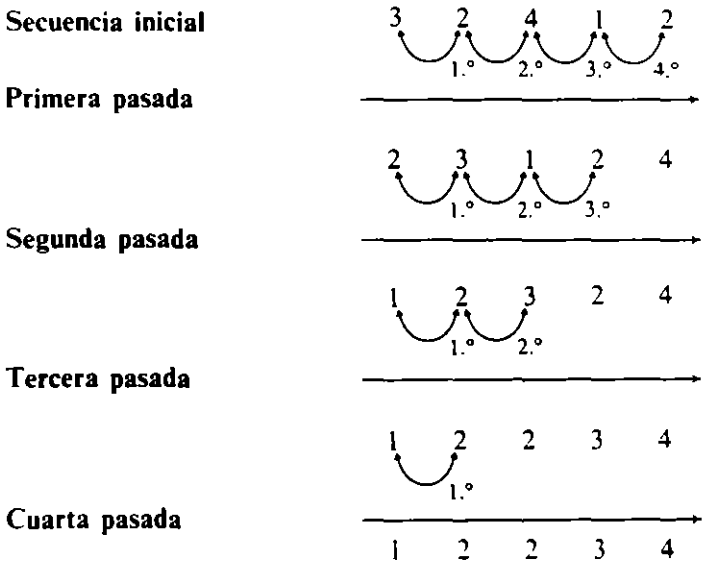
- **Recorrido izquierda-derecha:**

Consiste en realizar pasadas sucesivas direccionando desde el primer elemento hasta el penúltimo, comparando cada uno de ellos con el siguiente.

Esta versión del método va colocando en cada pasada el mayor elemento de los tratados en la última posición, quedando colocado y por tanto excluido de los elementos a tratar en la siguiente pasada.



Veamos la evolución de una secuencia de números mediante este método:

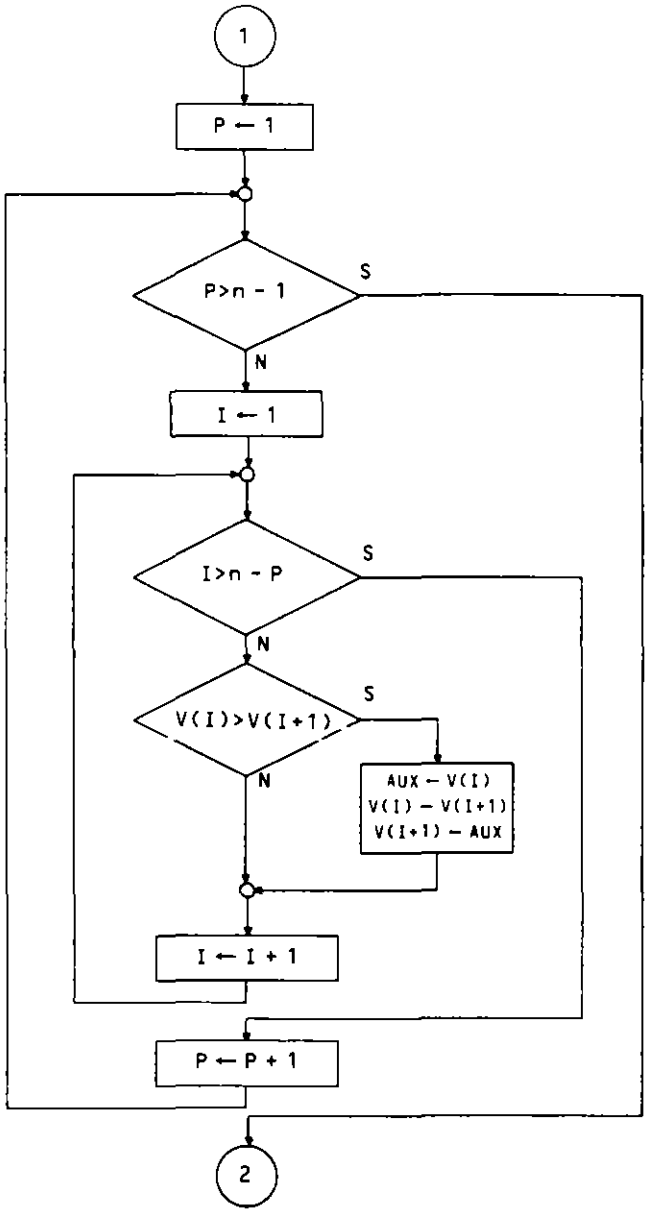


Obsérvese que cada pasada asegura el posicionamiento de un elemento por la derecha; por tanto, serán necesarias $n-1$ pasadas para asegurar en cualquier caso la ordenación.

- **Objetos:**

P es un contador de pasadas.
I direcciona los elementos.
AUX se utiliza para los intercambios.

- **Ordinograma:**



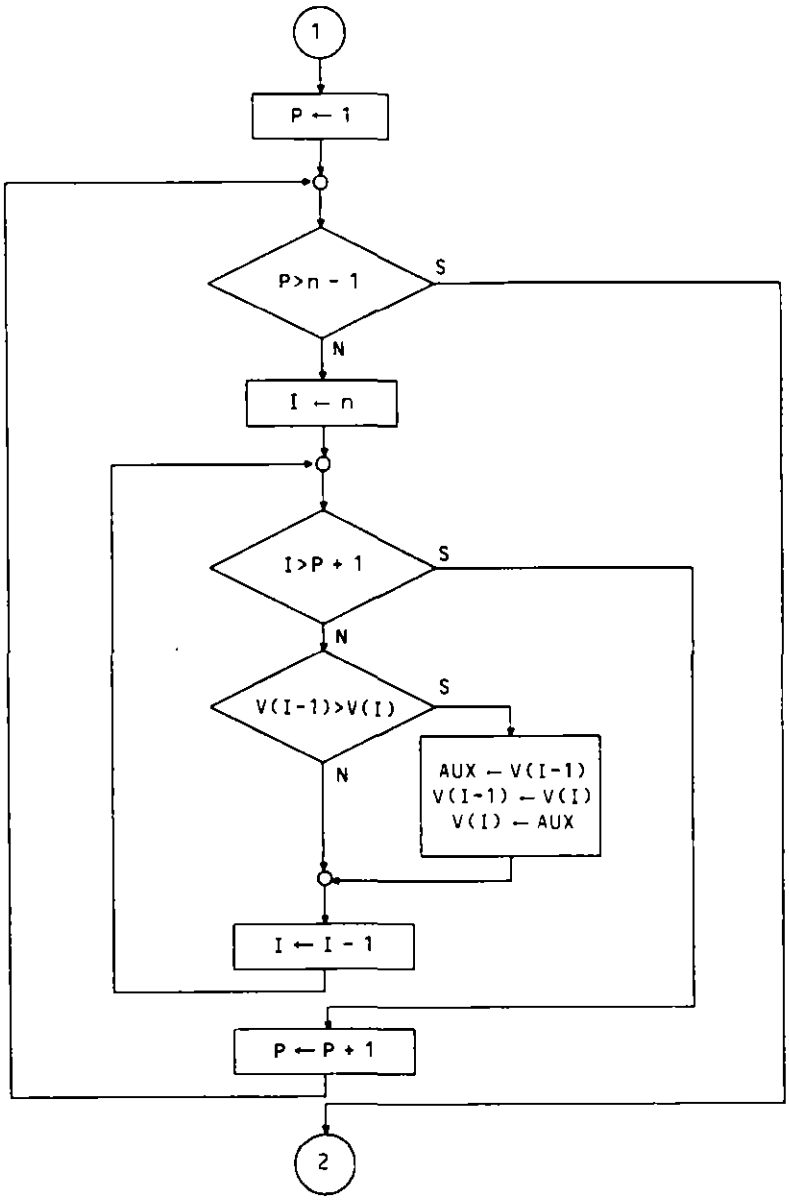
- **Pseudocódigo:**

```
...
para P de 1 a n-1 hacer
  para I de 1 a n - P hacer
    si V(I) > V(I+1)
      entonces
        AUX ← V(I)
        V(I) ← V(I+1)
        V(I+1) ← AUX
    finsi
  finpara
finpara
...
```

- **Recorrido derecha-izquierda:**

Esta versión es simétrica a la anterior desplazando los elementos menores hacia la izquierda.

- **Ordinograma:**



- **Pseudocódigo:**

```
...
para P de 1 a n - 1 hacer
  para I de n a P + 1 con incremento -1 hacer
    si V(I - 1) > V(I)
      entonces
        AUX ← V(I - 1)
        V(I - 1) ← V(I)
        V(I) ← AUX
    finsi
  finpara
finpara
...
```

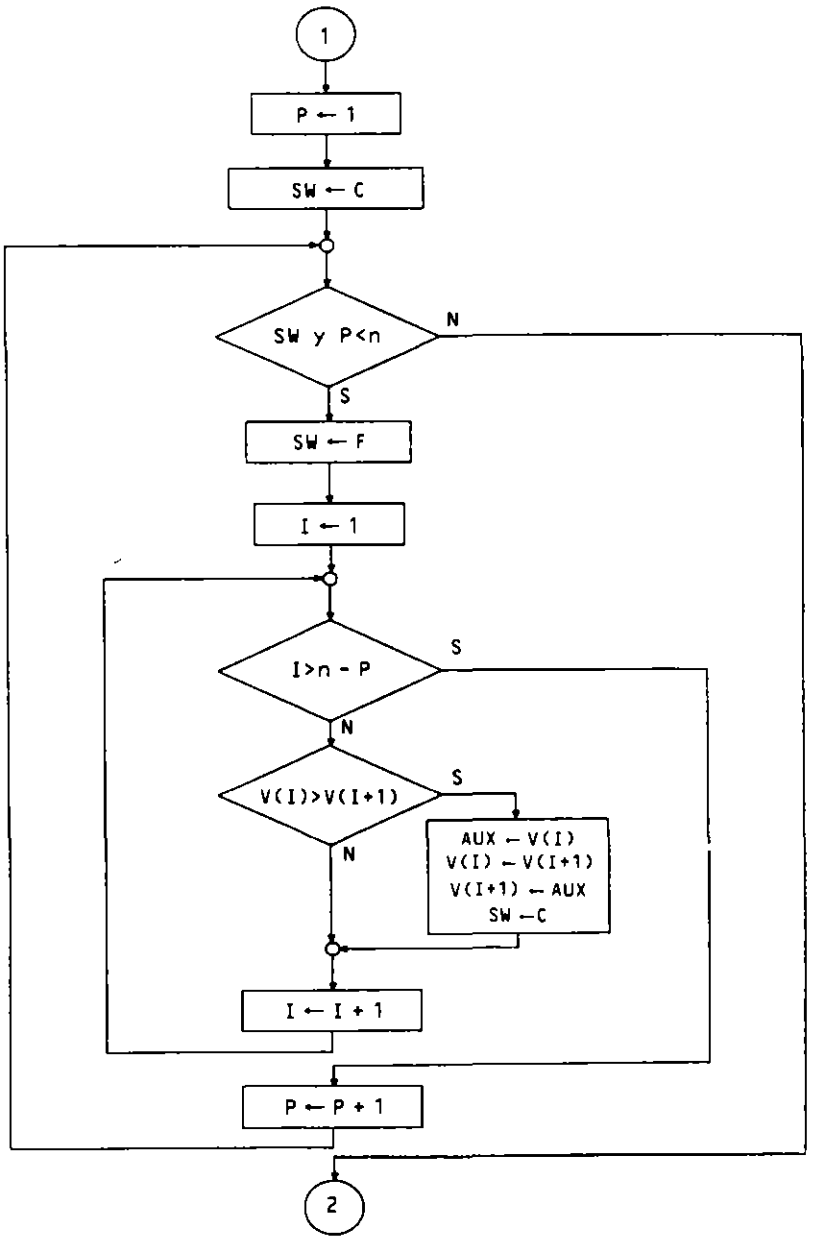
6.4.4. ORDENACION POR INTERCAMBIO DIRECTO CON TEST DE COMPROBACION (SWITCH)

Es una mejora de los métodos de intercambio directo en la que se comprueba mediante un **switch** (interruptor) si el vector está totalmente ordenado después de cada pasada, terminando la ejecución en caso afirmativo.

La comprobación de la ordenación en cada pasada consiste en detectar si se han producido intercambios o no, de tal forma que sólo se realizarán las pasadas necesarias que dependerán del grado de desorden existente en los datos iniciales.

$$SW \begin{cases} F \rightarrow \text{vector ordenado} \\ C \rightarrow \text{vector desordenado} \end{cases}$$

- **Ordinograma:**



- **Pseudocódigo:**

```
...
P ← 1
SW ← CIERTO
mientras SW y P < n hacer
  SW ← FALSO
  para I de 1 a n - P hacer
    si V(I) > V(I+1)
      entonces
        AUX ← V(I)
        V(I) ← V(I+1)
        V(I+1) ← AUX
        SW ← CIERTO
    finsi
  finpara
  P ← P + 1
finmientras
...
```

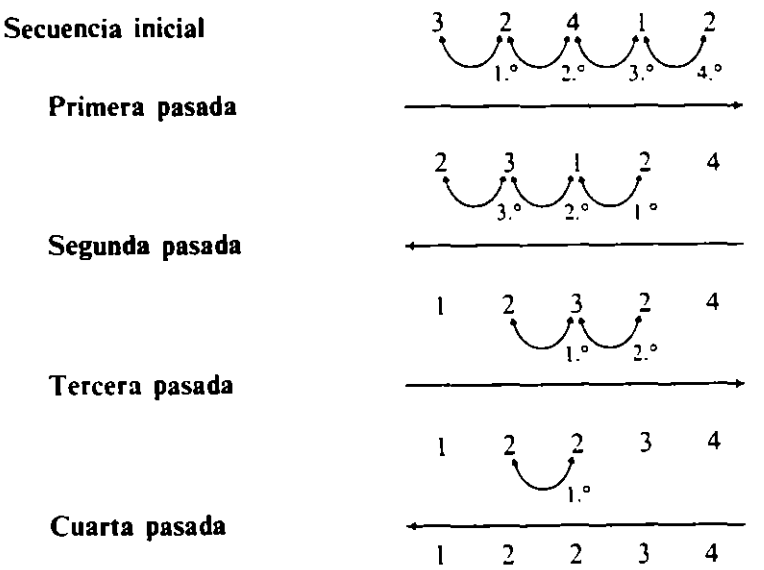
6.4.5. ORDENACION POR INTERCAMBIO DIRECTO. METODO DE LA SACUDIDA

Consiste en mezclar las dos versiones del método de la burbuja alternativamente, de tal forma que se realiza una pasada de izquierda a derecha y a continuación otra de derecha a izquierda, recortándose los elementos a tratar por ambos lados del vector.

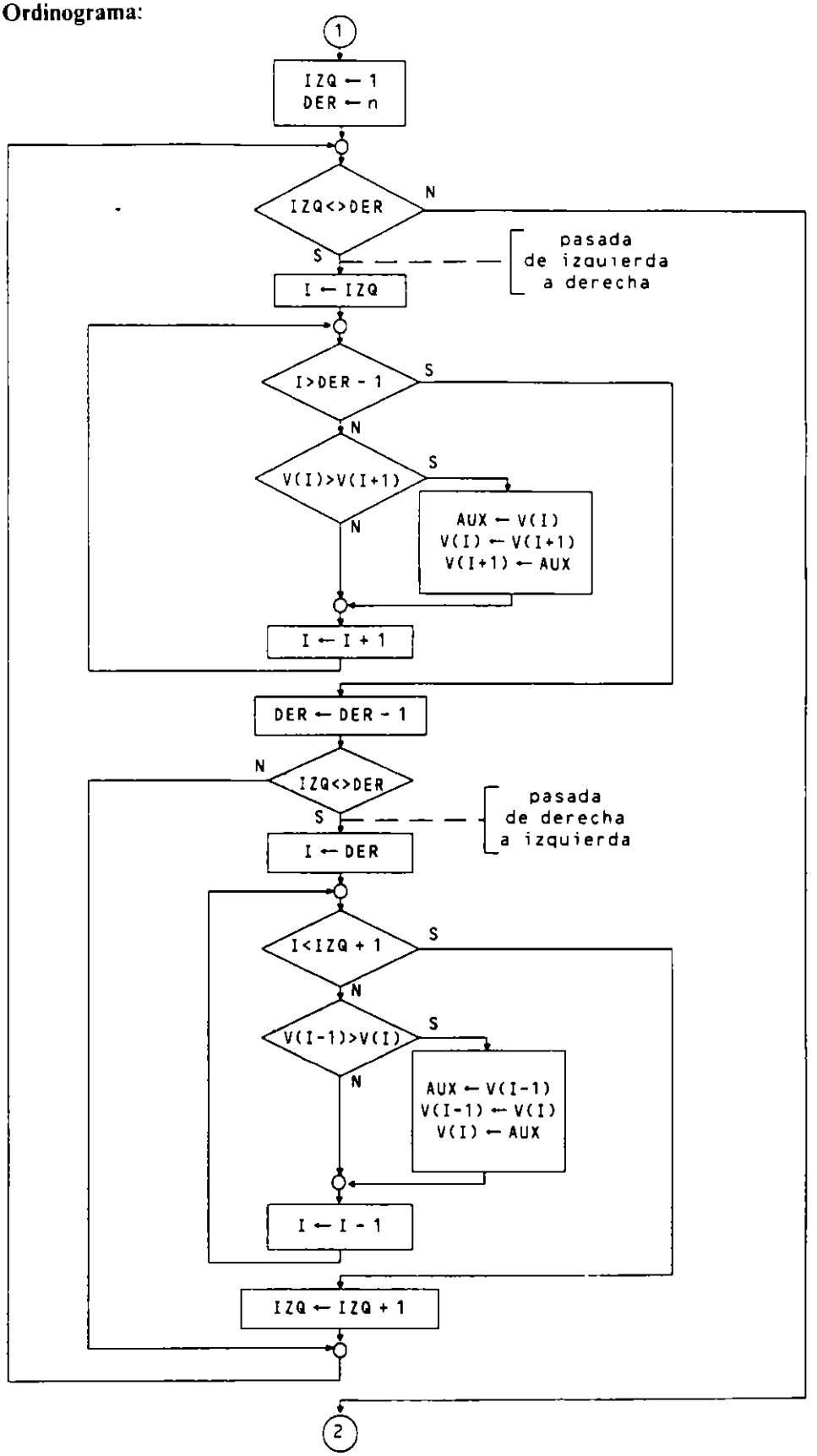
Para la realización del algoritmo se utilizan dos apuntadores, IZQ y DER, que nos indican en cada momento la zona del vector que queda por ordenar.

En cada pasada de izquierda a derecha recortamos una comparación por la derecha (DER ← DER − 1) y en cada pasada de derecha a izquierda lo hacemos por la izquierda (IZQ ← IZQ + 1).

Veamos la evolución de una secuencia de números mediante este método:



- **Ordinograma:**



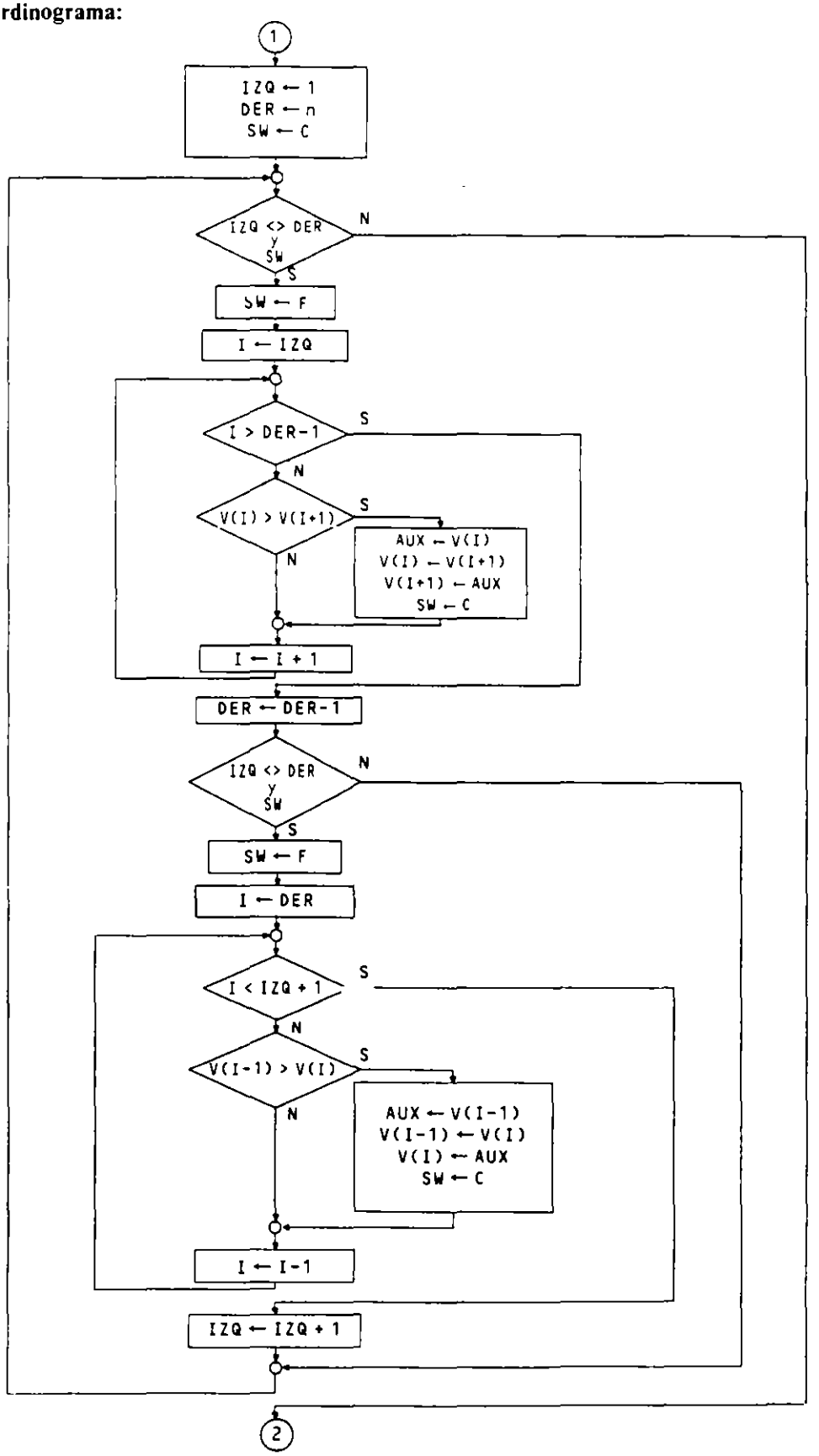
- **Pseudocódigo:**

```
...
IZQ ← 1
DER ← n
mientras IZQ <> DER hacer
  ** pasada de izquierda a derecha
  para I de IZQ a DER - 1 nacer
    si V(I) > V(I+1)
      entonces
        AUX ← V(I)
        V(I) ← V(I+1)
        V(I+1) ← AUX
    finsi
  finpara
  DER ← DER-1
  si IZQ <> DER
     entonces
       ** pasada de derecha a izquierda
       para I de DER a IZQ + 1 con incremento -1 hacer
         si V(I-1) > V(I)
            entonces
              AUX ← V(I-1)
              V(I-1) ← V(I)
              V(I) ← AUX
         finsi
       finpara
       IZQ ← IZQ + 1
  finsi
finmientras
...
```

Este método aumenta su efectividad si se le incluye un switch para detectar si en una pasada, tanto de izquierda a derecha como de derecha a izquierda, se encuentran todos los elementos ordenados; es decir, si en una de sus pasadas no se producen intercambios, podemos terminar sabiendo que toda la información contenida se encuentra ordenada.

$$SW \begin{cases} F \rightarrow \text{vector ordenado} \\ C \rightarrow \text{vector no ordenado} \end{cases}$$

- **Ordinograma:**



- **Pseudocódigo:**

```
...
IZQ ← 1
DER ← n
SW ← CIERTO
mientras IZQ <> DER y SW hacer
  SW ← FALSO
  para I de IZQ a DER-1 hacer
    si V(I) > V(I+1)
       entonces
         AUX ← V(I)
         V(I) ← V(I+1)
         V(I+1) ← AUX
         SW ← CIERTO
    finsi
  finpara
  DER ← DER-1
  si IZQ <> DER y SW
     entonces
       SW ← FALSO
       para I de DER a IZQ + 1 con incremento -1 hacer
         si V(I-1) > V(I)
            entonces
              AUX ←V(I-1)
              V(I-1) ← V(I)
              V(I) ← AUX
              SW ← CIERTO
         finsi
       finpara
       IZQ ← IZQ+1
  finsi
finmientras
...
```

6.4.6. ORDENACION POR INTERCAMBIO CON INCREMENTOS DECRECIENTES. METODO SHELL

Es una mejora del método de intercambio directo en la cual se produce un acercamiento de los elementos descolocados hacia su posición correcta en saltos de mayor longitud.

Se repite un mismo proceso con una distancia de comparación que inicialmente es la mitad de la longitud del vector y que se va reduciendo a la mitad en cada repetición hasta que dicha distancia vale 1.

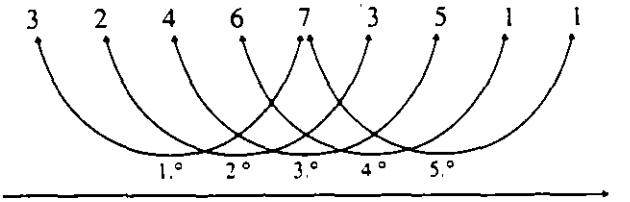
Cada pasada termina al detectarse mediante un switch la ordenación a la distancia correspondiente.

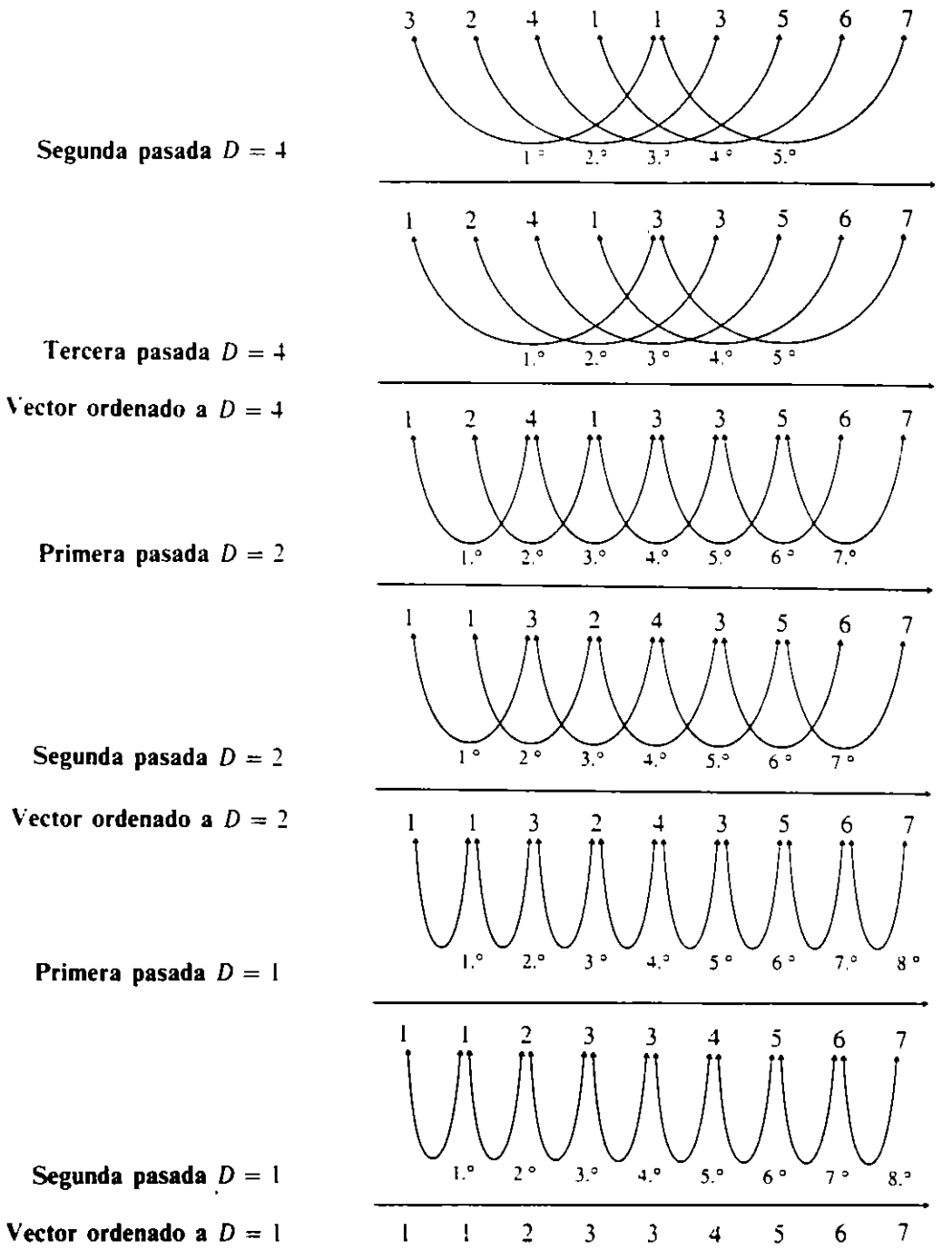
En definitiva, se trata de repetir el método de la burbuja comparando cada elemento con el situado D elementos más a la derecha, siendo D la distancia de comparación que varía entre $n\backslash 2$ y 1.

Veamos la evolución de una secuencia de números por este método:

Secuencia inicial 3 2 4 6 7 3 5 1 1

Primera pasada $D = 4$





- **Objetos:**

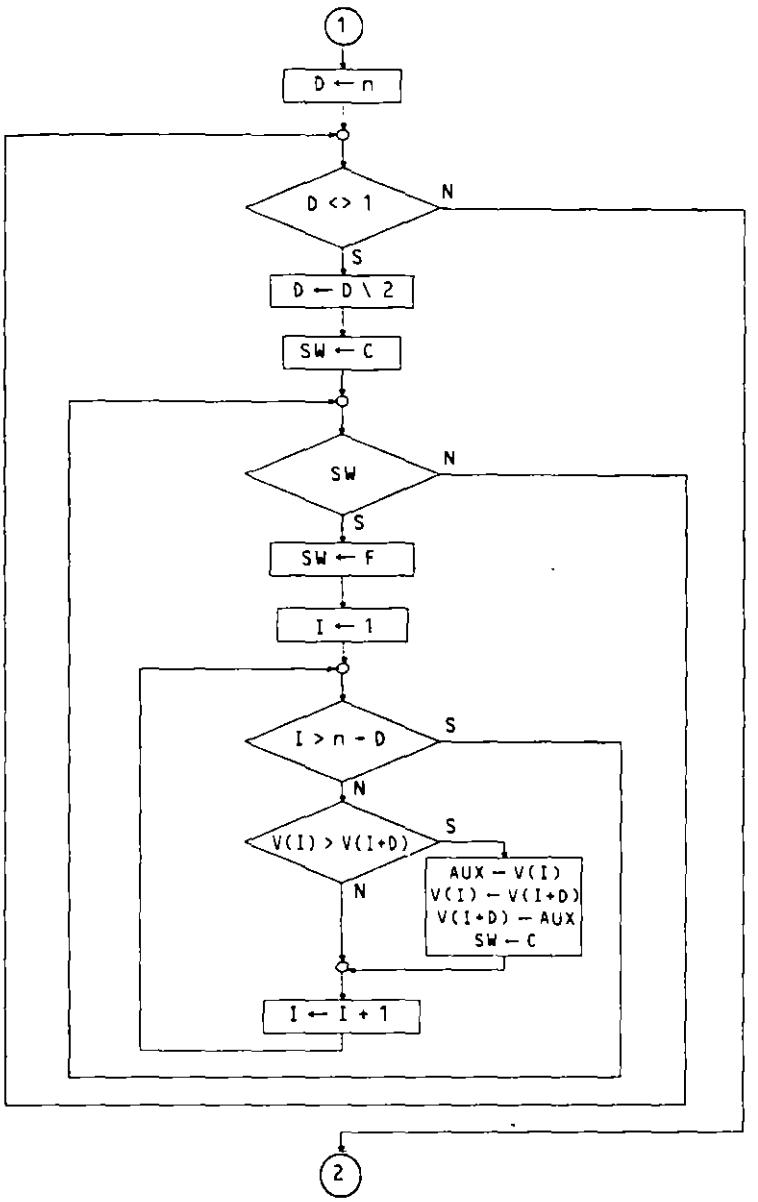
D es una variable que indica la distancia de comparación.

SW comprueba la ordenación a distancia *D*.

F ordenado a distancia *D*.

C desordenado a distancia *D*.

- **Ordinograma:**

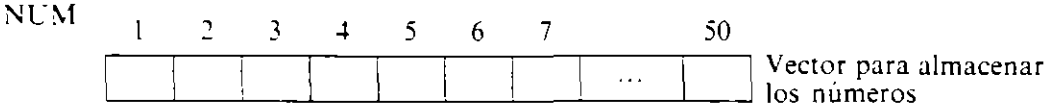


- **Pseudocódigo:**

```
...
D ← n
mientras D <> 1 hacer
  D ← D \ 2
  SW ← CIERTO
  mientras SW hacer
    SW ← FALSO
    para I de 1 a n-D hacer
      si V(I) > V(I+D)
        entonces
          AUX ← V(I)
          V(I) ← V(I+D)
          V(I+D) ← AUX
          SW ← CIERTO
      finsi
    finpara
  finmientras
finmientras
...
```

EJERCICIOS RESUELTOS

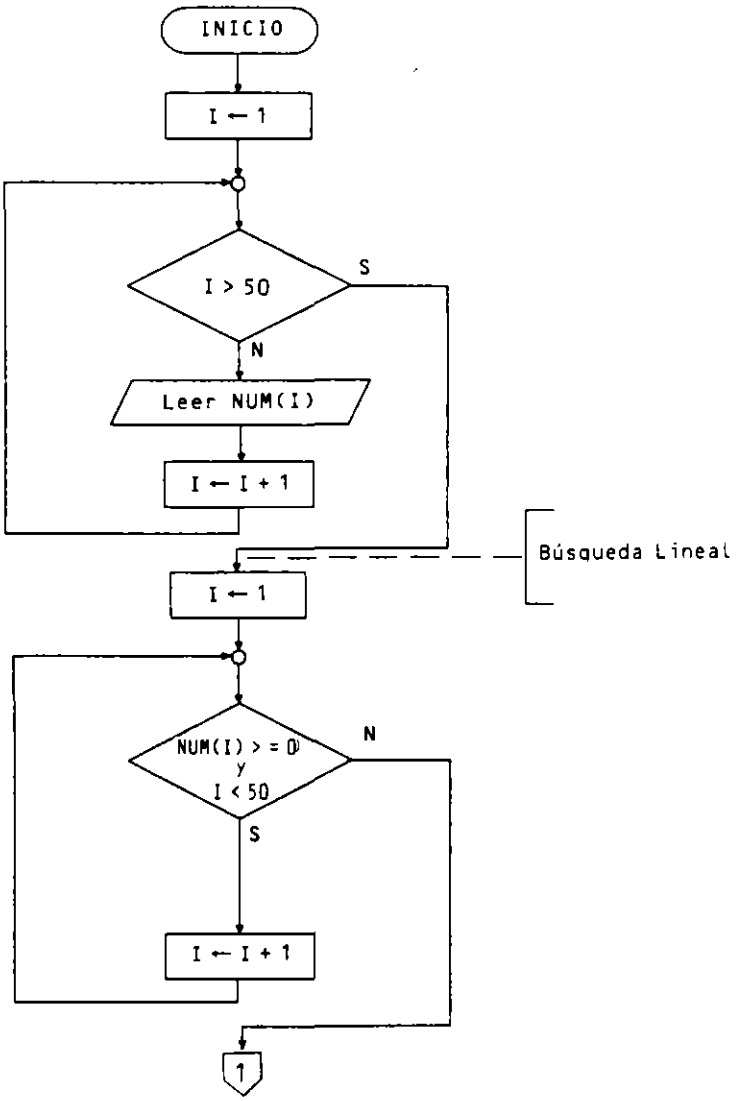
1. *Algoritmo que lee una secuencia de 50 números cargándolos en un vector y a continuación encuentra la posición que ocupa el primer número negativo en caso de existir. Si no hay números negativos se escribirá un mensaje indicándolo*

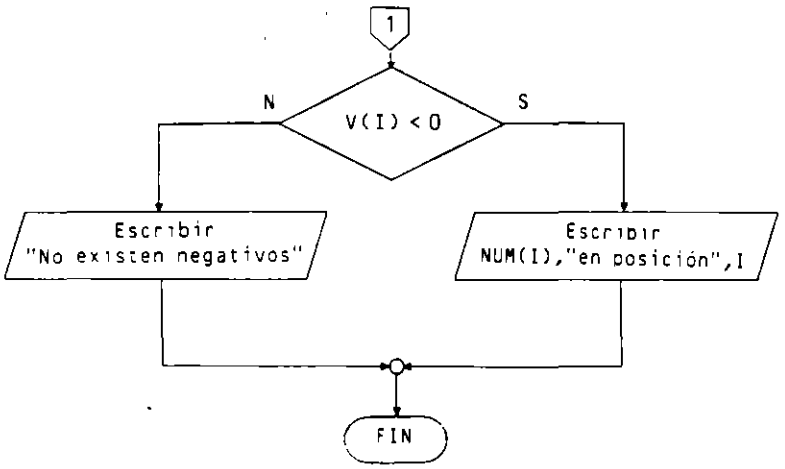


- **Objetos:**

NUM es el vector anteriormente descrito.
I es una variable para direccionar los elementos del vector.

- **Ordinograma**





- **Pseudocódigo**

```
Programa BUSCAR NEGATIVO
Entorno:
  NUM es tabla (50) numérica entera
  I es numérica entera
Algoritmo:
  ** Carga de los 50 números
  para I de 1 a 50 hacer
    escribir "Introducir elemento en posición ", I
    leer NUM(I)
  finpara
  ** Búsqueda lineal del primer número negativo
  I ← 1
  mientras NUM(I) >= 0 y I < 50 hacer
    I ← I + 1
  finmientras
  si V(I) < 0
     entonces
       escribir NUM(I), " en posición ", I
     sino
       escribir " No existen negativos "
  finsi
Finprograma
```

- **Codificación COBOL:**

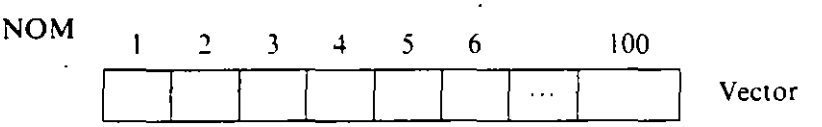
```
IDENTIFICATION DIVISION.
PROGRAM-ID. BUSCAR-NEGATIVO.
DATA DIVISION.
WORKING-STORAGE SECTION.
01   TABLA-NUM.
     05 NUM PIC S999 OCCURS 50 TIMES.
01   VARIABLE.
     05 I PIC 99.
```

```
        PROCEDURE DIVISION.
        PROCESO.
* Carga de los 50 números
     PERFORM VARYING I FROM 1 BY 1 UNTIL I > 50
       DISPLAY "INTRODUCIR ELEMENTO EN POSICION ", I
       ACCEPT NUM(I)
     END-PERFORM
* Búsqueda lineal del primer número negativo
     MOVE 1 TO I
     PERFORM UNTIL NUM(I) < 0 OR I = 50
       ADD 1 TO I
     END-PERFORM
     IF V(I) < 0
       THEN DISPLAY NUM(I), " EN POSICION ", I
       ELSE DISPLAY " NO EXISTEN NEGATIVOS "
     END-IF
     STOP RUN.
```

- **Codificación Pascal:**

```
PROGRAM BUSCAR_NEGATIVO (INPUT,OUTPUT);
VAR
  NUM : ARRAY[1..50] OF INTEGER;
    I : INTEGER;
BEGIN (*BUSCAR_NEGATIVO*)
  FOR I := 1 TO 50 DO
    BEGIN (*1*)
      WRITELN ('Introducir elemento ', I);
      READLN (NUM[I])
    END; (*1*)
  I := 1;
  WHILE (NUM[I] >= 0) AND (I < 50) DO
    I := I + 1;
  IF NUM[I] < 0 THEN
       WRITELN(NUM[I], 'en posición', I)
     ELSE
       WRITELN ('No existen negativos')
END. (*BUSCAR_NEGATIVO*)
```

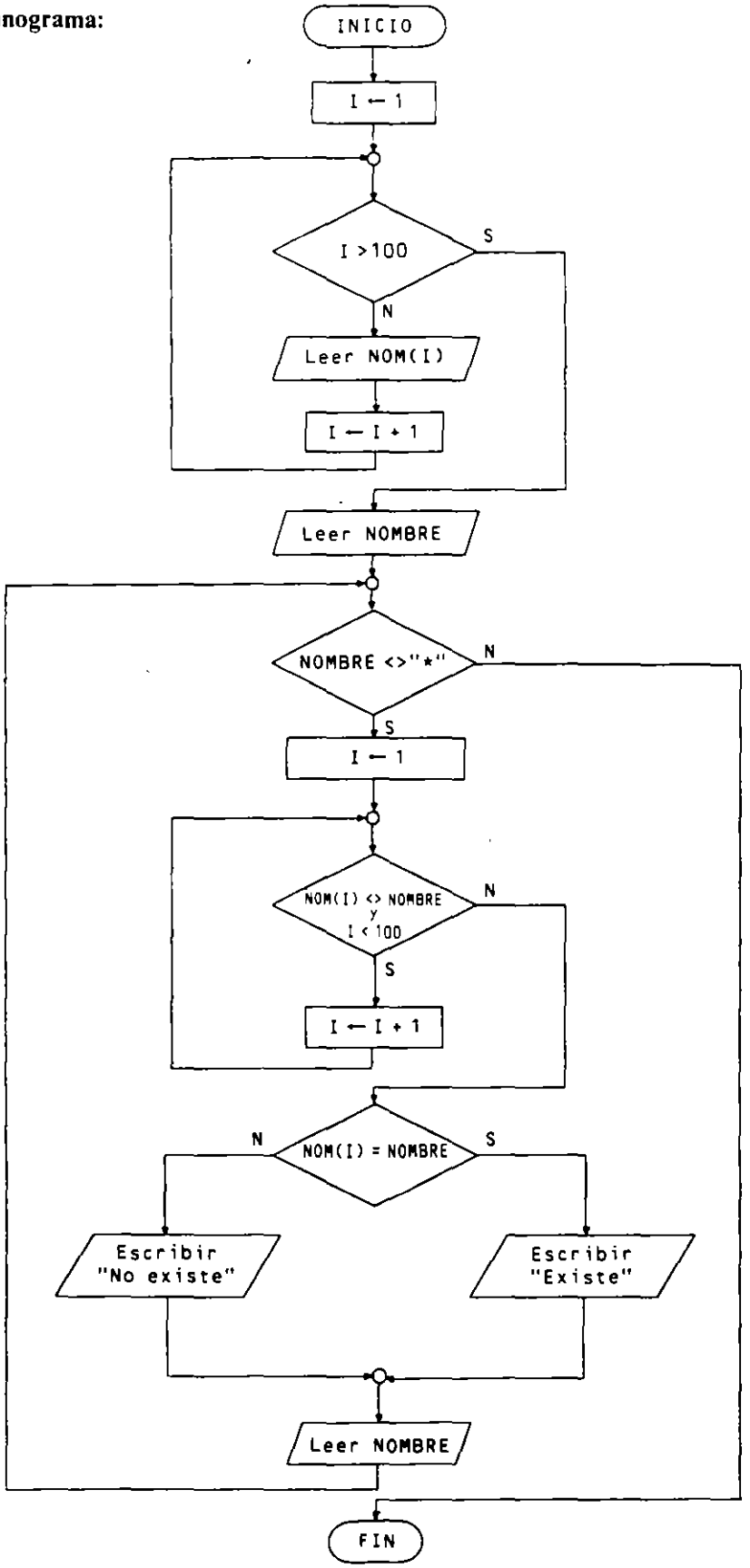
2. *Algoritmo que carga una lista de 100 nombres en un vector NOM de 100 elementos alfanuméricos y a continuación permite sucesivas consultas para comprobar si un nombre está o no en la lista. El final de las consultas se detectará al introducir un *.*



- **Objetos:**

NOM vector anterior.
I variable para direccionar los elementos del vector.
NOMBRE variable para almacenar el nombre a buscar.

- **Ordinograma:**



- **Pseudocódigo:**

```
Programa BUSCAR NOMBRES
Entorno:
  NOM es tabla (100) alfanumérica
  NOMBRE es alfanumérica
  I es numérica entera
Algoritmo:
  ** Carga del vector
  para I de 1 a 100 hacer
    escribir "Introducir nombre ", I
    leer NOM(I)
  finpara
  ** Consultas
  escribir " Introducir nombre a buscar o * "
  leer NOMBRE
  mientras NOMBRE <> "*" hacer
    ** Búsqueda lineal
    I ← 1
    mientras NOM(I) <> NOMBRE y I < 100 hacer
      I ← I + 1
    finmientras
    si NOM(I) = NOMBRE
       entonces
         escribir "existe"
       sino
         escribir "No existe"
    finsi
    escribir "Introducir otro nombre o * "
    leer NOMBRE
  finmientras
Finprograma
```

- **Codificación COBOL:**

```
IDENTIFICATION DIVISION.
PROGRAM-ID. BUSCAR-NOMBRES.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  VECTOR.
    05 NOM PIC X(30) OCCURS 100 TIMES.
01  VARIABLES.
    05 NOMBRE PIC X(30).
    05 I PIC 999.
PROCEDURE DIVISION.
PROCESO.
    PERFORM VARYING I FROM 1 BY 1 UNTIL I > 100
       DISPLAY " INTRODUCIR NOMBRE ", I
       ACCEPT NOM(I)
    END-PERFORM
    DISPLAY " INTRODUCIR NOMBRE A BUSCAR O * "
    ACCEPT NOMBRE
    PERFORM UNTIL NOMBRE = "*"
       MOVE 1 TO I
       PERFORM UNTIL NOM(I) = NOMBRE OR I = 100
          ADD 1 TO I
       END-PERFORM
       IF NOM(I) = NOMBRE THEN
          DISPLAY " EXISTE"
```

```
            ELSE
            DISPLAY " NO EXISTE"
          END-IF
          DISPLAY "INTRODUCIR OTRO NOMBRE O * "
          ACCEPT NOMBRE
        END-PERFORM
        STOP RUN.
```

- **Codificación Pascal:**

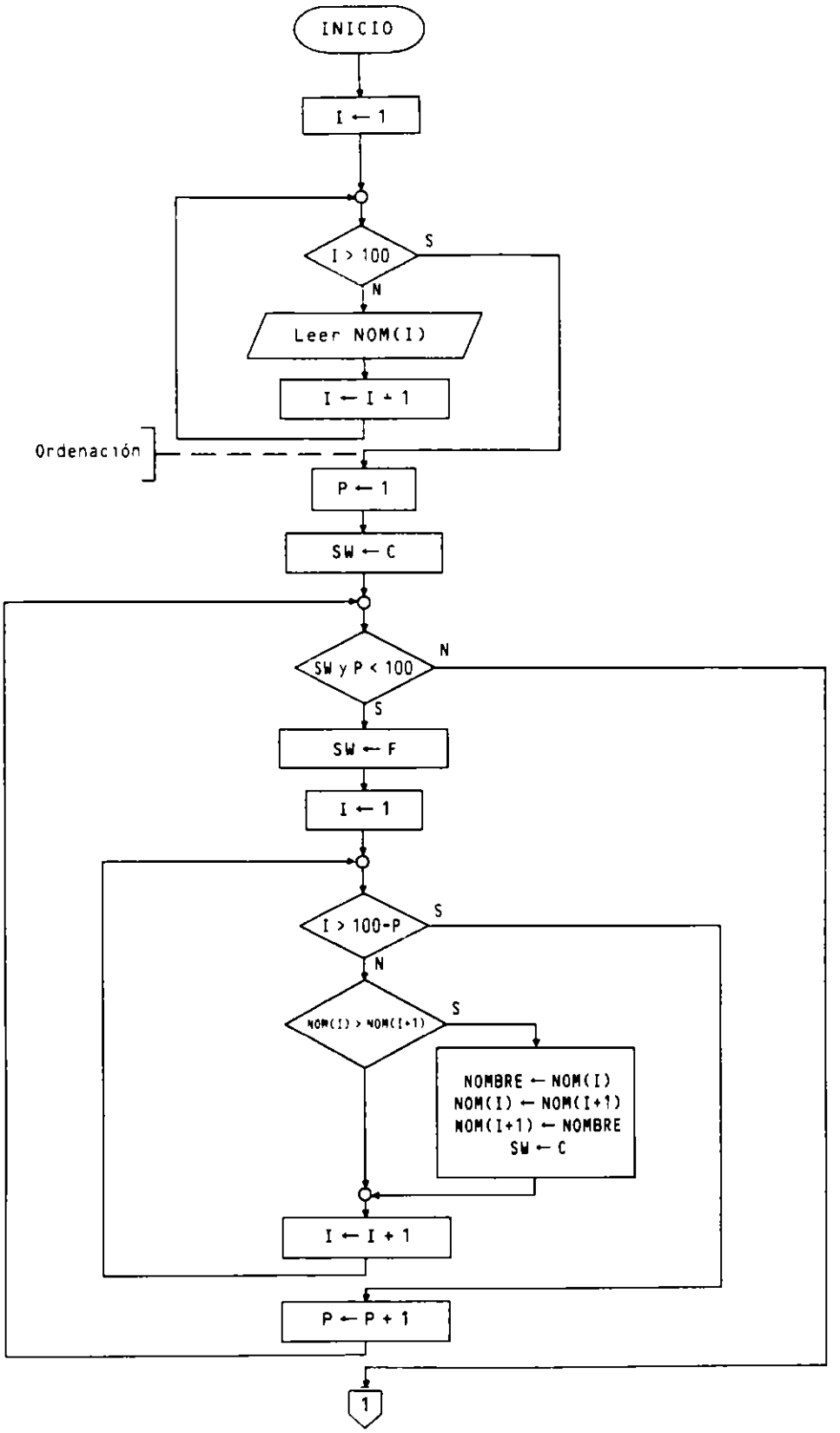
```
PROGRAM BUSCAR_NOMBRES (INPUT,OUTPUT);
VAR
  NOM : ARRAY [1..100] OF STRING[30];
  NOMBRE : STRING[30];
  I : INTEGER ;
BEGIN (* BUSCAR_NOMBRES *)
  FOR I := 1 TO 100 DO
    BEGIN (*1*)
      WRITELN ('Introducir nombre ', I);
      READLN (NOM[I])
    END; (*1*)
  WRITELN ('Introducir nombre a buscar o * ');
  READLN (NOMBRE);
  WHILE NOMBRE <> '*' DO
    BEGIN (*2*)
      I := 1;
      WHILE (NOM[I] <> NOMBRE) AND (I < 100) DO
        I := I + 1;
      IF NOM[I] = NOMBRE
        THEN
          WRITELN ('Existe')
        ELSE
          WRITELN ('No existe');
      WRITELN ('Introducir otro nombre o * ');
      READLN (NOMBRE)
    END (*2*)
END. (* BUSCAR_NOMBRES *)
```

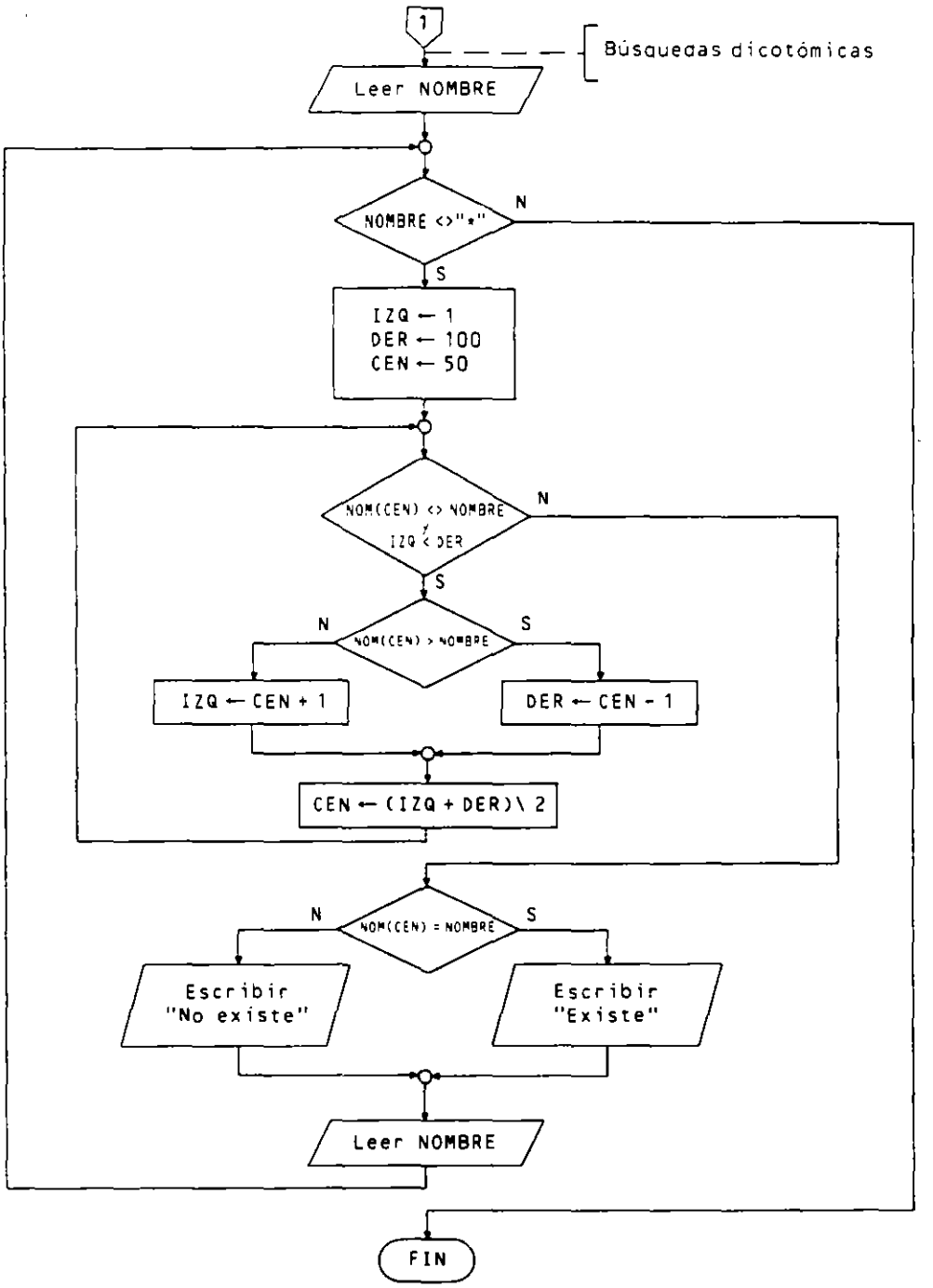
3. *Repetir el ejercicio anterior ordenando los 100 nombres ascendentemente antes de realizar las sucesivas consultas, que en este caso se harán de forma dicotómica.*

- **Objetos:** Además de los objetos del ejercicio anterior utilizaremos:

P contador de pasadas para la ordenación.
SW como test para la ordenación
  C → desordenado
  F → ordenado
Los intercambios utilizan como auxiliar NOMBRE.
IZQ, DER, CEN son los apuntadores para la búsqueda dicotómica.

- **Ordinograma:**





- **Pseudocódigo:**

```
Programa BUSCAR NOMBRES II
Entorno:
  NOM es tabla (100) alfanumérica
  NOMBRE es alfanumérica
  SW es lógica
  I, P, IZQ, DER, CEN son numéricas enteras
Algoritmo:
  para I de 1 a 100 hacer
```

```
    escribir "Introducir nombre ", I
    leer NOM(I)
  finpara
  ** Ordenación
  P ← 1
  SW ← CIERTO
  mientras SW y P < 100 hacer
    SW ← FALSO
    para I de 1 a 100 - P hacer
      si NOM(I) > NOM(I+1)
         entonces
           NOMBRE ← NOM(I)
           NOM(I) ← NOM(I+1)
           NOM(I+1) ← NOMBRE
           SW ← CIERTO
      finsi
    finpara
    P ← P + 1
  finmientras
  ** Consultas
  escribir "Introducir nombre a buscar o * "
  leer NOMBRE
  mientras NOMBRE <> "*" hacer
    IZQ ← 1
    DER ← 100
    CEN ← 50
    mientras NOM(CEN) <> NOMBRE y IZQ < DER hacer
      si NOM(CEN) > NOMBRE
         entonces
           DER ← CEN - 1
         sino
           IZQ ← CEN + 1
      finsi
      CEN ← (IZQ + DER) \ 2
    finmientras
    si NOM(CEN) = NOMBRE
       entonces
         escribir "Existe"
       sino
         escribir "No existe"
    finsi
    escribir "Introducir otro nombre o * "
    leer NOMBRE
  finmientras
Finprograma
```

- **Codificación COBOL:**

```
IDENTIFICATION DIVISION.
PROGRAM-ID. BUSCAR-NOMBRES-II.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  VECTOR.
    05 NOM PIC X(30) OCCURS 100 TIMES.
01  VARIABLES.
     05 NOMBRE PIC X(30).
     05 I      PIC 999.
     05 P      PIC 999.
     05 SW     PIC 9.
```

```
       05 IZQ    PIC 999.
       05 DER    PIC 999.
       05 CEN    PIC 999.
 PROCEDURE DIVISION.
 PROCESO.
     PERFORM VARYING I FROM 1 BY 1 UNTIL I > 100
        DISPLAY "INTRODUCIR NOMBRE ", I
        ACCEPT NOM(I)
     END-PERFORM
*  Ordenación
     MOVE 1 TO P
     MOVE 0 TO SW
     PERFORM UNTIL SW = 1 OR P = 100
        MOVE 1 TO SW
        PERFORM VARYING I FROM 1 BY 1 UNTIL I  > 100 - P
           IF NOM(I) > NOM(I + 1)
              MOVE NOM(I) TO NOMBRE
              MOVE NOM(I + 1) TO NOM(I)
              MOVE NOMBRE TO NOM(I + 1)
              MOVE 0 TO SW
           END-IF
        END-PERFORM
        ADD 1 TO P
     END-PERFORM
*  Consultas
     DISPLAY "INTRODUCIR NOMBRE A BUSCAR 0 * "
     ACCEPT NOMBRE
     PERFORM UNTIL NOMBRE = "*"
            MOVE 1 TO IZQ
            MOVE 100 TO DER
            MOVE 50 TO CEN
            PERFORM UNTIL NOM(CEN) = NOMBRE OR IZQ >= DER
               IF NOM(CEN) > NOMBRE THEN
                  SUBTRACT 1 TO CEN GIVING DER
                ELSE
                  ADD 1 TO CEN GIVING IZQ
               END-IF
               COMPUTE CEN = (IZQ + DER) / 2
            END-PERFORM
            IF NOM(CEN) = NOMBRE THEN
                DISPLAY "EXISTE"
            ELSE
                DISPLAY "NO EXISTE"
            END-IF
            DISPLAY "INTRODUCIR OTRO NOMBRE 0 * "
            ACCEPT NOMBRE
     END-PERFORM
     STOP RUN.
```

- **Codificación Pascal:**

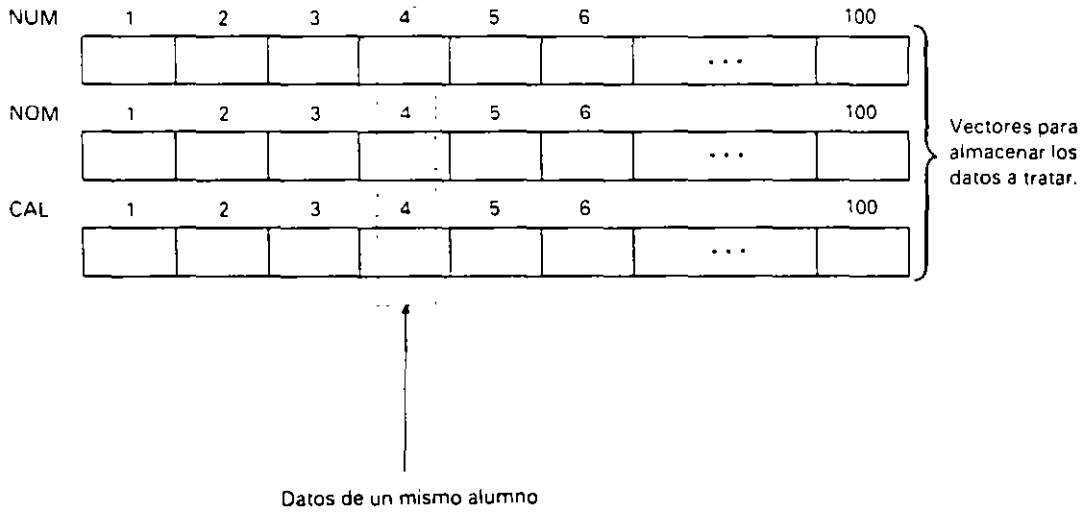
```
PROGRAM BUSCAR_NOMBRES_II (INPUT,OUTPUT);
VAR
  NOM : ARRAY [1..100] OF STRING[30];
  NOMBRE : STRING [30];
  I, P, IZQ, DER, CEN : INTEGER;
  SW : BOOLEAN;
BEGIN (* BUSCAR_NOMBRE_II *)
  FOR I := 1 TO 100 DO
```

```
    BEGIN (*1*)
      WRITELN('Introducir nombre ', I);
      READLN(NOM[I])
    END; (*1*)
  (* Ordenación *)
  P := 1;
  SW := TRUE;
  WHILE SW AND (P < 100) DO
    BEGIN (*2*)
      SW := FALSE;
      FOR I := 1 TO 100 - P DO
        IF NOM[I] > NOM[I+1]
          THEN
            BEGIN (*3*)
              NOMBRE := NOM[I];
              NOM[I] := NOM[I+1];
              NOM[I+1] := NOMBRE;
              SW := TRUE
            END; (*3*)
      P := P + 1
    END; (*2*)
  (* Consultas *)
  WRITELN('Introducir nombre a buscar o *');
  REALDN(NOMBRE);
  WHILE NOMBRE <> '*' DO
    BEGIN (*4*)
      IZQ := 1;
      DER := 100;
      CEN := 50;
      WHILE (NOM[CEN] <> NOMBRE) AND (IZQ < DER) DO
        BEGIN (*5*)
          IF NOM[CEN] > NOMBRE
            THEN
              DER := CEN - 1
            ELSE
              IZQ := CEN + 1 ;
          CEN := (IZQ + DER) DIV 2
        END; (*5*)
      IF NOM[CEN] = NOMBRE
        THEN
          WRITELN('Existe')
        ELSE
          WRITELN('No existe');
      WRITELN('Introducir otro nombre o *')
      REALDN (NOMBRE)
    END (*4*)
  END. (* BUSCAR_NOMBRES_II *)
```

4. *Algoritmo que carga en memoria los datos de 100 alumnos compuestos de número de matrícula, nombre y calificación en una asignatura. Se quieren sacar tres listados consecutivos y ordenados: el primero ascendentemente por número de matrícula, el segundo alfabéticamente por nombre (ascendente) y el tercero por nota de forma descendente. Todos los listados van precedidos por una CABECERA (que no se especifica hasta la codificación).*

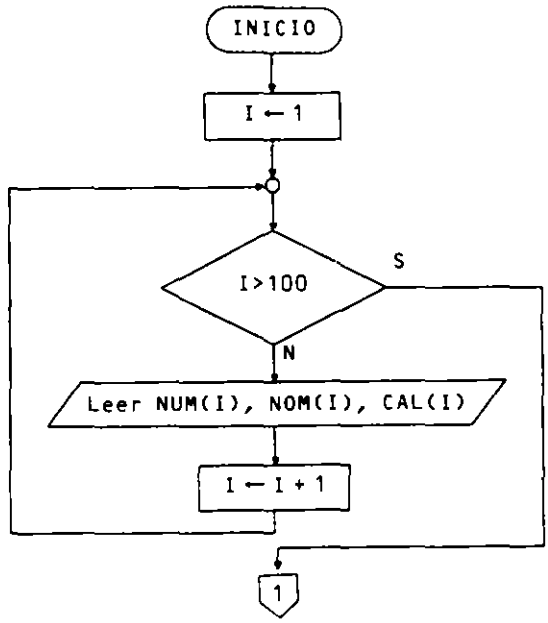
- **Objetos:**

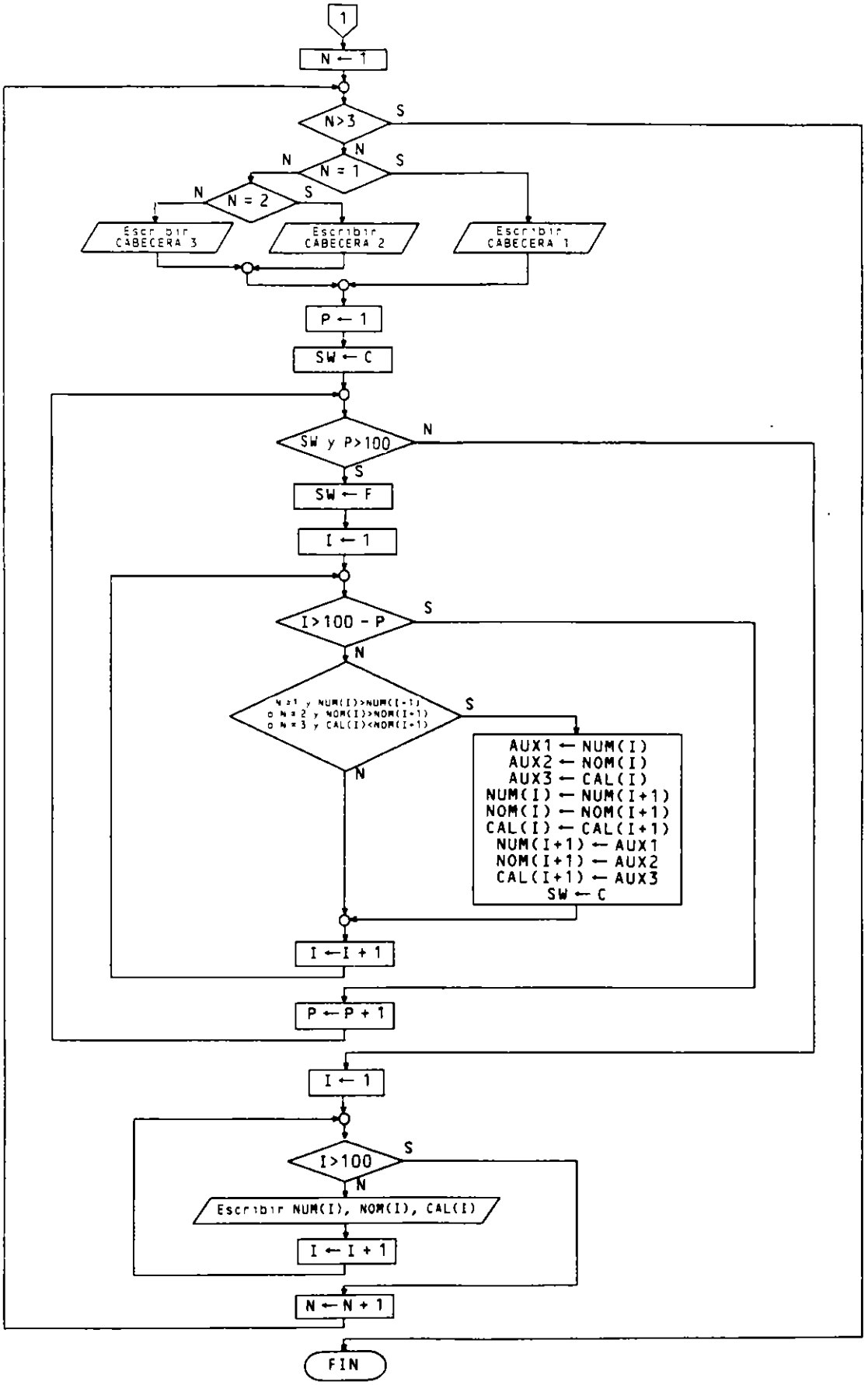
I. Variable para direccionar elementos de los vectores.
N. Variable para contar los listados.
P. Variable para contar pasadas.
SW. Switch para la ordenación.
AUX1. AUX3 Variables numéricas para intercambiar elementos de los vectores NUM y CAL.
AUX2. Variable alfanumérica para intercambiar elementos del vector NOM.



Realizamos la ordenación por el método de la burbuja con test.

- **Ordinograma:**




1
N ← 1
N>3
S
N
N = 1
S
N
N = 2
S
N
Escribir CABECERA 3
Escribir CABECERA 2
Escribir CABECERA 1
P ← 1
SW ← C
SW y P>100
N
S
SW ← F
I ← 1
I>100 - P
S
N
N=1 y NUM(I)>NUM(I-1)
o N=2 y NOM(I)>NOM(I-1)
o N=3 y CAL(I)<NOM(I-1)
S
N
AUX1 ← NUM(I)
AUX2 ← NOM(I)
AUX3 ← CAL(I)
NUM(I) ← NUM(I+1)
NOM(I) ← NOM(I+1)
CAL(I) ← CAL(I+1)
NUM(I+1) ← AUX1
NOM(I+1) ← AUX2
CAL(I+1) ← AUX3
SW ← C
I ← I + 1
P ← P + 1
I ← 1
I>100
S
N
Escribir NUM(I), NOM(I), CAL(I)
I ← I + 1
N ← N + 1
FIN

- **Pseudocódigo:**

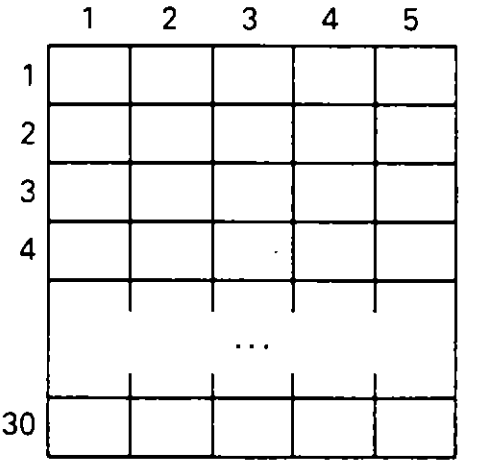
```
Programa LISTADOS
Entorno:
  NUM es tabla (100) numérica entera
  NOM es tabla (100) alfanumérica
  CAL es tabla (100) numerica real
  I, N, P, AUX1 son numericas enteras
  AUX2 es alfanumérica
  AUX3 es numérica real
  SW es lógica
Algoritmo:
  ** Lectura de datos
  para I de 1 a 100 hacer
    escribir "Número de matricula del alumno ", I
    leer NUM(I)
    escribir "Nombre"
    leer NOM(I)
    escribir "Calificación"
    leer CAL(I)
  finpara
  ** Bucle para confeccionar los listados
  para N de 1 a 3 hacer
    si N = 1
       entonces
          escribir CABECERA1
       sino
          si N = 2
             entonces
                escribir CABECERA2
             sino
                escribir CABECERA3
          finsi
    finsi
    ** Ordenación burbuja con switch
    P ← 1
    SW ← CIERTO
    mientras SW y P < 100 hacer
       SW ← FALSO
       para I de 1 a 100 - P hacer
         si  N = 1 y NUM(I) > NUM(I+1)
           o N = 2 y NOM(I) > NOM(I+1)
           o N = 3 y CAL(I) < CAL(I+1)
            entonces
              AUX1 ← NUM(I)
              AUX2 ← NOM(I)
              AUX3 ← CAL(I)
              NUM(I) ← NUM(I+1)
              NOM(I) ← NOM(I+1)
              CAL(I) ← CAL(I+1)
              NUM(I+1) ← AUX1
              NOM(I+1) ← AUX2
              CAL(I+1) ← AUX3
              SW ← CIERTO
         finsi
       finpara
       P ← P + 1
    finmientras
    para I de 1 a 100 hacer
      escribir NUM(I), NOM(I), CAL(I)
```

```
        finpara
      finpara
    Finprograma
```

5. *Algoritmo que carga en memoria las notas de los 30 alumnos de un grupo en sus cinco asignaturas. Los datos se almacenan en una matriz NOTAS de 30 filas (una por cada alumno) y cinco columnas (una por cada asignatura). Los nombres de los alumnos se almacenan en un vector alfanumérico NOM de 30 elementos (uno por cada alumno).*

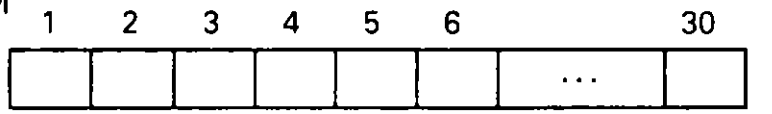
Se quiere sacar un listado de notas de tal forma que aparezcan ordenados descendentemente por la primera asignatura (columna 1).

NOTAS



matriz para almacenar las notas

NOM



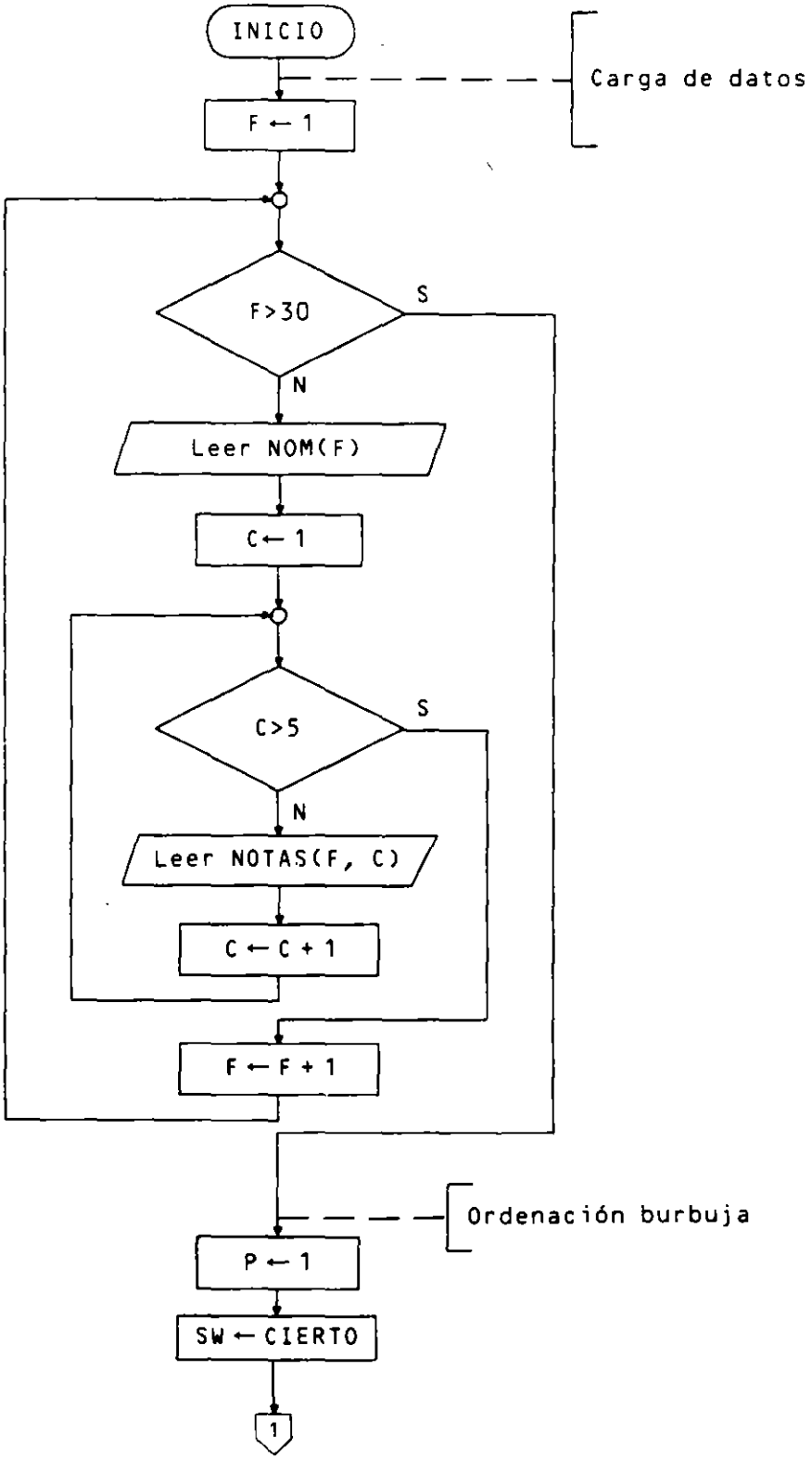
vector para los nombres

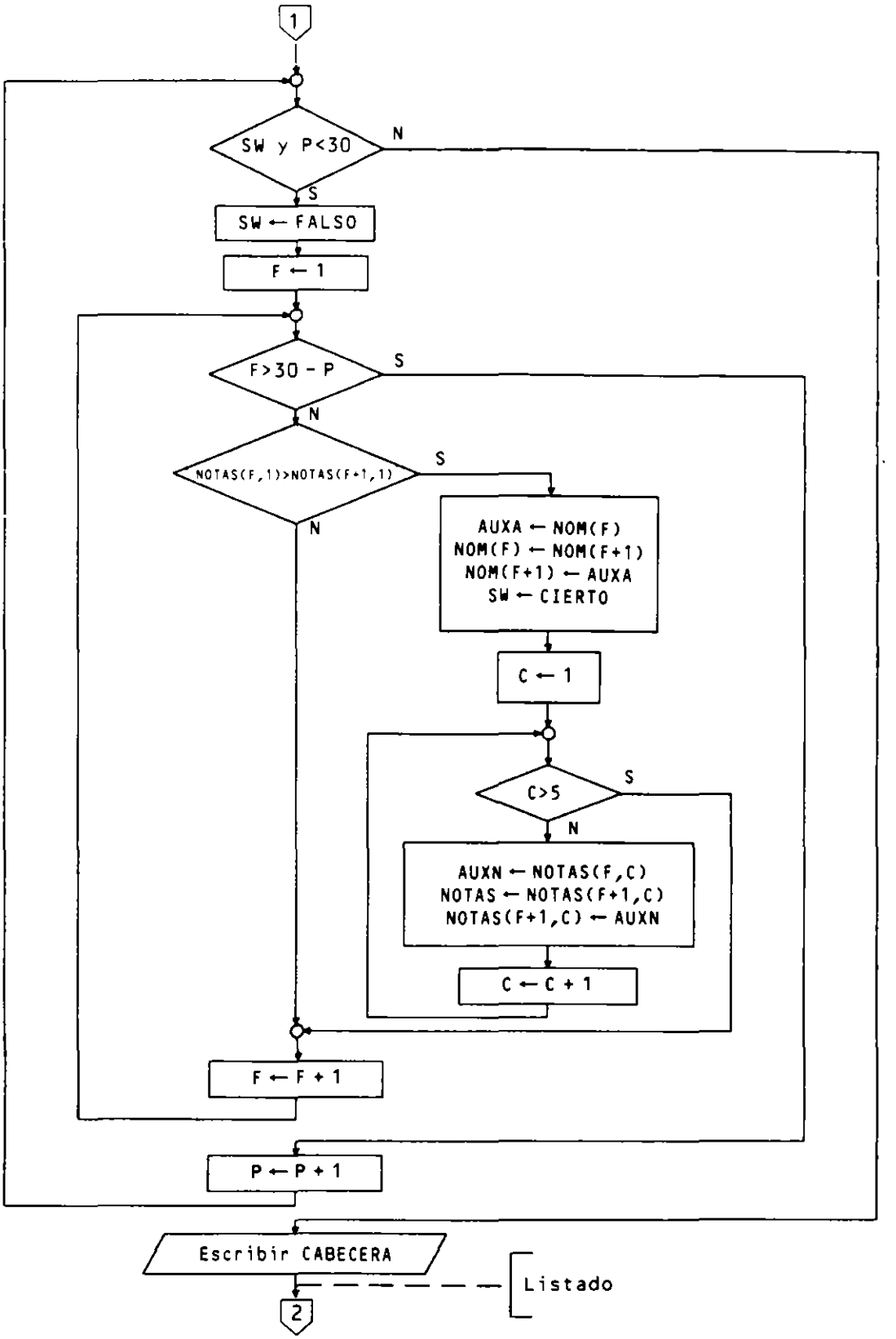
- **Objetos:**

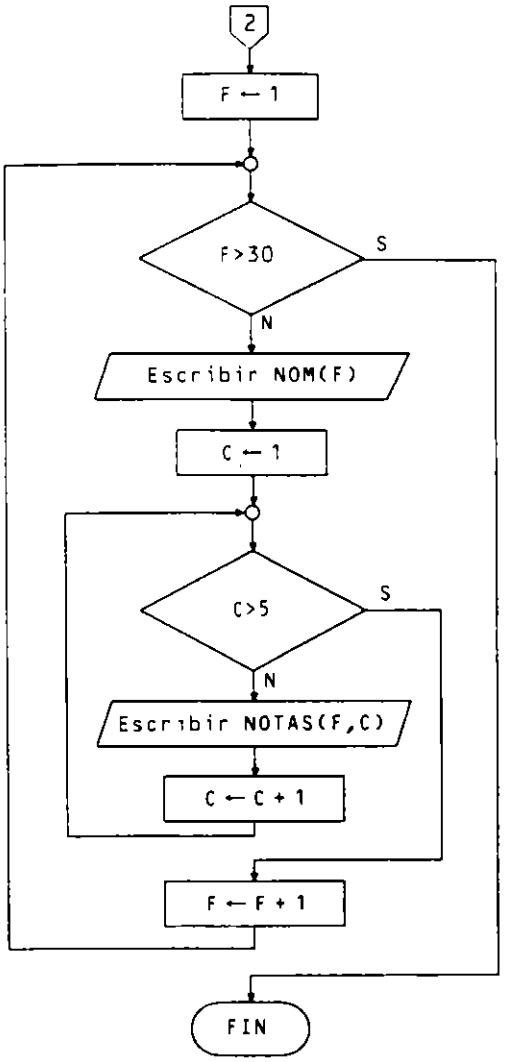
F y C son variables para direccionar filas y columnas (alumnos y asignaturas).
P es una variable para contar pasadas.
SW en un switch para la ordenación.

AUXN } auxiliares
AUXA } para los intercambios

- **Ordinograma:**




1
SW y P<30
N
S
SW ← FALSO
F ← 1
F>30 - P
S
N
NOTAS(F,1)>NOTAS(F+1,1)
S
N
AUXA ← NOM(F)
NOM(F) ← NOM(F+1)
NOM(F+1) ← AUXA
SW ← CIERTO
C ← 1
C>5
S
N
AUXN ← NOTAS(F,C)
NOTAS ← NOTAS(F+1,C)
NOTAS(F+1,C) ← AUXN
C ← C + 1
F ← F + 1
P ← P + 1
Escribir CABECERA
Listado
2



- **Pseudocódigo:**

```
Programa LISTADO ORDENADO
Entorno:
  NOTAS es tabla (30,5) numérica real
  NOM es tabla (30) alfanumérica
  F,C,P son numéricas enteras
  AUXN es numérica real
  AUXA es alfanumérica
  SW es lógica
Algoritmo:
  ** Carga de datos
  para F de 1 a 30 hacer
     escribir "Nombre del alumno ", F
     leer NOM(F)
     para C de 1 a 5 hacer
        escribir "Nota del alumno ", F, "en la asignatura ", C
        leer NOTAS(F,C)
     finpara
  finpara
```

```
    ** Ordenación burbuja
    P ← 1
    SW ← CIERTO
    mientras SW y P < 30 hacer
       SW ← FALSO
       para F de 1 a 30 - P hacer
       si NOTAS(F,1) > NOTAS(F+1,1)
         entonces
           AUXA ← NOM(F)
           NOM(F) ← NOM(F+1)
           NOM(F+1) ← AUXA
           SW ← CIERTO
           para C de 1 a 5 hacer
              AUXN ← NOTAS(F,C)
              NOTAS(F,C) ← NOTAS(F+1,C)
              NOTAS(F+1,C) ← AUXN
           finpara
       finsi
       P ← P + 1
    finmientras
    ** Listado
    escribir CABECERA
    para F de 1 a 30 hacer
      escribir NOM(F)
      para C de 1 a 5 hacer
        escribir NOTAS(F,C)
      finpara
    finpara
  Finprograma
```

EJERCICIOS PROPUESTOS

1. Programa que carga dos vectores de 35 componentes, almacenando en el primero una lista de nombres de personas y en el segundo las edades correspondientes a cada una de ellas. A continuación permite consultas sucesivas de edades para nombres introducidos por teclado.

2. Programa que carga una matriz alfanumérica de 80 filas y 2 columnas conteniendo en cada fila un nombre de persona y su número de teléfono. A continuación realiza una clasificación ascendente por orden alfabético de nombres, y, finalmente, imprime la lista ordenada de nombres y teléfonos.

3. Programa que permite sucesivas consultas en la tabla ordenada del ejercicio anterior.

4. Programa que carga un vector numérico de 100 componentes y obtiene e imprime los 10 valores menores y los 10 mayores.

5. Programa que realiza una ordenación de un vector numérico imprimiendo todos los estados intermedios del mismo.

6. Programa que realiza la clasificación completa de una matriz numérica en orden creciente (de izquierda a derecha y de arriba a abajo).

7. Programa que clasifica simultáneamente dos vectores numéricos de igual dimensión, el primero en orden creciente y el segundo en orden decreciente.

8. Programa que carga una matriz de 100 filas y tres columnas, con primer apellido, segundo apellido y nombre de 100 personas, realizando una clasificación alfabética completa e imprimiendo la lista de nombres clasificada.

9. Programa que lee una frase y averigua si es «palíndroma». Una frase es palíndroma si se lee igual de izquierda a derecha que de derecha a izquierda (sin considerar los espacios en blanco).

   • **Ejemplos:**

   «LE SACO SUS OCAS EL».
   «DABALE ARROZ A LA ZORRA EL ABAD».

10. Dado un vector numérico de 100 componentes. Programa que clasifica simultáneamente en orden creciente sus componentes pares y en orden decreciente las impares.

CAPITULO 7

Diseño descendente: Subprogramas

7.1. INTRODUCCION

Los problemas reales que se plantean a un departamento de informática requieren programas de una cierta complejidad y a veces de gran tamaño.

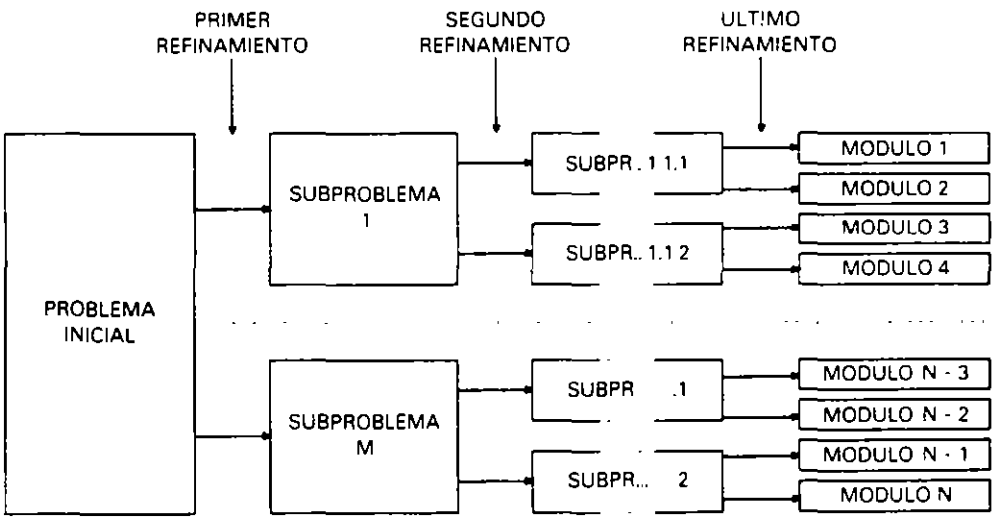
Abordar el diseño de un programa de estas características de una forma directa es una tarea, en la mayoría de los casos, bastante difícil.

Lo más adecuado es descomponer el problema, ya desde su fase de análisis, en partes cuya resolución sea más asequible. La programación de cada una de estas partes se realiza independientemente de las otras, incluso, en ocasiones, por diferentes personas.

De esta forma se pueden resolver problemas extremadamente complejos.

Por otro lado, la depuración y puesta a punto del programa hace necesario que el listado del mismo sea fácilmente comprensible. En este sentido conviene subdividir el programa de tal manera que cada parte sea suficientemente reducida y sencilla para su desarrollo y mantenimiento.

El **diseño descendente** o diseño **top-down** consiste en una serie de descomposiciones sucesivas del problema inicial, que describen el refinamiento progresivo del repertorio de instrucciones que van a formar parte del programa.

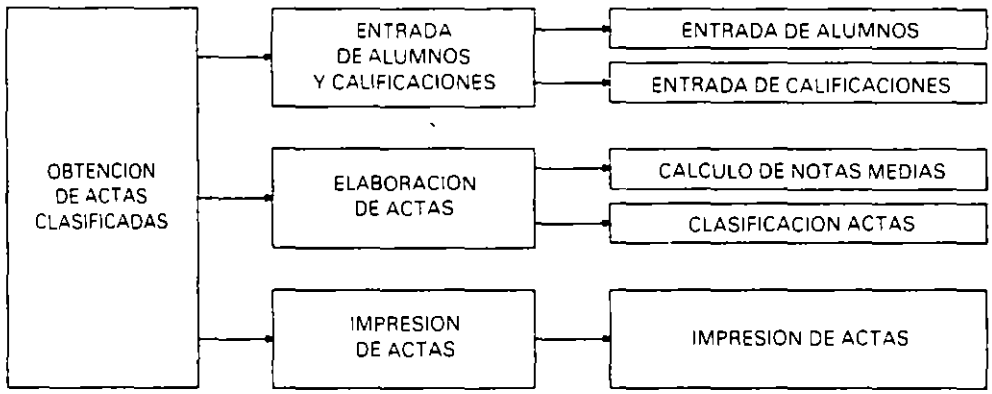


La utilización de esta técnica de diseño tiene los siguientes objetivos básicos:

— Simplificación del problema y de los subprogramas resultantes de cada descomposición.

— Las diferentes partes del problema pueden ser programadas de modo independiente e incluso por diferentes personas.
— El programa final queda estructurado en forma de bloques o módulos, lo que hace más sencilla su lectura y mantenimiento.

Ejemplo: *Obtención de las actas de evaluación final, ordenadas alfabéticamente, de los alumnos de un centro docente, a partir de un archivo de alumnos, un archivo de asignaturas y un archivo de calificaciones.*



Con la utilización de esta técnica de diseño surgen los conceptos de:

— **Programa principal y subprogramas.**
— **Declaración y llamada de subprograma.**
— **Subprogramas internos y externos.**
— **Objetos globales y locales.**
— **Parámetros o variables de enlace.**
— **Recursividad.**

7.2. PROGRAMA PRINCIPAL Y SUBPROGRAMAS

Un programa diseñado mediante esta técnica quedará constituido por dos partes claramente diferenciadas:

- **Programa principal**

Describe la solución completa del problema y consta principalmente de **llamadas a subprogramas**. Estas llamadas son indicaciones al procesador de que debe continuar la ejecución del programa en el subprograma llamado, regresando al punto de partida una vez lo haya concluido.

El programa principal puede contener, además, instrucciones primitivas y sentencias de control, que son ejecutables de modo inmediato por el procesador.

Un programa principal contendrá pocas líneas, y en él se verán claramente los diferentes pasos del proceso que se ha de seguir para la obtención de los resultados deseados.

- **Subprogramas**

A éstos se les suele denominar **declaración de subprogramas.** Figuran agrupados en distinto lugar al del programa principal.

Su estructura coincide básicamente con la de un programa, con alguna diferencia en el encabezamiento y finalización. En consecuencia, un subprograma puede tener sus propios subprogramas correspondientes a un refinamiento del mismo.

La función de un subprograma es resolver de modo independiente una parte del problema. Es importante que realice una función concreta en el contexto del problema; no obstante, a veces se convierte en subprograma un conjunto de instrucciones, cuando éstas se tendrían que repetir varias veces en diferentes lugares del programa. De esta manera, el conjunto de instrucciones a repetir aparece una sola vez en el listado del programa.

Un subprograma es ejecutado por el procesador sólo cuando es llamado por el programa principal o por otro subprograma.

7.3. SUBPROGRAMAS INTERNOS

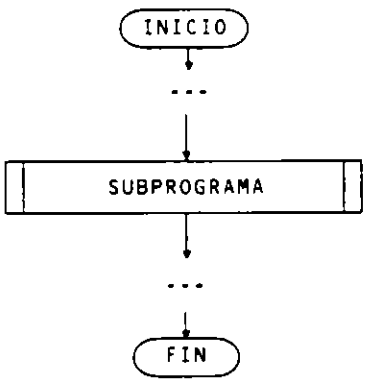
Son **subprogramas internos** los que figuran junto con el programa principal (en el mismo listado).

Se denominan de diferentes maneras en los distintos lenguajes de programación:

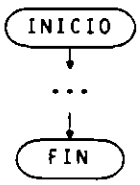
— Procedimientos en **COBOL**, activados mediante sentencia PERFORM (fuera de línea).
— Procedimientos y funciones en **Pascal**, invocados por medio de su nombre.

Su representación es la siguiente:

- **Ordinograma:**



Subprograma



- **Pseudocódigo:**

```
Programa PRINCIPAL
Entorno:
  ...
Algoritmo:
  ...
  SUBP
  ...
Finprograma
Subprograma SUBP
Entorno:
  ...
Algoritmo:
  ...
Finsubprograma
```

- **Codificación COBOL:**

```
IDENTIFICATION DIVISION.
PROGRAMA-ID. PRINCIPAL.
...
PROCEDURE DIVISION.
PROCESO.
    ...
    PERFORM SUBPROGRAMA
    ...
    STOP RUN.
SUBPROGRAMA.
    ...
```

- **Codificación Pascal:**

```
PROGRAM PRINCIPAL (INPUT, OUTPUT);
    ...
    PROCEDURE SUBPROGRAMA;
       ...
    BEGIN (* SUBPROGRAMA *)
    ...
    END; (* SUBPROGRAMA *)
    ...
BEGIN (* PRINCIPAL *)
    ...
    SUBPROGRAMA;
    ...
END. (* PRINCIPAL *)
```

7.4. SUBPROGRAMAS EXTERNOS

Son aquellos que figuran físicamente separados del programa principal, es decir, en distintos archivos fuente.

Pueden ser compilados separadamente, e incluso pueden haber sido codificados en un lenguaje de programación distinto al del programa principal.

Generalmente se enlazan con el programa principal en la fase de montaje (*linkage*), cuando ya son módulos objeto, es decir, traducidos a lenguaje máquina.

La representación y manejo de subprogramas externos varia mucho dependiendo del lenguaje de programación utilizado e incluso de las diferentes versiones de compiladores existentes.

En **ordinograma** y **pseudocódigo** se utilizará igual representación que para los subprogramas internos.

- **Codificación COBOL:**

Un subprograma externo tiene prácticamente la misma estructura de un programa principal. Las mínimas diferencias entre uno y otro corresponden a la forma de declarar los parámetros o variables que se van a utilizar para comunicar datos y resultados, como se verá en el apartado de variables de enlace. Otra diferencia significativa consiste en que la instrucción que se emplea para terminar la ejecución del programa principal, **STOP RUN**, se sustituye por **EXIT PROGRAM** en los subprogramas externos.

La instrucción para llamar a un subprograma externo tiene la siguiente sintaxis:

```
CALL "SUBPEX"
```

Siendo «SUBPEX» el nombre del subprograma externo, que ha sido escrito en COBOL.

Si se utilizan variables de enlace, éstas se indican en la llamada y en la cabecera de la división de procedimientos del subprograma mediante la cláusula USING:

```
CALL "SUBPEX" USING parámetros
...
PROCEDURE DIVISION USING parámetros.
```

Ejemplo:

```
* Archivo PRINCI.CBL que contiene el programa principal
 IDENTIFICATION DIVISION.
 PROGRAM-ID. PRINCIPAL.
 ...
 PROCEDURE DIVISION.
 PROCESO.
     ...
     CALL "SUBPEX" USING parámetros
     ...
     STOP RUN.

* Archivo SUBPEX.CBL que contiene un subprograma externo
 IDENTIFICATION DIVISION.
 PROGRAM-ID. SUBPEX.
 ...
 PROCEDURE DIVISION USING parámetros.
 PROCESO.
     ...
     EXIT PROGRAM.
```

- **Codificación Pascal:**

En Pascal estándar no se incluye el uso de subprogramas externos. Los distintos compiladores, no obstante, lo han implementado de distintas maneras. Por ejemplo, el compilador CDC Pascal, diseñado para grandes equipos, permite que un programa llame a subprogramas externos (procedimientos o funciones) que estén en una biblioteca precompilada, declarándolos en el programa principal mediante su cabecera seguida de la palabra **extern**. El compilador UCSD Pascal, diseñado por la Universidad de San Diego en California y el Turbo Pascal de Borland International Inc. lo han implementado de igual manera: el subprograma o subprogramas han de incluirse en uno o varios módulos denominados **unit**, y estos módulos se conectan al programa principal mediante la directiva **uses**, pudiendo entonces utilizar los procedimientos y funciones declarados en las units de la misma forma que si fueran internos.

Ejemplo:

```
(* Archivo PRINCI.PAS que contiene el programa principal *)
PROGRAM PRINCIPAL (INPUT, OUTPUT);
USES SUBPEX;
...
BEGIN (* PRINCIPAL *)
  ...
  SUBP (parámetros); (* Llamada a un procedimiento de SUBPEX *)
  ...
END.
```

```
(* Archivo SUBPEX.PAS que contiene la unidad SUBPEX, dentro de la cual se
   ha declarado el procedimiento SUBP *)
UNIT SUBPEX;
INTERFACE
  PROCEDURE SUBP (parámetros); (* cabecera del procedimiento *)
IMPLEMENTATION
  PROCEDURE SUBP (parámetros); (* declaración completa *)
  ...
  BEGIN (* SUBP *)
    ...
  END; (* SUBP *)
END. (* SUBPEX *)
```

7.5. OBJETOS GLOBALES Y LOCALES

Los diferentes objetos que manipula un programa (constantes, variables, tablas, archivos, subprogramas, etc.) se clasifican según su ámbito, es decir, según la porción de programa y/o subprogramas en que son conocidos y, por tanto, pueden ser utilizados.

Son **objetos globales** los declarados en el programa principal, cuyo ámbito se extiende al mismo y a todos sus subprogramas.

Son **objetos locales** a un subprograma los declarados en dicho subprograma, cuyo ámbito está restringido a él mismo y a los subprogramas declarados en él.

De los lenguajes estudiados sólo incluye esta clasificación el **Pascal** para los procedimientos y funciones. Para el lenguaje **COBOL**, todos los objetos son globales respecto a los posibles subprogramas internos o párrafos, y todos los objetos son locales respecto a los subprogramas externos.

Ejemplo: *Declaración de variables globales y locales en lenguaje Pascal y su ámbito de utilización.*

```
PROGRAM PRINCIPAL (INPUT, OUTPUT);
  VAR A,B ...
  PROCEDURE SUBP1;
    VAR C,D ...
    PROCEDURE SUBP11;
      VAR E,F ...
      BEGIN (* SUBP11 *)
      ...
      END; (* SUBP11 *)
    PROCEDURE SUBP12;
      VAR G,H ...
      BEGIN (* SUBP12 *)
      ...
      END; (* SUBP12 *)
    BEGIN (* SUBP1 *)
    ...
    END; (* SUBP1 *)
  PROCEDURE SUBP2;
    VAR I,J ...
    BEGIN (* SUBP2 *)
    ...
    END; (* SUBP2 *)
  BEGIN (* PRINCIPAL *)
    ...
  END. (* PRINCIPAL *)
```

Variables	Ambito
A,B (globales)	PRINCIPAL, SUBP1, SUBP11, SUBP12, SUBP2
C,D (locales a SUBP1)	SUBP1, SUBP11, SUBP12
E,F (locales a SUBP11)	SUBP11
G,H (locales a SUBP12)	SUBP12
I,J (locales a SUBP2)	SUBP2

7.6. VARIABLES DE ENLACE (PARAMETROS)

Todo programa utiliza unos datos de entrada y produce unos resultados, los primeros provienen de las unidades de entrada y los segundos son enviados a las unidades de salida, siendo ambas clases de unidades dispositivos externos.

Aunque un subprograma también puede realizar operaciones de entrada y salida con las unidades periféricas cuando sea necesario, en general sus datos de entrada y sus resultados provienen y son enviados del y al programa o subprograma llamante, respectivamente.

Para esta labor se utilizan las **variables de enlace o parámetros**. Es decir, cada vez que se realiza una llamada a un subprograma, los datos de entrada le son pasados por medio de determinadas variables, y, análogamente, cuando termina la ejecución del subprograma, los resultados regresan mediante otras o mediante las mismas variables.

Los parámetros pueden ser de dos tipos:

- **Parámetros formales:** Variables locales de un subprograma utilizadas para la recepción y el envío de los datos.
- **Parámetros actuales:** Variables y datos enviados, en cada llamada de subprograma, por el programa o subprograma llamante.

Los parámetros formales son siempre fijos para cada subprograma, mientras que los parámetros actuales pueden ser cambiados para cada llamada. En cualquier caso ha de haber una correspondencia entre los parámetros formales y actuales en su número, colocación y tipo.

Su representación es la siguiente:

- **Pseudocódigo:**

Los parámetros formales se declaran a continuación de la cabecera del subprograma, antes del «entorno» del mismo. Los parámetros actuales figurarán junto a la llamada de subprograma entre paréntesis y separados por comas.

```
Programa PRINCIPAL
Entorno:
  ** variables globales
Algoritmo:
  ** instrucciones
  ...
  SUBP (parámetros-actuales)
  ...
Finprograma

Subprograma SUBP
  parámetros-formales
Entorno:
  ** variables locales
Algoritmo:
  ** instrucciones
Finsubprograma
```

- **Codificación COBOL:**

Solamente utilizan parámetros con los subprogramas externos. Los formales se declaran en el subprograma dentro de la «LINKAGE SECTION», con nivel 01 o 77, y se nombran en la cabecera de la «PROCEDURE DIVISION» precedidos de la palabra «USING». Los actuales figuran en el programa principal junto a la llamada del subprograma (instrucción CALL) precedidos asimismo de «USING».

```
* Archivo del programa principal
 IDENTIFICATION DIVISION.
 PROGRAM-ID. PRINCIPAL.
 ...
 DATA DIVISION.
 WORKING-STORAGE SECTION.
* declaración de variables del programa principal
 PROCEDURE DIVISION.
 PROCESO.
* instrucciones
      ...
      CALL "SUBPEX" USING parámetros-actuales
      ...
      STOP RUN.
* procedimientos
 ...

* Archivo del subprograma externo
 IDENTIFICATION DIVISION.
 PROGRAM-ID. SUBPEX.
 ...
 DATA DIVISION.
 WORKING-STORAGE SECTION.
* declaración de variables del subprograma
 LINKAGE SECTION.
* declaración de parámetros formales
 PROCEDURE DIVISION USING parámetros-formales
 PROCESO.
* instrucciones
      ...
      EXIT PROGRAM.
* procedimientos
 ...
```

- **Codificación Pascal:**

La declaración de los parámetros formales se realiza en la cabecera del subprograma a continuación del nombre del mismo y entre paréntesis. Los parámetros actuales figuran en la llamada del subprograma entre paréntesis y separados por comas:

```
PROGRAM PRINCIPAL (INPUT, OUTPUT);
  ...
  (* declaración de variables globales *)
  ...
  PROCEDURE SUBPROGRAMA (parámetros-formales);
     ...
     (* declaración de variables locales *)
     ...
     BEGIN (* SUBPROGRAMA *)
     ...
```

```
    END;  (* SUBPROGRAMA *)
    ...
BEGIN (* PRINCIPAL *)
  ...
  SUBPROGRAMA (parámetros-actuales);
  ...
END. (* PRINCIPAL *)
```

7.7. PASO DE PARAMETROS

El proceso de emisión y recepción de datos y resultados mediante variables de enlace se denomina **paso de parámetros**.

El paso de parámetros puede realizarse de dos maneras diferentes:

- **Paso por valor:** Para suministrar datos de entrada al subprograma.
- **Paso por referencia:** Para entrada y salida o sólo salida.

Un parámetro actual pasado por valor es un dato, o una variable global que contiene un dato, de entrada para el subprograma. Esta variable no puede ser modificada por el subprograma, que copia su valor en el parámetro formal correspondiente para poder utilizarlo.

Un parámetro actual pasado por referencia es una variable del programa o subprograma llamante, que puede contener o no un dato para el subprograma llamado, el cual coloca un resultado en esa variable, que queda a disposición del llamante una vez concluida la ejecución del subprograma.

Los parámetros formales se comportan como variables locales, con la particularidad de que en cada llamada del subprograma se identifican con los parámetros actuales, según su colocación: esto es, cada parámetro actual, si es pasado por valor, se copia en el parámetro formal correspondiente, y, si es pasado por referencia, proporciona su dirección de memoria al parámetro formal asociado.

Es de destacar el hecho de que, desde el punto de vista físico, en el paso por valor no se proporciona la variable al subprograma, sino solamente su contenido, evitando así su modificación, y en el paso por referencia se proporciona la dirección o referencia de la variable, con lo que el subprograma la utiliza como propia, modificándola si es necesario, para dejar en ella los resultados que ha de devolver.

La utilización de parámetros por referencia supone ahorro de memoria, puesto que la variable local correspondiente no existe físicamente, sino que se asocia a la global en cada llamada. También supone el riesgo de modificar por error una variable global sin desearlo.

Ejemplo: *Programa que calcula la longitud de una serie de circunferencias a partir de la longitud de su radio. El programa termina cuando se introduce el valor 0.*

- **Pseudocódigo:**

En la declaración de los parámetros formales se antepone el prefijo «**recibe**» a los parámetros por valor y el prefijo «**transforma**» a los parámetros por referencia.

```
Programa CIRCUNFERENCIAS
Entorno:
  RADIO, LONGITUD son numéricas reales.
Algoritmo:
  escribir "Cálculo de longitudes de circunferencias."
  iterar
     escribir "Escriba la longitud del radio o 0 para terminar:"
     leer RADIO
```

```
      salir si RADIO = 0
      CALCULAR-LONGITUD (RADIO, LONGITUD)
      escribir "Longitud = ", LONGITUD
  finiterar
Finprograma
Subprograma CALCULAR-LONGITUD
  recibe     R numérica
  transforma L numérica
Entorno:
  PI es constante numérica con valor 3.141592
Algoritmo:
  L ← 2 * PI * R
Finsubprograma
```

- **Codificación COBOL:**

En la instrucción CALL de llamada de subprograma, y dentro de la cláusula USING, se preceden los parámetros por valor con la opción **BY CONTENT**, y los parámetros por referencia con la opción **BY REFERENCE** o sin nada.

```
* Archivo del programa principal
 IDENTIFICATION DIVISION.
 PROGRAMA-ID. CIRCUNFERENCIAS.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 01  VARIABLES.
     05  RADIO    PIC 9(5)V9999.
     05  LONGITUD PIC 9(6)V9999.
 01  CAMPOS-EDITADOS.
     05  EDI-LONG PIC Z(5)9.9999.
 PROCEDURE DIVISION.
 PROCESO.
     DISPLAY "Cálculo de longitudes de circunferencias."
     DISPLAY "Escriba la longitud del radio o 0 para terminar:"
     ACCEPT RADIO
     PERFORM UNTIL RADIO = 0
       CALL "CALCULAR"
         USING BY CONTENT RADIO, BY REFERENCE LONGITUD
       MOVE LONGITUD TO EDI-LONG
       DISPLAY "Longitud = ", EDI-LONG
       DISPLAY "Escriba la longitud del radio o 0 para terminar:"
       ACCEPT RADIO
     END-PERFORM
     STOP RUN.

* Archivo del subprograma externo
 IDENTIFICATION DIVISION.
 PROGRAM-ID. CALCULAR.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 01  PI  PIC  9V999999  VALUE  3.141592.
 LINKAGE SECTION.
 01  R   PIC  9(5)V9999.
 01  L   PIC  9(6)V9999.
 PROCEDURE DIVISION USING R, L.
 PROCESO.
     COMPUTE L = 2 * PI * R
     EXIT PROGRAM.
```

- **Codificación Pascal:**

En la cabecera del subprograma (procedimiento o función) se precede la declaración de cada parámetro formal que se pasa por referencia por la partícula **VAR**. Los parámetros que figuran sin esta partícula son por valor

```
PROGRAM CIRCUNFERENCIAS (INPUT, OUTPUT);
  VAR RADIO, LONGITUD: REAL;
  PROCEDURE CALCULAR_LONGITUD (R: REAL; VAR L: REAL);
    CONST PI = 3.141592;
  BEGIN (* CALCULAR_LONGITUD *)
    L := 2 * PI * R
  END; (* CALCULAR_LONGITUD *)
BEGIN (* CIRCUNFERENCIAS *)
  WRITELN ('Cálculo de longitudes de circunferencias.');
  WRITE ('Escriba la longitud del radio o 0 para terminar: ');
  READLN (RADIO);
  WHILE RADIO <> 0 DO BEGIN
    CALCULAR_LONGITUD (RADIO, LONGITUD);
    WRITELN ('Longitud = ', LONGITUD:11:4);
    WRITE ('Escriba la longitud del radio o 0 para terminar: ');
    READLN (RADIO)
  END
END.  (* CIRCUNFERENCIAS *)
```

7.8. RECURSIVIDAD

La recursividad es una técnica potente de programación que puede utilizarse en lugar de la iteración (bucles) para resolver determinado tipo de problemas. Consiste en permitir que un subprograma se llame a sí mismo para resolver una versión reducida del problema original.

Frente a una determinada gama de problemas se puede optar por una solución iterativa (no recursiva) o una solución recursiva. Existen situaciones en las que el uso de la recursividad permite soluciones (programas) mucho más simples (y elegantes). No obstante, no conviene abusar de esta herramienta, pues podría dar lugar a resultados impredecibles y de difícil comprensión.

El uso de esta técnica es apropiado especialmente cuando el problema a resolver o la estructura de datos a procesar tienen una clara definición recursiva.

Por ejemplo, si se desea calcular el factorial de un número n, entero positivo, se hará a partir de su definición:

$$0! = 1$$
$$n! = n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 3 \cdot 2 \cdot 1, \quad \text{si} \quad n > 0$$

Esta definición daría lugar a una solución iterativa, que podemos representar mediante el siguiente subprograma:

```
Subprograma CALCULAR-FACTORIAL
  recibe      N numérica entera ** mayor o igual que 0
  transforma F numérica entera
Algoritmo:
  F ← 1
  mientras N > 0 hacer
    F ← F * N
    N ← N - 1
  finmientras
Finsubprograma
```

Pero existe esta otra definición «recursiva» de la función factorial:

$$0! = 1$$
$$n! = n \cdot (n-1)!, \quad \text{si } n > 0$$

Esta segunda definición, para el cálculo del factorial de un número mayor que cero, hace referencia a la propia función, lo cual da lugar a una solución recursiva, que representamos con el siguiente subprograma:

```
Subprograma CALCULAR-FACTORIAL
  recibe     N numérica entera
  transforma F numérica entera
Entorno:
  FAUX es numerica entera
Algoritmo:
  si N = 0
    entonces F ← 1
    sino     CALCULAR-FACTORIAL (N - 1, FAUX)
             F ← N * FAUX
  finsi
Finsubprograma
```

Se dice que un subprograma es recursivo si entre sus instrucciones tiene una llamada a sí mismo.

Esta herramienta sólo está disponible en algunos de los lenguajes de programación más conocidos, entre ellos en el lenguaje Pascal, en el que se pueden codificar tanto procedimientos como funciones recursivas:

```
(* Codificación de n! mediante un procedimiento recursivo *)
PROCEDURE CALCULAR_FACTORIAL (N: INTEGER; VAR F: INTEGER);
  VAR FAUX: INTEGER;
BEGIN
  IF N = 0
    THEN F := 1
    ELSE BEGIN
           CALCULAR_FACTORIAL (N - 1, FAUX);
           F := N * FAUX
         END
END;

(* Codificación de n! mediante una función recursiva *)
FUNCTION FACTORIAL (N: INTEGER): INTEGER;
BEGIN
  IF N = 0
    THEN FACTORIAL := 1
    ELSE FACTORIAL := N * FACTORIAL (N - 1)
END;
```

EJERCICIOS RESUELTOS

1. *Las permutaciones: Programa que lee un número* n *entero positivo y obtiene las permutaciones de los* n *primeros números naturales.*

La solución consiste en generar ordenadamente las permutaciones. La primera que se ha de generar es la formada por los n números en el orden natural y la última que se obtiene es la formada por los números en el orden inverso.

Primera: $1, 2, 3, \ldots, n-1, n$.
Ultima: $n, n-1, n-2, \ldots, 2, 1$.

Diremos que una permutación Q es posterior a otra P si comparando sus números, uno a uno, de izquierda a derecha se cumple que en el primer número diferente el número de Q es mayor que el de P

Para generar la permutación Q que sigue a una dada P se procede de la siguiente manera:

Sea la permutación $P = a_1, a_2, \ldots a_n$.

1. Se busca de derecha a izquierda el primer número a_i que sea menor que el que le sigue ($a_i < a_{i-1}$).
2. Se busca de derecha a izquierda el primer número a_j que sea mayor que a_i ($a_j > a_i$).
3. Se intercambian ambos números.
4. Se invierten los números que siguen a la posición i.

Por ejemplo, si se están generando las permutaciones de 7:
Sea $P = 5, 6, 3, 7, 4, 2, 1$.

1. $a_i = 3$ ($3 < 7$).
2. $a_j = 4$ ($4 > 3$).
3. 5, 6, 4, 7, 3, 2, 1.
4. 5, 6, 4, 1, 2, 3, 7.

Por tanto $Q = 5, 6, 4, 1, 2, 3, 7$.

- **Pseudocódigo:**

```
Programa PERMUTACIONES
**
Entorno:
  MAX es constante numérica entera con valor 8  ** 8! = 40320
  P es tabla(MAX) numérica entera
  N, I, J son numéricas enteras
**
Algoritmo:
  escribir "Permutaciones de los n primeros números naturales"
  escribir "Escriba el número n entre 1 y 8: "
  leer N
  GENERAR-PRIMERA (P, N)
  ESCRIBIR-PERMUTACION (P, N)
  iterar
    BUSCAR-MENOR (P, N, I)
    salir si I = 0
    BUSCAR-MAYOR (P, N, I, J)
    INTERCAMBIAR (P, I, J)
    INVERTIR-FINAL (P, N, I, J)
    ESCRIBIR-PERMUTACION (P, N)
```

```
  finiterar
Finprograma
**
Subprograma GENERAR-PRIMERA
  transforma P tabla(MAX) numérica entera
  recibe     N numérica entera
Entorno:
  K es numérica entera
Algoritmo:
  para K de 1 a N hacer
    P(K) ← K
  finpara
Finsubprograma
**
Subprograma ESCRIBIR-PERMUTACION
  recibe P tabla(MAX) numérica entera
  recibe N numérica entera
Entorno:
  K es numérica entera
Algoritmo:
  para K de 1 a N hacer
    escribir P(K)
  finpara
  saltar-linea  ** se escribe cada permutación en una línea
Finsubprograma
**
Subprograma BUSCAR-MENOR
  recibe     P tabla(MAX) numérica entera
  recibe     N numérica entera
  transforma I numérica entera
Algoritmo:
  I ← N - 1
  mientras P(I) > P(I + 1) y I > 1 hacer
    I ← I - 1
  finmientras
  si P(I) > P(I + 1)
    entonces I ← 0
  finsi
Finsubprograma
**
Subprograma BUSCAR-MAYOR
  recibe     P tabla(MAX) numérica entera
  recibe     N numérica entera
  recibe     I numérica entera
  transforma J numérica entera
Algoritmo:
  J ← N
  mientras P(J) < P(I) hacer
    J ← J - 1  ** se llega como máximo a J = I + 1
  finmientras
Finsubprograma
**
Subprograma INTERCAMBIAR
  transforma P tabla(MAX) numérica entera
  recibe     I numérica entera
  recibe     J numérica entera
Entorno:
  K es numérica entera
Algoritmo:
  K ← P(I)
```

```
    P(I) ← P(J)
    P(J) ← K
  Finsubprograma
  **
  Subprograma INVERTIR-FINAL
    transforma P tabla(MAX) numérica entera
    recibe     N numérica entera
    recibe     I numérica entera
    recibe     J numérica entera
  Algoritmo:
    I ← I + 1
    J ← N
    mientras I < J hacer
      INTERCAMBIAR (P, I, J)
      I ← I + 1
      J ← J - 1
    finmientras
  Finsubprograma
```

- **Codificación Pascal:**

```
PROGRAM PERMUTACIONES (INPUT, OUTPUT);
  CONST MAX = 8;  (* 8! = 40320 *)
  TYPE PERM = ARRAY [1..MAX] OF INTEGER;
  VAR P: PERM;
      N, I, J: INTEGER;
  (**)
  PROCEDURE GENERAR_PRIMERA (VAR P: PERM; N: INTEGER);
    VAR K: INTEGER;
  BEGIN (* GENERAR_PRIMERA *)
    FOR K := 1 TO N DO P[K] := K
  END;  (* GENERAR_PRIMERA *)
  (**)
  PROCEDURE ESCRIBIR_PERMUTACION (P: PERM; N: INTEGER);
    VAR K: INTEGER;
  BEGIN (* ESCRIBIR_PERMUTACION *)
    FOR K := 1 TO N DO WRITE (P[K]:3);
    WRITELN
  END;  (* ESCRIBIR_PERMUTACION *)
  (**)
  PROCEDURE BUSCAR_MENOR (P: PERM; N: INTEGER; VAR I: INTEGER);
  BEGIN (* BUSCAR_MENOR *)
    I := N - 1;
    WHILE (P[I] > P[I + 1]) AND (I > 1) DO I := I - 1;
    IF P[I] > P[I + 1] THEN I := 0
  END;  (* BUSCAR_MENOR *)
  (**)
  PROCEDURE BUSCAR_MAYOR(P: PERM; N, I: INTEGER; VAR J: INTEGER);
  BEGIN (* BUSCAR_MAYOR *)
    J := N;
    WHILE P[J] < P[I] DO J := J - 1
  END;  (* BUSCAR_MAYOR *)
  (**)
  PROCEDURE INTERCAMBIAR (VAR P: PERM; I, J: INTEGER);
    VAR K: INTEGER;
  BEGIN (* INTERCAMBIAR *)
    K    := P[I];
    P[I] := P[J];
    P[J] := K
  END;  (* INTERCAMBIAR *)
```

```
  (**)
  PROCEDURE INVERTIR_FINAL (VAR P: PERM; N, I, J: INTEGER);
  BEGIN (* INVERTIR_FINAL *)
    I := I + 1;
    J := N;
    WHILE I < J DO
    BEGIN
      INTERCAMBIAR (P, I, J);
      I := I + 1;
      J := J - 1
    END
  END;  (* INVERTIR_FINAL *)
(**)
BEGIN (* PERMUTACIONES *)
  WRITELN ('Permutaciones de los n primeros números naturales');
  WRITE ('Escriba el número n entre 1 y 8: ');
  READLN (N);
  WRITELN;
  GENERAR_PRIMERA (P, N);
  ESCRIBIR_PERMUTACION (P, N);
  BUSCAR_MENOR (P, N, I);
  WHILE I > 0 DO
    BEGIN
    BUSCAR_MAYOR (P, N, I, J);
    INTERCAMBIAR (P, I, J);
    INVERTIR_FINAL (P, N, I, J);
    ESCRIBIR_PERMUTACION (P, N);
    BUSCAR_MENOR (P, N, I)
  END
END.  (* PERMUTACIONES *)
```

2. *El problema de las ocho reinas: Programa que obtiene todas las posibilidades de colocar ocho reinas sobre un tablero de ajedrez sin que ninguna de ellas amenace a cualquiera de las otras.*

Cada reina colocada en el tablero amenaza a todas las que estén situadas en alguna casilla perteneciente a la fila, columna o diagonales en que se encuentra la primera.

Una primera idea sería generar todas las posibles configuraciones seleccionando las que sean válidas. Esta idea se desecha en cuanto se contabiliza el número de configuraciones posibles:

$$\binom{64}{8} = 4.420.165.368$$

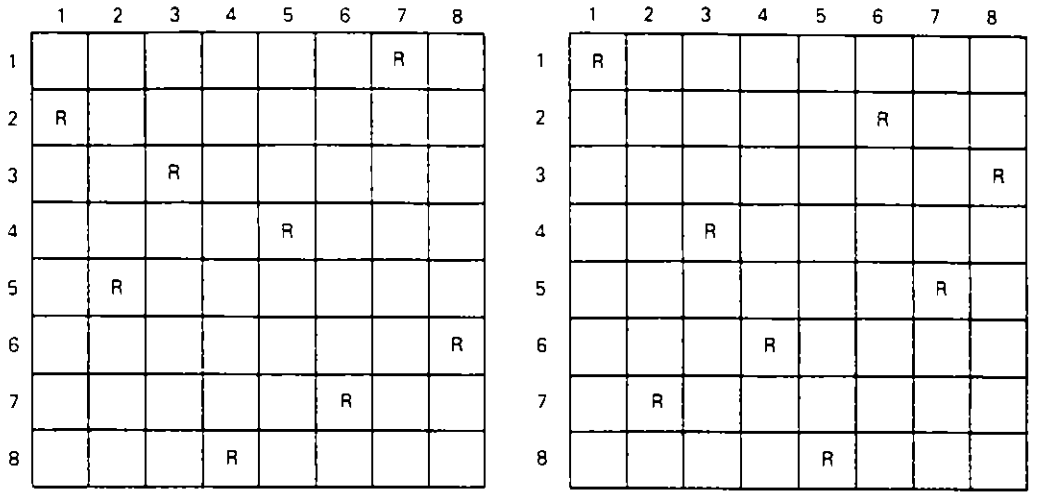
Si seleccionamos inicialmente sólo las configuraciones en las que haya una sola reina por fila, el número de posibilidades se limita bastante:

$$8^8 = 16.777.216$$

Pero aún se puede reducir mucho más si consideramos únicamente las configuraciones en las que haya una sola reina por fila y columna. El número de posibilidades en este caso es:

$$8! = 40.320$$

Por ejemplo, son configuraciones de este tipo las dos siguientes:



La primera configuración no es válida, pues se amenazan las reinas de las posiciones (2, 1) y (7, 6), pero la segunda es una de las configuraciones válidas que ha de obtener el programa.

El segundo aspecto a considerar es la estructura de datos que se va a utilizar para representar el tablero de ajedrez.

Una primera estructura de datos válida sería una tabla de ocho filas y ocho columnas de tipo booleano, asignando *cierto* a los cuadros ocupados y *falso* a los libres.

No obstante, para generar las configuraciones y examinarlas, es mucho más apropiada una estructura de vector de ocho componentes numéricas enteras, en el que el índice representa el número de fila y el valor asignado el número de columna.

La representación de las dos configuraciones antes expuestas queda como sigue:

7 1 3 5 2 8 6 4 1 6 8 3 7 4 2 5

donde cada número indica la columna y su orden es la fila para cada una de las ocho reinas colocadas en el tablero.

Con esta estructura de datos, y con la limitación expuesta de considerar únicamente las configuraciones con una sola reina por fila y columna, podemos darnos cuenta de que el total de configuraciones a examinar se representa con todas las permutaciones de los ocho primeros números naturales, con lo que podemos utilizar el mismo programa del ejercicio anterior.

De esta manera sólo queda decidir cómo se examina cada configuración para ver si es válida o no. Para ello utilizamos la propiedad de que para cada diagonal ascendente la suma de sus índices es constante y para cada diagonal descendente la resta de sus índices también lo es.

- **Pseudocódigo:**

```
Programa OCHO-REINAS
**
Entorno:
  T es tabla(8) numérica entera
  I, J son numéricas enteras
  HAY-AMENAZA es booleana
```

```
Algoritmo:
  escribir "Configuraciones de 8 reinas sin amenazas:"
  GENERAR-PRIMERA (T)
  ** no examinamos la primera que sabemos que no es válida
  iterar
    BUSCAR-MENOR (T, I)
    salir si I = 0
    BUSCAR-MAYOR (T, I, J)
    INTERCAMBIAR (T, I, J)
    INVERTIR-FINAL (T, I, J)
    EXAMINAR-TABLERO (T, HAY-AMENAZA)
    si no HAY-AMENAZA
      entonces ESCRIBIR-TABLERO (T)
    finsi
  finiterar
Finprograma
** ...
** No incluimos aquí los subprogramas que son similares a los
** del ejercicio anterior
** ...
Subprograma EXAMINAR-TABLERO
  recibe      T tabla(8) numérica entera
  transforma HAY-AMENAZA booleana
Entorno:
  I, J, DAI, DDI son numéricas enteras
** DAI suma de los índices de la diagonal ascendente de la reina I
** DDI diagonal descendente de la reina I
Algoritmo:
  HAY-AMENAZA ← FALSO
  I ← 2
  mientras I <= 8 y no HAY-AMENAZA hacer
    DAI ← I + T(I)
    DDI ← I - T(I)
    J ← 1
    mientras J + T(J) <> DAI y J - T(J) <> DDI hacer
      J ← J + 1
    finmientras
    si J <> I
      entonces HAY-AMENAZA ← CIERTO
    finsi
    I ← I + 1
  finmientras
Finsubprograma
```

- **Codificación COBOL:**

```
IDENTIFICATION DIVISION.
PROGRAM-ID. OCHO-REINAS.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  TABLERO.
    05  T          PIC 9  OCCURS 8 COMP.
01  VARIABLES.
    05  I          PIC 9          COMP.
    05  J          PIC 9          COMP.
    05  AUX        PIC 9          COMP.
    05  DAI        PIC S99        COMP.
    05  DDI        PIC S99        COMP.
    05  DAJ        PIC S99        COMP.
    05  DDJ        PIC S99        COMP.
```

```
      05  HAY-AMENAZA PIC XX.
      05  EDI-T       PIC 9BB.
 PROCEDURE DIVISION.
 PROCESO.
      DISPLAY "Configuraciones de 8 reinas sin amenazas:"
      PERFORM GENERAR-PRIMERA
      PERFORM BUSCAR-MENOR
      PERFORM UNTIL I = 0
        PERFORM BUSCAR-MAYOR
        PERFORM INTERCAMBIAR
        PERFORM INVERTIR-FINAL
        PERFORM EXAMINAR-TABLERO
        IF HAY-AMENAZA = "NO"
          THEN PERFORM ESCRIBIR-TABLERO
        END-IF
        PERFORM BUSCAR-MENOR
      END-PERFORM
      STOP RUN.
*
 GENERAR-PRIMERA.
      PERFORM VARYING I FROM 1 BY 1 UNTIL I > 8
        MOVE I TO T (I)
      END-PERFORM.
*
 BUSCAR-MENOR.
      MOVE 7 TO I
      PERFORM UNTIL T (I) < T (I + 1) OR I = 1
        SUBTRACT 1 FROM I
      END-PERFORM
      IF T (I) > T (I + 1)
        THEN MOVE 0 TO I
      END-IF.
*
 BUSCAR-MAYOR.
      MOVE 8 TO J
      PERFORM UNTIL T (J) > T (I)
        SUBTRACT 1 FROM J
      END-PERFORM.
*
 INTERCAMBIAR.
      MOVE T (I) TO AUX
      MOVE T (J) TO T(I)
      MOVE AUX   TO T (J).
*
 INVERTIR-FINAL.
      ADD  1 TO I
      MOVE 8 TO J
      PERFORM UNTIL I >= J
        PERFORM INTERCAMBIAR
        ADD       1 TO   I
        SUBTRACT 1 FROM J
      END-PERFORM.
*
 EXAMINAR-TABLERO.
      MOVE "NO" TO HAY-AMENAZA
      MOVE 2    TO I
      PERFORM UNTIL I > 8 OR HAY-AMENAZA = "SI"
        COMPUTE DAI = I + T (I)
        COMPUTE DDI = I - T (I)
        MOVE 1 TO J
```

```
            COMPUTE DAJ = J + T (J)
            COMPUTE DDJ = J - T (J)
            PERFORM UNTIL DAI = DAJ OR DDI = DDJ
              ADD 1 TO J
              COMPUTE DAJ = J + T (J)
              COMPUTE DDJ = J - T (J)
            END-PERFORM
            IF J NOT = I
              THEN MOVE "SI" TO HAY-AMENAZA
            END-IF
            ADD 1 TO I
          END-PERFORM.
      *
       ESCRIBIR-TABLERO.
          PERFORM VARYING I FROM 1 BY 1 UNTIL I > 8
            MOVE T (I) TO EDI-T
            DISPLAY EDI-T NO ADVANCING
          END-PERFORM
          DISPLAY " ".
```

3. *El problema de las elecciones: Programa que realiza el reparto de escaños en unas elecciones por medio de la regla D'HONT.*

   El programa recibe como datos de entrada:

   — Nombre de una circunscripción electoral.
   — Censo electoral.
   — Número de escaños a distribuir.
   — Número de grupos presentados a las elecciones.
   — Nombre de cada grupo y su número de votos.
   — Número de votos en blanco.
   — Número de votos nulos.

   El programa proporcionará como resultado la siguiente salida (por pantalla o por impresora):

ACTA ELECTORAL

Circunscripción . . . . . .

Censo electoral	Número de escaños	
Votos emitidos	. . . . .	% sobre el censo
Abstención	. . . . .	% sobre el censo
Votos válidos	. . . . .	% sobre emitidos
Votos en blanco	. . . . .	% sobre emitidos
Votos nulos	. . . . .	% sobre emitidos

DISTRIBUCION DE ESCAÑOS

Grupo	Votos	% censo	% emitidos	Escaños
. . . . .	. . . . .	. . . . .	. . . . .	. . . . .
. . . . .	. . . . .	. . . . .	. . . . .	. . . . .
. . . . .	. . . . .	. . . . .	. . . . .	. . . . .
. . . . .	. . . . .	. . . . .	. . . . .	. . . . .

El listado de distribución de escaños se presentará ordenado por número de votos.

El reparto de escaños se realiza utilizando una matriz con una fila por cada grupo y tantas columnas como número de escaños a repartir.

A cada componente de la matriz se le asigna el número resultante de dividir la cantidad de votos obtenidos por el grupo correspondiente a esa fila, dividido por el número de la columna.

Los escaños se asignan a los grupos en cuyas filas están los máximos valores de la matriz, uno por cada máximo, hasta completar el número de escaños.

La estructura de datos utilizada para el reparto de N escaños entre M grupos será la matriz:

DHONT

	1	2	. . .	N
Grupo 1	Votos 1/1	Votos 1/2	. . .	Votos 1/N
Grupo 2	Votos 2/1	Votos 2/2	. . .	Votos 2/N
. . . .	.	. . .	. . .	. . .
Grupo M	Votos M/1	Votos M/2	. . .	Votos M/N

Para almacenar los resultados, que habrá que ordenar posteriormente, se utilizarán los vectores:

	1	2	. . .	M
GRUPO	Grupo 1	Grupo 2	. . .	Grupo M
VOTOS	Votos 1	Votos 2	. . .	Votos M
PVCEN	Pvcen 1	Pvcen 2	. . .	Pvcen M
PVEMI	Pvemi 1	Pvemi 2	. . .	Pvemi M
ESCAN	Escan 1	Escan 2	. . .	Escan M

- **Pseudocódigo:**

```
Programa ELECCIONES
Entorno:
  DHONT es tabla(15,40) numérica real
  ** Máximo de 15 grupos y 40 escaños
  GRUPO es tabla(15) alfanumérica
  ** Nombres de los grupos participantes
  VOTOS es tabla(15) numérica entera
  ** Número de votos de cada grupo
  PVCEN es tabla(15) numérica real
```

```
  ** Porcentaje de votos de cada grupo sobre el censo
  PVEMI es tabla(15) numérica real
  ** Porcentaje de votos de cada grupo sobre emitidos
  ESCAN es tabla(15) numérica entera
  ** Número de escaños obtenidos por cada grupo
  CIR es alfanumérica ** Nombre de la circunscripción
  CENSO, NESCAN, NGRUP, VEMI, ABSTE, VVALI, VBLAN,
  VNULO, I, J son numéricas enteras
  PEMICEN, PABSCEN, PVALI, PBLAN, PNULO son numéricas reales
Algoritmo:
  ENTRAR DATOS
  ORDENAR GRUPOS
  CALCULAR PORCENTAJES
  REPARTIR ESCANOS
  IMPRIMIR LISTADO
Finprograma
**
Subprograma ENTRAR DATOS
Algoritmo:
  leer CIR, CENSO, NESCAN, NGRUP
  VVALI ← 0
  para I de 1 a NGRUP hacer
    leer GRUPO(I), VOTOS(I); VVALI ← VVALI + VOTOS(I)
  finpara
  leer VBLAN, VNULO
  VEMI ← VVALI + VBLAN + VNULO; ABSTE ← CENSO - VEMI
Finsubprograma
**
Subprograma ORDENAR GRUPOS
** Ordena simultáneamente los vectores GRUPO y VOTOS
** en orden descendente según los valores del segundo
Entorno:
  A es alfanumérica; N es numérica entera
Algoritmo:
  para I de NGRUP - 1 a 1 con incremento -1 hacer
    para J de 1 a I hacer
      si VOTOS(J) < VOTOS(J + 1) entonces
        N ← VOTOS(J); VOTOS(J) ← VOTOS(J + 1)
        VOTOS(J + 1) ← N; A ← GRUPO(J)
        GRUPO(J) ← GRUPO(J + 1); GRUPO(J + 1) ← A
      finsi
    finpara
  finpara
Finsubprograma
**
Subprograma CALCULAR PORCENTAJES
Algoritmo:
  PEMICEN ← VEMI * 100 / CENSO
  PABSCEN ← ABSTE * 100 / CENSO
  PVALI ← VVALI * 100 / VEMI
  PBLAN ← VBLAN * 100 / VEMI
  PNULO ← VNULO * 100 / VEMI
  para I de 1 a NGRUP hacer
    PVCEN(I) ← VOTOS(I) * 100 / CENSO
    PVEMI(I) ← VOTOS(I) * 100 / VEMI
  finpara
Finsubprograma
**
Subprograma REPARTIR ESCANOS
Entorno:
```

```
    MAXIMO es numérica real
    F, C, FI, CO son numericas enteras
  Algoritmo:
    ** Carga de la matriz DHONT
    para F de 1 a NGRUP hacer
      para C de 1 a NESCAN hacer
        DHONT(F,C) ← VOTOS(F) / C
      finpara
    finpara
    ** Inicialización del vector ESCAN
    para I de 1 a NGRUP hacer
      ESCAN(I) ← 0
    finpara
    ** Reparto de escaños
    para I de 1 a NESCAN hacer
      MAXIMO ← -1
      ** Obtención de un máximo
      para F de 1 a NGRUP hacer
        para C de 1 a NESCAN hacer
          si DHONT(F,C) > MAXIMO entonces
            MAXIMO ← DHONT(F,C); FI ← F; CO ← C
          finsi
        finpara
      finpara
      ESCAN(FI) ← ESCAN(FI) + 1
      DHONT(FI,CO) ← -1
    finpara
  Finsubprograma
  **
  Subprograma IMPRIMIR LISTADO
  Algoritmo:
    escribir CIR, CENSO, NESCAN, VEMI, PEMICEN, ABSTE,
      PABSCEN, VVALI, PVALI, VBLAN, PBLAN, VNULO, PNULO
    para I de 1 a NGRUP hacer
      escribir GRUPO(I),VOTOS(I),PVCEN(I),PVEMI(I),ESCAN(I)
    finpara
  Finsubprograma
```

- **Codificación COBOL:**

```
IDENTIFICATION DIVISION.
PROGRAM-ID. ELECCIONES.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  TABLA-DHONT.
    05  FILA-DHONT OCCURS 15.
        08  DHONT PIC 9(7)V99 OCCURS 40.
01  TABLA-GRUPO.
    05  GRUPO     PIC X(10)     OCCURS 15.
01  TABLA-VOTOS.
    05  VOTOS     PIC 9(7)      OCCURS 15.
01  TABLA-PVCEN.
    05  PVCEN     PIC 99V99     OCCURS 15.
01  TABLA-PVEMI.
    05  PVEMI     PIC 99V99     OCCURS 15.
01  TABLA-ESCAN.
    05  ESCAN     PIC 99        OCCURS 15.
01  VARIABLES.
    05  CIR       PIC X(15).
    05  CENSO     PIC 9(7).
```

```
       05  NESCAN  PIC 99.
       05  NGRUP   PIC 99.
       05  NG      PIC 99.
       05  VEMI    PIC 9(7).
       05  ABSTE   PIC 9(7).
       05  VVALI   PIC 9(7).
       05  VBLAN   PIC 9(7).
       05  VNULO   PIC 9(7).
       05  I       PIC 99.
       05  J       PIC 99.
       05  F       PIC 99.
       05  C       PIC 99.
       05  FI      PIC 99.
       05  CO      PIC 99.
       05  N       PIC 9(7).
       05  A       PIC X(10).
       05  PEMICEN PIC 99V99.
       05  PABSCEN PIC 99V99.
       05  PVALI   PIC 99V99.
       05  PBLAN   PIC 99V99.
       05  PNULO   PIC 99V99.
       05  MAX     PIC 9(7)V99.
       05  EDIP    PIC Z9.99.
       05  EDIQ    PIC Z9.99.
PROCEDURE DIVISION.
PROCESO.
       PERFORM ENTRAR-DATOS
       PERFORM ORDENAR-GRUPOS
       PERFORM CALCULAR-PORCENTAJES
       PERFORM REPARTIR-ESCANOS
       PERFORM IMPRIMIR-LISTADO
       STOP RUN.
*
ENTRAR-DATOS.
       DISPLAY "Nombre de la circunscripción: "
       ACCEPT CIR
       DISPLAY "Censo electoral: "
       ACCEPT CENSO
       DISPLAY "Número de escaños: "
       ACCEPT NESCAN
       DISPLAY "Número de grupos presentados: "
       ACCEPT NGRUP
       MOVE 0 TO VVALI
       PERFORM VARYING I FROM 1 BY 1 UNTIL I > NGRUP
         DISPLAY "Nombre del grupo nº ", I, ":"
         ACCEPT GRUPO(I)
         DISPLAY "Número de votos:"
         ACCEPT VOTOS(I)
         ADD VOTOS(I) TO VVALI
       END-PERFORM
       DISPLAY "Votos en blanco: "
       ACCEPT VBLAN
       DISPLAY "Votos nulos: "
       ACCEPT VNULO
       ADD VVALI, VBLAN, VNULO GIVING VEMI
       SUBTRACT VEMI FROM CENSO GIVING ABSTE.
*
ORDENAR-GRUPOS.
       SUBTRACT 1 FROM NGRUP GIVING NG
       PERFORM VARYING I FROM NG BY -1 UNTIL I < 1
```

```
        PERFORM VARYING J FROM 1 BY 1 UNTIL J > I
          IF VOTOS(J) < VOTOS(J + 1) THEN
            MOVE VOTOS(J)     TO N
            MOVE VOTOS(J + 1) TO VOTOS(J)
            MOVE N            TO VOTOS(J + 1)
            MOVE GRUPO(J)     TO A
            MOVE GRUPO(J + 1) TO GRUPO(J)
            MOVE A            TO GRUPO(J + 1)
          END-IF
        END-PERFORM
      END-PERFORM.
*
 CALCULAR-PORCENTAJES.
      COMPUTE PEMICEN = VEMI  * 100 / CENSO
      COMPUTE PABSCEN = ABSTE * 100 / CENSO
      COMPUTE PVALI   = VVALI * 100 / VEMI
      COMPUTE PBLAN   = VBLAN * 100 / VEMI
      COMPUTE PNULO   = VNULO * 100 / VEMI
      PERFORM VARYING I FROM 1 BY 1 UNTIL I > NGRUP
        COMPUTE PVCEN(I) = VOTOS(I) * 100 / CENSO
        COMPUTE PVEMI(I) = VOTOS(I) * 100 / VEMI
      END-PERFORM.
*
 REPARTIR-ESCANOS.
      PERFORM VARYING F FROM 1 BY 1 UNTIL F > NGRUP
        PERFORM VARYING C FROM 1 BY 1 UNTIL C > NESCAN
          DIVIDE VOTOS(F) BY C GIVING DHONT(F, C)
        END-PERFORM
      END-PERFORM
      PERFORM VARYING I FROM 1 BY 1 UNTIL I > NGRUP
        MOVE 0 TO ESCAN(I)
      END-PERFORM
      PERFORM VARYING I FROM 1 BY 1 UNTIL I > NESCAN
        MOVE -1 TO MAX
        PERFORM VARYING F FROM 1 BY 1 UNTIL F > NGRUP
          PERFORM VARYING C FROM 1 BY 1 UNTIL C > NESCAN
            IF DHONT(F, C) > MAX THEN
              MOVE DHONT(F, C) TO MAX
              MOVE F           TO FI
              MOVE C           TO CO
            END-IF
          END-PERFORM
        END-PERFORM
        ADD 1 TO ESCAN(FI)
        MOVE -1 TO DHONT(FI, CO)
      END-PERFORM.
*
 IMPRIMIR-LISTADO.
      DISPLAY " " ERASE
      DISPLAY "                          ",
              "A C T A     E L E C T O R A L"
      DISPLAY "                          "
              "----------------------------"
      DISPLAY " "
      DISPLAY "                              Circunscripción: ", CIR
      DISPLAY " "
      DISPLAY "              Censo electoral: ", CENSO,
              "       Número de escaños: ", NESCAN
      MOVE PEMICEN TO EDIP
      DISPLAY "            Votos emitidos   ", VEMI,
```

```
            "      ", EDIP, " % sobre el censo"
        MOVE PABSCEN TO EDIP
        DISPLAY "            Abstención       ", ABSTE,
                "      ", EDIP, " % sobre el censo"
        MOVE PVALI TO EDIP
        DISPLAY "            Votos validos    ", VVALI,
                "      ", EDIP, " % sobre emitidos"
        MOVE PBLAN TO EDIP
        DISPLAY "            Votos en blanco  ", VBLAN,
                "      ", EDIP, " % sobre emitidos"
        MOVE PNULO TO EDIP
        DISPLAY "            Votos nulos      ", VNULO,
                "      ", EDIP, " % sobre emitidos"
        DISPLAY " "
        DISPLAY " "
        DISPLAY "              ",
                "D I S T R I B U C I O N   D E   E S C A Ñ O S"
        DISPLAY "              ",
                "-----------------------------------------------"
        DISPLAY " "
        DISPLAY "           GRUPO       VOTOS       % CENSO    ",
                "% EMITIDOS  ESCAÑOS"
        DISPLAY "           ---------------------------------",
                "-------------------"
        PERFORM VARYING I FROM 1 BY 1 UNTIL I > NGRUP
          MOVE PVCEN(I) TO EDIP
          MOVE PVEMI(I) TO EDIQ
          DISPLAY "             ", GRUPO(I), VOTOS(I), "      ",
                  EDIP, "       ", EDIQ, "         ", ESCAN(I)
        END-PERFORM.
```

- **Codificación Pascal:**

```
PROGRAM ELECCIONES (INPUT, OUTPUT);
  TYPE MAT  = ARRAY[1..15,1..40] OF REAL;
       VEC1 = ARRAY[1..15] OF STRING[10];
       VEC2 = ARRAY[1..15] OF INTEGER;
       VEC3 = ARRAY[1..15] OF REAL;
  VAR DHONT : MAT;
      GRUPO : VEC1;
      VOTOS, ESCAN : VEC2;
      PVCEN, PVEMI : VEC3;
      CIR : STRING[15];
      CENSO, NESCAN, NGRUP, VEMI, ABSTE,
      VVALI, VBLAN, VNULO, I, J : INTEGER;
      PEMICEN, PABSCEN, PVALI, PBLAN, PNULO : REAL;
(**)
  PROCEDURE ENTRARDATOS;
    BEGIN (*ENTRARDATOS*)
      WRITE('Nombre de la circunscripción: ');
      READLN(CIR);
      WRITE('Censo electoral: '); READLN(CENSO);
      WRITE('Número de escaños: '); READLN(NESCAN);
      WRITE('Número de grupos presentados: ');
      READLN(NGRUP);
      VVALI := 0;
      FOR I := 1 TO NGRUP DO
        BEGIN (*1*)
          WRITE('Nombre del grupo nº ', I, ': ');
          READLN(GRUPO[I]);
```

```
          WRITE('Número de votos: ');
          READLN(VOTOS[I]);
          VVALI := VVALI + VOTOS[I]
        END; (*1*)
      WRITE('Votos en blanco: '); READLN(VBLAN);
      WRITE('Votos nulos: '); READLN(VNULO);
      VEMI := VVALI + VBLAN + VNULO;
      ABSTE := CENSO - VEMI
  END; (*ENTRARDATOS*)
(**)
  PROCEDURE ORDENARGRUPOS;
    VAR A : STRING[10];
        N : INTEGER;
    BEGIN (*ORDENARGRUPOS*)
      FOR I := NGRUP - 1 DOWNTO 1 DO
        FOR J := 1 TO I DO
          IF VOTOS[J] < VOTOS[J + 1] THEN
            BEGIN (*2*)
              N := VOTOS[J];
              VOTOS[J] := VOTOS[J + 1];
              VOTOS[J + 1] := N;
              A := GRUPO[J];
              GRUPO[J] := GRUPO[J + 1];
              GRUPO[J + 1] := A
            END(*2*)
    END; (*ORDENARGRUPOS*)
(**)
  PROCEDURE CALCULARPORCENTAJES;
    BEGIN (*CALCULARPORCENTAJES*)
      PEMICEN := VEMI  * 100 / CENSO;
      PABSCEN := ABSTE * 100 / CENSO;
      PVALI   := VVALI * 100 / VEMI;
      PBLAN   := VBLAN * 100 / VEMI;
      PNULO   := VNULO * 100 / VEMI;
      FOR I  := 1 TO NGRUP DO
        BEGIN (*3*)
          PVCEN[I] := VOTOS[I] * 100 / CENSO;
          PVEMI[I] := VOTOS[I] * 100 / VEMI
        END (*3*)
    END; (*CALCULARPORCENTAJES*)
(**)
  PROCEDURE REPARTIRESCANOS;
    VAR MAX : REAL;
        F, C, FI, CO : REAL;
    BEGIN (*REPARTIRESCANOS*)
      FOR F := 1 TO NGRUP DO
        FOR C := 1 TO NESCAN DO
          DHONT[F,C] := VOTOS[F] / C;
      FOR I := 1 TO NGRUP DO ESCAN[I] := 0;
      FOR I := 1 TO NESCAN DO
        BEGIN (*4*)
          MAX := -1;
          FOR F := 1 TO NGRUP DO
            FOR C := 1 TO NESCAN DO
              IF DHONT[F,C] > MAX THEN
                BEGIN (*5*)
                  MAX := DHONT[F,C];
                  FI := F; CO := C
                END; (*5*)
          ESCAN[FI] := ESCAN[FI] + 1;
```

```
            DHONT[FI,CO] := -1
          END (*4*)
      END; (*REPARTIRESCANOS*)
(**)
  PROCEDURE IMPRIMIRLISTADO;
    BEGIN (*IMPRIMIRLISTADO*)
      WRITELN(' ':25,'A C T A   E L E C T O R A L');
      WRITELN(' ':25,'-----------------------');
      WRITELN;
      WRITELN(' ':25, 'Circunscripción: ', CIR);WRITELN;
      WRITELN(' ':12, 'Censo electoral:  ', CENSO:7,
              ' ':6,  'Número de escaños: ', NESCAN);
      WRITELN(' ':12, 'Votos emitidos    ', VEMI:7,
              ' ':6,  PVEMI:5:2, ' % sobre el censo');
      WRITELN(' ':12, 'Abstención        ', ABSTE:7,
              ' ':6,  PABSCEN:5:2, ' % sobre el censo');
      WRITELN(' ':12, 'Votos válidos     ', VVALI:7,
              ' ':6,  PVALI:5:2, ' % sobre emitidos');
      WRITELN(' ':12, 'Votos en blanco   ', VBLAN:7,
              ' ':6,  PBLAN:5:2, ' % sobre emitidos');
      WRITELN(' ':12, 'Votos nulos       ', VNULO:7,
              ' ':6,  PNULO:5:2, ' % sobre emitidos');
      WRITELN;
      WRITELN;
      WRITELN(' ':15, 'D I S T R I B U C I O N   D E',
                      '   E S C A Ñ O S');
      WRITELN(' ':15, '--------------------------',
                      '--------------'); WRITELN;
      WRITELN(' ':12, 'GRUPO    VOTOS      % CENSO',
              '    % EMITIDOS ESCAÑOS');
      WRITELN(' ':12, '--------------------------',
              '----------------------');
      FOR I := 1 TO NGRUP DO
        WRITELN(' ':12, GRUPO[I]:10, VOTOS[I]:7, ' ':5,
          PVCEN[I]:5:2, ' ':8, PVEMI[I]:5:2, ' ':7,
          ESCAN[I]:3)
    END; (*IMPRIMIRLISTADO*)
(**)
  BEGIN (* ELECCIONES *)
    ENTRARDATOS;
    ORDENARGRUPOS;
    CALCULARPORCENTAJES;
    REPARTIRESCANOS;
    IMPRIMIRLISTADO
  END. (* ELECCIONES *)
```

EJERCICIOS PROPUESTOS

1. Programa que realiza cálculo matricial, incluyendo las siguientes operaciones:
   - — Suma de dos matrices.
   - — Producto de una constante por una matriz.
   - — Producto de dos matrices.
   - — Potencia de una matriz.

2. Programa que calcula superficies y volúmenes de las siguientes figuras geométricas.
   - — Cilindros.
   - — Conos.
   - — Esferas.
   - — Prismas.
   - — Pirámides.

3. Programa que lee un número entero positivo e imprime su correspondiente representación alfabética (problema de los cheques).

4. Programa que lee un número entero positivo e imprime su correspondiente representación en números romanos.

5. Programa que simula una calculadora de bolsillo, incluyendo las siguientes operaciones, que pueden ir encadenadas:
   - — Suma.
   - — Resta.
   - — Producto.
   - — División.
   - — Raiz cuadrada.

6. Subprograma que calcula recursivamente la potencia entera de un número real (X elevado a n).

7. Subprograma que calcula recursivamente la media aritmética de una serie de números enteros no nulos introducidos por teclado terminando con un cero.

8. Subprograma que genera recursivamente las permutaciones de n elementos ($n > 0$ entero). Para ello considerar el proceso de generar las permutaciones de los elementos $a_1, a_2, \ldots, a_m$ formado por m subprocesos de generación de las permutaciones de $a_1, a_2, \ldots, a_{m-1}$ a las que se añade am, habiendo intercambiado inicialmente en el subproceso i-ésimo los elementos a_i y a_m.

CAPITULO 8

Técnicas de programación estructurada

8.1. INTRODUCCION

Las técnicas de desarrollo y diseño de programas que se utilizan en la programación convencional tienen inconvenientes, sobre todo a la hora de verificar y modificar un programa. En la actualidad están adquiriendo gran importancia las técnias de programación, cuyo objetivo principal es el de facilitar la comprensión del programa, y además permiten, de forma rápida, las ampliaciones y modificaciones que surjan en la fase de explotación del ciclo de vida de un programa o una aplicación informática.

Como hemos visto en el capítulo anterior, una forma de simplificar los programas, haciendo más sencilla su lectura y mantenimiento, es utilizar la técnica del diseño descendente de programas (TOP-DOWN).

En los últimos años la técnica más utilizada que sigue las directrices TOP-DOWN es la *programación estructurada*.

La programación estructurada fue desarrollada en sus principios por Edsgar W. Dijkstra en sus **Notes on Structured Programming** y se basa en el denominado **Teorema de la Estructura** desarrollado en 1966 por Böhm y Jacopini, que se ratificó con los trabajos de Harlan D. Mills.

En la programación convencional se suele hacer un uso indiscriminado y sin control de las instrucciones de salto condicional e incondicional, lo cual produce cierta complejidad en la lectura y en las modificaciones de un programa. Eliminar estas dificultades es uno de los propósitos de la programación estructurada y, por ello, en ocasiones, se ha definido como la técnica de la programación sin saltos condicionales e incondicionales. Esto no es rigurosamente cierto, y por tanto no lo tomaremos como definición, sino como una norma general.

Como consecuencia del párrafo anterior podemos indicar que todo programa estructurado puede ser leído de principio a fin sin interrupciones en la secuencia normal de lectura.

Al mismo tiempo que se obtiene una mayor clarificación del programa por medio de estas técnicas, la puesta a punto del mismo es mucho más rápida, así como la confección de su documentación.

Los programadores en la fase de diseño realizan cada tarea en *módulos* o *bloques*, los cuales pueden estandarizar y así formar su propia biblioteca de programas para su utilización en sucesivas aplicaciones.

En los distintos departamentos de informática existentes no siempre se dispone de los mismos programadores con respecto al tiempo que se pretende que dure una aplicación, por lo cual es de suma importancia que un programa realizado por una persona sea fácil de modificar y mantener por otra. En este sentido, la programación estructurada ofrece muchas ventajas para lograr estos objetivos.

Un programa estructurado es:

— Fácil de leer y comprender.
— Fácil de codificar en una amplia gama de lenguajes y en diferentes sistemas.

— Fácil de mantener.
— Eficiente, aprovechando al máximo los recursos de la computadora.
— Modularizable.

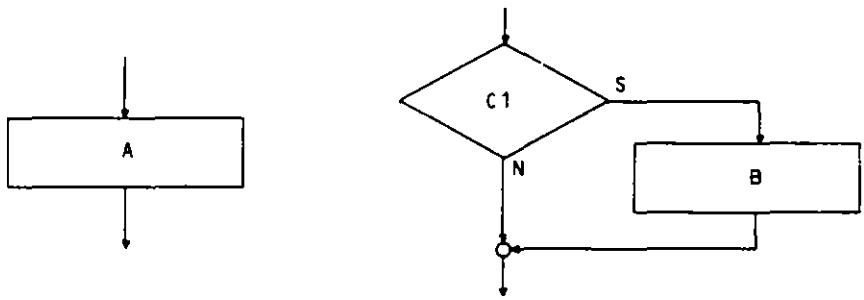
8.2. TEOREMA DE LA ESTRUCTURA

En la actualidad existen diversas definiciones de la programación estructurada, pero todas ellas giran en torno al teorema de la estructura que, como ya hemos dicho, se debe a Bohm y Jacopini.

Para un buen entendimiento del mismo realizamos la definición previa de diagrama propio, programa propio y equivalencia de programas que intervienen en su enunciado directa o indirectamente.

- **Diagrama propio.**

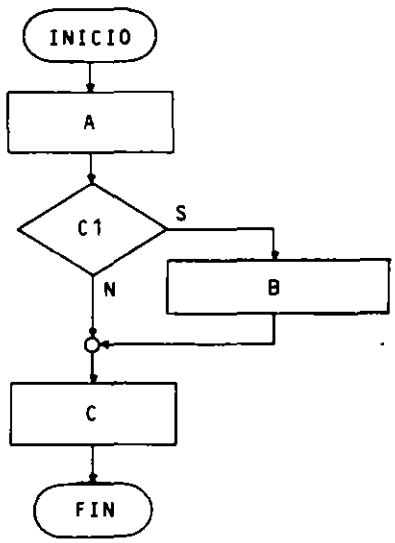
Es aquel que posee un solo punto de entrada y uno solo de salida.



- **Programa propio.**

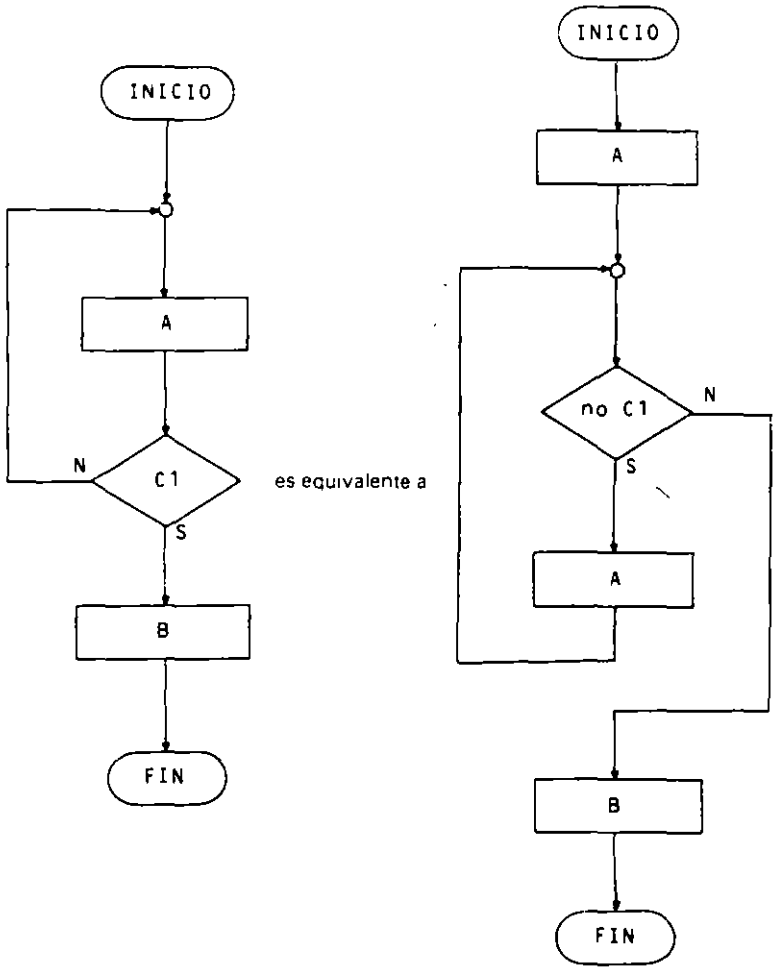
Es aquel programa que cumple las siguientes condiciones:

— Posee un solo inicio y un solo fin.
— Todo elemento del programa es accesible, es decir, existe al menos un camino desde el inicio al fin que pasa a través de él.
— No posee bucles infinitos.



- **Equivalencia de programas.**

Dos programas son equivalentes si realizan, ante cualquier situación de datos, el mismo trabajo pero de distinta forma.



- **Teorema de la estructura.**

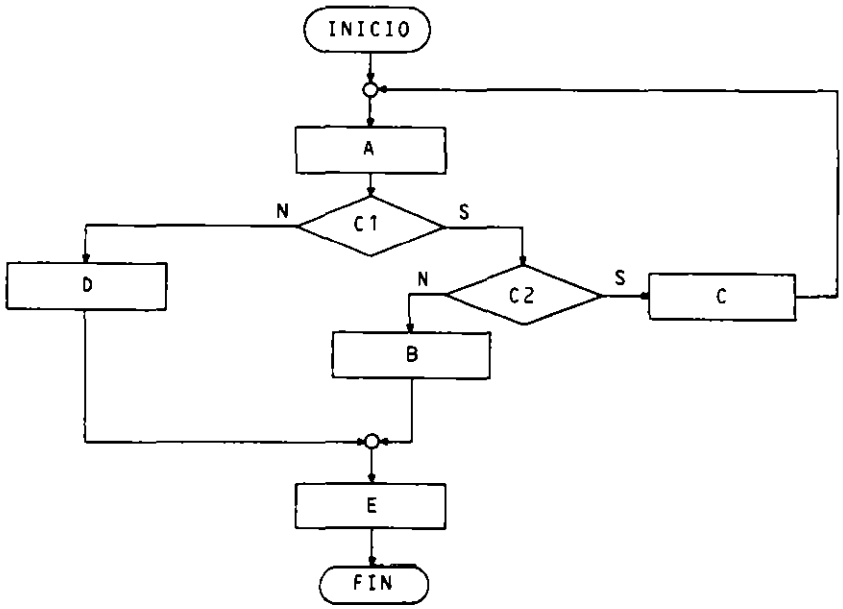
Todo programa propio, realice el trabajo que realice, tiene siempre al menos un programa propio equivalente que sólo utiliza las estructuras básicas de la programación, que son:

— La secuencia.
— La selección.
— La repetición.

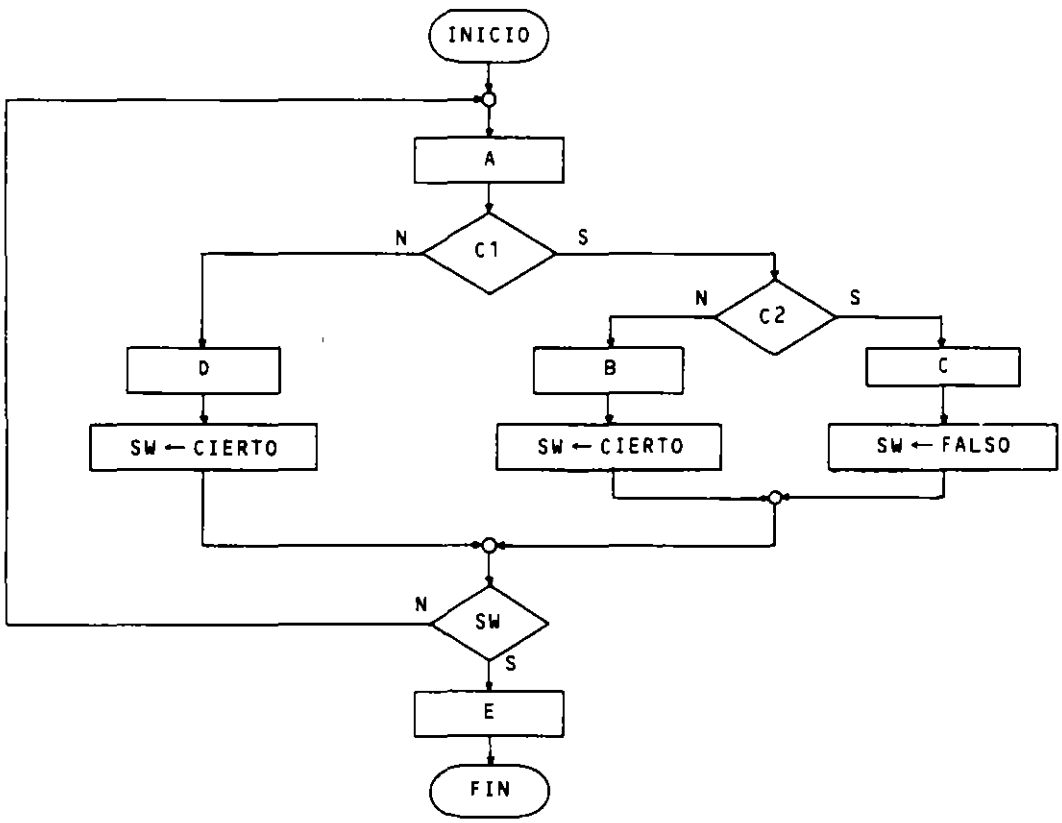
En definitiva, el teorema nos viene a decir que, diseñando programas con sentencias primitivas (lectura, escritura y asignación) y estructuras básicas, no sólo podremos hacer cualquier trabajo sino que además conseguiremos mejorar la creación, lectura, comprensión y mantenimiento de los programas.

Ejemplo: *Aplicación del teorema de la estructura a un algoritmo.*

El siguiente programa propio no utiliza sólo estructuras básicas, como puede verse. Encontrar un programa propio equivalente que sólo utilice dichas estructuras.



El equivalente estructurado, entre otros, puede ser:



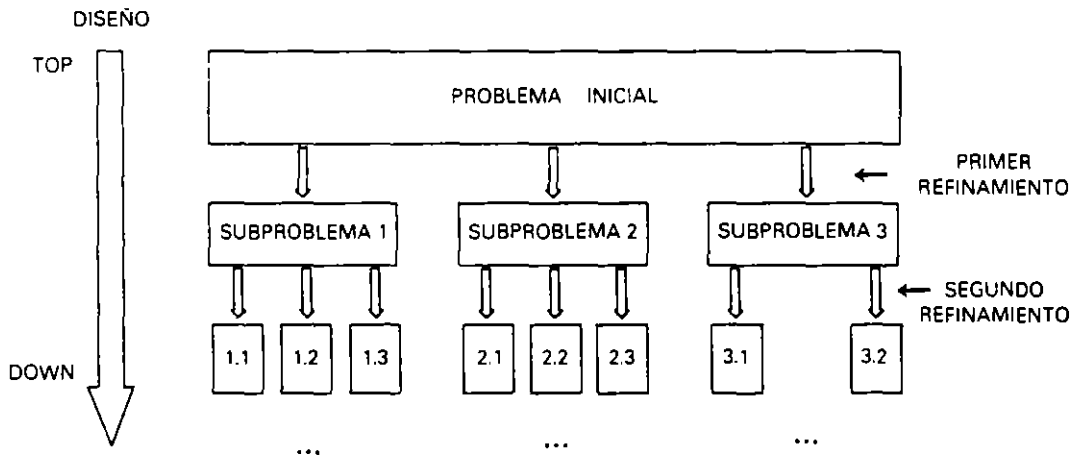
8.3. HERRAMIENTAS DE LA PROGRAMACION ESTRUCTURADA

Además de elementos comunes con otros métodos de programación (objetos, variables auxiliares, operadores, etc.), la programación estructurada utiliza:

— Diseño descendente (TOP-DOWN).
— Recursos abstractos.
— Estructuras básicas.

■ Diseño top-down

Como ya hemos visto en el capítulo anterior, los programas se diseñan de lo general a lo particular por medio de sucesivos refinamientos o descomposiciones que nos van acercando a las instrucciones finales del programa.



■ Utilización de recursos abstractos

Es el complemento perfecto para el diseño TOP-DOWN donde se utiliza el concepto de abstracción; es decir, en cada descomposición se supone que todas las partes resultantes están resueltas, dejando su realización para el siguiente refinamiento y considerando que todas ellas pueden llegar a estar definidas en instrucciones y estructuras disponibles en los lenguajes de programación.

■ Estructuras básicas

Como se indicó anteriormente, el teorema de la estructura dice que toda acción se puede realizar utilizando tres estructuras básicas de control, la estructura secuencial, alternativa y repetitiva. Esta afirmación es cierta y demostrable, aunque su demostración se sale de los objetivos de este libro; por tanto, asumimos su cumplimiento.

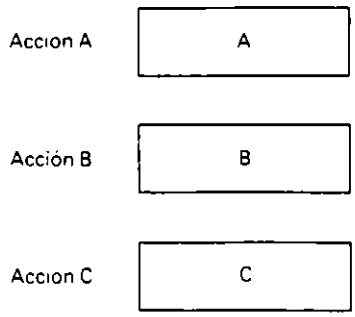
Para la representación gráfica de las estructuras utilizaremos el concepto de acción cuyo significado es totalmente general.

Una **acción** puede representar:

— Ninguna operación.
— Una operación sencilla; por ejemplo, el movimiento de un valor de un campo a otro, una operación de salida, etc.
— Un proceso de cualquier tipo; por ejemplo, una ordenación de datos.

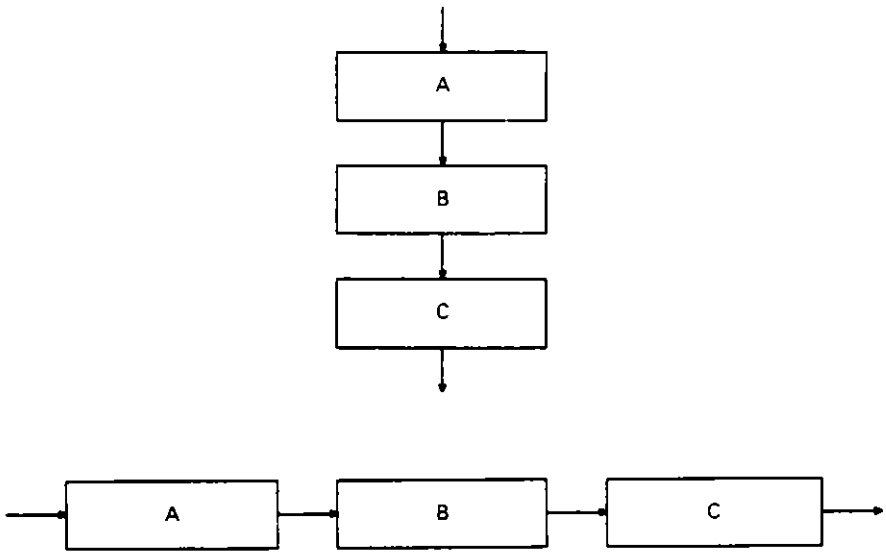
Con ello interpretaremos que una determinada acción representada en una de las tres estructuras puede estar compuesta por una o más estructuras en su interior.

La notación utilizada para representar dichas acciones constará de rectángulos horizontales en cuyo interior pondremos letras mayúsculas.



- **Estructura secuencial**

Es una estructura con una entrada y una salida en la cual figuran una serie de acciones cuya ejecución es lineal y en el orden en que aparecen. A su vez, todas las acciones tienen una única entrada y una única salida.

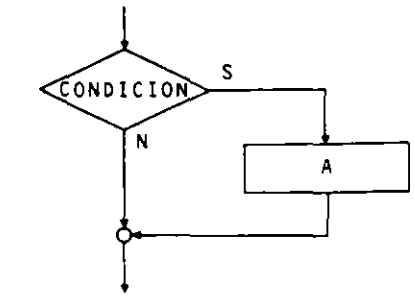


- **Estructura alternativa**

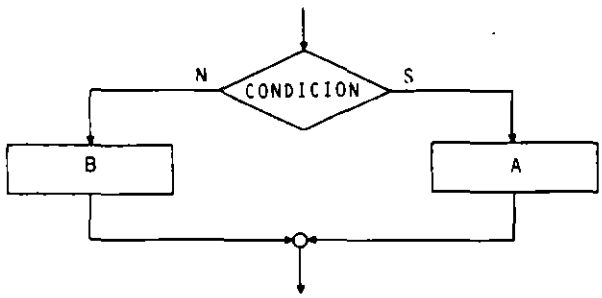
Es una estructura con una sola entrada y una sola salida en la cual se realiza una acción de entre varias, según una condición, o se realiza una acción según el cumplimiento o no de una determinada condición. Esta condición puede ser simple o compuesta.

Las estructuras alternativas pueden ser:

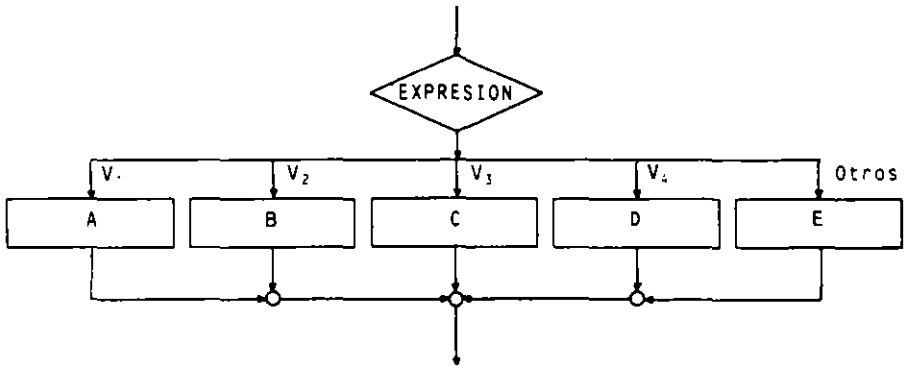
— De dos salidas, en la que una de ellas puede ser la acción nula.
— De tres o más salidas, que también se llama múltiple.



Alternativa simple



Alternativa doble



Alternativa múltiple

- **Estructura repetitiva**

Es una estructura con una entrada y una salida en la cual se repite una acción un número determinado o indeterminado de veces, dependiendo en este caso del cumplimiento de una condición.

Las estructuras repetitivas pueden ser:

— Estructura para (FOR).
— Estructura mientras (WHILE).
— Estructura hasta (UNTIL).
— Estructura iterar (LOOP).

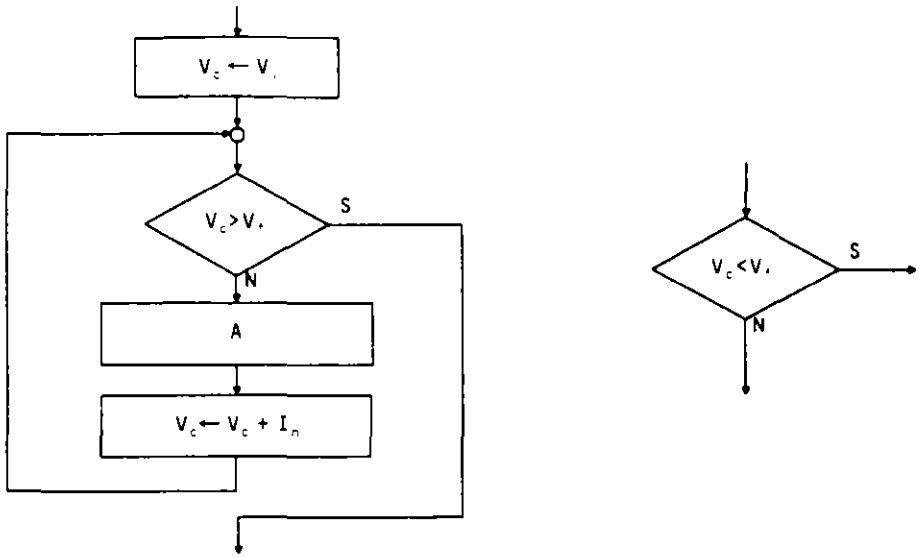
- **Estructura PARA (FOR)**

En esta estructura se repite una acción un número fijo de veces representado normalmente por N. Es necesario para el control de la repetición utilizar una variable de control V_c y los valores que asignaremos a la misma inicialmente V_i y su correspondiente valor final V_f. El incremento de la variable de control V_c es normalmente 1, pero puede tomar otros valores positivos y negativos, en cuyos casos es necesario indicarlo por medio de I_n.

El número de repeticiones N está dado por la fórmula

$$N = \text{parte entera}\left(\frac{V_f - V_i}{I_n}\right) + 1$$

Si $N = 0$ se obtiene un valor negativo, se dice que el bucle es inactivo y no se repite ninguna vez.



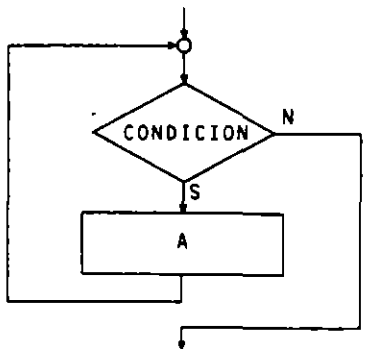
En este tipo de estructuras existen una serie de normas de obligado cumplimiento, como son:

— El I_n no puede ser 0 (bucle infinito).
— V_c no puede modificarse en el rango del bucle (acción A).

- **Estructura MIENTRAS (WHILE)**

En esta estructura se repite una acción mientras se cumpla la condición que controla el bucle. La característica principal de esta estructura es la de que la condición es evaluada siempre antes de cada repetición.

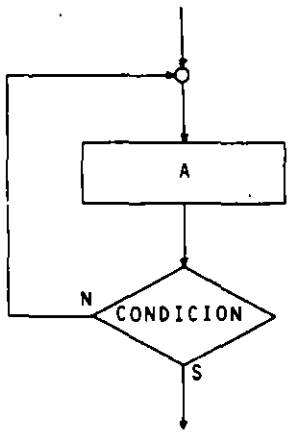
El número de repeticiones oscila entre 0 e infinito, dependiendo de la evaluación de la condición, cuyos argumentos en los casos de repetición, al menos una vez, deberán modificarse dentro del bucle, pues de no ser así el número de repeticiones será infinito y nos encontraremos en un bucle sin salida.



- **Estructura HASTA (UNTIL)**

En esta estructura se repite una acción hasta que se cumpla la condición que controla el bucle, la cual se evalúa después de cada ejecución del mismo.

El número de repeticiones oscila entre 1 e infinito, dependiendo de la evaluación de la condición, cuyos argumentos en los casos de repetición, al menos dos veces, deberán modificarse dentro del bucle, pues de no ser así el número de repeticiones será infinito y nos encontraremos en un bucle sin salida.

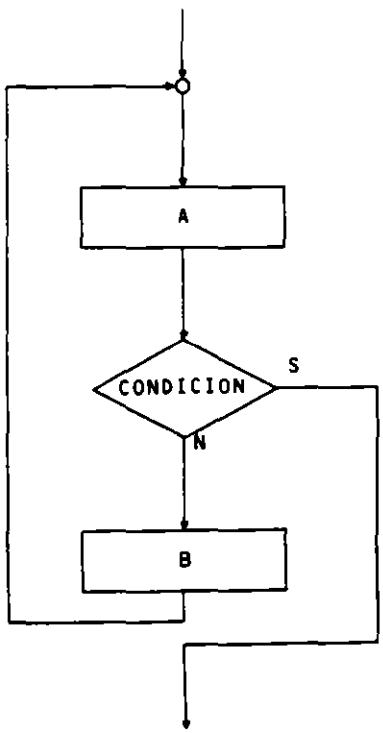


- **Estructura ITERAR (LOOP)**

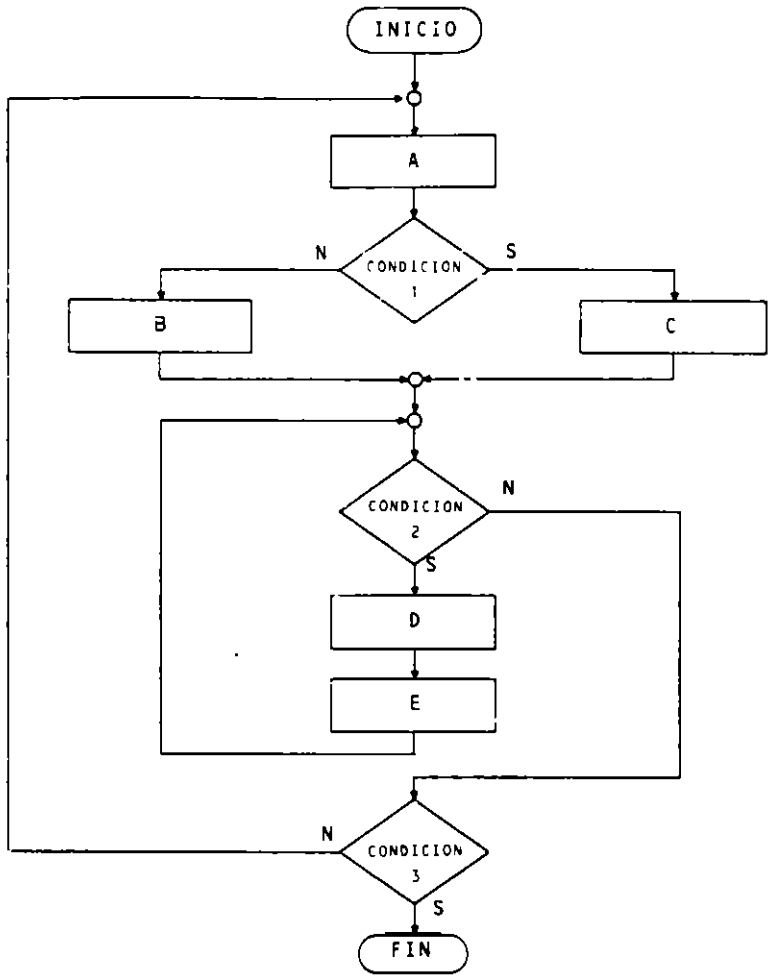
En esta estructura se repiten alternativamente dos acciones, evaluando la condición de salida entre ambas.

El número de repeticiones oscila, para la acción A, entre 1 e infinito, y para la acción B, entre 0 e infinito, cumpliéndose que siempre se repite A una vez más que B.

Los bucles anteriores son casos particulares de éste.



Ejemplo de ordinograma estructurado:



8.4. METODO DE WARNIER

Se trata de un método para la representación de programas cuyo resultado final se denomina diagrama de Warnier. En él podemos utilizar toda la terminología estudiada hasta ahora en lo que respecta a identificadores, constantes, variables, expresiones y operadores, teniendo en cuenta que la característica fundamental en relación con todo lo anteriormente visto es la forma de diseñar el programa, que será descendentemente, y la representación utilizada.

Este método se basa en el empleo de llaves de distintos tamaños que relacionan entre sí todas las tareas y operaciones.

La representación del algoritmo se basa en los siguientes puntos:

— Un programa se representa por un solo diagrama en el cual se engloban todas las operaciones necesarias para la resolución del problema. Estas operaciones están colocadas secuencialmente a la derecha de una llave, en cuya parte izquierda figura el nombre del programa.
— En la parte superior de la llave anterior figurará el comentario INICIO.
— En la parte inferior figurará FIN.
— La forma de conectar con distintas páginas es a través de la palabra PROCESO seguida de un número o un nombre que tenga relación con las operaciones que se realizan en la siguiente página. Estas palabras figurarán en el diagrama

principal (diagrama que ocupa la primera página). En las siguientes figurará un diagrama sujeto a las mismas normas, salvo que el nombre del programa será la palabra anteriormente citada. Las sucesivas conexiones se hacen de forma similar.

— Las estructuras tienen dos formas de representación, de las cuales utilizaremos la más sencilla.

■ Estructura secuencial

En esta estructura las acciones se sitúan a la derecha de la llave y desde arriba hacia abajo, según el orden de ejecución

```
{  A
(1 vez)

   B
(1 vez)

   C
(1 vez)
```

Como en la mayoría de los casos, en una estructura secuencial no aparecen acciones repetidas de forma consecutiva, la representaremos:

```
{A
 B
 C
```

■ Estructura alternativa

En esta estructura se ejecutará una acción dependiendo de la condición.

```
{ CONDICION  {A
  (0-1 vez)

  NO CONDICION  {B
  (0-1 vez)
```

También se puede representar de la forma:

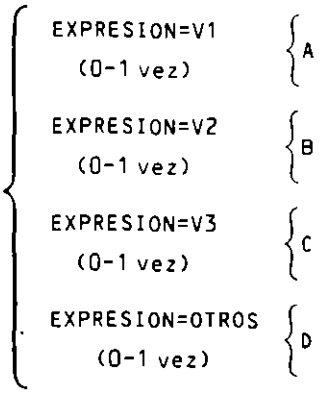
```
CONDICION { SI {A
          { NO {B
```

En los casos en que una de las acciones sea nula se representa mediante un guión (-).

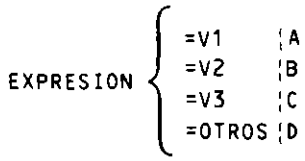
Por ejemplo, si en caso de cumplimiento de una condición se realiza una acción A y en caso contrario la acción nula (ninguna operación), la representación es:

```
CONDICION { SI {A
          { NO {-
```

La estructura alternativa múltiple se puede representar indistintamente por:



O también:

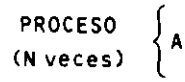


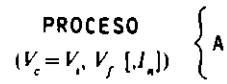
■ **Estructura repetitiva**

Como hemos visto anteriormente, esta estructura puede ser de varias formas, cuya representación en este método es la siguiente:

- **Estructura PARA (FOR)**

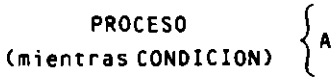
En esta estructura podemos indicar el número de repeticiones por *N* o a través de una variable de control, indicando sus valores inicial y final, así como el incremento cuando éste sea distinto de 1.





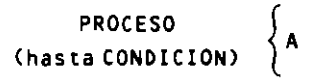
- **Estructura MIENTRAS (WHILE)**

En este caso la acción se repite mientras se cumpla la condición que se evaluará siempre antes de cada repetición.



- **Estructura HASTA (UNTIL)**

En este caso la acción se repite hasta que se cumpla la condición que se evaluará siempre después de cada repetición.



- **Estructura ITERAR (LOOP)**

En este caso se repiten dos acciones alternativamente hasta que se cumpla la condición que se evaluará siempre entre ambas acciones.

```
PROCESO { A
          (hasta CONDICION)
          B
```

Ejemplo: *Diagrama de Warnier de un algoritmo que lee cinco veces tres números, A, B y C, y los escribe ordenados ascendentemente, indicando en cada caso si los números fueron introducidos ordenados o no.*

```
PROGRAMA ORDENACION {
    INICIO
    PROCESO (5 veces) {
        SW ← FALSO
        leer A, B, C
        A > B { SI { AUX ← A
                     A ← B
                     B ← AUX
                     SW ← CIERTO
              { NO { -
        B > C { SI { AUX ← B
                     B ← B
                     C ← AUX
                     SW ← CIERTO
              { NO { -
        A > B { SI { AUX ← A
                     A ← B
                     B ← AUX
              { NO { -
        escribir A, B, C
        escribir "Los números"
        SW { SI { escribir "si"
           { NO { escribir "no"
        escribir "estaban ordenados"
    FIN
```

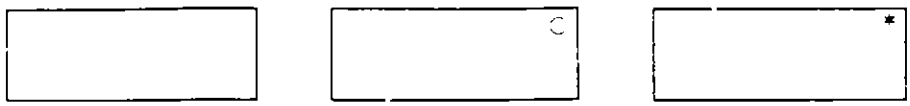
8.5. METODO DE JACKSON

Se trata de un método de representación de programas en forma de árbol denominado diagrama arborescente de Jackson. En él, al igual que en el método anterior, se puede utilizar la terminología común a todos los métodos de representación.

Un diagrama de Jackson consta de:

— Definición detallada de los datos de entrada y salida incluyendo los archivos lógicos utilizados.
— Representación del proceso o algoritmo.

La simbología utilizada se basa en el empleo de rectángulos horizontales, que pueden presentar los siguientes aspectos.



La lectura del diagrama se hace recorriendo el árbol en preorden (RID), lo que supone realizar:

— Situarse en la raiz (R).
— Recorrer el subárbol izquierdo (I).
— Recorrer el subárbol derecho (D).

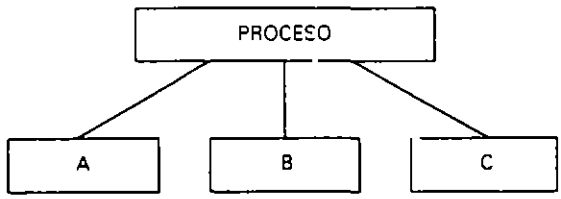
Cada subárbol se recorre igualmente en preorden hasta llegar a las hojas o nodos terminales del árbol.

La representación del algoritmo se basa en los siguientes puntos:

— Un programa se representa por un solo diagrama en el que se incluyen todas las operaciones a realiar para la resolución del problema. La forma de conectar una página con la siguiente es similar a los métodos anteriores, es decir, mediante la palabra PROCESO, seguida de un número o un nombre encerrados en un rectángulo.
— Todo diagrama comienza con un rectángulo en cuyo interior figura el nombre del programa.
— Para conseguir una clara interpretación del diagrama es necesario realizarlo de la forma más simétrica posible.

■ Estructura secuencial

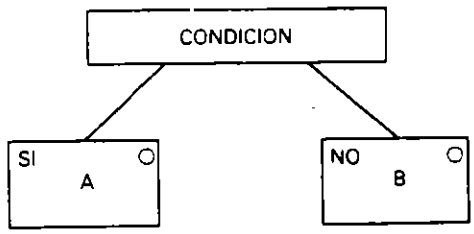
En esta estructura se ejecutan las acciones A, B y C de izquierda a derecha.



■ Estructura alternativa

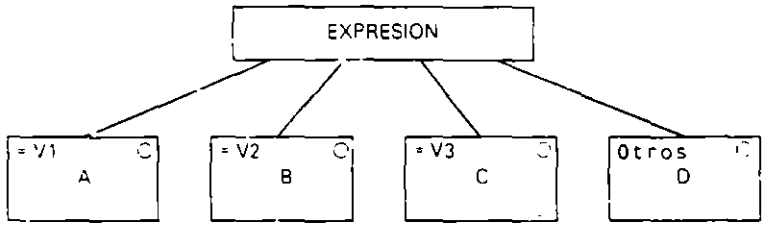
Se ejecuta una acción entre varias, según la evaluación de una condición.

- **Estructura alternativa doble:**



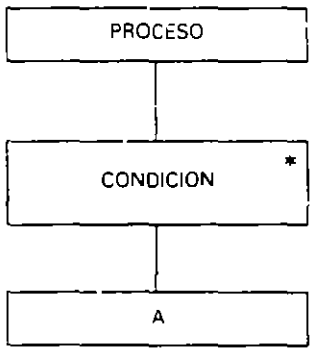
Si B es la acción nula se escribe un guión (—).

- **Estructura alternativa múltiple:**



Estructura repetitiva

Se repite una acción dependiendo de una condición de fin de repetición



- **Estructura PARA (FOR)**

La CONDICION puede ser:

— N veces
— $V_c = V_i, V_f\ [,I_n]$

- **Estructura MIENTRAS (WHILE)**

Se escribirá:

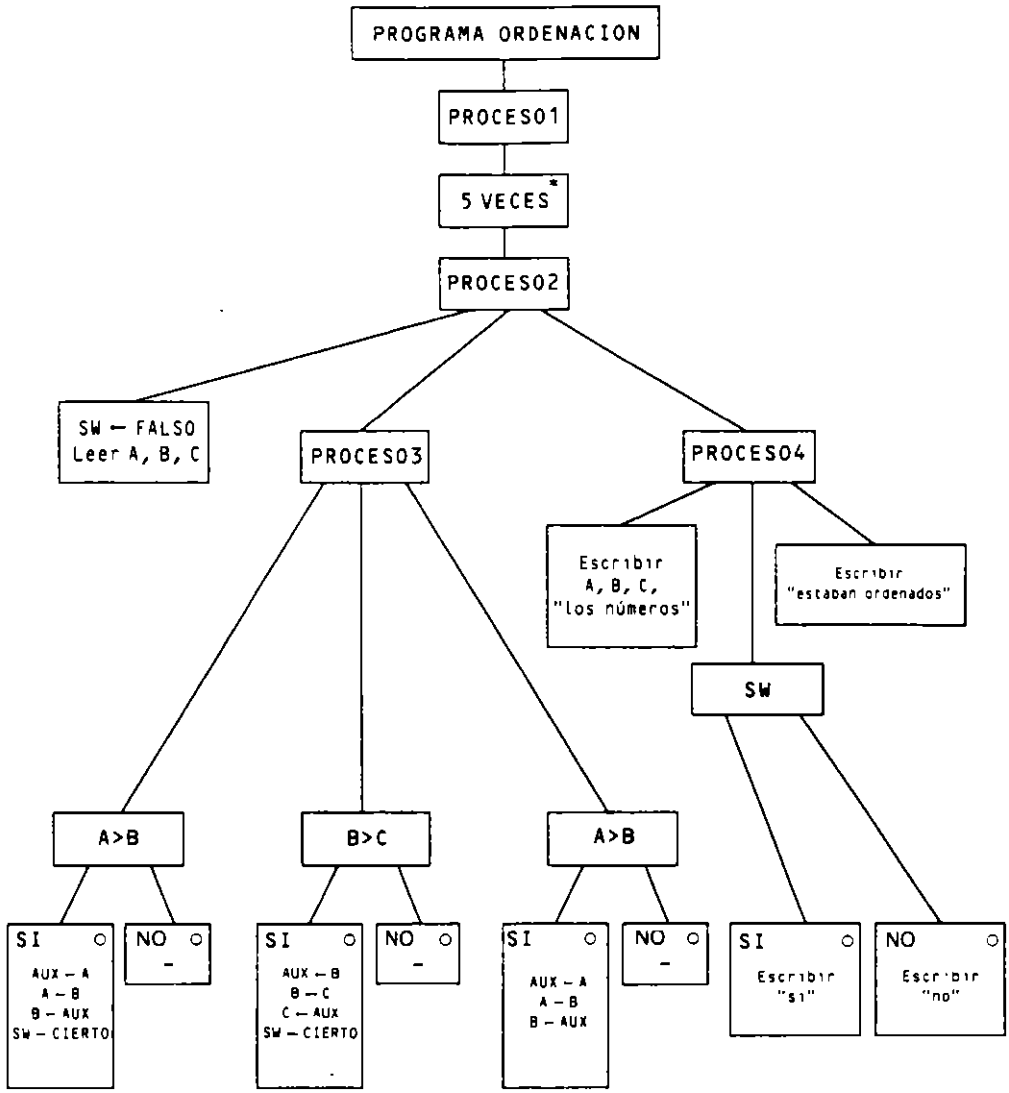
— Mientras CONDICION.

- **Estructura HASTA (UNTIL)**

Se escribirá:

— Hasta CONDICION.

Ejemplo: *Diagrama arborescente de Jackson que representa un algoritmo que lee cinco veces tres números y los escribe ordenados ascendentemente, indicando en cada ordenación si los números fueron introducidos ordenados o no.*



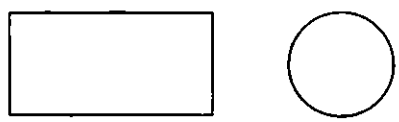
8.6. METODO DE BERTINI

Al igual que Jackson, la representación de programas es en forma de árbol denominado diagrama arborescente de Bertini. En este caso también se puede utilizar la terminología común ya estudiada.

Un diagrama de Bertini consta de:

— Definición detallada de los datos de entrada y salida, incluyendo los archivos lógicos utilizados.
— Representación del proceso o algoritmo.

La simbología utilizada se basa en el empleo de rectángulos horizontales y círculos.



La lectura del diagrama se hace recorriendo el árbol en orden inverso (RDI), lo que supone realizar:

— Situarse en la raíz (R).
— Recorrer el subárbol derecho (D).
— Recorrer el subárbol izquierdo (I).

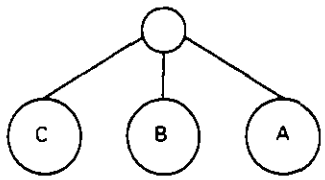
Cada subárbol se recorre igualmente en orden inverso hasta llegar a las hojas o nodos terminales del árbol.

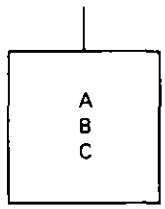
La representación del algoritmo se basa en los siguientes puntos:

— Un programa se representa por un solo diagrama en el que se incluyen todas las operaciones a realizar para la resolución del problema. La forma de conectar una página con la siguiente es similar a los métodos anteriores, es decir, mediante la palabra PROCESO seguida de un número o un nombre encerrados en un círculo.
— Todo diagrama comienza con un símbolo terminal, similar al de los ordinogramas, en cuyo interior figura el nombre del programa.

□ Estructura secuencial

En esta estructura se ejecutan las acciones A, B y C de derecha a izquierda en la primera representación, y de arriba abajo en la segunda, que se utiliza para una secuencia de acciones primitivas.

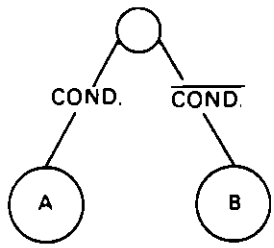




□ Estructura alternativa

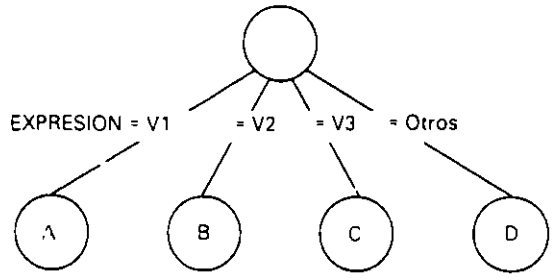
Se ejecuta una acción entre varias según la evaluación de una condición.

- **Estructura alternativa doble:**



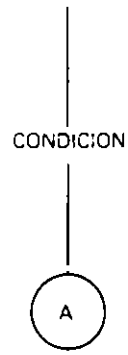
Si B es la acción nula se escribe un guión (−).

- **Estructura alternativa múltiple:**



■ **Estructura repetitiva**

Se repite una acción dependiendo de una condición de terminación.



- **Estructura PARA (FOR)**

La CONDICION puede ser:

— N veces.
— $V_c = V_i, V_f [, I_n]$

- **Estructura MIENTRAS (WHILE)**

Se escribirá:

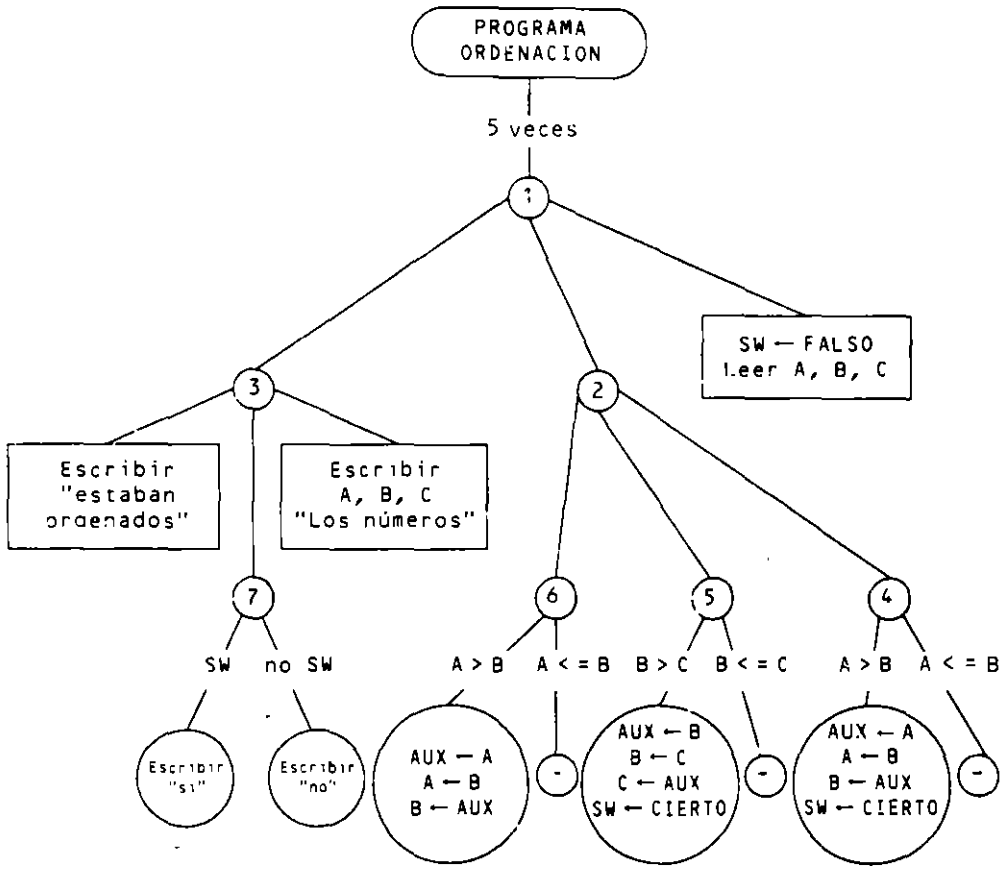
— Mientras CONDICION.

- **Estructura HASTA (UNTIL)**

Se escribirá:

— Hasta CONDICION.

Ejemplo: *Diagrama arborescente de Bertini que representa un algoritmo que lee cinco veces tres números A, B y C y los escribe ordenados ascendentemente, indicando en cada caso si los números fueron introducidos ordenados o no.*



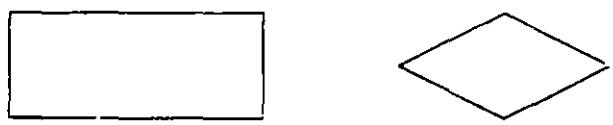
8.7. METODO DE TABOURIER

Se trata de una representación de programas en forma de árbol denominado diagrama de Tabourier. Para ello se puede utilizar asimismo la terminología común.

Un diagrama de Tabourier consta de:

— Definición detallada de los datos de entrada y salida, incluyendo los archivos lógicos utilizados.
— Representación del proceso o algoritmo.

La simbología utilizada se basa en el empleo de rectángulos y rombos horizontales.



La lectura del diagrama se hace recorriendo el árbol en preorden (RID), lo que supone realizar:

— Situarse en la raíz (R).
— Recorrer el subárbol izquierdo (I).
— Recorrer el subárbol derecho (D).

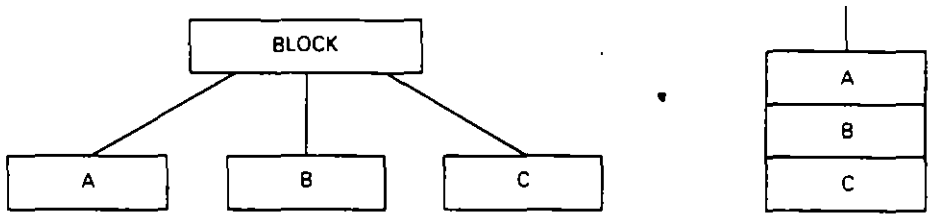
Cada subárbol se recorre igualmente en preorden hasta llegar a las hojas o nodos terminales del árbol.

La representación del algoritmo se basa en los siguientes puntos:

— Un programa se representa por un solo diagrama en el que se incluyen todas las operaciones a realizar para la resolución del problema. La forma de conectar una página con la siguiente es similar a los métodos anteriores, es decir, mediante la palabra PROCESO seguida de un número o un nombre encerrado en un rectángulo.
— Todo diagrama comienza con un rectángulo dividido horizontalmente, en el que en su parte superior figura el nombre del programa y en su parte inferior la palabra BLOCK.

■ Estructura secuencial

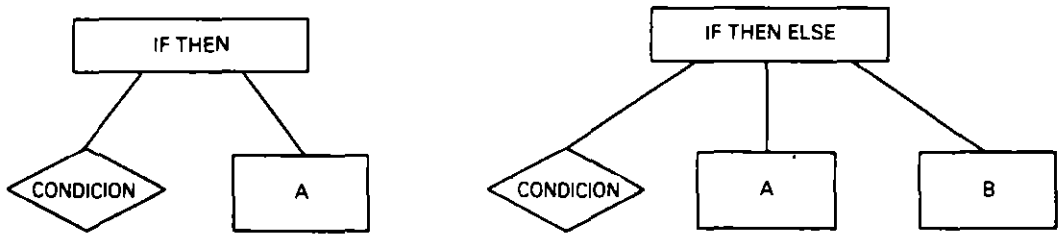
Las acciones A, B y C se ejecutan de izquierda a derecha o, si utilizamos la segunda representación (sólo para acciones primitivas), de arriba a abajo.



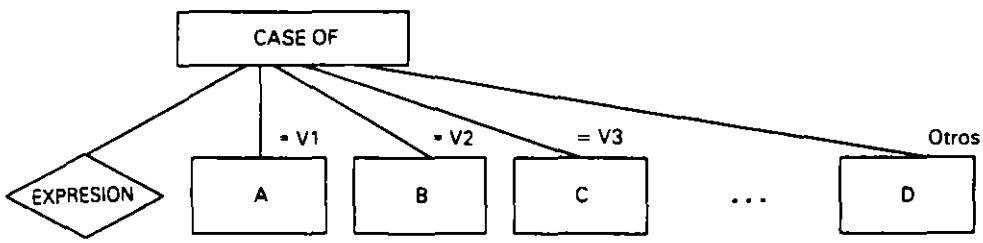
■ Estructura alternativa

Se ejecuta una acción entre varias según la evaluación de una condición.

- **Estructuras alternativas simple y doble:**



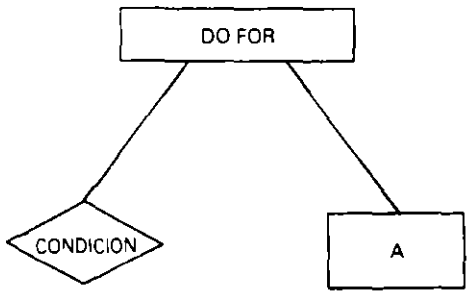
- **Estructura alternativa múltiple:**



■ **Estructura repetitiva**

Se repite una acción dependiendo de una condición de terminación

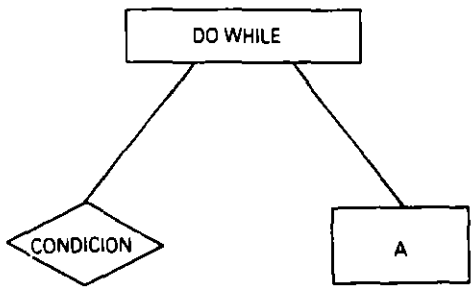
- **Estructura PARA (FOR)**



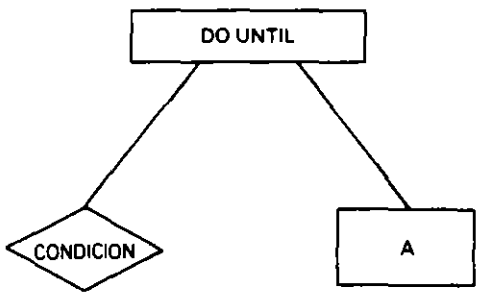
La CONDICION puede ser:

— N veces.
— $V_i = V_i, V_f [, I_n]$

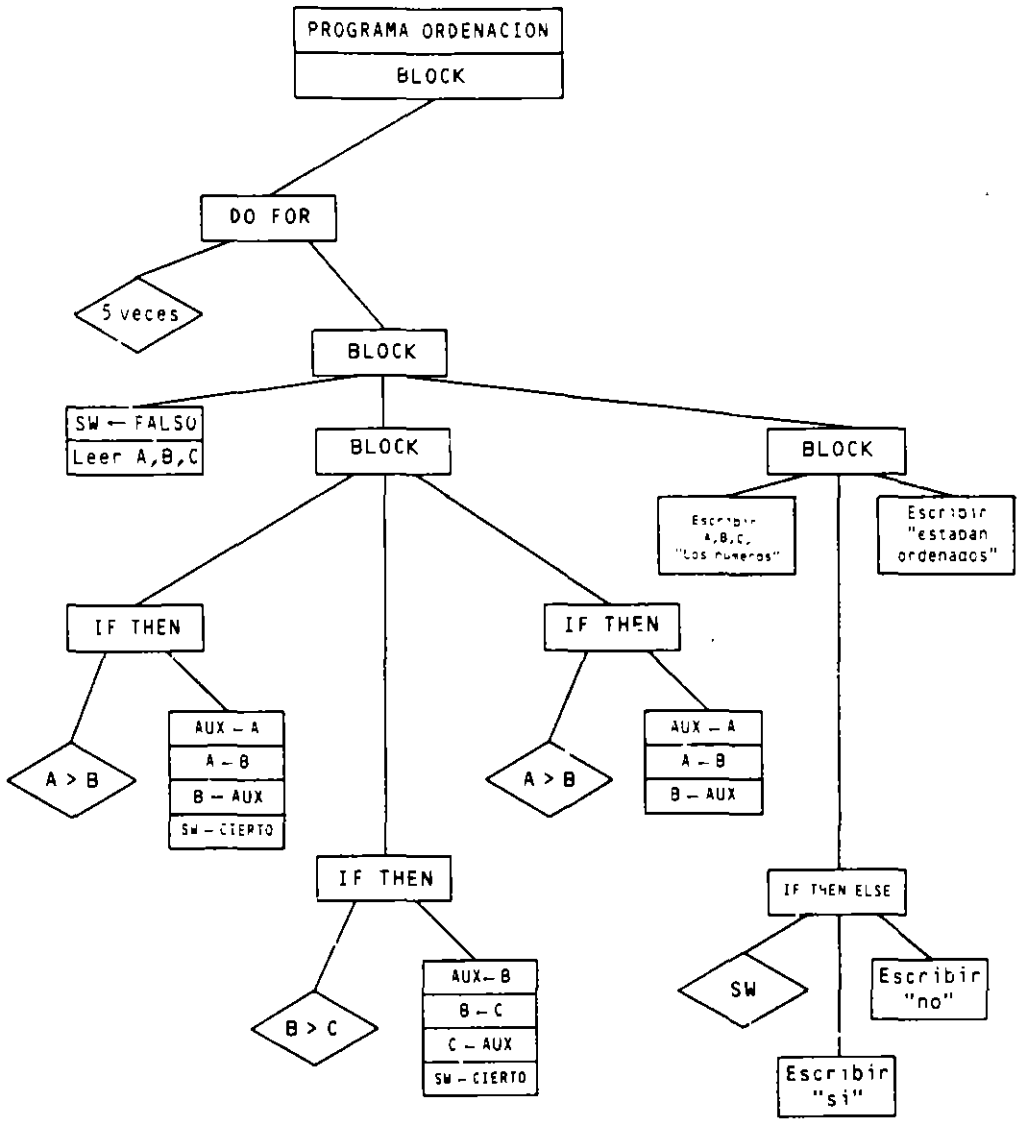
- **Estructura MIENTRAS (WHILE)**



- **Estructura HASTA (UNTIL)**



Ejemplo: *Diagrama de Tabourier que representa un algoritmo que lee cinco veces tres números, A, B y C, y los escribe ordenados ascendentemente, indicando en cada caso si los números fueron introducidos ordenados o no.*



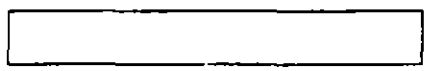
8.8. METODO DE CHAPIN (NASSI/SHNEIDERMAN)

Se trata de un método de representación de programas en forma de bloque compacto. También en este caso se puede utilizar la terminologia común.

Un diagrama de Chapin consta de:

— Definición detallada de los datos de entrada y salida incluyendo los archivos lógicos utilizados.
— Representación del proceso o algoritmo.

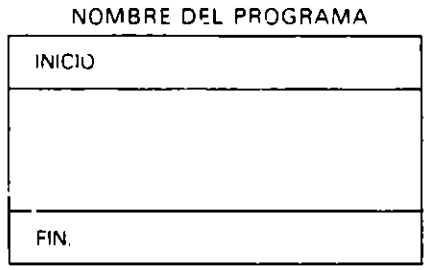
La simbología utilizada se basa en el empleo de rectángulos.



La lectura del diagrama se hace de arriba a abajo.

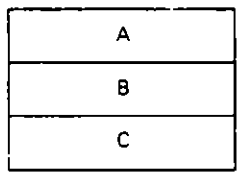
La representación del algoritmo se basa en los siguientes puntos:

— Un programa se representa por un solo diagrama en el que se incluyen todas las operaciones a realizar para la resolución del problema. La forma de conectar una página con la siguiente es similar a los métodos anteriores, es decir, mediante la palabra PROCESO seguida de un número o un nombre encerrados en uno de los rectángulos que componen el algoritmo.
— Todo diagrama comienza en un rectángulo en el que en su parte superior y fuera de el figura el nombre del programa. El rectángulo superior contiene la palabra INICIO y el inferior FIN.



■ Estructura secuencial

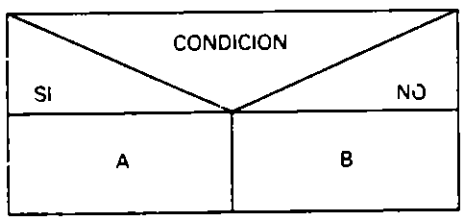
Las acciones A, B y C son ejecutadas de arriba a abajo.



■ Estructura alternativa

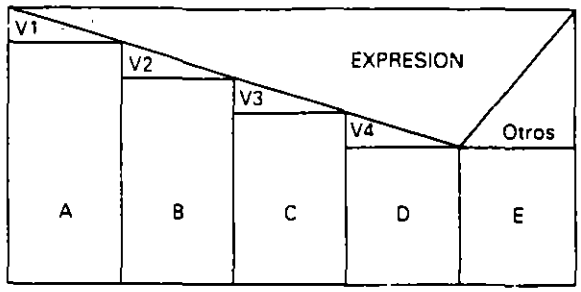
Se ejecuta una u otra acción según el resultado de la evaluación de una condición.

- **Estructura alternativa doble:**



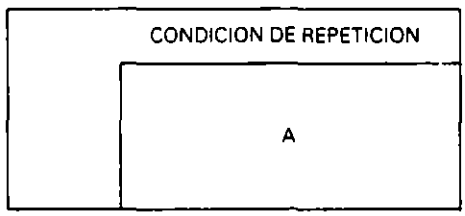
Si B es la acción nula se escribe un guión (—).

- **Estructura alternativa múltiple:**



■ **Estructura repetitiva**

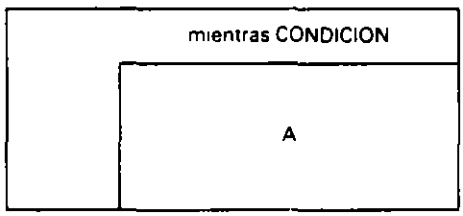
- **Estructura PARA (FOR)**



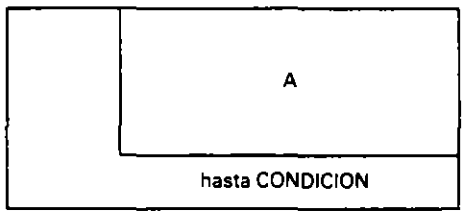
La CONDICION puede ser:

— N veces.
— $V_c = V_i, V_f [, I_n]$

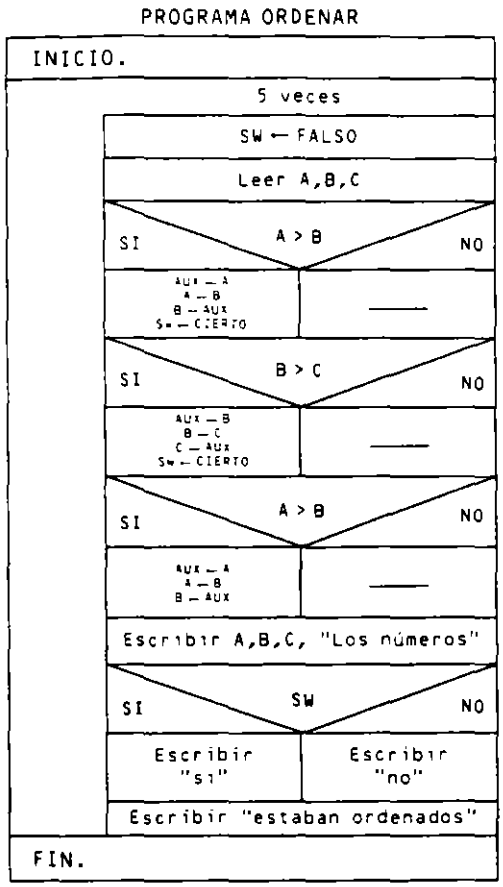
- **Estructura MIENTRAS (WHILE)**



- **Estructura HASTA (UNTIL)**



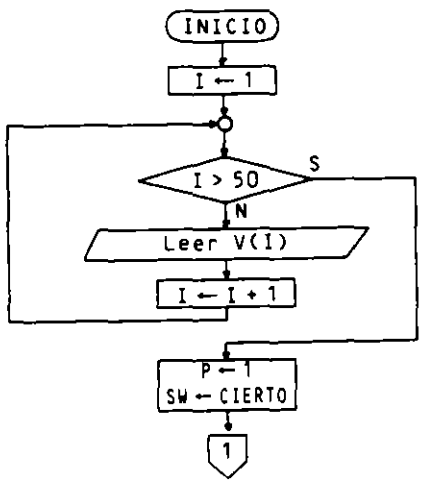
Ejemplo: *Diagrama de Chapin o N-S que representa un algoritmo que lee cinco veces tres números A, B y C y los escribe ordenados ascendentemente, indicando en cada caso si los números fueron introducidos ordenados o no.*

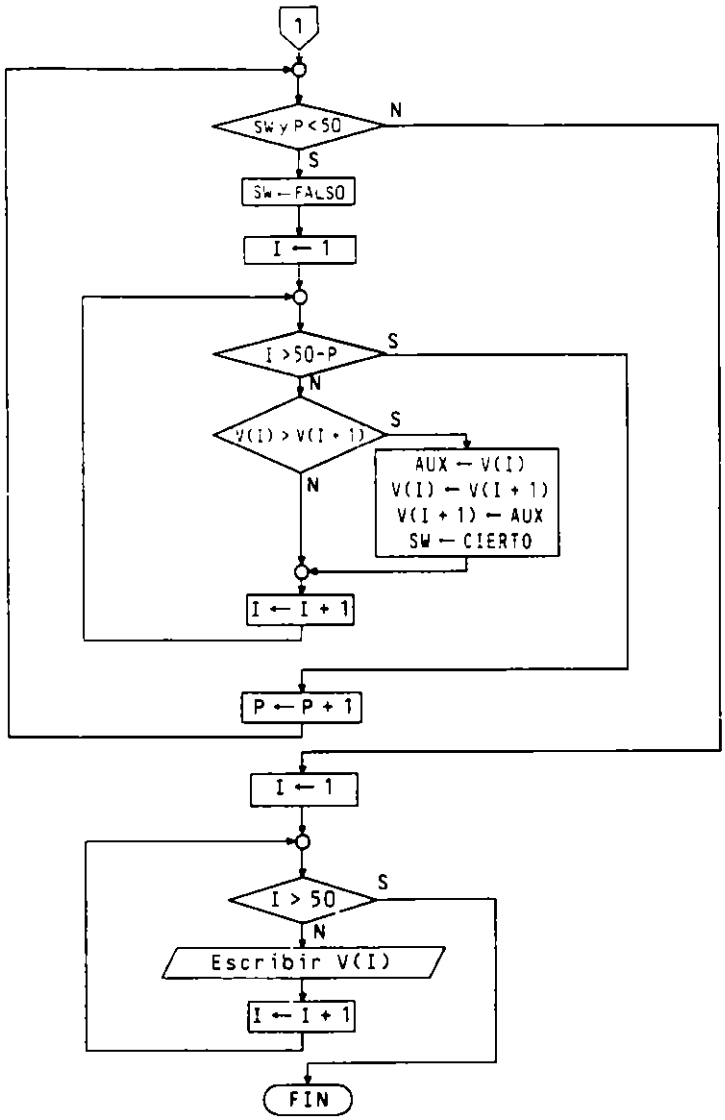


EJERCICIOS RESUELTOS

1. *Algoritmo que almacena una serie de 50 números en un vector V de 50 componentes, los ordena ascendentemente por el método de intercambio directo con* switch *y por último los escribe en dicho orden.*

a) Ordinograma:

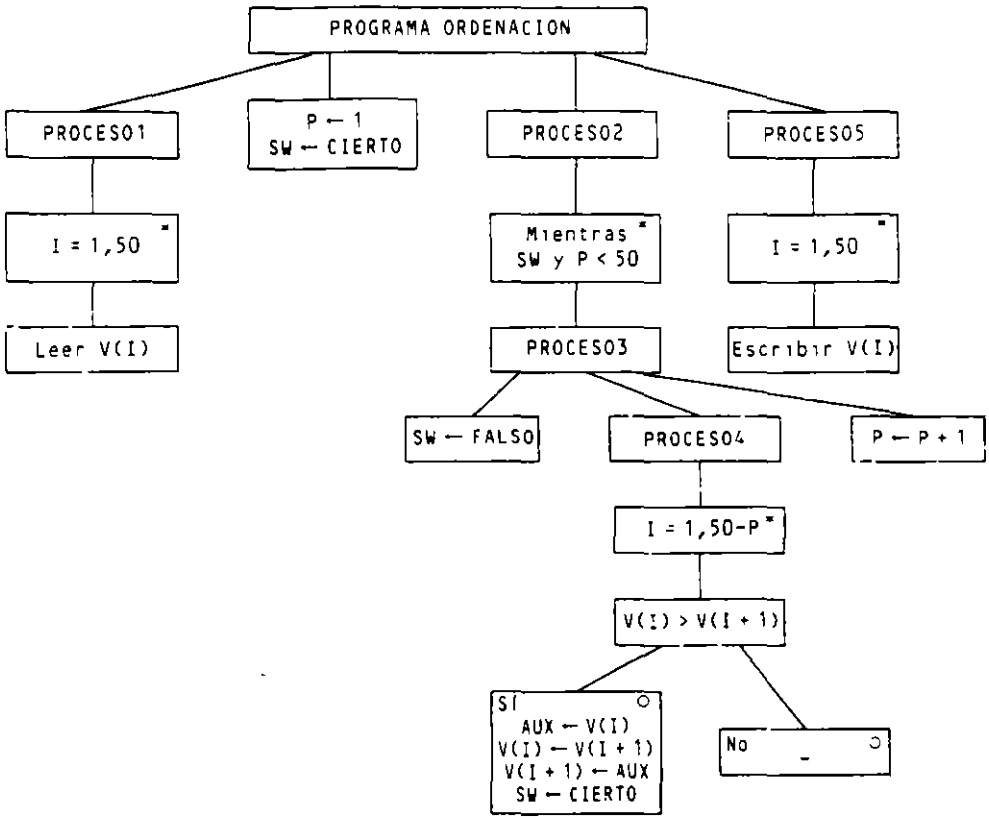




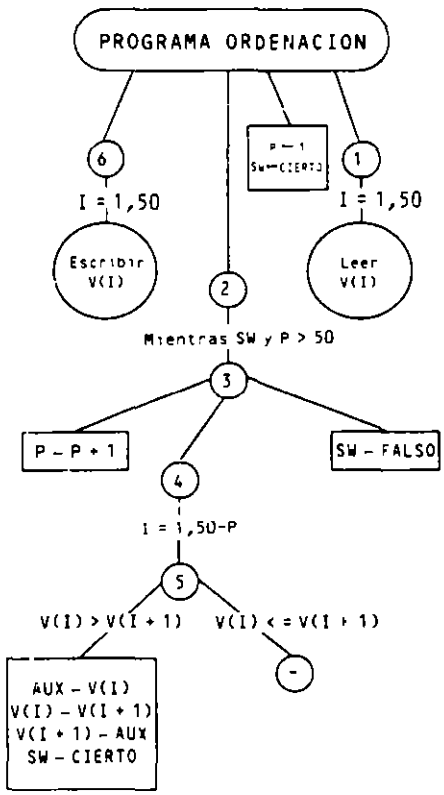
b) Diagrama de Warnier:

```
PROGRAMA ORDENACION
  INICIO
  PROCESO (I = 1,50)
    Leer V(I)
  P ← 1
  SW ← CIERTO
  Proceso (mientras SW y P<50)
    SW ← FALSO
    Proceso (I = 1,50-P)
      V(I)>V(I + 1)
        SI
          AUX ← V(I)
          V(I) ← V(I + 1)
          V(I + 1) ← AUX
          SW ← CIERTO
        NO
          -
    P ← P + 1
  PROCESO (I = 1,50)
    Escribir V(I)
  FIN
```

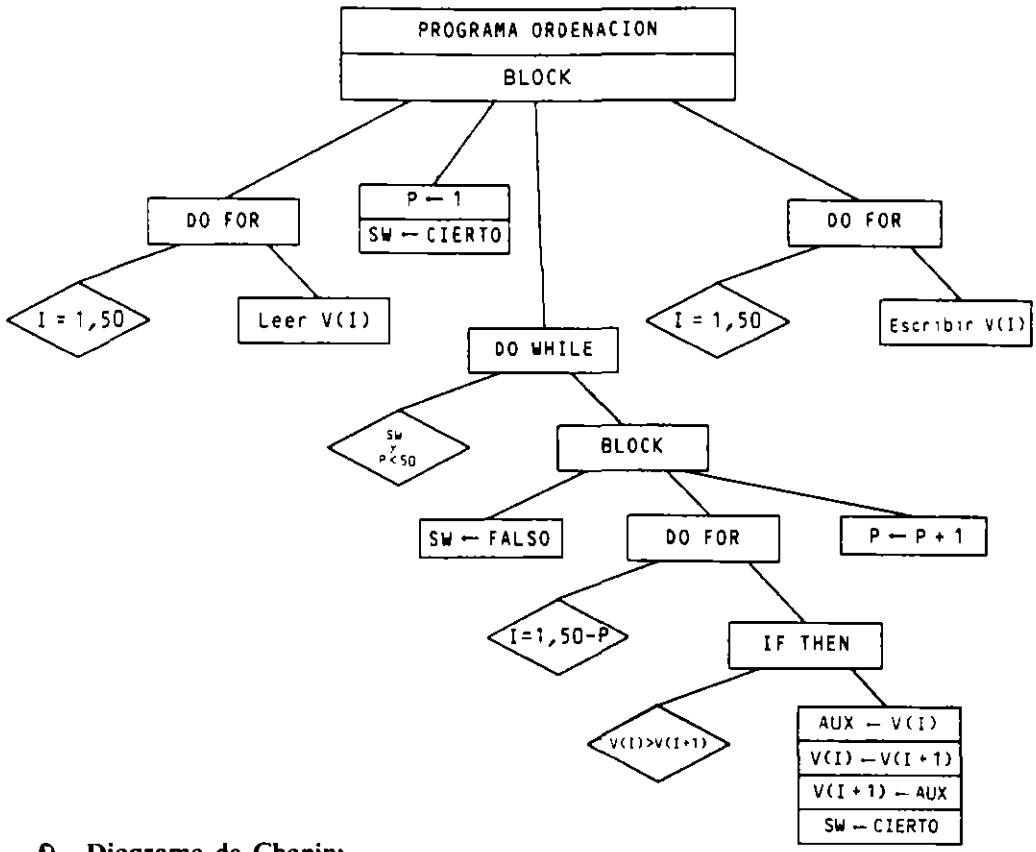
c) **Diagrama de Jackson:**



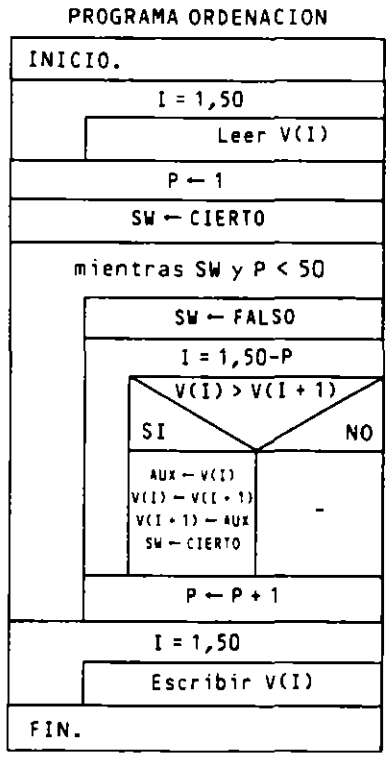
d) **Diagrama de Bertini:**



e) **Diagrama de Tabourier:**



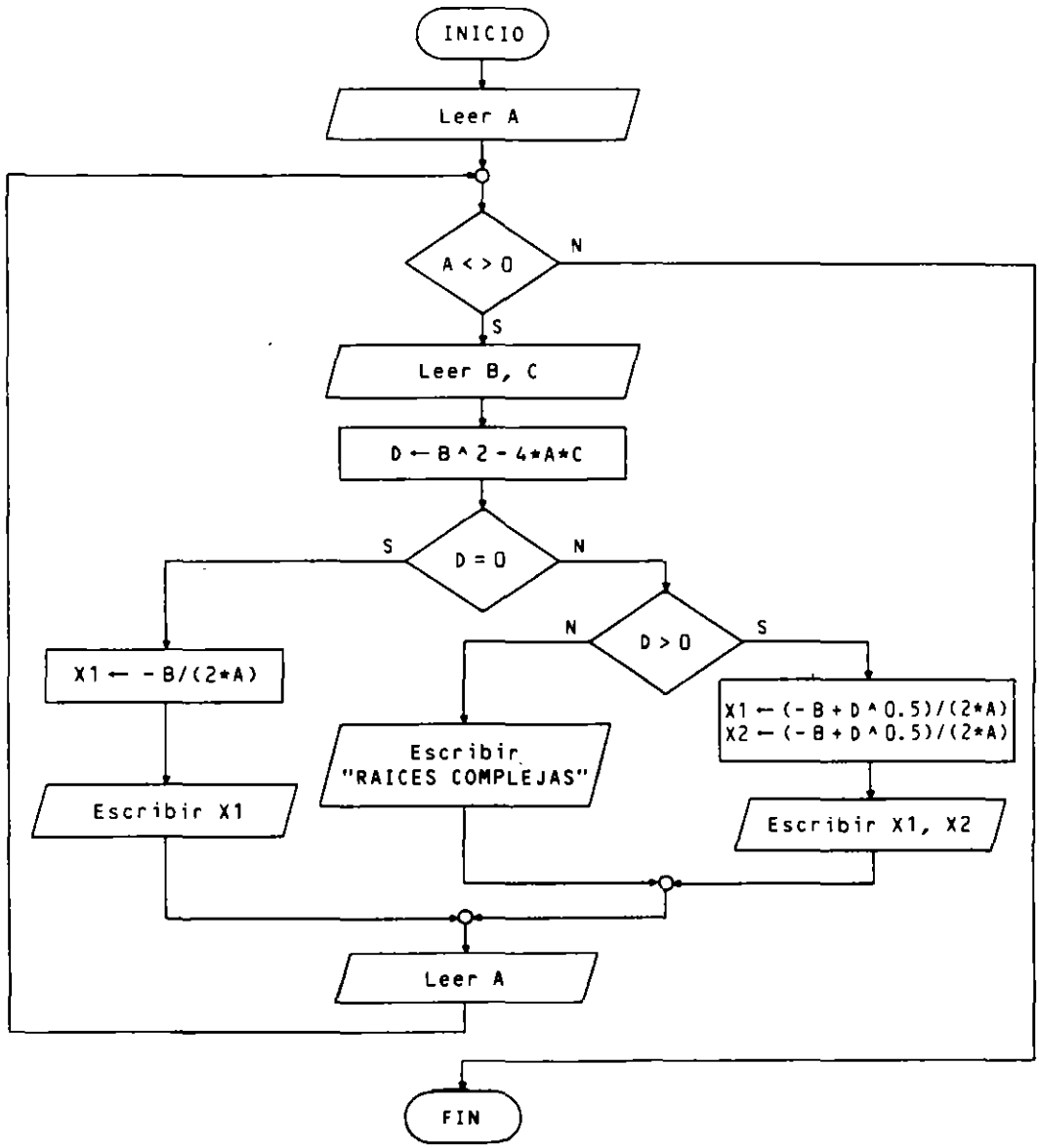
f) **Diagrama de Chapin:**



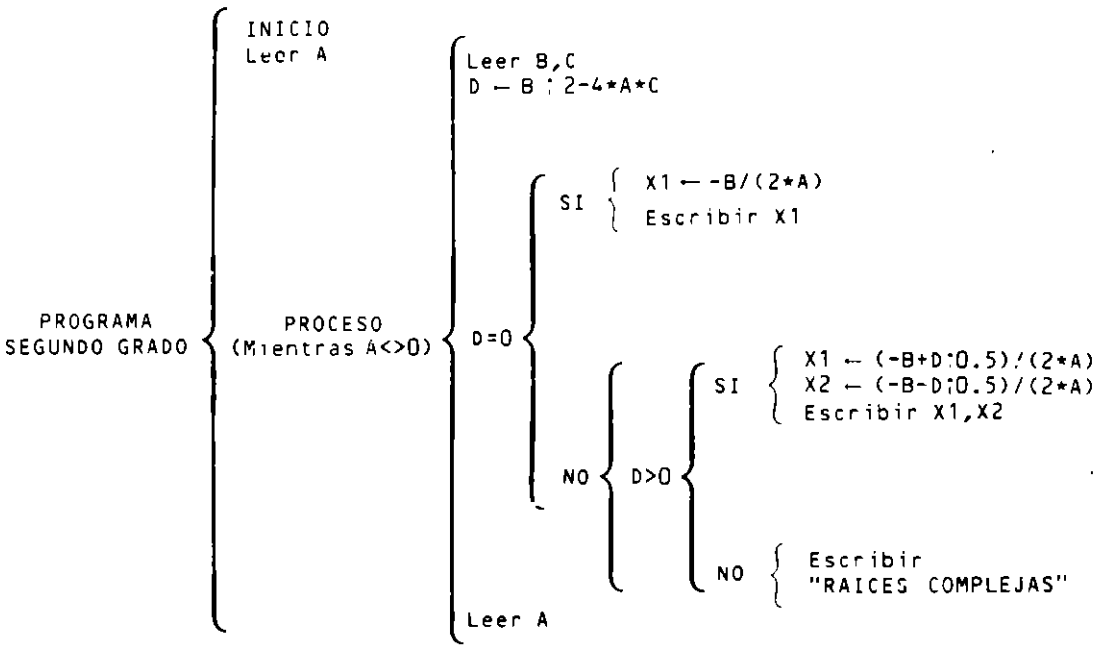
2. *Algoritmo que lee sucesivas ternas de valores A, B y C como coeficientes de ecuaciones de segundo grado y nos calcula y escribe para cada una de ellas el valor de sus raíces.*

   El final de los datos de entrada será cuando el coeficiente A valga 0.

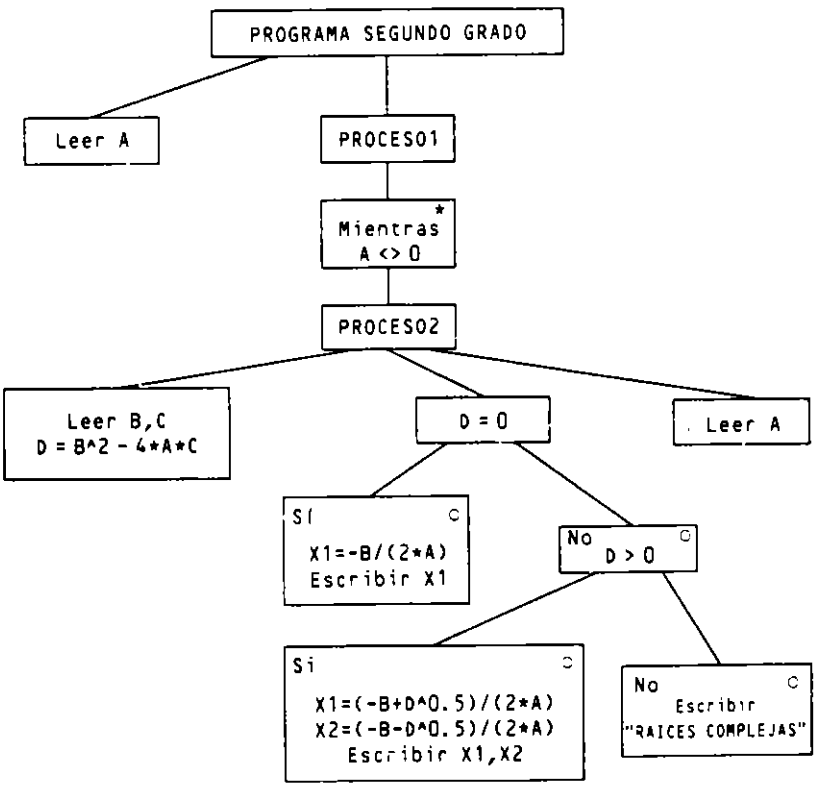
   a) Ordinograma:



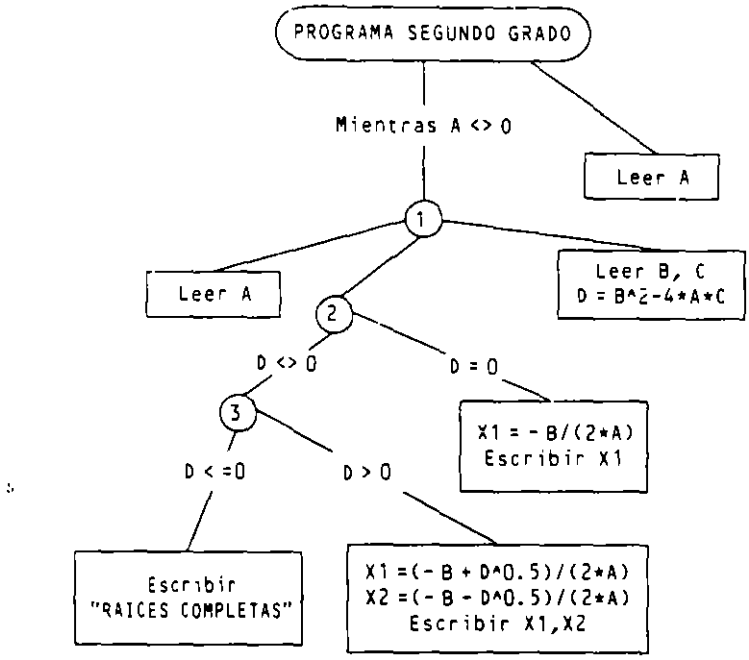
b) Diagrama de Warnier:



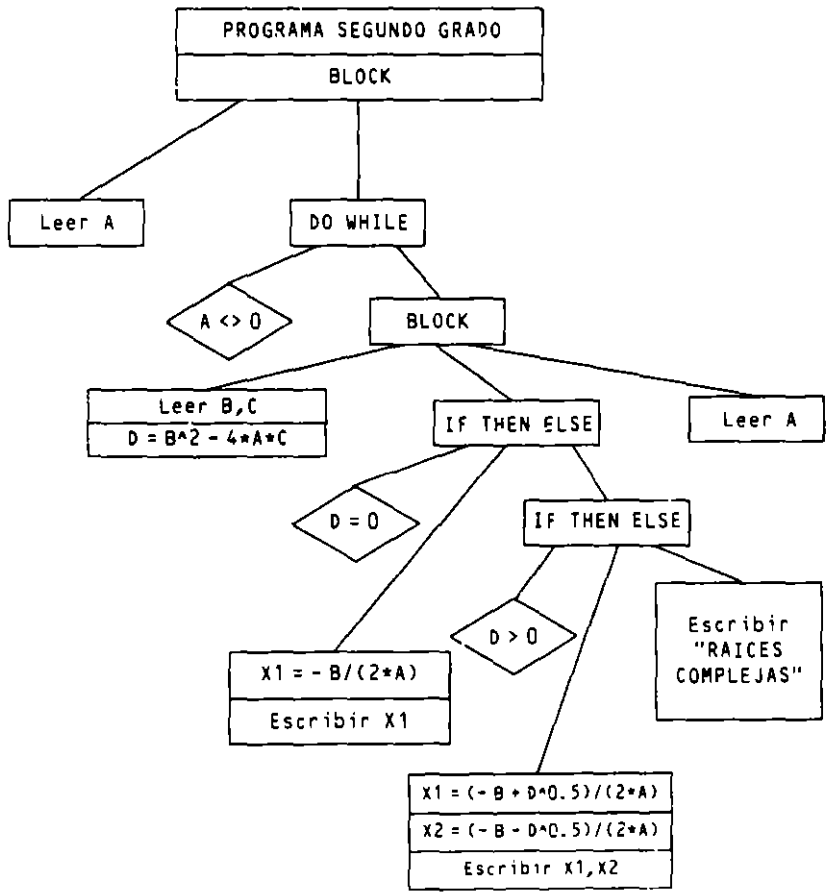
c) Diagrama de Jackson:



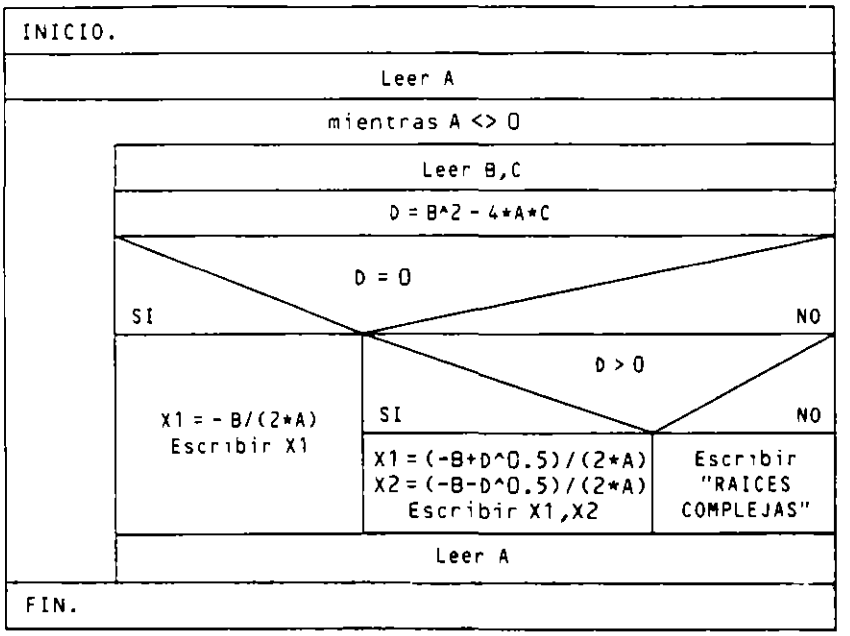
d) Diagrama de Bertini:



e) Diagrama de Tabourier:



f) Diagrama de Chapin:



EJERCICIOS PROPUESTOS

1. Representar por cualquiera de los métodos de diseño estructurado los ejercicios propuestos del Capítulo 4 que figuran con los números 3, 4, 5, 6, 7, 8, 9 y 10.
2. Representar por cualquiera de los métodos de diseño estructurado los ejercicios propuestos del Capítulo 5.
3. Representar por cualquiera de los métodos de diseño estructurado los ejercicios propuestos del Capítulo 6.
4. Representar por cualquiera de los métodos de diseño estructurado los ejercicios propuestos del Capítulo 7.

CAPITULO 9

Estructuras de datos externas (archivos)

9.1. INTRODUCCION

Los objetos tratados por un programa que hemos visto hasta ahora tienen dos limitaciones importantes. Por un lado, la cantidad de datos que pueden almacenar es bastante reducida por ser limitada la memoria central de la computadora. Además, su existencia está condicionada al tiempo que dure la ejecución del programa; es decir, cuando termina el programa, todos sus datos desaparecen de la memoria central.

Para abordar un aspecto importante de la Programación, que trata de la manipulación y almacenamiento de grandes cantidades de datos para futuros usos, se utilizan las estructuras de datos externas denominadas **ficheros** o **archivos**.

Los archivos no están contenidos en la memoria central de la computadora, sino que residen en soportes externos que establecen comunicación con ella al ser solicitada.

Su nombre corresponde al concepto clásico de conjunto de fichas que contienen información relativa a un mismo tema. Por ejemplo, el archivo de un hospital, que contiene los historiales clínicos de los enfermos, o el archivo de una biblioteca, que contiene información sobre los libros existentes en la misma.

Los soportes donde residen estos archivos pueden ser una carpeta, un armario, etc., existiendo algunas reglas o criterios de clasificación y manipulación.

Desde el punto de vista informático, un archivo es algo similar, residente en un soporte de información externo, como un disco o una cinta magnética.

Un archivo se compone de **registros** (equivalentes a las fichas), siendo éstos la unidad de acceso y de tratamiento.

Esta estructura es fundamental, debido a que nos permite almacenar cualquier tipo de información, como datos, textos, gráficos, programas, etc., y mantenerla durante todo el tiempo que sea necesaria.

En este capítulo estudiaremos la creación y manejo de archivos de datos, consistentes en un conjunto de datos homogéneos que contienen información relativa a un mismo tema.

9.2. CONCEPTOS Y DEFINICIONES

Un **archivo** o **fichero** es un conjunto de informaciones estructuradas en unidades de acceso denominadas registros, todos del mismo tipo.

Un **registro (registro lógico)** es una estructura de datos formada por uno o más elementos denominados **campos**, que pueden ser de diferentes tipos y que, a su vez, pueden estar compuestos por **subcampos**.

Los archivos contienen información relativa a un conjunto de individuos u objetos por regla general, estando ubicada la información correspondiente a cada uno de ellos en un registro.

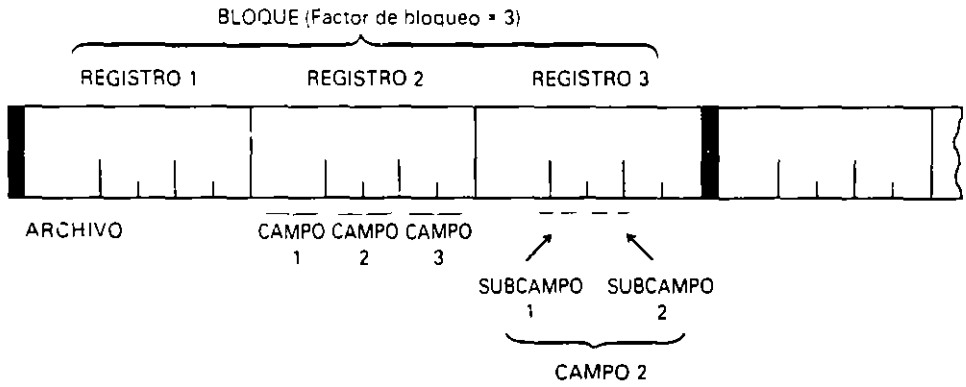
Denominamos registro **buffer** a un espacio de memoria interna que reserva el sistema para el intercambio de registros entre el archivo y el programa.

Se denomina **clave** o **identificativo** a un campo especial del registro que sirve para identificarlo.

Algunos archivos no tienen campo clave, mientras que otros pueden tener varios, denominándose los respectivos campos clave primaria, secundaria, etc.

Otro concepto relacionado con los archivos es el de **bloque (registro físico)**, correspondiente a la cantidad de información que se transfiere en cada operación de lectura o escritura sobre un archivo. Su tamaño depende de las características físicas de la computadora utilizada.

Se denomina **factor de bloqueo** al número de registros lógicos que contiene cada bloque.



9.3. CARACTERISTICAS DE LOS ARCHIVOS

Las principales características de esta estructura de datos son:

- Residencia en soportes de información externos, también denominados memorias secundarias o masivas, como son los discos y las cintas magnéticas.
- Independencia de las informaciones respecto de los programas. Es decir, la existencia de un archivo no está limitada al tiempo de ejecución del programa que lo crea, sino que permanece cuando éste termine, y, además, puede ser utilizado por otros programas en cualquier otro momento.
- Permanencia de la información almacenada. A diferencia de los datos almacenados en la memoria interna, que desaparecen cuando se desconecta la computadora, un archivo y la información contenida en el mismo no desaparece hasta que se borre explícitamente mediante una instrucción de programa u orden del sistema operativo.
- Portabilidad de los datos entre diferentes computadoras.
- Gran capacidad de almacenamiento, teóricamente ilimitada. Aunque la capacidad de un soporte es limitada, el tamaño del archivo no lo es, porque podría ocupar tantas unidades de soporte como fuesen necesarias. Un archivo que ocupa varias unidades de soporte, por ejemplo varios disquetes, se denomina **archivo multivolumen.**

9.4. CLASIFICACION DE LOS ARCHIVOS SEGUN SU USO

Los archivos se clasifican, según su uso, en tres grupos:

- **Archivos permanentes.** Contienen información que varía poco a lo largo del tiempo. Existen tres tipos de estos archivos:

- **Archivos de constantes.** La variación de su información es prácticamente nula, utilizándose principalmente para consultas; por ejemplo, un archivo con los datos geográficos de las provincias (situación, capital, ciudades, población, etc.).
- **Archivos de situación o maestros.** Su información refleja el estado o situación actual de una entidad, grupo o alguno de sus aspectos en un momento determinado. Estos archivos necesitan ser actualizados con cierta periodicidad. Un archivo de este tipo puede ser el que contiene la información sobre las mercancías existentes en un almacén.
- **Archivos históricos.** Son archivos de los tipos anteriores que se retiran del proceso para futuros usos estadísticos o consultas; por ejemplo, el archivo de situación de los préstamos de libros de una determinada biblioteca al final del año pasado.

■ **Archivos de movimientos.** Se crean para actualizar los archivos maestros. Sus registros, denominados movimientos o transacciones, son de tres tipos: **altas, bajas** y **modificaciones**.

Una vez realizado el proceso de actualización, el archivo pierde su utilidad y se hace desaparecer, para comenzar la creación de uno nuevo; por ejemplo, el archivo de los préstamos y devoluciones de libros realizados durante un día, que actualiza al maestro de libros al final del mismo día.

■ **Archivos de maniobra o trabajo.** Tienen una vida limitada, normalmente menor que la duración de la ejecución de un programa. Se utilizan como auxiliares de los anteriores. Por ejemplo, si se desea obtener un listado de ciudades, ordenado por población, se hará mediante un archivo auxiliar del geográfico que permita la ordenación previa al listado.

9.5. ORGANIZACION DE ARCHIVOS

Los archivos se organizan para su almacenamiento y acceso según las necesidades de las aplicaciones que los van a utilizar y el tipo de soporte utilizado.

Las tres principales organizaciones de archivos son:

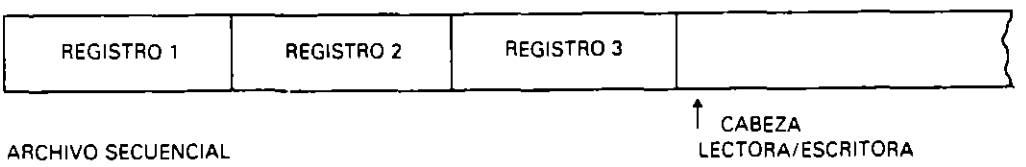
— Secuencial.
— Aleatoria o directa.
— Secuencial indexada.

9.5.1. ORGANIZACION SECUENCIAL

Es aquella en la cual los registros ocupan posiciones consecutivas de memoria y que sólo se puede acceder a ellos de uno en uno a partir del primero.

En un archivo secuencial no se pueden hacer operaciones de escritura cuando se está leyendo ni operaciones de lectura cuando se está escribiendo.

Por otro lado, para actualizarlos es preciso crear nuevos archivos donde se copien los antiguos junto con las actualizaciones.



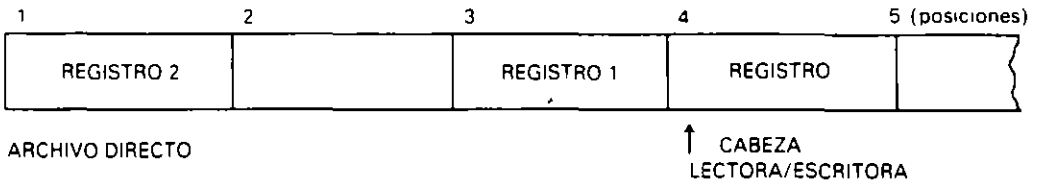
9.5.2. ORGANIZACION ALEATORIA O DIRECTA

Las informaciones se colocan y se acceden aleatoriamente mediante su posición, es decir, indicando el lugar relativo que ocupan dentro del conjunto de posiciones posibles.

En esta organización se pueden leer y escribir registros en cualquier orden y en cualquier lugar.

Presenta el inconveniente de que es tarea del programador establecer la relación entre la posición que ocupa un registro y su contenido; además, puede desaprovecharse parte del espacio destinado al archivo, ya que pueden quedar huecos libres entre unos registros y otros.

Su principal ventaja es la rapidez de acceso a un registro cualquiera, ya que para ello no es preciso pasar por los anteriores.



9.5.3. ORGANIZACION SECUENCIAL INDEXADA

Un archivo con esta organización consta de tres áreas:

— Area de índices.
— Area primaria.
— Area de excedentes (*overflow*).

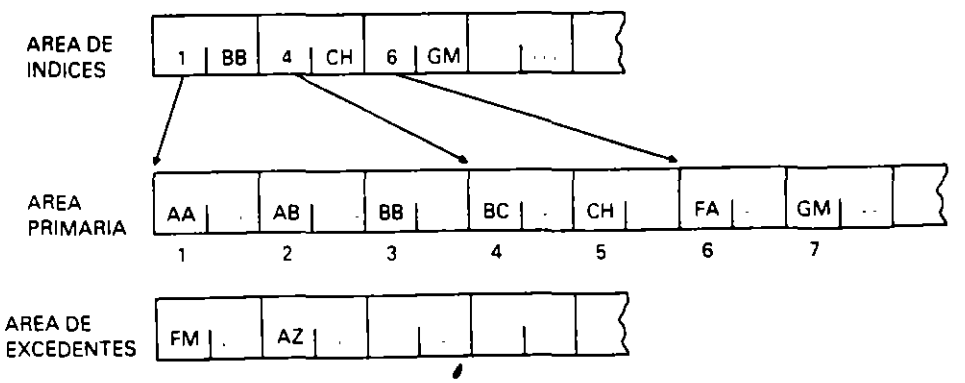
El área primaria contendrá los registros de datos, clasificados en orden ascendente por su campo clave.

El área de índices es un archivo secuencial creado por el sistema, en el que cada registro establece una división (segmento) en el área primaria, y contiene la dirección de comienzo del segmento y la clave más alta del mismo. De esta manera el sistema accede de forma directa a un segmento del área primaria a partir del área de índices, de forma similar a la búsqueda de un capítulo de un libro a partir de su índice.

Por último se reserva un espacio, llamado área de excedentes, para añadir nuevos registros que no pueden ser colocados en el área primaria cuando se produce una actualización del archivo.

Esta organización presenta la ventaja de un rápido acceso, y además el sistema se encarga de relacionar la posición de cada registro con su contenido por medio del área de índices. También es trabajo del sistema la gestión de las áreas de índices y excedentes.

Los inconvenientes que presenta son la necesidad de espacio adicional para el área de índices y el desaprovechamiento de espacio que resulta de quedar huecos intermedios libres después de sucesivas actualizaciones.



9.6. OPERACIONES SOBRE ARCHIVOS

Las operaciones generales que se realizan sobre un archivo son:

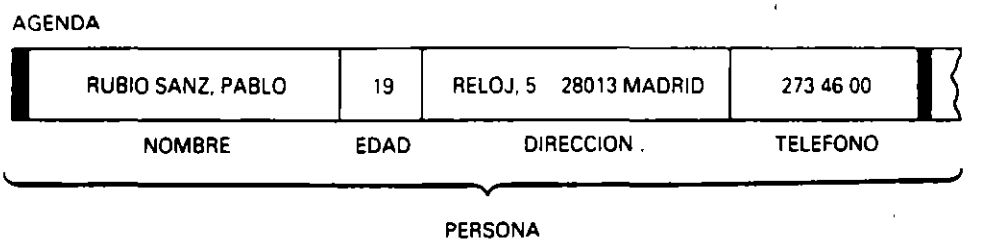
— **Creación.** Escritura de todos sus registros.
— **Copia.** Creación de un archivo cuyo contenido es idéntico al de otro ya existente.
— **Consulta.** Lectura de todos o de algunos de sus registros.
— **Actualización.** Inserción, supresión o modificación de algunos de sus registros.
— **Clasificación.** Reubicación de los registros de forma que queden ordenados según determinados criterios.
— **Concatenación.** Se obtiene un archivo a partir de otros dos del mismo tipo como resultado de colocar los registros del segundo a continuación de los del primero.
— **Mezcla o fusión.** Consiste en juntar todos los registros de dos o más archivos ordenados manteniendo la ordenación en el resultante.
— **Partición.** Consiste en la descomposición de un archivo en dos o más según algún criterio determinado.
— **Reorganización.** Los registros de un archivo que ha sido actualizado sucesivas veces se reubican para colocarlos de la mejor forma posible, aprovechando los posibles huecos que hubiese.
— **Borrado.** Eliminación total del archivo, dejando libre el espacio del soporte que ocupaba.

Las operaciones más usuales a nivel de registro son:

— **Inserción.** Añadir un nuevo registro al archivo.
— **Supresión.** Quitar un registro del archivo.
— **Modificación.** Alterar la información de un registro.
— **Consulta.** Leer el contenido de un registro.

9.7. INSTRUCCIONES PARA MANEJO DE ARCHIVOS

Utilizamos como ejemplo para el estudio de las instrucciones un archivo denominado AGENDA, cuyos registros contienen cuatro campos con el NOMBRE, EDAD, DIRECCION y TELEFONO de una serie de personas.



En primer lugar es preciso declarar el archivo, su nombre y la estructura de sus registros.

- **Pseudocódigo:**

```
AGENDA es archivo de PERSONA
PERSONA es registro compuesto de
            NOMBRE es alfanumérico
            EDAD es numerico entero
            DIRECCION es alfanumérico
            TELEFONO es alfanumérico
          finregistro
```

- **COBOL:** Dentro de la FILE SECTION de la DATA DIVISION.

```
FD  AGENDA.
01  PERSONA.
    05 NOMBRE    PIC X(20).
    05 EDAD      PIC 99.
    05 DIRECCION PIC X(30).
    05 TELEFONO  PIC X(12).
```

El archivo lógico AGENDA se asigna a un dispositivo físico en la ENVIRONMENT DIVISION:

```
SELECT AGENDA ASSIGN TO DISK, "AGENDA.DAT".
```

- **Pascal:**

```
TYPE RPERSONA = RECORD
                  NOMBRE    : PACKED ARRAY [1..20] OF CHAR;
                  EDAD      : INTEGER;
                  DIRECCION : PACKED ARRAY [1..30] OF CHAR;
                  TELEFONO  : PACKED ARRAY [1..12] OF CHAR
                END;
    FAGENDA = FILE OF RPERSONA;

VAR AGENDA  : FAGENDA;
    PERSONA : RPERSONA;
```

El archivo lógico AGENDA se asigna a un dispositivo físico con la siguiente instrucción, que se debe ejecutar antes de la apertura del mismo:

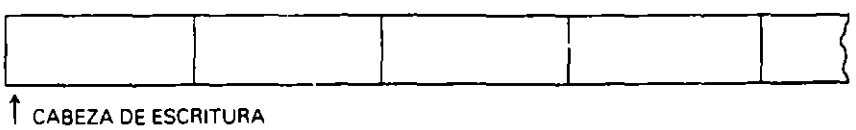
```
ASSIGN (AGENDA, 'AGENDA.DAT')
```

9.7.1. CREACION DE ARCHIVOS SECUENCIALES

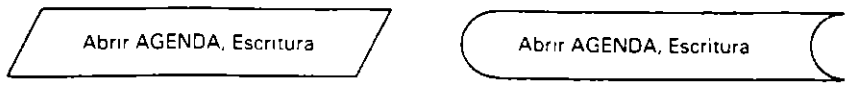
- **Apertura:**

Reserva un archivo secuencial en exclusividad para el programa que la ejecuta. Coloca la cabeza de escritura en su primer registro, quedando preparado para ser creado.

Si el archivo no existía, lo crea, y si ya existía, borra todo su contenido, salvo que el sistema disponga de protección para que esto último no ocurra.



- **Ordinograma:**



- **Pseudocódigo:**

```
abrir AGENDA para escritura
```

- **COBOL:**

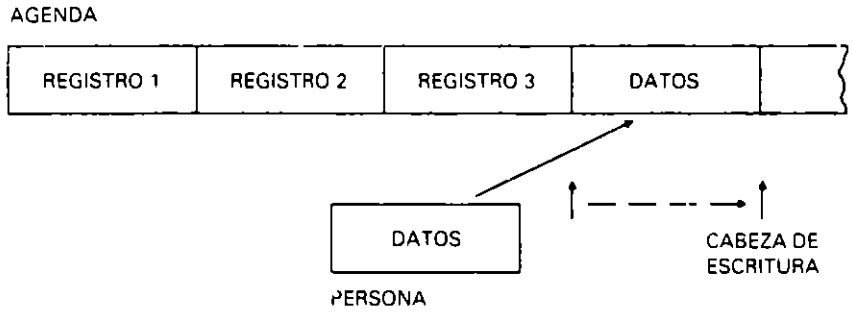
```
OPEN OUTPUT AGENDA
```

- **Pascal:**

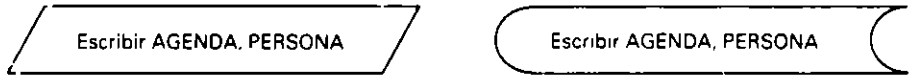
```
REWRITE (AGENDA)
```

- **Escritura de un registro:**

Coloca en el lugar donde esté la cabeza de escritura el contenido del registro «buffer», al que previamente se le habrán asignado los datos, avanzando la cabeza de escritura al siguiente registro del archivo.



- **Ordinograma:**



- **Pseudocódigo:**

```
escribir AGENDA, PERSONA
```

- **COBOL:**

```
WRITE PERSONA
```

- **Pascal:**

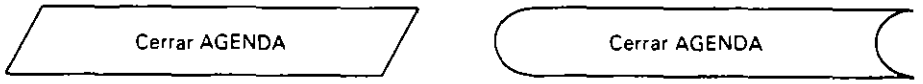
```
WRITE (AGENDA, PERSONA)
```

- **Cierre:**

Libera el archivo AGENDA del programa. Coloca una marca especial de «fin de archivo» que será detectada en posteriores lecturas. El archivo queda a disposición de cualquier programa que lo solicite.

AGENDA

- **Ordinograma:**



- **Pseudocódigo:**

```
cerrar AGENDA
```

- **COBOL:**

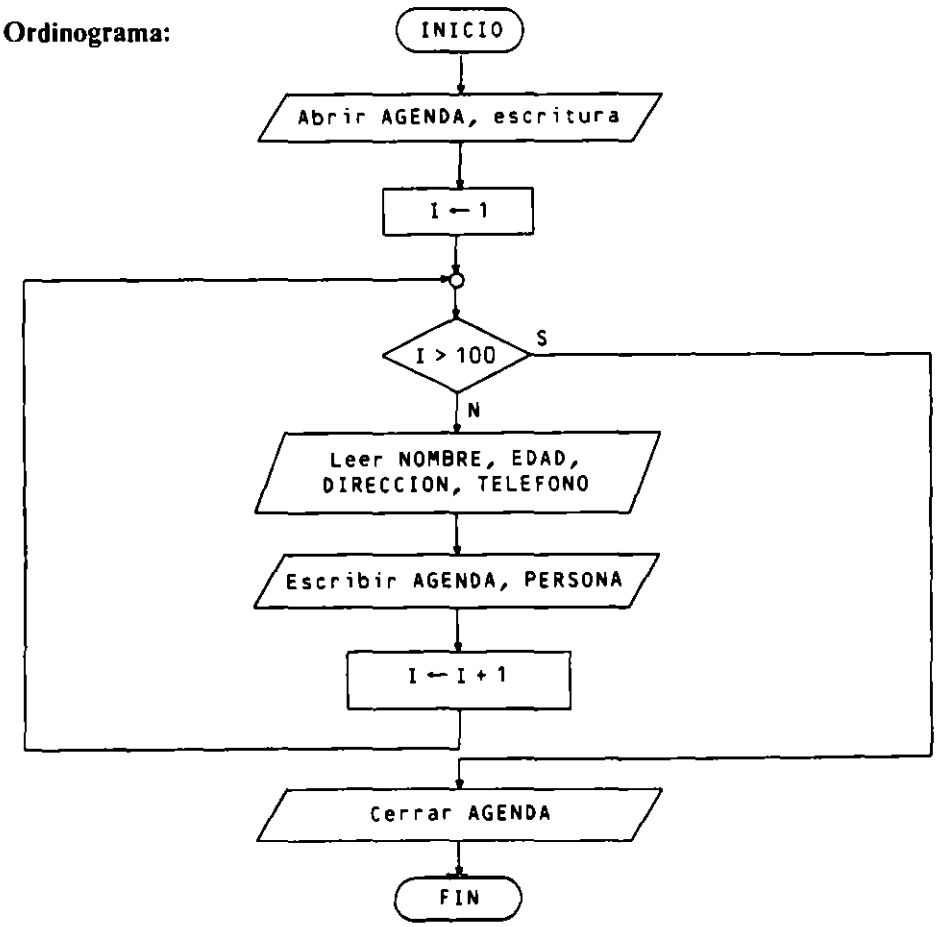
```
CLOSE AGENDA
```

- **Pascal:**

```
CLOSE (AGENDA)
```

Ejemplo: *Creación del archivo AGENDA en disco, con los datos de 100 personas, introducidos por teclado.*

- **Ordinograma:**



- **Pseudocódigo:**

```
Programa CREACION AGENDA
Entorno:
  AGENDA es archivo de PERSONA
  PERSONA es registro compuesto de
              NOMBRE es alfanumérico
              EDAD es numérico entero
              DIRECCION es alfanumérico
              TELEFONO es alfanumérico
            finregistro
  I es numérica entera
Algoritmo:
  abrir AGENDA para escritura
  para I de 1 a 100 hacer
    leer NOMBRE, EDAD, DIRECCION, TELEFONO
    escribir AGENDA, PERSONA
  finpara
  cerrar AGENDA
Finprograma
```

- **Codificación COBOL:**

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID. CREACION-AGENDA.
*
 ENVIRONMENT DIVISION.
 INPUT-OUTPUT SECTION.
 FILE-CONTROL.
     SELECT AGENDA ASSIGN TO DISK, "AGENDA.DAT".
*
 DATA DIVISION.
 FILE SECTION.
 FD  AGENDA.
 01  PERSONA.
     05 NOMBRE    PIC X(20).
     05 EDAD      PIC 99.
     05 DIRECCION PIC X(30).
     05 TELEFONO  PIC X(12).
 WORKING-STORAGE SECTION.
 01  I PIC 999.
*
 PROCEDURE DIVISION.
 PROCESO.
     OPEN OUTPUT AGENDA
     PERFORM VARYING I FROM 1 BY 1 UNTIL I > 100
       DISPLAY "Datos de la persona núm.", I
       PERFORM LEER-DATOS
       WRITE PERSONA
     END-PERFORM
     CLOSE AGENDA
     STOP RUN.
*
 LEER-DATOS.
     DISPLAY "Nombre: " NO ADVANCING
     ACCEPT NOMBRE
     DISPLAY "Edad: " NO ADVANCING
     ACCEPT EDAD
     DISPLAY "Dirección: " NO ADVANCING
```

```
ACCEPT DIRECCION
DISPLAY "Teléfono: " NO ADVANCING
ACCEPT TELEFONO.
```

- **Codificación Pascal:**

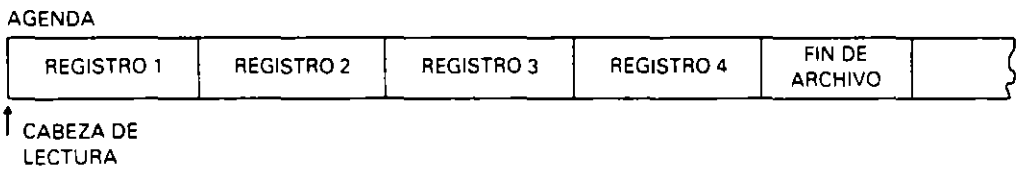
```
PROGRAM CREACION_AGENDA (INPUT, OUTPUT, AGENDA);
USES CRT; (* Para poder utilizar CLRSCR *)
CONST
  LONG_NOMBRE = 20;
  LONG_DIR    = 30;
  LONG_TEL    = 12;
TYPE
  TIPO_NOMBRE  = PACKED ARRAY [1..LONG_NOMBRE] OF CHAR;
  TIPO_DIR     = PACKED ARRAY [1..LONG_DIR] OF CHAR;
  TIPO_TEL     = PACKED ARRAY [1..LONG_TEL] OF CHAR;
  TIPO_PERSONA = RECORD
                   NOMBRE    : TIPO_NOMBRE;
                   EDAD      : INTEGER;
                   DIRECCION : TIPO_DIR;
                   TELEFONO  : TIPO_TEL
                 END;
  TIPO_AGENDA = FILE OF TIPO_PERSONA;
VAR
  AGENDA  : TIPO_AGENDA;
  PERSONA : TIPO_PERSONA;
  I : INTEGER;
(**)
PROCEDURE LEER_DATOS (VAR PERSONA: TIPO_PERSONA);
  VAR
    J: INTEGER;
  BEGIN (* LEER_DATOS *)
    WITH PERSONA DO
      BEGIN (*1*)
        WRITE ('Nombre: ');
        J := 0;
        WHILE NOT EOLN AND (J < LONG_NOMBRE) DO
          BEGIN (*2*)
            J := J + 1;
            READ (NOMBRE[J])
          END;  (*2*)
        READLN;
        FOR J := J + 1 TO LONG_NOMBRE DO NOMBRE[J] := ' ';
        WRITE ('Edad: ');
        READLN (EDAD);
        WRITE ('Dirección: ');
        J := 0;
        WHILE NOT EOLN AND (J < LONG_DIR) DO
          BEGIN (*3*)
            J := J + 1;
            READ (DIRECCION[J])
          END;  (*3*)
        READLN;
        FOR J := J + 1 TO LONG_DIR DO DIRECCION[J] := ' ';
        WRITE ('Teléfono: ');
        J := 0;
        WHILE NOT EOLN AND (J < LONG_TEL) DO
          BEGIN (*4*)
            J := J + 1;
            READ (TELEFONO[J])
```

```
          END;  (*4*)
        READLN;
        FOR J := J + 1 TO LONG_TEL DO TELEFONO[J] = ' '
      END (*1*)
  END;  (* LEER_DATOS *)
(**)
BEGIN (* CREACION_AGENDA *)
  ASSIGN (AGENDA, 'AGENDA.DAT');
  REWRITE (AGENDA);
  FOR I := 1 TO 100 DO
    BEGIN (*5*)
      CLRSCR;
      WRITELN ('Datos de la persona núm. ', I);
      LEER_DATOS (PERSONA);
      WRITE (AGENDA, PERSONA)
    END;  (*5*)
  CLOSE (AGENDA)
END.  (* CREACION_AGENDA *)
```

9.7.2. LECTURA DE ARCHIVOS SECUENCIALES

- **Apertura:**

Reserva un archivo secuencial en exclusividad para un programa. Coloca la cabeza lectora sobre el primer registro, quedando el archivo preparado para ser leído.



- **Ordinograma:**

- **Pseudocódigo:**

```
abrir AGENDA para lectura
```

- **COBOL:**

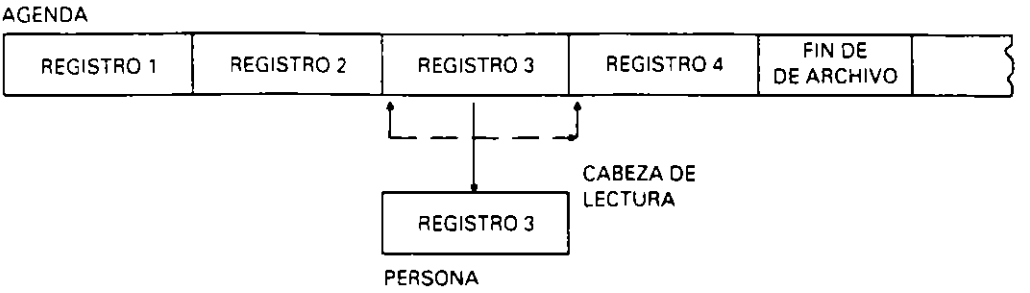
```
OPEN INPUT AGENDA
```

- **Pascal:**

```
RESET (AGENDA)
```

- **Lectura de un registro:**

Coloca el contenido del registro apuntado por la cabeza lectora sobre la variable «buffer» del archivo, avanzando la cabeza lectora al siguiente registro.



- **Ordinograma:**

- **Pseudocódigo:**

```
Leer AGENDA, PERSONA
```

- **COBOL:**

```
READ AGENDA
```

- **Pascal**

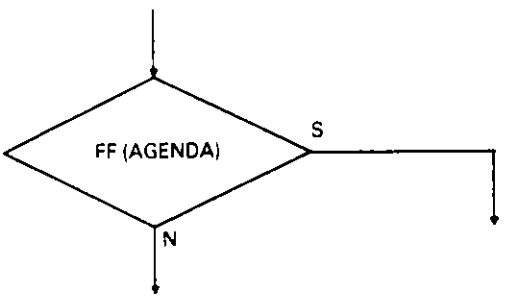
```
READ (AGENDA, PERSONA)
```

- **Comprobación de final de archivo:**

Es una función booleana que toma el valor CIERTO si la cabeza lectora señala la marca especial de fin de archivo, es decir, si no quedan registros por leer, y toma el valor FALSO en caso contrario.

Siempre habrá que hacer esta comprobación antes de leer un nuevo registro, pues si se ejecuta una instrucción de lectura cuando no quedan registros por leer se producirá un error y se interrumpirá la ejecución del programa.

- **Ordinograma:**



- **Pseudocódigo:**

```
ff (AGENDA)
```

- **COBOL:**

Se realiza dentro de la instrucción de lectura mediante la cláusula opcional «AT END», que permite incluir las instrucciones que se han de realizar si se da la condición de fin de archivo. Asimismo, mediante la cláusula «NOT AT END», se pueden incorporar en la lectura las instrucciones que se han de realizar si no se ha detectado el final de archivo y, por tanto, se ha leído un registro.

```
READ AGENDA
  AT END instrucciones en caso de fin de archivo
  NOT AT END instrucciones en caso de lectura correcta
END-READ
```

- **Pascal:**

```
EOF (AGENDA)
```

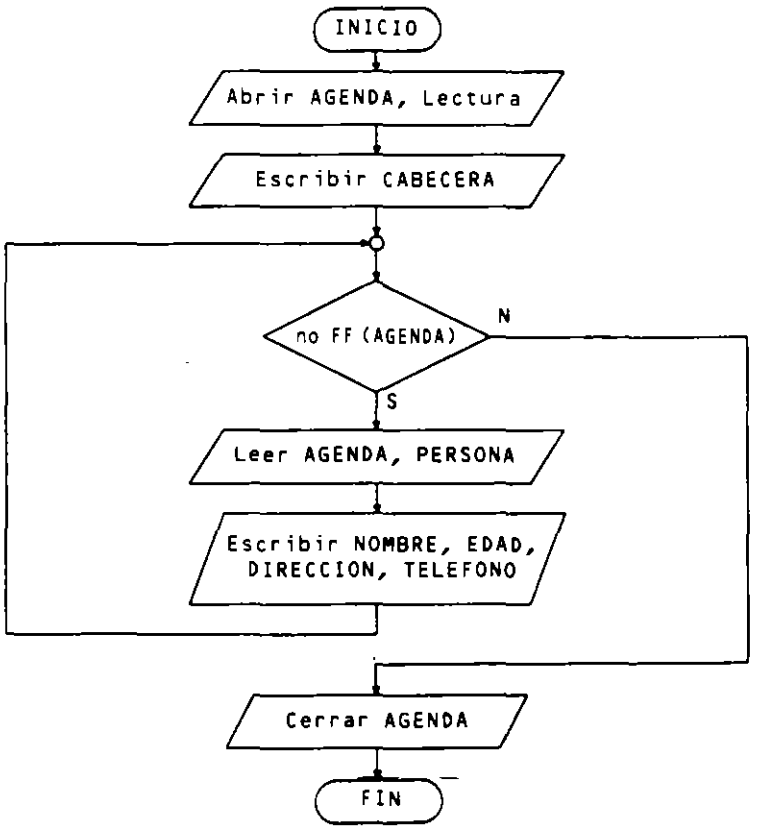
- **Cierre:**

Libera al archivo del programa. El archivo queda igual que estaba antes de ser leído, sin ninguna modificación, tanto si se ha leído completo como si no. Asimismo queda a disposición de cualquier programa que lo solicite.

La notación es igual que para el caso de escritura.

Ejemplo: *Listado por impresora del archivo AGENDA del ejemplo anterior.*

- **Ordinograma:**



- **Pseudocódigo:**

```
Programa LISTADO AGENDA
Entorno:
  AGENDA es archivo de PERSONA
  PERSONA es registro compuesto de
              NOMBRE es alfanumérico
              EDAD es numérico entero
              DIRECCION es alfanumérico
              TELEFONO es alfanumérico
            finregistro
Algoritmo:
  escribir CABECERA
  abrir AGENDA para lectura
  mientras no ff(AGENDA) hacer
    leer AGENDA, PERSONA
    escribir NOMBRE, EDAD, DIRECCION, TELEFONO
  finmientras
  cerrar AGENDA
Finprograma
```

- **Codificación COBOL:**

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID. LISTADO-AGENDA.
*
 ENVIRONMENT DIVISION.
 INPUT-OUTPUT SECTION.
 FILE-CONTROL.
     SELECT AGENDA ASSIGN TO DISK, "AGENDA.DAT".
     SELECT LISTADO ASSIGN TO PRINTER, "PRN:".
*
 DATA DIVISION.
 FILE SECTION.
 FD  AGENDA.
 01  PERSONA.
     05 NOMBRE    PIC X(20).
     05 EDAD      PIC 99.
     05 DIRECCION PIC X(30).
     05 TELEFONO  PIC X(12).
 FD  LISTADO.
 01  LINEA PIC X(80).
 WORKING-STORAGE SECTION.
 01  LIN-PERSONA.
     05 NOMBRE    PIC X(20).
     05 FILLER    PIC X(4)  VALUE SPACES.
     05 EDAD      PIC 99.
     05 FILLER    PIC X(4)  VALUE SPACES.
     05 DIRECCION PIC X(30).
     05 FILLER    PIC X(4)   VALUE SPACES.
     05 TELEFONO  PIC X(12).
     05 FILLER    PIC X(4) VALUE SPACES.
 01  CABECERA1.
     05 FILLER    PIC X(20) VALUE "LISTADO DE LA AGENDA".
     05 FILLER    PIC X(60) VALUE SPACES.
 01  CABECERA2.
     05 FILLER    PIC X(6)  VALUE "Nombre".
     05 FILLER    PIC X(18) VALUE SPACES.
     05 FILLER    PIC X(4)  VALUE "Edad".
     05 FILLER    PIC X(2)  VALUE SPACES.
```

```
    05 FILLER     PIC X(9)  VALUE "Dirección".
    05 FILLER     PIC X(25) VALUE SPACES.
    05 FILLER     PIC X(8)  VALUE "Teléfono".
    05 FILLER     PIC X(8)  VALUE SPACES.
01  FIN-ARCHIVO  PIC XX.
PROCEDURE DIVISION.
PROCESO.
    OPEN INPUT AGENDA, OUTPUT LISTADO
    WRITE LINEA FROM CABECERA1
    WRITE LINEA FROM CABECERA2
    MOVE "NO" TO FIN-ARCHIVO
    PERFORM UNTIL FIN-ARCHIVO = "SI"
      READ AGENDA
        AT END
          MOVE "SI" TO FIN-ARCHIVO
        NOT AT END
          MOVE CORRESPONDING PERSONA TO LIN-PERSONA
          WRITE LINEA FROM LIN-PERSONA
      END-READ
    END-PERFORM
    CLOSE AGENDA, LISTADO
    STOP RUN.
```

- **Codificación Pascal:**

```
PROGRAM LISTADO_AGENDA (AGENDA, LST);
USES PRINTER; (* Para poder utilizar LST *)
CONST
  LONG_NOMBRE = 20;
  LONG_DIR    = 30;
  LONG_TEL    = 12;
TYPE
  TIPO_NOMBRE  = PACKED ARRAY [1..LONG_NOMBRE] OF CHAR;
  TIPO_DIR     = PACKED ARRAY [1..LONG_DIR] OF CHAR;
  TIPO_TEL     = PACKED ARRAY [1..LONG_TEL] OF CHAR;
  TIPO_PERSONA = RECORD
                   NOMBRE    : TIPO_NOMBRE;
                   EDAD      : INTEGER;
                   DIRECCION : TIPO_DIR;
                   TELEFONO  : TIPO_TEL
                 END;
  TIPO_AGENDA = FILE OF TIPO_PERSONA;
VAR
  AGENDA  : TIPO_AGENDA;
  PERSONA : TIPO_PERSONA;
(**)
BEGIN (* LISTADO_AGENDA *)
  ASSIGN (AGENDA, 'AGENDA. DAT');
  WRITELN (LST, 'LISTADO DE LA AGENDA');
  WRITELN (LST, 'Nombre', ' ':18, 'Edad Dirección', ' ':25,
                'Teléfono');
  RESET (AGENDA);
  WHILE NOT EOF (AGENDA) DO
    BEGIN (*1*)
      READ (AGENDA, PERSONA);
      WITH PERSONA DO
        WRITELN (LST, NOMBRE, EDAD, DIRECCION, TELEFONO)
    END; (*1*)
  CLOSE (AGENDA)
END.  (* LISTADO_AGENDA *)
```

9.7.3. LECTURA-ESCRITURA DE ARCHIVOS DIRECTOS

- **Apertura:**

Reserva un archivo directo en exclusividad para un programa. Si no existia anteriormente, lo crea. El archivo queda preparado para realizar operaciones de lectura o escritura sobre el mismo.

- **Ordinograma:**

- **Pseudocódigo:**

```
abrir AGENDAD directo
```

- **COBOL:**

Previamente a la apertura se habrá declarado en la sentencia SELECT un dispositivo de acceso directo, el tipo de acceso y la clave.

```
SELECT AGENDAD
  ASSIGN TO DISK, "AGENDAD.DAT"
  ORGANIZATION IS RELATIVE
  ACCESS IS RANDOM
  RELATIVE KEY IS NUM.
```

La clave declarada NUM es un campo numérico declarado en la WORKING-STORAGE SECTION.

```
OPEN I-O AGENDAD
```

- **Pascal:**

Se abre igual que para lectura, permitiendo operaciones de lectura y escritura.

```
RESET (AGENDAD)
```

- **Escritura de un registro:**

Coloca en una posición cualquiera del archivo, fijada en el programa, el contenido del registro buffer, al que previamente se le habrán asignado los datos.

- **Ordinograma:**

- **Pseudocódigo:**

```
escribir AGENDAD, PERSONAD, NUM
```

- **COBOL:**

Se supone asignado el valor de la posición deseada a NUM.

```
WRITE PERSONAD
```

- **Pascal:**

```
SEEK (AGENDAD, NUM);
WRITE (AGENDAD, PERSONAD)
```

- **Lectura de un registro:**

Coloca el contenido del registro cuya posición se indique en el registro buffer

- **Ordinograma:**

- **Pseudocódigo:**

```
Leer AGENDAD, PERSONAD, NUM
```

- **COBOL:**

Se supone asignado el valor de la posición deseada a NUM.

```
READ AGENDAD
```

- **Pascal:**

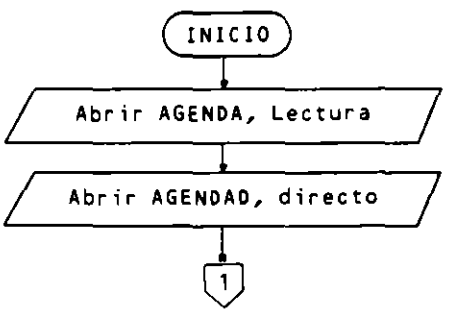
```
SEEK (AGENDAD, NUM);
READ (AGENDAD, PERSONAD)
```

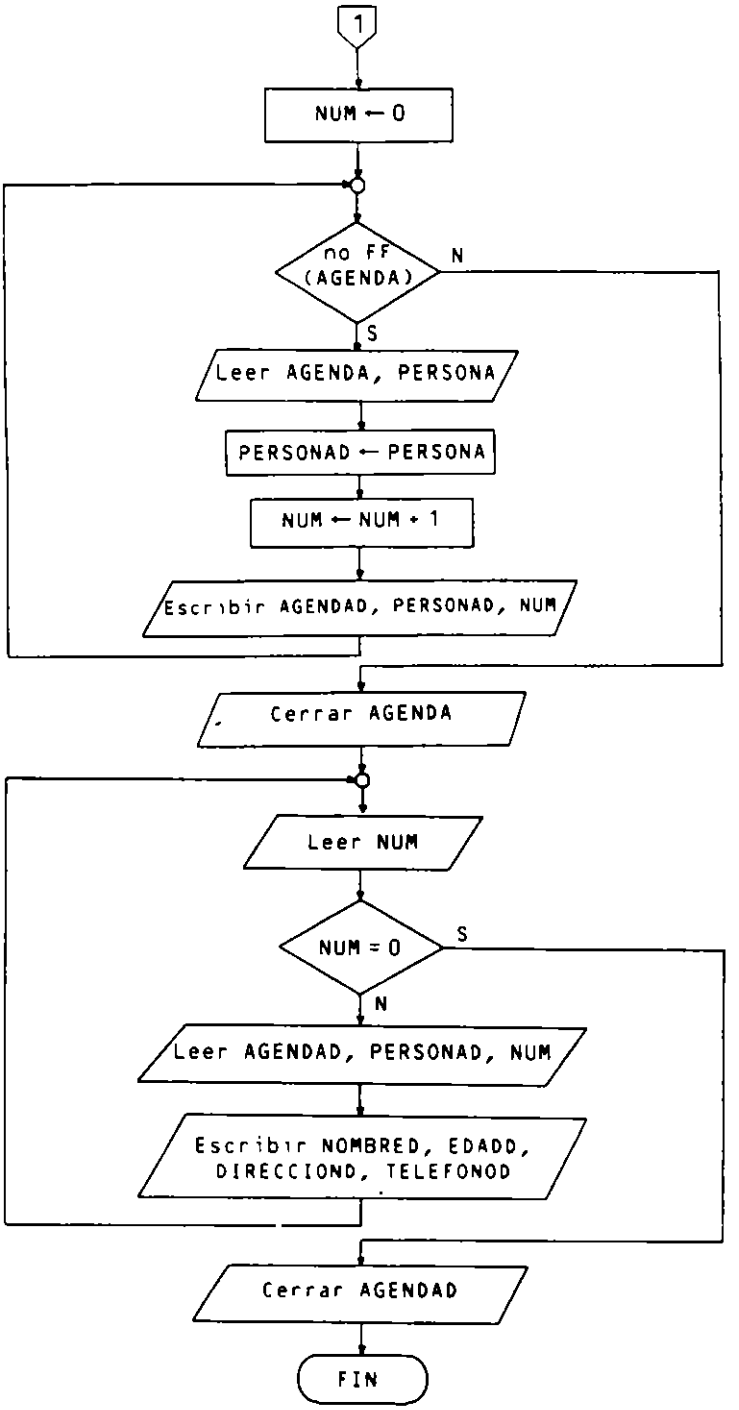
- **Cierre:**

Igual que en los casos anteriores.

Ejemplo: *Programa que copia el archivo secuencial AGENDA del ejemplo anterior en un archivo directo AGENDAD, conservando cada registro su posición relativa. Concluida la copia, el programa permitirá consultas al archivo directo, introduciendo por teclado el número de registro y terminando al introducir un 0.*

- **Ordinograma:**





- **Pseudocódigo:**

```
Programa AGENDA DIRECTA
Entorno:
  AGENDA es archivo de PERSONA
  PERSONA es registro compuesto de
              NOMBRE es alfanumérico
              EDAD es numérico entero
              DIRECCION es alfanumérico
```

```
                TELEFONO es alfanumérico
              finregistro
  AGENDAD es archivo de PERSONAD
  PERSONAD es registro compuesto de
                NOMBRED es alfanumérico
                EDADD es numérico entero
                DIRECCIOND es alfanumérico
                TELEFONOD es alfanumérico
              finregistro
  NUM es numérica entera
Algoritmo:
  abrir AGENDA para lectura
  abrir AGENDAD directo
  ** Copia de AGENDA en AGENDAD
  NUM ← 0
  mientras no ff (AGENDA) hacer
    leer AGENDA, PERSONA
    PERSONAD ← PERSONA
    NUM ← NUM + 1
    escribir AGENDAD, PERSONAD, NUM
  finmientras
  cerrar AGENDA
  ** AGENDAD contendrá 100 registros
  ** Consultas de AGENDAD
  iterar
    escribir "Núm. de registro (entre 1 y 100), para terminar 0"
    leer NUM
    salir si NUM = 0
    leer AGENDAD, PERSONAD, NUM
    escribir NOMBRED, EDADD, DIRECCIOND, TELEFONOD
  finiterar
  cerrar AGENDAD
Finprograma
```

- **Codificación COBOL:**

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID. AGENDA-DIRECTA.
*
 ENVIRONMENT DIVISION.
 INPUT-OUTPUT SECTION.
 FILE-CONTROL.
     SELECT AGENDA
       ASSIGN TO DISK, "AGENDA.DAT".
     SELECT OPTIONAL AGENDAD
       ASSIGN TO DISK, "AGENDAD.DAT"
       ORGANIZATION IS RELATIVE
       ACCESS IS RANDOM
       RELATIVE KEY IS NUM.
*
 DATA DIVISION.
 FILE SECTION.
 FD  AGENDA.
 01  PERSONA.
     05 NOMBRE     PIC X(20).
     05 EDAD       PIC 99.
     05 DIRECCION  PIC X(30).
     05 TELEFONO   PIC X(12).
 FD  AGENDAD.
 01  PERSONAD.
```

```
           05 NOMBRED     PIC X(20).
           05 EDADD       PIC 99.
           05 DIRECCIOND  PIC X(30).
           05 TELEFONOD   PIC X(12).
       WORKING-STORAGE SECTION.
       01  PERSONAS.
           05 NOMBRED     PIC X(20).
           05 FILLER      PIC X(2)   VALUE SPACES.
           05 EDADD       PIC 29.
           05 FILLER      PIC X(2)   VALUE SPACES.
           05 DIRECCIOND  PIC X(30).
           05 FILLER      PIC X(2)   VALUE SPACES.
           05 TELEFONOD   PIC X(12).
       01  NUM            PIC 999.
       01  FIN-ARCHIVO    PIC XX.
      *
       PROCEDURE DIVISION.
       PROCESO.
           OPEN INPUT AGENDA, I-O AGENDAD
      * Copia de AGENDA en AGENDAD
           MOVE 0 TO NUM
           MOVE "NO" TO FIN-ARCHIVO
           PERFORM UNTIL FIN-ARCHIVO = "SI"
             READ AGENDA
               AT END
                 MOVE "SI" TO FIN-ARCHIVO
               NOT AT END
                 ADD 1 TO NUM
                 WRITE PERSONAD FROM PERSONA
               END-READ
           END-PERFORM
           CLOSE AGENDA
      * Consultas de AGENDAD
           DISPLAY Núm. de registro (entre 1 y 100) para terminar 0"
           ACCEPT NUM
           PERFORM UNTIL NUM = 0
             READ AGENDAD
             MOVE CORRESPONDING PERSONAD TO PERSONAS
             DISPLAY PERSONAS
             DISPLAY "Núm. de registro (entre 1 y 100) para terminar 0"
             ACCEPT NUM
           END-PERFORM
           CLOSE AGENDAD
           STOP RUN.
```

- **Codificación Pascal:**

El compilador utilizado permite tratar un archivo secuencial abierto para lectura como directo, accediendo a sus registros por su posición mediante la instrucción «seek», pudiendo ser leídos o escritos los registros accedidos. Por tanto, no haría falta crear un nuevo archivo. No obstante, se presenta a continuación una codificación equivalente al ejercicio planteado para poder compararlo.

```
PROGRAM AGENDA_DIRECTA (INPUT, OUTPUT, AGENDA, AGENDAD);
CONST
  LONG_NOMBRE  = 20;
  LONG_DIR     = 30;
  LONG_TEL     = 12;
TYPE
  TIPO_NOMBRE  = PACKED ARRAY [1..LONG_NOMBRE] OF CHAR;
```

```
  TIPO_DIR     = PACKED ARRAY [1..LONG_DIR] OF CHAR;
  TIPO_TEL     = PACKED ARRAY [1..LONG_TEL] OF CHAR;
  TIPO_PERSONA = RECORD
                   NOMBRE    : TIPO_NOMBRE;
                   EDAD      : INTEGER;
                   DIRECCION : TIPO_DIR;
                   TELEFONO  : TIPO_TEL
                 END;
  TIPO_AGENDA = FILE OF TIPO_PERSONA;
VAR
  AGENDA, AGENDAD : TIPO_AGENDA;
  PERSONA: TIPO_PERSONA;
  NUM: INTEGER;
(**)
BEGIN (* AGENDA_DIRECTA *)
  ASSIGN (AGENDA,  'AGENDA.DAT');
  ASSIGN (AGENDAD, 'AGENDAD.DAT');
  RESET  (AGENDA);
  RESET  (AGENDAD);
  (* Copia de AGENDA en AGENDAD *)
  NUM := 0;
  WHILE NOT EOF (AGENDA) DO
    BEGIN (*1*)
      READ (AGENDA, PERSONA);
      NUM := NUM + 1;
      SEEK (AGENDAD, NUM);
      WRITE (AGENDAD, PERSONA)
    END; (*1*)
  CLOSE (AGENDA);
  (* Consultas de AGENDAD *)
  WRITE ('Núm. de registro (entre 1 y 100) para terminar 0');
  READLN (NUM);
  WHILE NUM <> 0 DO
    BEGIN (*2*)
      SEEK (AGENDAD, NUM);
      READ (AGENDA, PERSONA);
      WITH PERSONA DO
        WRITELN (NOMBRE, ' ', EDAD, ' ', DIRECCION, ' ',
                 TELEFONO);
      WRITE ('Número de registro (entre 1 y 100)',
             ' para terminar 0');
      READLN (NUM)
    END; (*2*)
  CLOSE (AGENDAD)
END. (* AGENDAD_DIRECTA *)
```

9.7.4. LECTURA-ESCRITURA DE ARCHIVOS INDEXADOS

- **Apertura:**

Reserva un archivo indexado en exclusividad para un programa. Si no existía anteriormente, lo crea. El archivo queda preparado para lectura, escritura o borrado de sus registros.

Esta organización no está disponible en la mayoría de compiladores del lenguaje Pascal, y en el lenguaje COBOL presenta algunas diferencias de unas versiones a otras.

- **Ordinograma:**

- **Pseudocódigo:**

```
abrir AGENDAI indexado
```

- **COBOL:**

Es preciso haber declarado en la cláusula SELECT el soporte, la organización, el modo de acceso y la clave, que ha de ser un campo alfanumérico del registro. Se hace como sigue:

```
SELECT AGENDAI
  ASSIGN TO DISK, "AGENDAI.DAT"
  ORGANIZATION IS INDEXED
  ACCESS MODE IS DYNAMIC
  RECORD KEY IS DNII.
```

La apertura del archivo se expresa:

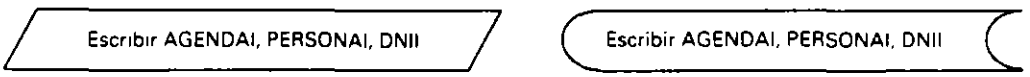
```
OPEN I-O AGENDAI
```

- **Escritura de un registro:**

Coloca en una posición del archivo, determinada por el sistema, el contenido del registro buffer, al que previamente se le habrán asignado los datos.

La ubicación de cada registro depende del valor asignado al campo clave.

- **Ordinograma:**



- **Pseudocódigo:**

```
escribir AGENDAI, PERSONAI, DNII
```

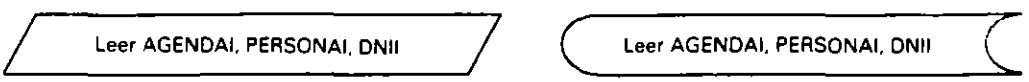
- **COBOL:**

```
WRITE PERSONAI
```

- **Lectura de un registro:**

Coloca el contenido del registro correspondiente a la clave de valor, previamente asignado, en el registro buffer.

- **Ordinograma:**



- **Pseudocódigo:**

```
leer AGENDAI, PERSONAI, DNII
```

- **COBOL**

```
READ AGENDAI
```

- **Reescritura de un registro:**

Coloca en su posición correspondiente el contenido de un registro que ya existía previamente y cuyos datos han sido modificados.

- **Ordinograma:**

- **Pseudocódigo:**

```
reescribir AGENDAI, PERSONAI, DNII
```

- **COBOL:**

```
REWRITE PERSONAI
```

- **Borrado de un registro:**

Elimina un registro existente, determinado por el valor asignado al campo clave.

- **Ordinograma:**

- **Pseudocódigo:**

```
borrar AGENDAI, PERSONAI, DNII
```

- **COBOL:**

```
DELETE AGENDAI
```

- **Cierre:**

Igual que en los casos anteriores.

- **Errores:**

En los archivos indexados se puede producir error de ejecución por alguna de las causas expuestas a continuación.

— **Escritura de un registro que ya existe.**
— **Lectura de un registro que no existe.**
— **Reescritura de un registro inexistente.**
— **Borrado de un registro inexistente.**

Estos errores se controlan en COBOL por medio de la cláusula INVALID KEY que acompaña a la instrucción de acceso, evitando que el programa sea abortado y ejecutando sus instrucciones en el caso de los errores antes citados.

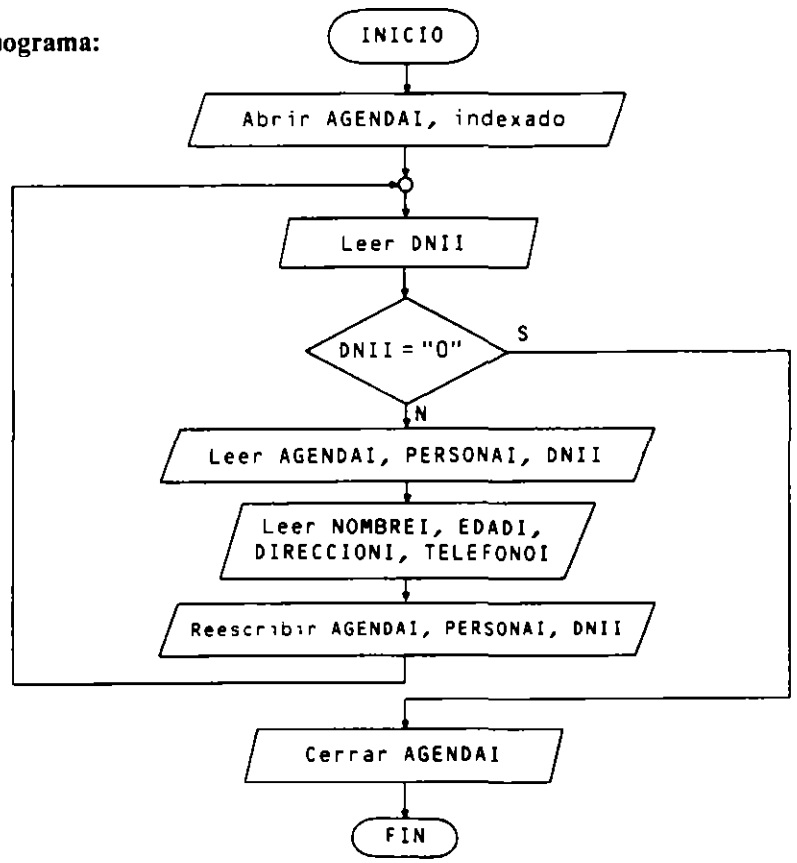
Asimismo se puede acompañar de la cláusula NOT INVALID KEY para incluir las instrucciones que se han de realizar en caso de que no se presente el error.

Las instrucciones de acceso a los registros con tratamiento de error son las siguientes:

```
WRITE PERSONAI
  INVALID KEY     instrucciones de tratamiento de error
  NOT INVALID KEY instrucciones en caso de escritura correcta
END-WRITE

READ AGENDAI
  INVALID KEY     instrucciones de tratamiento de error
  NOT INVALID KEY instrucciones de proceso del registro leido
END-READ

REWRITE PERSONAI
  INVALID KEY     instrucciones de tratamiento de error
  NOT INVALID KEY instrucciones en caso de reescritura correcta
END-REWRITE

DELETE AGENDAI
  INVALID KEY     instrucciones de tratamiento de error
  NOT INVALID KEY instrucciones en caso de borrado correcto
END-DELETE
```

Ejemplo: *Dado el archivo AGENDAI que contiene los datos de una serie de personas cuya clave es el número del DNI. Programa que permite modificar el contenido de algunos de sus registros a partir de datos introducidos por teclado.*

- **Ordinograma:**



- **Pseudocódigo:**

```
Programa ACTUALIZACION DE INDEXADO
Entorno:
  AGENDAI es archivo de PERSONAI
  PERSONAI es registro compuesto de
                DNII es clave alfanumérico
                NOMBREI es alfanumérico
                EDADI es numérico entero
                DIRECCIONI es alfanumérico
                TELEFONOI es alfanumérico
              finregistro
Algoritmo:
  abrir AGENDAI indexado
  iterar
    escribir "Escriba DNI de persona a modificar
              o 0 para terminar"
    leer DNII
    salir si DNII = "0"
    leer AGENDAI, PERSONAI, DNII
    ** Presentación de datos y modificación
    escribir NOMBREI; leer NOMBREI
    escribir EDADI; leer EDADI
    escribir DIRECCIONI; leer DIRECCIONI
    escribir TELEFONOI; leer TELEFONOI
    reescribir AGENDAI, PERSONAI, DNII
  finiterar
  cerrar AGENDAI
Finprograma
```

- **Codificación COBOL:**

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID. ACTUALIZACION-INDEXADO.
*
 ENVIRONMENT DIVISION.
 INPUT-OUTPUT SECTION.
 FILE-CONTROL.
     SELECT AGENDAI
       ASSIGN TO DISK, "AGENDAI.DAT"
       ORGANIZATION IS INDEXED
       ACCESS MODE IS DYNAMIC
       RECORD KEY IS DNII.
*
 DATA DIVISION.
 FILE SECTION.
 FD  AGENDAI.
 01  PERSONAI.
     05 DNII          PIC X(8).
     05 NOMBREI       PIC X(20).
     05 EDADI         PIC 99.
     05 DIRECCIONI    PIC X(30).
     05 TELEFONOI     PIC X(12).
 WORKING-STORAGE SECTION.
 01  VARIABLES.
     05 NUMDNI        PIC X(8).
     05 CONFIRMACION PIC X.
     05 ESPERA        PIC X.
*
 PROCEDURE DIVISION.
```

```
PROCESO.
    OPEN I-O AGENDAI
    DISPLAY " " ERASE
    DISPLAY "Escriba el núm. de DNI de la persona a modificar ",
            "o 0 para terminar: " NO ADVANCING
    ACCEPT NUMDNI
    PERFORM UNTIL NUMDNI = "0"
      MOVE NUMDNI TO DNII
      READ AGENDAI
        INVALID KEY
          DISPLAY " No existe ese D.N.I."
          DISPLAY " Pulse <RETURN> para continuar"
          ACCEPT ESPERA
        NOT INVALID KEY
          PERFORM MODIFICAR
      END-READ
      DISPLAY " " ERASE
      DISPLAY "Escriba el núm. de DNI de la persona a modificar ",
              "o 0 para terminar: " NO ADVANCING
      ACCEPT NUMDNI
    END-PERFORM
    CLOSE AGENDAI
    STOP RUN.
*
MODIFICAR.
    DISPLAY "Nuevo nombre: "     NO ADVANCING
    ACCEPT NOMBRE     UPDATE
    DISPLAY "Nueva edad: "       NO ADVANCING
    ACCEPT EDAD       UPDATE
    DISPLAY "Nueva dirección: " NO ADVANCING
    ACCEPT DIRECCION UPDATE
    DISPLAY "Nuevo teléfono: "  NO ADVANCING
    ACCEPT TELEFONO  UPDATE
    DISPLAY " Los datos modificados son correctos (S/N)? "
                                 NO ADVANCING
    ACCEPT CONFIRMACION
    IF CONFIRMACION = "S"
      THEN REWRITE PERSONA
    END-IF.
```

Nota:

La cláusula UPDATE en la instrucción ACCEPT muestra el valor almacenado en una variable y permite modificarlo parcialmente, totalmente o no modificarlo (pulsando <RET> sin escribir nada).

EJERCICIO RESUELTO

Programa que gestiona de forma interactiva un diccionario INGLES/FRANCES/ESPAÑOL. Para ello se dispone de un archivo secuencial con 1000 registros, conteniendo cada uno tres palabras de igual significado en inglés, francés y español, respectivamente.

El programa cargará este archivo en una tabla, permitiendo sucesivas consultas a la misma.

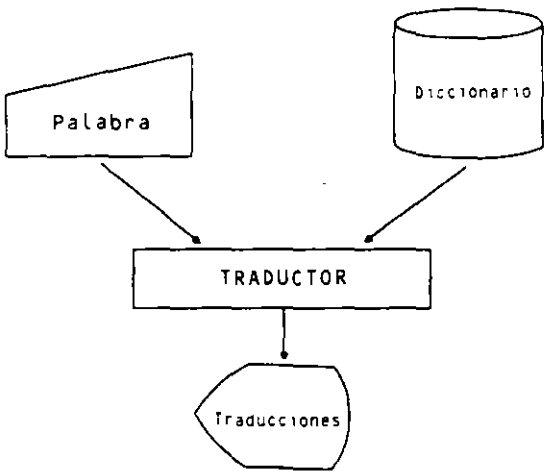
El primer dato de entrada indica cuál de los tres diccionarios se desea utilizar en

primer lugar. A continuación se introduce una palabra de ese idioma y el programa proporciona sus traducciones si dicha palabra figura en la tabla.

Este proceso se puede repetir hasta que se desee, siendo posible el cambio de un diccionario a otro.

La entrada de datos se realizará mediante elecciones en sucesivos «menús» que se presentarán por pantalla.

- **Organigrama:**



Una vez cargado el archivo en la tabla, la selección de un diccionario se hace sobre la siguiente pantalla:

```
   TRADUCTOR DE INGLES/FRANCES/ESPAÑOL

1.  DICCIONARIO DE INGLES
2.  DICCIONARIO DE FRANCES
3.  DICCIONARIO DE ESPAÑOL
4.  TERMINAR

Escriba opción: _
```

A continuación se presentará una pantalla según el diccionario elegido solicitando la palabra a traducir o la vuelta a la pantalla anterior. Por ejemplo, si se eligió la opción número 1, la pantalla es:

```
          DICCIONARIO DE INGLES
   Escriba la palabra en inglés que desea
traducir o el carácter "*" para volver al
menú principal

   Palabra o "*": _
```

Esta pantalla permanecerá hasta que se introduzca el carácter "*". El programa utilizará las líneas inferiores de ésta para imprimir las traducciones solicitadas o un mensaje de que la palabra no está en el diccionario. Por ejemplo, si se introduce la palabra WOMAN, el resultado puede ser una de las dos siguientes pantallas:

```
        DICCIONARIO DE INGLES
  Escriba la palabra en inglés que desea
traducir o el carácter "*" para volver al
menú principal

  Palabra o "*": _

     Inglés:  WOMAN
     Francés: FEMME
     Español: MUJER
```

```
        DICCIONARIO DE INGLES
  Escriba la palabra en inglés que desea
traducir o el carácter "*" para volver al
menú principal

  Palabra o "*": _

     La palabra WOMAN no figura en este
  diccionario.
```

La estructura de datos interna que contiene el diccionario es una tabla D de 1000 filas y tres columnas, de componentes alfanuméricas, conteniendo en cada fila las tres palabras de igual significado, la primera en inglés, la segunda en francés y la tercera en español.

D	1	2	3
1	RECORD	REGISTRE	REGISTRO
2	INPUT	ENTREE	ENTRADA
3	ARRAY	TABLE	TABLA
...	...	...	...
999	COMPUTER	ORDINATEUR	ORDENADOR
1000	PROGRAM	PROGRAMME	PROGRAMA

- **Pseudocódigo:**

```
Programa DICCIONARIO
Entorno:
  D es tabla(1000,3) alfanumérica
  OPCION, I, J, K, L son numéricas enteras
  PALABRA, ID1, ID2, ID3 son alfanuméricas
Algoritmo:
  CARGAR DICCIONARIO
  iterar
    PEDIR IDIOMA
    salir si OPCION = 4
    SELECCIONAR DICCIONARIO
    iterar
      PEDIR PALABRA
```

```
        salir si PALABRA = "*"
        BUSCAR PALABRA
        IMPRIMIR TRADUCCIONES
      finiterar
  finiterar
Finprograma
**
Subprograma CARGAR DICCIONARIO
Entorno:
  FIDICI es archivo de REDICI
  REDICI es registro compuesto de
              IG es alfanumérico
              FR es alfanumérico
              ES es alfanumérico
            finregistro
  F es numérica entera
Algoritmo:
  abrir FIDICI para lectura
  para F de 1 a 1000 hacer
    leer FIDICI, REDICI
    D(F,1) = IG; D(F,2) = FR; D(F,3) = ES
  finpara
  cerrar FIDICI
Finsubprograma
**
Subprograma PEDIR IDIOMA
Algoritmo:
  escribir MENU PRINCIPAL
  iterar
    leer OPCION
    salir si 1 <= OPCION y OPCION <= 4
    escribir "Opción incorrecta"
  finiterar
Finsubprograma
**
Subprograma SELECCIONAR DICCIONARIO
Algoritmo:
  si OPCION = 1
    entonces I ← 1; ID1 ← "Inglés:  "
             J ← 2; ID2 ← "Francés: "
             K ← 3; ID3 ← "Español: "
    sino si OPCION = 2
           entonces
             I ← 2; ID1 ← "Francés: "
             J ← 1; ID2 ← "Inglés:  "
             K ← 3; ID3 ← "Español: "
           sino
             I ← 3; ID1 ← "Español: "
             J ← 1; ID2 ← "Inglés:  "
             K ← 2; ID3 ← "Francés: "
         finsi
  finsi
Finsubprograma
**
Subprograma PEDIR PALABRA
Algoritmo:
  escribir PANTALLA PETICION
  leer PALABRA
Finsubprograma
**
```

```
Subprograma BUSCAR PALABRA
Algoritmo:
  L ← 0
  repetir
    L ← L + 1
  hasta D(L,I) = PALABRA o L = 1000
Finsubprograma
**
Suborograma IMPRIMIR TRADUCCIONES
Algoritmo:
  si D(L,I) = PALABRA
    entonces escribir ID1, D(L,I)
             escribir ID2, D(L,J)
             escribir ID3, D(L,K)
    sino escribir "   La palabra ", PALABRA, " no figura"
         escribir "en este diccionario."
  finsi
Finsubprograma
```

- **Codificación COBOL:**

```
 IDENTIFICATION DIVISION.
 PROGRAMA-ID. DICCIONARIO.
*    *****
 ENVIRONMENT DIVISION.
 INPUT-OUTPUT SECTION.
 FILE-CONTROL.
     SELECT FIDICI ASSIGN TO DISK, "FIDICI.DAT".
*    *****
 DATA DIVISION.
 FILE SECTION.
 FD  FIDICI.
 01  REDICI.
     05 IG PIC X(15).
     05 FR PIC X(15).
     05 ES PIC X(15).
 WORKING-STORAGE SECTION.
 01  TABLA-D.
     05  FILA-D    OCCURS 1000 TIMES.
         10 D PIC X(15) OCCURS    3 TIMES.
 01  VARIABLES.
     05  OPCION PIC 9.
         88  TERMINAR  VALUE 4.
     05  I PIC 9.
     05  J PIC 9.
     05  K PIC 9.
     05  F PIC 9999.
     05  L PIC 9999.
     05  PALABRA PIC X(15).
     05  ID1     PIC X(9).
     05  ID2     PIC X(9).
     05  ID3     PIC X(9).
*    *****
 PROCEDURE DIVISION.
 PROCESO.
     PERFORM CARGAR-DICCIONARIO
     PERFORM LEER-IDIOMA THRU FIN-LEER-IDIOMA
     PERFORM UNTIL TERMINAR
       DISPLAY " " ERASE
       PERFORM SELECCIONAR-DICCIONARIO
```

```
           DISPLAY " " ERASE
           PERFORM LEER-PALABRA
           PERFORM UNTIL PALABRA = "*"
             PERFORM BUSCAR-PALABRA
             PERFORM VISUALIZAR
             PERFORM LEER-PALABRA
           END-PERFORM
           PERFORM LEER-IDIOMA THRU FIN-LEER-IDIOMA
         END-PERFORM
         STOP RUN.
*        *****
 CARGAR-DICCIONARIO.
         OPEN INPUT FIDICI
         PERFORM VARYING F FROM 1 BY 1 UNTIL F > 1000
           READ FIDICI
           MOVE IG TO D(F, 1)
           MOVE FR TO D(F, 2)
           MOVE ES TO D(F, 3)
         END-PERFORM
         CLOSE FIDICI.
*        *****
 LEER-IDIOMA.
         DISPLAY " " ERASE
         DISPLAY " TRADUCTOR INGLES/FRANCES/ESPAÑOL"
                                                  LINE  8 POSITION 22
         DISPLAY "1. DICCIONARIO DE INGLES"       LINE 10 POSITION 26
         DISPLAY "2. DICCIONARIO DE FRANCES"      LINE 11 POSITION 26
         DISPLAY "3. DICCIONARIO DE ESPAÑOL"      LINE 12 POSITION 26
         DISPLAY "4. TERMINAR"                    LINE 13 POSITION 26
         DISPLAY "    Escriba opción: "           LINE 15 POSITION 26.
 LEER-OPCION
         ACCEPT OPCION                            LINE 15 POSITION 47
         PERFORM UNTIL OPCION >= 1 AND <= 4
           DISPLAY "    Opción incorrecta"       LINE 16 POSITION 26
           DISPLAY "                     "       LINE 15 POSITION 47
           ACCEPT OPCION                          LINE 15 POSITION 47
         END-PERFORM.
 FIN-LEER-IDIOMA.
         EXIT.
*        *****
 SELECCIONAR-DICCIONARIO.
         EVALUATE OPCION
           WHEN 1
             MOVE 1 TO I, MOVE "Inglés:  " TO ID1
             MOVE 2 TO J, MOVE "Francés: " TO ID2
             MOVE 3 TO K, MOVE "Español: " TO ID3
           WHEN 2
             MOVE 2 TO I, MOVE "Francés: " TO ID1
             MOVE 1 TO J, MOVE "Inglés:  " TO ID2
             MOVE 3 TO K, MOVE "Español: " TO ID3
           WHEN 3
             MOVE 3 TO I, MOVE "Español: " TO ID1
             MOVE 1 TO J, MOVE "Inglés:  " TO ID2
             MOVE 2 TO K, MOVE "Francés: " TO ID3
         END-EVALUATE.
*        *****
 LEER-PALABRA.
         DISPLAY "DICCIONARIO DE "                LINE  8 POSITION 31,
                 ID1                              LINE  8 POSITION 46
         DISPLAY "Escriba la palabra en "         LINE 10 POSITION 30,
```

```
              ID1                                    LINE 10 POSITION 52
       DISPLAY "que desea traducir o el carácter"
                                                     LINE 11 POSITION 25
       DISPLAY "'*' para ir al menú principal"       LINE 12 POSITION 25
       DISPLAY "                                     "
                                                     LINE 13 POSITION 26
       DISPLAY "Palabra o '*': "                     LINE 15 POSITION 28
       ACCEPT  PALABRA                               LINE 15 POSITION 44.
*      *****
 BUSCAR-PALABRA.
       MOVE 1 TO L
       PERFORM UNTIL D(L, I) = PALABRA OR L = 1000
         ADD 1 TO L
       END-PERFORM.
*      *****
 VISUALIZAR.
       DISPLAY "                                     "
                                                     LINE 16 POSITION 28
       DISPLAY "                                     "
                                                     LINE 17 POSITION 28
       DISPLAY "                                     "
                                                     LINE 18 POSITION 28
       IF D(L, I) = PALABRA
         THEN DISPLAY
              ID1 LINE 16 POSITION 31, D(L, I) LINE 16 POSITION 40,
              ID2 LINE 17 POSITION 31, D(L, J) LINE 17 POSITION 40,
              ID3 LINE 18 POSITION 31, D(L, K) LINE 18 POSITION 40,
         ELSE DISPLAY
              "La palabra "                        LINE 17 POSITION 31,
              PALABRA                              LINE 17 POSITION 42,
              "no figura en este diccionario"      LINE 18 POSITION 28
       END-IF
       DISPLAY "                                     "
                                                     LINE 15 POSITION 44.
```

- **Codificación Pascal:**

```
PROGRAM DICCIONARIO (INPUT, OUTPUT, FIDICI);
  USES CRT;
  TYPE STRING15 = PACKED ARRAY [1..15] OF CHAR;
       STRING9  = PACKED ARRAY [1..9]  OF CHAR;
  VAR D : ARRAY[1..1000, 1..3] OF STRING15;
      OPCION, I, J, K, L : INTEGER;
      PALABRA : STRING15;
      ID1, ID2, ID3 : STRING9;
(* ***** *)
  PROCEDURE CARGARDIC;
    TYPE REG = RECORD
                 IG : STRING15;
                 FR : STRING15;
                 ES : STRING15
               END;
         FICH = FILE OF REG;
    VAR FIDICI : FICH;
        REDICI : REG;
        F : INTEGER;
    BEGIN (*CARGARDIC*)
      ASSIGN(FIDICI, 'FIDICI.DAT');
      RESET(FIDICI);
      FOR F := 1 TO 1000 DO
```

```
        BEGIN (*1*)
          READ(FIDICI, REDICI);
          D[F,1] := REDICI.IG;
          D[F,2] := REDICI.FR;
          D[F,3] := REDICI.ES
        END; (*1*)
      CLOSE(FIDICI)
    END; (*CARGARDIC*)
(* ***** *)
  PROCEDURE PEDIRID;
    BEGIN (*PEDIRID*)
      CLRSCR; (*Borrado de pantalla*)
      GOTOXY(22,8); (*Posicionamiento del cursor*)
      WRITE(' TRADUCTOR INGLES/FRANCES/ESPAÑOL');
      GOTOXY(22,10);
      WRITE('      1. DICCIONARIO DE INGLES');
      GOTOXY(22,11);
      WRITE('      2. DICCIONARIO DE FRANCES');
      GOTOXY(22,12);
      WRITE('      3. DICCIONARIO DE ESPAÑOL');
      GOTOXY(22,13);
      WRITE('      4. TERMINAR');
      GOTOXY(22,15);
      WRITE('          Escriba opción: ');
      READ(OPCION);
      WHILE (OPCION < 1) OR (OPCION > 4) DO
        BEGIN (*2*)
          GOTOXY(22,16);
          WRITE('       Opción incorrecta');
          GOTOXY(47,15);
          READ (OPCION)
        END (*2*)
    END; (*PEDIRID*)
(* ***** *)
  PROCEDURE SELEDIC;
    BEGIN (*SELEDIC*)
      IF OPCION = 1 THEN
              BEGIN (*3*)
                I := 1; ID1 := 'Inglés:  ';
                J := 2; ID2 := 'Francés: ';
                K := 3; ID3 := 'Español: '
              END (*3*)
      ELSE IF OPCION = 2 THEN
              BEGIN (*4*)
                I := 2; ID1 := 'Inglés:  ';
                J := 1; ID2 := 'Francés: ';
                K := 3; ID3 := 'Español: '
              END (*4*)
           ELSE
              BEGIN (*5*)
                I := 3; ID1 := 'Inglés:  ';
                J := 1; ID2 := 'Francés: ';
                K := 2; ID3 := 'Español: '
              END (*5*)
    END; (*SELEDIC*)
(* ***** *)
  PROCEDURE PEDIRPAL;
    PROCEDURE LEERPAL
      VAR J: INTEGER;
      BEGIN (*LEERPAL*)
```

```
        J := 0;
        WHILE NOT EOLN AND (J < 15) DO
          BEGIN (*6*)
            J := J + 1;
            READ(PALABRA[J])
          END;  (*6*)
        READLN;
        FOR J := J + 1 TO 15 DO PALABRA[J] := ' '
      END;  (*LEERPAL*)
  BEGIN (*PEDIRPAL*)
    GOTOXY(22,8);
    WRITE('         DICCIONARIO DE ', ID1);
    GOTOXY(22,10);
    WRITE('         Escriba la palabra en ', ID1);
    GOTOXY(22,11);
    WRITE('    que desea traducir o el carácter');
    GOTOXY(22,12);
    WRITE('    "*" para volver al menú principal');
    GOTOXY(22,14);
    WRITE('       Palabra o "*": ');
    LEERPAL
    END; (*PEDIRPAL*)
(* ***** *)
  PROCEDURE BUSCARPAL;
    BEGIN (*BUSCARPAL*)
      L := 0;
      REPEAT L := L + 1
      UNTIL (D[L,I] = PALABRA) OR (L = 1000)
    END; (*BUSCARPAL*)
(* ***** *)
  PROCEDURE IMPRIMIR;
  BEGIN (*IMPRIMIR*)
    GOTOXY(43,14); CLREOL; (*Borra la linea*)
    GOTOXY(22,16); CLREOL;
    GOTOXY(22,17); CLREOL;
    GOTOXY(22,18); CLREOL;
    IF D[L,I] = PALABRA THEN
       BEGIN (*7*)
         GOTOXY(31,16); WRITE(ID1, D[L,I]);
         GOTOXY(31,17); WRITE(ID2, D[L,J]);
         GOTOXY(31,18); WRITE(ID3, D[L,K])
       END (*7*)
    ELSE
       BEGIN (*8*)
         GOTOXY(31,17); WRITE('La palabra ', PALABRA);
         GOTOXY(31,18); WRITE('no figura en este ',
                              'diccionario.')
       END (*8*)
    END; (*IMPRIMIR*)
(* ***** *)
BEGIN (* DICCIONARIO *)
  CARGARDIC;
  PEDIRID;
  WHILE OPCION <> 4 DO
    BEGIN (*9*)
      CLRSCR; SELEDIC;
      CLRSCR; PEDIRPAL;
      WHILE PALABRA <> '*' DO
        BEGIN (*10*)
          BUSCARPAL;
```

```
            IMPRIMIR;
            PEDIRPAL
          END; (*10*)
        PEDIRID
      END (*9*)
END. (* DICCIONARIO *)
```

EJERCICIOS PROPUESTOS

1. Programa para crear un archivo secuencial LIBROS a partir de datos introducidos por teclado, cuyos registros estarán compuestos por los siguientes campos:

   ```
   SIGNATURA alfanumérico
   AUTOR/ES alfanumérico
   TITULO alfanumérico
   EDITORIAL alfanumérico
   AÑO DE EDICION numérico entero
   TEMA alfanumérico
   NUMERO DE PAGINAS numérico entero
   ISBN alfanumérico
   ```

2. Programa que copia el archivo del ejercicio anterior en otro INDEXLIBROS, de organización secuencial indexada, tomando el campo SIGNATURA como clave.

3. Programa que, a partir de uno de los archivos anteriores, proporciona un listado de libros de una editorial que se toma como dato de entrada.

4. Programa que genera un archivo directo VENTAS a partir de los datos que figuran en los albaranes de las ventas efectuadas en una empresa durante un mes, los cuales tienen numeración consecutiva, empezando por el número 1, que se tomará como dirección de almacenamiento. Los datos del albarán que figurarán en el archivo son:

   ```
   NUMERO DE ALBARAN numérico entero
   CODIGO DEL VENDEDOR alfanumérico
   CODIGO DEL CLIENTE alfanumérico
   CODIGO DEL ARTICULO alfanumérico
   NUMERO DE UNIDADES VENDIDAS numérico entero
   FECHA DE LA VENTA alfanumérico
   DESCUENTO APLICADO numérico real
   ```

5. Sabiendo que en la empresa del ejercicio anterior existen 10 vendedores y cinco artículos diferentes. Programa que calcula e imprime un listado con las cantidades vendidas de cada artículo por cada vendedor.

6. Programa que recibe como datos de entrada los precios de los cinco artículos y el archivo VENTAS, y confecciona una factura por cada uno de los registros, imprimiendo finalmente el total obtenido por todas las ventas.

CAPITULO 10

Métodos de tratamiento de archivos

10.1. INTRODUCCION

La característica principal de las aplicaciones de gestión es el manejo de gran cantidad de datos. En la mayoría de los casos estas aplicaciones manejan varios archivos, siendo las operaciones de **búsqueda** las más usuales.

En archivos secuenciales, la lectura de un registro para su consulta, modificación o supresión va precedida necesariamente de la búsqueda del mismo.

En archivos directos, la búsqueda de un registro cuya posición es desconocida se realiza secuencialmente, de la misma forma que en los secuenciales

Se tiene una situación análoga a la anterior en los archivos indexados cuando se desea acceder a un registro a partir de un campo que no es la clave, desconociendo el valor de ésta.

Las operaciones de entrada-salida en una aplicación consumen la mayor parte del tiempo de proceso en comparación con el resto de las operaciones internas, por lo cual interesa que dichas operaciones se reduzcan al mínimo imprescindible. Esta optimización, en el caso de las búsquedas, se consigue si el archivo ha sido previamente clasificado.

Los algoritmos de **clasificación** de archivos son muy diversos, dependiendo del tamaño y de las características del archivo a clasificar.

En la mayoría de los sistemas operativos actuales se dispone de programas estándar para realizar la clasificación de un archivo (SORT); no obstante es conveniente conocer el mecanismo de estos algoritmos, por lo que estudiaremos algunos de ellos.

Muchos algoritmos de clasificación externa se basan en la realización de sucesivas particiones y mezclas del archivo a clasificar, por lo que también se presentan los algoritmos básicos para ello.

La **mezcla** de archivos, además de utilizarse para la clasificación, se emplea para otras aplicaciones, disponiendo los sistemas operativos de programas estándar para realizarla (MERGE).

Otras aplicaciones importantes de gestión para las que se exponen métodos básicos son la **actualización**, las **rupturas de secuencia** y la **sincronización de archivos.**

10.2. BUSQUEDA EN ARCHIVOS SECUENCIALES

Se utiliza un algoritmo de búsqueda cuando se desea obtener el contenido de un registro de un archivo, a partir del valor de uno de sus campos o subcampos, que denominaremos «clave de búsqueda».

Como se ha indicado anteriormente, la búsqueda en las organizaciones directa e indexada, cuando sea necesaria, se realizará de forma secuencial, utilizándose los algoritmos que veremos a continuación.

Los algoritmos consisten en un recorrido lineal del archivo, variando la condición de terminación según si el archivo está ordenado o no por la clave de búsqueda.

10.2.1. BUSQUEDA EN ARCHIVOS DESORDENADOS

Se recorre el archivo desde el primer registro hasta encontrar aquel donde el valor de su campo clave coincide con el buscado o hasta que se acabe el archivo, en cuyo caso se debe indicar la inexistencia de dicho registro.

Si existiesen varios registros con el mismo valor del campo clave, igual al buscado, se obtendrá el primero de ellos.

Sea un archivo F, cuyos registros R contienen un campo C, que es la clave de búsqueda, y un valor X a buscar en C.

- **Pseudocódigo:**

```
** En los algoritmos del presente capitulo
** Se utilizan interruptores (SW) numéricos
...
abrir F para lectura
SW ← 0
mientras SW = 0 y no FF(F) hacer
  leer F, R
  si C = X
    entonces SW ← 1
  finsi
finmientras
si SW = 1
  entonces escribir R
  sino escribir "No existe"
finsi
cerrar F
...
```

Si se desea obtener todos los registros cuyo campo C vale X se realizará el siguiente algoritmo que recorre F hasta el final:

- **Pseudocódigo:**

```
...
abrir F para lectura
SW ← 0
mientras no FF(F) hacer
  leer F, R
  si C = X
    entonces escribir R
             SW ← 1
  finsi
finmientras
si SW = 0
  entonces escribir "No existen"
finsi
cerrar F
...
```

10.2.2. BUSQUEDA EN ARCHIVOS ORDENADOS

Los algoritmos del apartado anterior son válidos para archivos ordenados. No obstante conviene aprovechar la ordenación para optimizarlos en cuanto a su tiempo de ejecución, ampliando la condición de terminación de la búsqueda al caso de sobrepasar el valor de la clave buscada (supondremos que la ordenación es ascendente).

- **Pseudocódigo:**

```
...
abrir F para lectura
SW ← 0
mientras SW = 0 y no FF(F) hacer
  leer F, R
  si C >= X
    entonces SW ← 1
  finsi
finmientras
si SW = 1
  entonces si C = X
              entonces escribir R
              sino escribir "No existe"
            finsi
  sino escribir "No existe"
finsi
cerrar F
...
```

Si se desea obtener todos los registros cuyo campo C vale X, éstos ocuparán posiciones consecutivas en el archivo, si existen, terminándose el recorrido al obtenerlos todos.

- **Pseudocódigo:**

```
...
abrir F para lectura
SW ← 0
EXISTE ← 0
mientras SW = 0 y no FF(F) hacer
  leer F, R
  si C > X
    entonces SW ← 1
    sino si C = X
           entonces escribir R
                    EXISTE ← 1
         finsi
  finsi
finmientras
si EXISTE = 0
  entonces escribir "No existen"
finsi
cerrar F
...
```

10.3. PARTICION DE ARCHIVOS

Consiste en repartir los registros de un archivo en otros dos o más, dependiendo de una determinada condición.

10.3.1. PARTICION POR CONTENIDO

La condición la determinan los valores de uno o más campos del registro.

Sea el archivo F, que se desea dividir en dos, F1 y F2, copiando en el primero los registros de F que contienen en el campo C el valor X y en el segundo los demás.

- **Pseudocódigo:**

```
...
abrir F para lectura
abrir F1 para escritura
abrir F2 para escritura
mientras no FF(F) hacer
  leer F, R
  si C = X
    entonces R1 ← R
              escribir F1, R1
    sino      R2 ← R
              escribir F2, R2
  finsi
finmientras
cerrar F, F1, F2
...
```

10.3.2. PARTICION EN SECUENCIAS

Los registros se distribuyen en secuencias alternativas, de igual o diferente longitud, según los casos.

Sea el archivo F, que se desea dividir en dos, F1 y F2, copiando en el primero los registros de F que ocupan posiciones impares y en el segundo los que ocupan posiciones pares (la longitud de las secuencias es 1).

- **Pseudocódigo:**

```
...
abrir F para lectura
abrir F1 para escritura
abrir F2 para escritura
SW ← -1
mientras no FF(F) hacer
  leer F, R
  SW ← -SW
  si SW = 1
    entonces R1 ← R
              escribir F1, R1
    sino      R2 ← R
              escribir F2, R2
  finsi
  finmientras
  cerrar F, F1, F2
  ...
```

Sea el archivo F, que se desea dividir en dos, F1 y F2, copiando alternativamente en uno y otro secuencias de registros de longitud N.

- **Pseudocódigo:**

```
...
abrir F para lectura
abrir F1 para escritura
abrir F2 para escritura
SW ← 1
C  ← 0
mientras no FF(F) hacer
  leer F, R
  si SW = 1
    entonces R1 ← R
```

```
                escribir F1, R1
      sino      R2 ← R
                escribir F2, R2
    finsi
    C ← C + 1
    si C = N
      entonces SW ← -SW
                C  ← 0
    finsi
  finmientras
  cerrar F, F1, F2
  ...
```

Existen otras particiones con diferentes condiciones y longitudes de secuencia cuyos algoritmos son similares a los anteriores.

10.4. MEZCLA DE ARCHIVOS

También denominada fusión o intercalación, consiste en reunir en un archivo los registros de dos o más archivos, manteniendo el posible orden que hubiese establecido.

Una mezcla de archivos desordenados consiste en intercalar secuencias de registros de una determinada longitud alternativamente en el archivo destino. Los algoritmos para hacerlo son los inversos de los de partición.

Las mezclas que se estudian a continuación son de aplicación a dos archivos ordenados ascendentemente por el valor de un campo que denominamos clave, y aunque se pueden generalizar para mezclar N archivos ordenados, es recomendable, en estos casos, realizar sucesivas mezclas de dos en dos por la complejidad de los algoritmos que resultan de la citada generalización.

10.4.1. MEZCLA CON REGISTRO CENTINELA

Se dice que un archivo tiene un registro «centinela» si se le ha añadido un registro al final con el único objetivo de utilizarlo como condición de fin de archivo.

Sean dos archivos, F1 y F2, con registros de igual estructura, ordenados por el campo clave C y con un registro centinela de clave máxima CMAX (el valor del campo C en todos los demás registros es inferior al del centinela). Se desea obtener un archivo F ordenado por el mismo campo que contenga los registros de ambos.

- **Pseudocódigo:**

```
...
abrir F1 para lectura
abrir F2 para lectura
abrir F para escritura
leer F1, R1
leer F2, R2
mientras C1 < CMAX o C2 < CMAX hacer
  si C1 < C2
    entonces
      R ← R1
      escribir F, R
      leer F1, R1
    sino
      R ← R2
      escribir F, R
      leer F2, R2
  finsi
```

```
finmientras
** escritura del centinela en F
R ← R1
escribir F, R
cerrar F1, F2, F
...
```

10.4.2. MEZCLA CONTROLADA POR VALOR DE CLAVE MAXIMO

Si los archivos que se desea mezclar no tienen registro centinela (registro con valor de clave CMAX, superior al valor del campo clave de todos los registros de los dos archivos), se puede simular esta circunstancia dentro del programa, haciendo que cada vez que se necesite leer en un archivo, si éste se ha terminado, se asigne el valor CMAX en el campo clave del registro correspondiente.

De esta manera, el algoritmo de mezcla es muy similar al anterior, con la diferencia de que habrá que comprobar la condición de fin de archivo (FF) antes de cada lectura para asignar el valor CMAX cuando la condición se cumpla.

Sean dos archivos, F1 y F2, con registros de igual estructura, ordenados por el campo clave C, cuyos valores son todos inferiores al valor CMAX. Se desea obtener un archivo F ordenado por el mismo campo que contenga los registros de ambos.

- **Pseudocódigo:**

```
...
abrir F1 para lectura
abrir F2 para lectura
abrir F para escritura
si FF(F1)
  entonces C1 ← CMAX
  sino     leer F1, R1
finsi
si FF(F2)
  entonces C2 ← CMAX
  sino     leer F2, R2
finsi
mientras C1 < CMAX o C2 < CMAX hacer
  si C1 < C2
    entonces
        R ← R1
        escribir F, R
              si FF(F1)
                entonces C1 ← CMAX
                sino     leer F1, R1
              finsi
    sino
        R ← R2
        escribir F, R
              si FF(F2)
                entonces C2 ← CMAX
                sino     leer F2, R2
              finsi
  finsi
finmientras
cerrar F1, F2, F
...
```

10.4.3. MEZCLA CONTROLADA POR FIN DE ARCHIVO

Sean dos archivos, F1 y F2, con registros de igual estructura y ordenados por el campo clave C (C1 para el registro R1 de F1 y C2 para el registro R2 de F2). Se desea obtener F ordenado por el mismo campo que contenga los registros de ambos.

El algoritmo consiste en el recorrido simultáneo de F1 y F2, copiando el registro menor de ambos en F y avanzando en el archivo correspondiente. Se controla la terminación de los archivos por medio de dos *switches*, SW1 y SW2.

A la salida del bucle de fusión, uno de los dos archivos no habrá sido copiado en su totalidad, por lo que es necesario añadir dos bucles de copia para completar la fusión con el resto del archivo no copiado. El procesador detecta cuál es el archivo no completado por el valor de su *switch*.

- **Pseudocódigo:**

```
...
abrir F1 para lectura
abrir F2 para lectura
abrir F para escritura
SW1 ← 0
SW2 ← 0
si FF(F1)
  entonces SW1 ← 1
  sino leer F1, R1
finsi
si FF(F2)
  entonces SW2 ← 1
  sino leer F2, R2
finsi
mientras SW1 = 0 y SW2 = 0 hacer
  si C1 < C2
    entonces
      R ← R1
      escribir F, R
      si FF(F1)
        entonces SW1 ← 1
        sino leer F1, R1
      finsi
    sino
      R ← R2
      escribir F, R
      si FF(F2)
        entonces SW2 ← 1
        sino leer F2, R2
      finsi
  finsi
finmientras
si SW1 = 0
  entonces
    iterar
      R ← R1
      escribir F, R
      salir si FF(F1)
      leer F1, R1
    finiterar
finsi
si SW2 = 0
  entonces
    iterar
```

```
        R — R2
        escribir F, R
        salir si FF(F2)
        leer F2, R2
      finiterar
finsi
cerrar F1, F2, F
...
```

10.5. CLASIFICACION DE ARCHIVOS

Se dice que un archivo está clasificado ascendente o descendentemente si tiene todos sus registros en secuencia ascendente o descendente respecto al valor de un campo que denominamos **clave de ordenación**.

Un algoritmo de clasificación tiene por objeto redistribuir los registros para que se cumpla la condición de ordenación.

Si el archivo a ordenar, por su tamaño, cabe íntegramente en la memoria central, se carga en una tabla y se realiza una clasificación interna recopiando el resultado en el archivo original.

Las clasificaciones externas se realizan si el archivo no cabe en la memoria central, teniendo la desventaja de que su tiempo de ejecución es mucho mayor por la gran cantidad de operaciones de entrada y salida que conlleva.

Algunos algoritmos de clasificacón son una combinación de ordenación externa e interna, aprovechando al máximo la capacidad de la memoria central.

Se presentan a continuación tres ejemplos de métodos de clasificación externa, de entre los muchos existentes, seleccionados por su sencillez.

10.5.1. CLASIFICACION POR MEZCLA DIRECTA

Se trata de la realización sucesiva de una partición y una mezcla que produce secuencias ordenadas de longitud cada vez mayor.

La primera partición se hace en secuencias de longitud 1, y la fusión correspondiente produce sobre el archivo inicial secuencias ordenadas de longitud 2.

A cada nueva partición y fusión se duplica la longitud de las secuencias ordenadas.

El proceso termina cuando la longitud de la secuencia ordenada excede la longitud del archivo a ordenar.

Ejemplo: *Sea un archivo F cuyos valores del campo clave son los que figuran a continuación. Se utilizan dos archivos auxiliares, F1 y F2, con la misma estructura que F para realizar las sucesivas particiones y fusiones.*

```
F: 15,18,7,75,14,13,43,40,51,93,75,26,64,27,13
```

Partición en secuencias de longitud 1.

```
F1: 15,  7, 14, 43, 51, 75, 64, 13
F2: 18, 75, 13, 40, 93, 26, 27
```

Fusión de secuencias de longitud 1.

```
F: 15,18, 7,75, 13,14, 40,43, 51,93, 26,75, 27,64, 13
```

Partición en secuencias de longitud 2.

```
F1: 15,18, 13,14, 51,93, 27,64
F2:  7,75, 40,43, 26,75, 13
```

Fusión de secuencias de longitud 2.

```
F: 7,15,18,75, 13,14,40,43, 26,51,75,93, 13,27,64
```

Partición en secuencias de longitud 4.

```
F1:  7,15,18,75, 26,51,75,93
F2: 13,14,40,43, 13,27,64
```

Fusión de secuencias de longitud 4.

```
F: 7,13,14,15,18,40,43,75, 13,26,27,51,64,75,93
```

Partición en secuencias de longitud 8.

```
F1:  7,13,14,15,18,40,43,75
F2: 13,26,27,51,64,75,93
```

Fusión de secuencias de longitud 8.

```
F: 7,13,13,14,15,18,26,27,40,43,51,64,75,75,93
```

El proceso se termina al detectarse que la nueva longitud de la secuencia para partición es mayor o igual que el número de registros de F.

10.5.2. CLASIFICACION POR MEZCLA EQUILIBRADA

Es una optimización del método anterior, consistente en realizar la partición tomando las secuencias ordenadas de máxima longitud posible y realizando la fusión de secuencias ordenadas alternativamente sobre dos archivos, lo que hace que la siguiente partición quede realizada.

Utiliza tres archivos auxiliares, junto con el original, siendo alternativamente dos de ellos de entrada y los otros dos de salida, para la realización simultánea de fusión y partición.

Durante el proceso de fusión-partición, dos o más secuencias ascendentes, que estén consecutivas, pueden constituir una única secuencia para el paso siguiente.

El proceso termina cuando en la realización de una fusión-partición el segundo archivo queda vacío. El archivo totalmente ordenado estará en el primero.

Ejemplo: *Sea F el archivo a ordenar y F1, F2 y F3 los auxiliares. Con las mismas claves del ejemplo anterior, el proceso es el siguiente.*

```
F: 15,18,7,75,14,13,43,40,51,93,75,26,64,27,13
F1:
```

Partición inicial.

```
F2: 15,18, 14, 40,51,93, 26,64, 13
F3: 7,75, 13,43, 75, 27
```

Primera fusión partición.

```
F: 7,15,18,75, 26,27,64
F1: 13,14,40,43,51,75,93, 13
```

Segunda fusión-partición.

```
F2: 7,13,14,15,18,40,43,51,75,75,93
F3: 13,26,27,64
```

Tercera fusión-partición.

```
F: 7,13,13,14,15,18,26,27,40,43,51,64,75,75,93
F1:
```

10.5.3. CLASIFICACION DE RAIZ

Se utiliza para claves de ordenación numéricas. Consiste en distribuir los registros en 10 archivos auxiliares, numerados del 0 al 9, según el dígito de la clave que corresponde al número de pasada que se está realizando. A continuación se concatenan los 10 archivos en el archivo original.

Los dígitos de la clave se toman de derecha a izquierda, completándose la ordenación en tantos pasos como dígitos tenga el campo clave.

Ejemplo: *Sea F el archivo de los ejemplos anteriores y F0, F1, . . ., F9 los 10 archivos auxiliares. El proceso necesitará dos pasadas compuestas por una partición y una concatenación.*

```
F: 15,18,07,75,14,13,43,40,51,93,75,26,64,27,13
```

Primera partición.

```
F0: 40
F1: 51
F2:
F3: 13,43,93,13
F4: 14,64
F5: 15,75,75
F6: 26
F7: 07,27
F8: 18
F9:
```

Primera concatenación.

```
F: 40,51,13,43,93,13,14,64,15,75,75,26,07,27,18
```

Segunda partición.

```
F0: 07
F1: 13,13,14,15,18
F2: 26,27
F3:
F4: 40,43
F5: 51
F6: 64
F7: 75,75
F8:
F9: 93
```

Segunda concatenación.

```
F: 07,13,13,14,15,18,26,27,40,43,51,64,75,75,93
```

10.6. ACTUALIZACION DE ARCHIVOS

Los archivos maestros o de situación son una clase de archivos permanentes que reflejan el estado actual o situación de una entidad o de un aspecto de la misma. Cuando se desarrolla la actividad normal de la entidad, los datos sufren modificaciones, por lo que es preciso actualizar estos archivos para que reflejen la nueva situación.

La actualización de un archivo maestro se realiza periódicamente con las transacciones o movimientos habidos, relativos a los datos almacenados, entre cada periodo de actualización.

Las transacciones pueden ser proporcionadas al proceso de actualización de forma interactiva por teclado, pero lo más normal es haberlas registrado previamente en un archivo de movimientos durante el tiempo transcurrido desde la última actualización. Este tiempo, que se fija para cada aplicación, puede variar desde un día, una semana, un mes, etc.

10.6.1. ACTUALIZACION DE ARCHIVOS SECUENCIALES

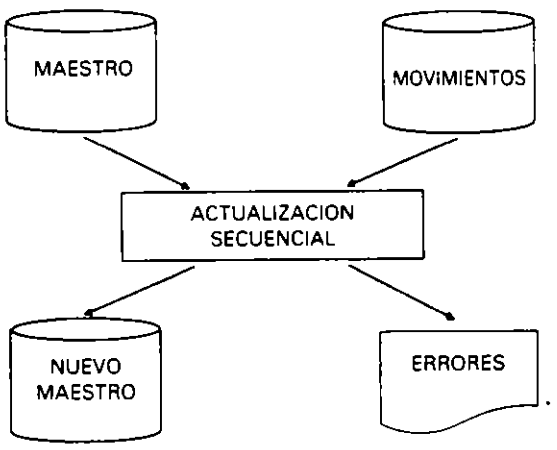
Un archivo secuencial no puede ser actualizado sobre sí mismo. Lo más que se puede hacer, excepcionalmente, es modificar el contenido de sus registros o añadir nuevos registros al final del mismo.

El proceso de actualización general consistirá en crear un nuevo archivo maestro a partir del antiguo, añadiendo las altas, suprimiendo las bajas, copiando los registros modificados en su caso y los que quedan como estaban. Simultáneamente se creará un archivo de errores, normalmente en impresora para tratamiento manual, donde se incluirán aquellos movimientos que no han podido ser reflejados en el nuevo archivo maestro por ser inconsistentes: altas de registros ya existentes y bajas o modificaciones de registros inexistentes.

Para poder llevar a cabo el proceso de actualización es necesario que los archivos maestro y movimientos estén clasificados por el mismo campo clave. Los archivos obtenidos, nuevo maestro y errores, mantendrán también esta ordenación.

Supondremos el caso en que no existen movimientos con clave repetida, es decir, para cualquier clave hay, como mucho, un movimiento.

- **Organigrama:**



Descripción del proceso

El proceso consiste en un recorrido simultáneo de los archivos MAESTRO y MOVIMIENTOS. De forma similar a como se hacía en la mezcla de archivos se lee inicialmente

un registro de cada uno de los archivos. sucesivamente se comparan sus valores del campo clave y se analiza cada situación que da lugar a un tratamiento particular. Si se acaba un archivo se almacena en el campo clave de su registro asociado un valor CMAX que es superior a todos los valores de clave posibles.

Este valor CMAX hace que se pueda continuar con el proceso del otro archivo que no se ha terminado sin necesidad de variar el algoritmo.

El proceso termina cuando en los dos registros asociados se tiene almacenado el valor CMAX. Esto implica que se han terminado ambos archivos y que todos sus registros han sido procesados.

Se describen a continuación las situaciones que se pueden presentar y el tratamiento para cada una de ellas.

— Clave del maestro menor que clave de movimientos:
  - Corresponde a un registro del maestro que queda como está: por tanto se copia en el nuevo maestro y se lee un nuevo registro del maestro.

— Clave del maestro igual a la de movimientos:
  - Si es una baja se prescinde del registro del maestro y se lee un nuevo registro de ambos archivos.
  - Si es un modificación se cambian los campos del registro del maestro. se copia en el nuevo maestro y se lee del maestro y movimientos.
  - Si es un alta corresponde a un error de alta existente: se registra el error. se copia el registro del maestro en el nuevo maestro y se leen nuevos registros del maestro y movimientos.

— Clave del maestro mayor que la de movimientos:
  - Si es un alta se procesa correctamente copiándola en el nuevo maestro y se lee un nuevo movimiento.
  - Si es una baja corresponde a un error de baja inexistente. se registra en el archivo de errores y se lee un nuevo movimiento.
  - Si es una modificación. también es un error de clave inexistente que se registra antes de leer el siguiente movimiento.

- **Pseudocódigo:**

```
Programa ACTUALIZACION-SECUENCIAL
Entorno:
  MAESTRO es archivo de REG-MAESTRO
  REG-MAESTRO es registro compuesto de
    CLAVE-MAESTRO ** de cualquier tipo ordinal
    RESTO-CAMPOS  ** sin especificar
  finregistro
  NUEVO-MAESTRO es archivo de REG-NUEVO
  REG-NUEVO es registro compuesto de
    CLAVE-NUEVO   ** del mismo tipo que CLAVE-MAESTRO
    RESTO-CAMPOS ** iguales que en el archivo MAESTRO
  finregistro
  MOVIMIENTOS es archivo de REG-MOVI
  REG-MOVI es registro compuesto de
    TIPO-MOVI    de tipo carácter ** "A", "B" o "M"
    DATOS-MOVI es registro compuesto de
      CLAVE-MOVI    ** del mismo tipo que CLAVE-MAESTRO
      RESTO-CAMPOS ** iguales que en el archivo MAESTRO
    finregistro
  finregistro
  ERRORES es archivo de REG-ERROR
  REG-ERROR es registro compuesto de
    TIPO-ERROR    ** del mismo tipo que TIPO-MOVI
    CLAVE-ERROR   ** del mismo tipo que CLAVE-MAESTRO
    RESTO-CAMPOS ** iguales que en el archivo MAESTRO
```

```
  finregistro
**
Algoritmo:
  abrir MAESTRO para lectura
  abrir MOVIMIENTOS para lectura
  abrir NUEVO-MAESTRO para escritura
  abrir ERRORES para escritura
  ** lectura del primer registro del archivo maestro
  LEER-MAESTRO
  ** lectura del primer movimiento
  LEER-MOVIMIENTO
  mientras CLAVE-MAESTRO < CMAX o CLAVE-MOVI < CMAX hacer
    si CLAVE-MAESTRO < CLAVE-MOVI
      entonces
        ** copia de un registro que queda igual
        REG-NUEVO ← REG-MAESTRO
        escribir NUEVO-MAESTRO, REG-NUEVO
        ** lectura del siguiente registro del archivo maestro
        LEER-MAESTRO
      sino
        si CAVE-MAESTRO = CLAVE-MOVI
          entonces
            opción TIPO-MOVI de
              "A" hacer
                ** error de alta existente
                REG-ERROR ← REG-MOVI
                escribir ERRORES, REG-ERROR
                REG-NUEVO ← REG-MAESTRO
                escribir NUEVO-MAESTRO, REG-NUEVO
              "B" hacer
                ** dar de baja, no copiar REG-MAESTRO
              "M" hacer
                ** modificar campos de REG-MAESTRO con DATOS-MOVI
                REG-NUEVO ← DATOS-MOVI
                escribir NUEVO-MAESTRO, REG-NUEVO
            finopción
            ** lectura del siguiente registro del archivo maestro
            LEER-MAESTRO
            ** lectura del siguiente movimiento
            LEER-MOVIMIENTO
          sino ** CLAVE-MAESTRO > CLAVE-MOVI
            opción TIPO-MOVI de
              "A" hacer
                ** dar de alta un nuevo registro
                REG-NUEVO ← DATOS-MOVI
                escribir NUEVO-MAESTRO, REG-NUEVO
              "B" hacer
                ** error de baja inexistente
                REG-ERROR ← REG-MOVI
                escribir ERRORES, REG-ERROR
              "M" hacer
                ** error de modificación inexistente
                REG-ERROR ← REG-MOVI
                escribir ERRORES, REG-ERROR
            finopción
            ** lectura del siguiente movimiento
            LEER-MOVIMIENTO
        finsi
    finsi
  finmientras
```

```
  cerrar MAESTRO
  cerrar MOVIMIENTOS
  cerrar NUEVO-MAESTRO
  cerrar ERRORES
Finprograma
**
Subprograma LEER-MAESTRO
Algoritmo:
  si ff(MAESTRO)
    entonces CLAVE-MAESTRO ← CMAX
    sino leer MAESTRO, REG-MAESTRO
  finsi
Finsubprograma
**
Subprograma LEER-MOVIMIENTO
Algoritmo:
  si ff(MOVIMIENTOS)
    entonces CLAVE-MOVI ← CMAX
    sino leer MOVIMIENTOS, REG-MOVI
  finsi
Finsubprograma
```

10.6.2. ACTUALIZACION DE ARCHIVOS DIRECTOS

Los archivos directos pueden ser actualizados sobre si mismos, escribiendo los nuevos registros (altas), borrando los sobrantes (bajas) y reescribiendo los que varíen sus datos (modificaciones). Los movimientos pueden proceder directamente del teclado (actualización interactiva) o de un archivo secuencial de movimientos que no necesita estar ordenado. En este caso el proceso consiste en recorrer el archivo de movimientos procesando cada una de sus transacciones.

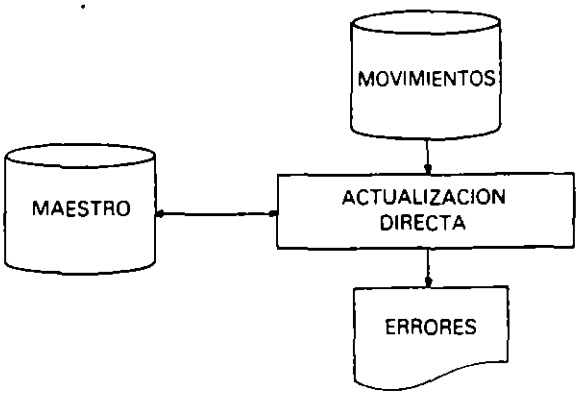
Si en el archivo maestro no se tiene una relación entre el campo clave y la posición de almacenamiento, cada movimiento requerirá una búsqueda previa en el archivo maestro para localizar el registro afectado, si existe, y una decisión sobre dónde colocar los nuevos registros a dar de alta.

En el caso anterior, si estuviesen ordenados por clave los archivos maestro y movimientos, el mejor algoritmo de actualización es el utilizado para los archivos secuenciales adaptado a esta organización y sobre el mismo archivo maestro.

Supondremos el caso de direccionamiento directo por clave; es decir, existe una relación entre la clave y la dirección de almacenamiento que permite el acceso mediante clave o mediante una transformación sencilla sin la aparición de *sinónimos* (claves diferentes a las que les corresponderia la misma dirección).

Por ejemplo, supondremos en nuestro caso que el valor de la clave coincide con la dirección de almacenamiento.

- **Organigrama:**



- **Pseudocódigo:**

```
Programa ACTUALIZACION-DIRECTA
Entorno:
  MAESTRO es archivo directo de REG-MAESTRO
  REG-MAESTRO es registro compuesto de
    CLAVE-MAESTRO es numerico entero ** coincide con la dirección
    RESTO-CAMPOS ** sin especificar
  finregistro
  MOVIMIENTOS es archivo de REG-MOVI
  REG-MOVI es registro compuesto de
    TIPO-MOVI  de tipo carácter ** "A", "B" o "M"
    DATOS-MOVI es registro compuesto de
      CLAVE-MOVI    ** del mismo tipo que CLAVE-MAESTRO
      RESTO-CAMPOS ** iguales que en el archivo MAESTRO
    finregistro
  finregistro
  ERRORES es archivo de REG-ERROR
  REG-ERROR es registro compuesto de
    TIPO-ERROR    ** del mismo tipo que TIPO-MOVI
    CLAVE-ERROR   ** del mismo tipo que CLAVE-MAESTRO
    RESTO-CAMPOS ** iguales que en el archivo MAESTRO
  finregistro
**
Algoritmo:
  abrir MAESTRO directo
  abrir MOVIMIENTOS para lectura
  abrir ERRORES para escritura
  mientras no ff(MOVIMIENTOS) hacer
    leer MOVIMIENTOS, REG-MOVI
    opción TIPO-MOVI de
      "A" hacer
        REG-MAESTRO ← DATOS-MOVI
        escribir MAESTRO, REG-MAESTRO, CLAVE-MAESTRO
        si CLAVE-INVALIDA
          entonces
            ** error de alta existente
            REG-ERROR ← REG-MOVI
            escribir ERRORES, REG-ERROR
        finsi
      "B" hacer
        CLAVE-MAESTRO ← CLAVE-MOVI
        borrar MAESTRO, CLAVE-MAESTRO
        si CLAVE-INVALIDA
          entonces
            ** error de baja inexistente
            REG-ERROR ← REG-MOVI
            escribir ERRORES, REG-ERROR
        finsi
      "M" hacer
        ** modificar campos de REG-MAESTRO con DATOS-MOVI
        REG-MAESTRO ← DATOS-MOVI
        reescribir MAESTRO, REG-MAESTRO, CLAVE-MAESTRO
        si CLAVE-INVALIDA
          entonces
            ** error de modificación inexistente
            REG-ERROR ← REG-MOVI
            escribir ERRORES, REG-ERROR
        finsi
    finopción
```

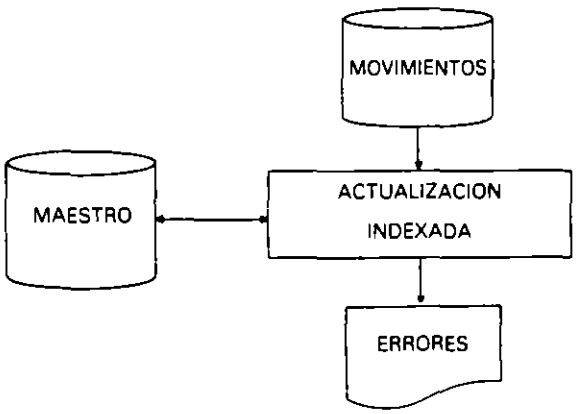
```
  finmientras
  cerrar MAESTRO
  cerrar MOVIMIENTOS
  cerrar ERRORES
Finprograma
```

10.6.3. ACTUALIZACION DE ARCHIVOS INDEXADOS

Es tal vez la más sencilla de realizar. Los movimientos se procesan sobre el mismo archivo maestro, no necesitándose que el archivo de movimientos esté ordenado, ya que el acceso al archivo maestro se hace de forma directa por el campo clave.

En caso de tener el archivo de movimientos ordenado, se podría utilizar una versión del algoritmo de actualización secuencial teniendo en cuenta que los movimientos se reflejarán en el mismo archivo maestro. Aunque este algoritmo sería más eficiente en cuanto a tiempo de proceso, su programación es más complicada que la del que se presenta a continuación.

- **Organigrama:**



- **Pseudocódigo:**

```
Programa ACTUALIZACION-INDEXADA
Entorno:
  MAESTRO es archivo indexado de REG-MAESTRO
  REG-MAESTRO es registro compuesto de
    CLAVE-MAESTRO es campo clave ** de cualquier tipo ordinal
    RESTO-CAMPOS  ** sin especificar
  finregistro
  MOVIMIENTOS es archivo de REG-MOVI
  REG-MOVI es registro compuesto de
    TIPO-MOVI    de tipo carácter ** "A", "B" o "M"
    DATOS-MOVI es registro compuesto de
      CLAVE-MOVI    ** del mismo tipo que CLAVE-MAESTRO
      RESTO-CAMPOS ** iguales que en el archivo MAESTRO
    finregistro
  finregistro
  ERRORES es archivo de REG-ERROR
  REG-ERROR es registro compuesto de
    TIPO-ERROR    ** del mismo tipo que TIPO-MOVI
    CLAVE-ERROR   ** del mismo tipo que CLAVE-MAESTRO
    RESTO-CAMPOS ** iguales que en el archivo MAESTRO
```

```
    finregistro
  Algoritmo:
    abrir MAESTRO indexado
    abrir MOVIMIENTOS para lectura
    abrir ERRORES para escritura
    mientras no ff(MOVIMIENTOS) hacer
      leer MOVIMIENTOS, REG-MOVI
      opción TIPO-MOVI de
        "A" hacer
          REG-MAESTRO ← DATOS-MOVI
          escribir MAESTRO, REG-MAESTRO, CLAVE-MAESTRO
          si CLAVE-INVALIDA
            entonces
              ** error de alta existente
              REG-ERROR ← REG-MOVI
              escribir ERRORES, REG-ERROR
          finsi
        "B" hacer
          CLAVE-MAESTRO ← CLAVE-MOVI
          borrar MAESTRO, CLAVE-MAESTRO
          si CLAVE-INVALIDA
            entonces
              ** error de baja inexistente
              REG-ERROR ← REG-MOVI
              escribir ERRORES, REG-ERROR
          finsi
        "M" hacer
          ** modificar campos de REG-MAESTRO con DATOS-MOVI
          REG-MAESTRO ← DATOS-MOVI
          reescribir MAESTRO, REG-MAESTRO, CLAVE-MAESTRO
          si CLAVE-INVALIDA
            entonces
              ** error de modificación inexistente
              REG-ERROR ← REG-MOVI
              escribir ERRORES, REG-ERROR
          finsi
      finopción
    finmientras
    cerrar MAESTRO
    cerrar MOVIMIENTOS
    cerrar ERRORES
  Finprograma
```

10.6.4. ACTUALIZACION INTERACTIVA

La actualización interactiva consiste en proporcionar los movimientos por teclado y procesarlos sobre el archivo maestro en ese mismo momento.

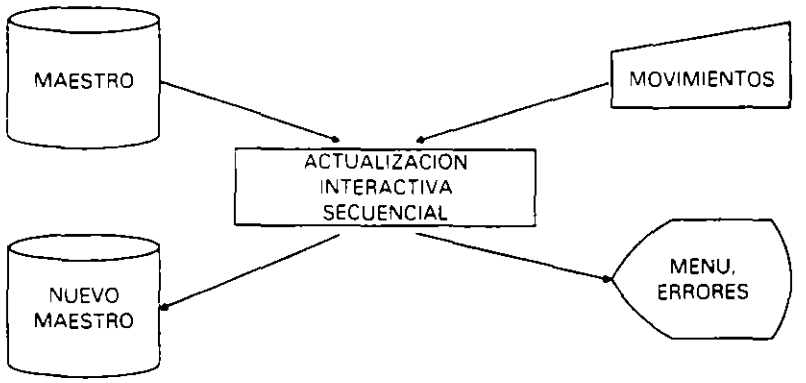
Para actualizar un maestro secuencial se requiere que los movimientos sean proporcionados en el mismo orden del maestro. Sin embargo, para la actualización de un directo (con direccionamiento directo) o de un indexado, los movimientos pueden ser introducidos en cualquier orden.

Es conveniente hacerlo mediante la presentación de un menú de opciones (altas, bajas y modificaciones).

En caso de alta se pedirán todos los nuevos datos. En caso de baja, una vez solicitada la clave y accedido el registro correspondiente, se presentará el registro a borrar, si existe, y se pedirá la confirmación. Por último, en caso de modificación, se hará de forma similar al anterior, pero pidiendo los datos nuevos de cada campo con la posibilidad de no modificarlos y una confirmación posterior para todo el registro modificado.

Actualización interactiva de archivo secuencial.

- **Organigrama:**



- **Pseudocódigo:**

```
Programa ACTUALIZACION-INTERACTIVA-SECUENCIAL
Entorno:
  MAESTRO es archivo de REG-MAESTRO
  REG-MAESTRO es registro compuesto de
    CLAVE-MAESTRO ** de cualquier tipo ordinal
    RESTO-CAMPOS  ** sin especificar
  finregistro
  NUEVO-MAESTRO es archivo de REG-NUEVO
  REG-NUEVO es registro compuesto de
    CLAVE-NUEVO  ** del mismo tipo que CLAVE-MAESTRO
    RESTO-CAMPOS ** iguales que en el archivo MAESTRO
  finregistro
  CLAVE-MOVI es del mismo tipo que CLAVE-MAESTRO
  CLAVE-ANTERIOR es del mismo tipo que CLAVE-MAESTRO
  OPERACION es numérica entera
  FINAL es carácter ** "S" o "N"
  CMAX es del mismo tipo que CLAVE-MAESTRO
  CMIN es del mismo tipo que CLAVE-MAESTRO
  ** CMAX almacena un valor superior a todos los valores posibles
  ** de los campos CLAVE-MAESTRO y CLAVE-MOVI, y CMIN inferior.
Algoritmo:
  abrir MAESTRO para lectura
  abrir NUEVO-MAESTRO para escritura
  CMAX ← máximo valor
  CMIN ← mínimo valor
  CLAVE-MAESTRO ← CMIN
  LEER-MAESTRO
  FINAL ← "N"
  mientras FINAL <> "S" hacer
    PRESENTAR-MENU
    SELECCIONAR-OPERACION
    REALIZAR-OPERACION
  finmientras
  cerrar MAESTRO
  cerrar NUEVO-MAESTRO
Finprograma
**
Subprograma PRESENTAR-MENU
```

```
Algoritmo:
  escribir " Actualización del archivo maestro"
  escribir " ----------------------------------"
  escribir "          1. Alta."
  escribir "          2. Baja."
  escribir "          3. Modificación."
  escribir "          4. Terminar."
  escribir "     Elija operación a realizar:"
Finsubprograma
Subprograma SELECCIONAR-OPERACION
**
Algoritmo:
  leer OPERACION
  mientras OPERACION < 1 o OPERACION > 4 hacer
    escribir " Operación inexistente, elija otra: "
    leer OPERACION
  finmientras
Finsubprograma
**
Subprograma REALIZAR-OPERACION
Algoritmo:
  opción OPERACION de
    1 hacer DAR-ALTA
    2 hacer DAR-BAJA
    3 hacer MODIFICAR
    4 hacer CONFIRMAR-FIN
  finopción
Finsubprograma
**
Subprograma DAR-ALTA
Algoritmo:
  escribir " Escriba la clave:"
  leer CLAVE-MOVI
  si CLAVE-MOVI <= CLAVE-ANTERIOR
    entonces escribir "Error: clave ya existente o fuera de orden"
    sino
      COPIAR-ANTERIORES
      si CLAVE-MOVI = CLAVE-MAESTRO
        entonces escribir "Error: clave ya existente o fuera de orden"
        sino  CLAVE-NUEVO ← CLAVE-MOVI
              escribir " Escriba el resto de campos:"
              leer RESTO-CAMPOS de REG-NUEVO
              escribir NUEVO-MAESTRO, REG-NUEVO
      finsi
  finsi
Finsubprograma
**
Subprograma DAR-BAJA
Algoritmo:
  escribir " Escriba la clave:"
  leer CLAVE-MOVI
  COPIAR-ANTERIORES
  si CLAVE-MAESTRO > CLAVE-MOVI
    entonces escribir "Error: clave no existente o fuera de orden"
    sino     LEER-MAESTRO
  finsi
Finsubprograma
**
Subprograma MODIFICAR
Algoritmo:
```

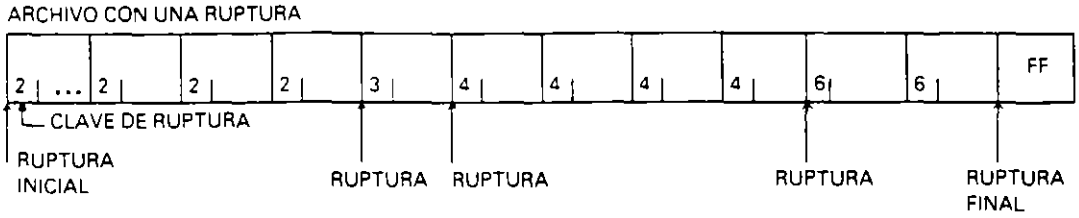
```
    escribir " Escriba la clave:"
    leer CLAVE-MOVI
    COPIAR-ANTERIORES
    si CLAVE-MAESTRO > CLAVE-MOVI
      entonces escribir "Error: clave no existente o fuera de orden"
      sino     CLAVE-NUEVO ← CLAVE-MOVI
               escribir "Escriba el resto de campos:"
               leer RESTO-CAMPOS de REG-NUEVO
               escribir NUEVO-MAESTRO, REG-NUEVO
               LEER-MAESTRO
    finsi
Finsubprograma
**
Subprograma CONFIRMAR-FIN
Algoritmo:
    escribir "¿Está seguro de que quiere terminar (S/N)? "
    leer FINAL
    si FINAL = "S"
      entonces mientras CLAVE-MAESTRO < CMAX hacer
                  REG-NUEVO ← REG-MAESTRO
                  escribir NUEVO-MAESTRO, REG-MAESTRO
                  LEER-MAESTRO
                finmientras
    finsi
Finsubprograma
**
Subprograma COPIAR-ANTERIORES
Algoritmo:
    mientras CLAVE-MAESTRO < CLAVE-MOVI hacer
      REG-NUEVO ← REG-MAESTRO
      escribir NUEVO-MAESTRO, REG-NUEVO
      LEER-MAESTRO
    finmientras
Finsubprograma
**
Subprograma LEER-MAESTRO
Algoritmo:
    CLAVE-ANTERIOR ← CLAVE-MAESTRO
    si ff(MAESTRO)
      entonces CLAVE-MAESTRO ← CMAX
      sino     leer MAESTRO, REG-MAESTRO
    finsi
Finsubprograma
```

Las actualizaciones interactivas de archivos directos e indexados, cuya complejidad es muy inferior a la secuencial, se dejan como ejercicio al lector.

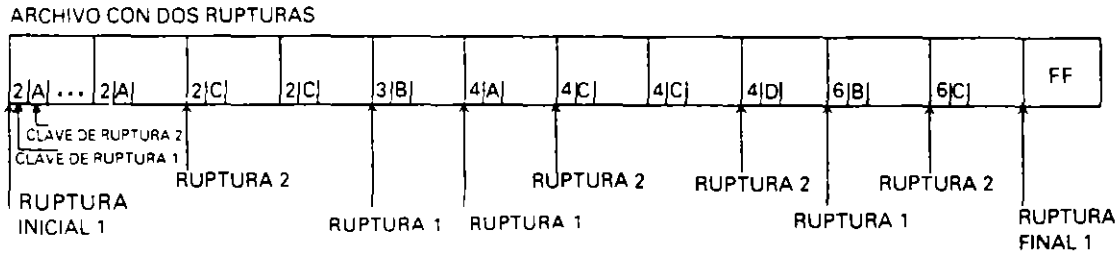
10.7. RUPTURAS DE SECUENCIA

Se define como ruptura de secuencia el tratamiento secuencial de un archivo por grupos homogéneos de registros consecutivos. La secuencia de registros consecutivos está determinada por el mismo valor de algún campo determinado.

Se denomina **clave de ruptura** al campo que produce un cambio de la secuencia. Este campo puede coincidir con alguna clave del archivo o no, pero, en cualquier caso, el archivo ha de estar ordenado respecto a ese campo.



Puede haber un anidamiento de rupturas, correspondiendo cada una de ellas a una subsecuencia obtenida por el cambio de una clave de ruptura secundaria, respecto de la cual el archivo tendrá una ordenación secundaria.



Ejemplo: *Se dispone de un archivo secuencial VENTAS que contiene los datos de las ventas realizadas por una determinada empresa durante un cierto periodo de tiempo. Sus campos son*

```
CODIGO DE VENDEDOR numérico entero
NOMBRE DEL VENDEDOR alfanumérico
DESCRIPCION DE LA VENTA alfanumérico
IMPORTE numérico entero
```

El archivo está ordenado ascendentemente por código de vendedor, habiendo tantos registros por vendedor como ventas haya realizado.

Se desea obtener un listado con el importe total obtenido por cada vendedor y un total general al final con el siguiente formato:

```
                RESUMEN DE VENTAS POR VENDEDOR
                ------------------------------
CODIGO           NOMBRE                          IMPORTE TOTAL
-------------------------------------------------------------
999999           XXXXXXXXXXXXXXXXXX                99999999
   ...           ...                                  ...
999999           XXXXXXXXXXXXXXXXXX                99999999
                                    ------------------------------
                                    TOTAL GENERAL:  999999999999
```

- **Pseudocódigo**

```
Programa TOTAL-VENDEDORES
Entorno:
  VENTAS es archivo de REG-VENTA
  REG-VENTA es registro compuesto de
    CODIGO es numérico entero
    NOMBRE es alfanumérico
    DESCRIPCION es alfanumérico
```

```
    IMPORTE es numérico entero
  finregistro
  TOTAL-VENDEDOR, TOTAL-GENERAL son numéricas enteras
  CODIGO-ANT es numérica entera
  NOMBRE-ANT es alfanumérica
**
Algoritmo:
  abrir VENTAS para lectura
  escribir CABECERA
  TOTAL-GENERAL ← 0
  TOTAL-VENDEDOR ← 0
  leer VENTAS, REG-VENTA
  CODIGO-ANT ← CODIGO
  NOMBRE-ANT ← NOMBRE
  mientras no ff(VENTAS) hacer
    mientras CODIGO-ANT = CODIGO y no ff(VENTAS) hacer
      TOTAL-VENDEDOR ← TOTAL-VENDEDOR + IMPORTE
      leer VENTAS, REG-VENTA
    finmientras
    ** ruptura de secuencia
    si CODIGO-ANT <> CODIGO
      entonces
        escribir CODIGO-ANT, NOMBRE-ANT, TOTAL-VENDEDOR
        TOTAL-GENERAL ← TOTAL-GENERAL + TOTAL-VENDEDOR
        TOTAL-VENDEDOR ← 0
        CODIGO-ANT ← CODIGO
        NOMBRE-ANT ← NOMBRE
    finsi
  finmientras
  TOTAL-VENDEDOR ← TOTAL-VENDEDOR + IMPORTE
  escribir CODIGO-ANT, NOMBRE-ANT, TOTAL-VENDEDOR
  TOTAL-GENERAL ← TOTAL-GENERAL + TOTAL-VENDEDOR
  escribir "      TOTAL GENERAL: ", TOTAL-GENERAL
Finprograma
```

Se obtiene un algoritmo más sencillo si se utiliza un valor CMAX, superior a todos los valores posibles de la clave, para asignarlo al campo CODIGO cuando se detecte el final de archivo:

```
Programa TOTAL-VENDEDORES
Entorno:
  VENTAS es archivo de REG-VENTA
  REG-VENTA es registro compuesto de
    CODIGO es numérico entero
    NOMBRE es alfanumérico
    DESCRIPCION es alfanumérico
    IMPORTE es numérico entero
  finregistro
  TOTAL-VENDEDOR, TOTAL-GENERAL son numéricas enteras
  CODIGO-ANT es numérica entera
  NOMBRE-ANT es alfanumérica
  CMAX es numérica entera
**
Algoritmo:
  abrir VENTAS para lectura
  escribir CABECERA
  TOTAL-GENERAL ← 0
  CMAX ← máximo valor
  LEER-VENTA
  mientras CODIGO < CMAX hacer
```

```
      TOTAL-VENDEDOR ← 0
      CODIGO-ANT ← CODIGO
      NOMBRE-ANT ← NOMBRE
      mientras CODIGO-ANT = CODIGO hacer
        TOTAL-VENDEDOR ← TOTAL-VENDEDOR + IMPORTE
        LEER-VENTA
      finmientras
      ** ruptura de secuencia
      escribir CODIGO-ANT, NOMBRE-ANT, TOTAL-VENDEDOR
      TOTAL-GENERAL ← TOTAL-GENERAL + TOTAL-VENDEDOR
    finmientras
    escribir "      TOTAL GENERAL: ", TOTAL-GENERAL
Finprograma
**
Subprograma LEER-VENTA
Algoritmo:
  si ff(VENTAS)
    entonces CODIGO ← CMAX
    sino     leer VENTAS, REG-VENTA
  finsi
Finsubprograma
```

10.8. SINCRONIZACION DE ARCHIVOS

Se define como sincronización o cruce de archivos al tratamiento secuencial de dos o más archivos, que tienen al menos un campo común, con el fin de obtener todos los datos relacionados con el citado campo, cuando su valor coincida en todos los archivos, y realizar algún proceso concreto con ellos.

Se denomina **clave de sincronización** al campo común que se utiliza en la aplicación de sincronización, y es necesario que los archivos que intervengan estén ordenados respecto a ese campo.

Si los archivos tienen organización indexada puede utilizarse como clave de sincronización la clave primaria o cualquiera de las claves secundarias que sea común a todos ellos.

En cualquier caso, el proceso a realizar consiste en recorrer simultáneamente los archivos, haciendo avanzar el de clave menor. Cada vez que se tiene el mismo valor del campo clave en todos ellos se dice que se ha producido una sincronización, procediéndose entonces al tratamiento del total de los datos de los registros correspondientes.

El proceso se detendrá cuando se termine con la lectura de cualquiera de los archivos, puesto que no se podrá obtener ninguna sincronización posterior.

Ejemplo: *Una empresa de selección de personal ha recopilado datos de solicitantes de empleo en tres archivos diferentes. Algunas personas tienen datos en los tres archivos, pero otras sólo figuran en uno o dos de ellos.*

Los tres archivos son secuenciales, tienen el campo común DNI y están clasificados ascendentemente respecto a este campo. El primer archivo contiene solamente datos personales, el segundo datos académicos y el tercero datos profesionales.

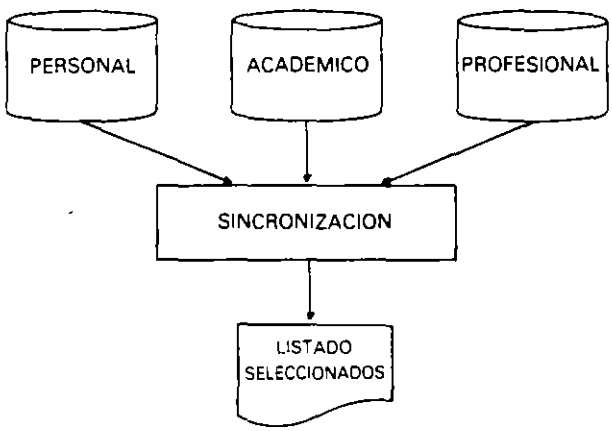
Los registros tienen la siguiente estructura:

```
REGISTRO PERSONAL
    D.N.I. alfanumérico
    NOMBRE alfanumérico
    EDAD numérico entero
    RESTO DATOS PERSONALES alfanumérico
```

```
REGISTRO ACADEMICO
     D.N.I. alfanumérico
     NIVEL DE ESTUDIOS carácter
(A = Sin estudios, B = Básicos, C = Medios, D = Superiores)
     RESTO DATOS ACADEMICOS alfanumérico

REGISTRO PROFESIONAL
     D.N.I. alfanumérico
     SITUACION LABORAL carácter
(A = Fijo, B = Temporal, C = Paro con subsidio, D = Paro sin subsidio)
     EXPERIENCIA booleano
     RESTO DATOS PROFESIONALES alfanumérico
```

Se desea un programa que obtenga un listado con el DNI y el nombre de las personas con edad superior a 21 años, con nivel de estudios superiores, sin experiencia laboral y que se encuentren en paro.

- **Organigrama:**



- **Pseudocódigo:**

```
Programa SELECCION
Entorno:
  PERSONAL es archivo de REG-PERSONAL
  REG-PERSONAL es registro compuesto de
    DNI1 es alfanumérico
    NOMBRE es alfanumérico
    EDAD es numérico entero
    RESTO-DATOS es alfanumérico
  finregistro
  ACADEMICO es archivo de REG-ACADEMICO
  REG-ACADEMICO es registro compuesto de
    DNI2 es alfanumérico
    NIVEL es carácter
    RESTO-DATOS es alfanumérico
  finregistro
  PROFESIONAL es archivo de REG-PROFESIONAL
  REG-PROFESIONAL es registro compuesto de
    DNI3 es alfanumérico
    SITUACION es carácter
    EXPERIENCIA es booleano
    RESTO-DATOS es alfanumérico
```

```
  finregistro
**
Algoritmo:
  ESCRIBIR-CABECERAS
  abrir PERSONAL para lectura
  abrir ACADEMICO para lectura
  abrir PROFESIONAL para lectura
  ** podemos suponer que ninguno de los archivos es vacio
  ** lectura de los primeros registros
  leer PERSONAL, REG-PERSONAL
  leer ACADEMICO, REG-ACADEMICO
  leer PROFESIONAL, REG-PROFESIONAL
  ** recorrido hasta que termine uno cualquiera de los archivos
  mientras no (ff(PERSONAL) o ff(ACADEMICO) o ff(PROFESIONAL)) hacer
    si DNI1 < DNI2 o DNI1 < DNI3
      entonces
        leer PERSONAL, REG-PERSONAL
      sino
        si DNI2 < DNI1 o DNI2 < DNI3
          entonces
            leer ACADEMICO, REG-ACADEMICO
          sino
            si DNI3 < DNI1 o DNI3 < DNI2
              entonces
                leer PROFESIONAL, REG-PROFESIONAL
              sino
                ** sincronismo: DNI1 = DNI2 = DNI3
                TRATAR-REGISTROS
                leer PERSONAL, REG-PERSONAL
                leer ACADEMICO, REG-ACADEMICO
                leer PROFESIONAL, REG-PROFESIONAL
            finsi
        finsi
    finsi
  finmientras
  ** tratamiento de los últimos registros leidos
  si DNI1 = DNI2 y DNI1 = DNI3
    entonces TRATAR-REGISTROS
  finsi
  cerrar PERSONAL
  cerrar ACADEMICO
  cerrar PROFESIONAL
Finprograma
**
Subprograma ESCRIBIR-CABECERAS
Algoritmo:
  escribir " Condiciones demandadas: "
  escribir "          Edad: Superior a 21 años."
  escribir "          Nivel de estudios: Superiores."
  escribir "          Experiencia laboral: No."
  escribir "          Situación laboral: En paro."
  escribir " Solicitantes que cumplen las condiciones demandadas: "
  escribir " D.N.I.          Nombre"
  escribir " ------          ------"
Finsubprograma
**
Subprograma TRATAR-REGISTROS
Algoritmo:
  si EDAD > 21 y NIVEL = "D" y
     (SITUACION = "C" o SITUACION = "D") y no EXPERIENCIA
```

```
    entonces
      escribir DNI1, "  ", NOMBRE
  fins
Finsubprograma
```

EJERCICIOS RESUELTOS

1. *Se dispone del archivo indexado ALUMNOS que almacena información sobre los alumnos matriculados en un centro docente. Sus registros tienen los campos.*

```
NUMERO DE MATRICULA de 10 caracteres es el campo clave
NOMBRE de 30 caracteres
ESPECIALIDAD de 11 caracteres
  (DELINEACION, ELECTRONICA, INFORMATICA, QUIMICA)
CURSO de un digito (1 a 5)
```

Se desea un programa que permita actualizarlo de forma interactiva registrando los movimientos que se proporcionen por teclado, consistentes en añadir nuevos registros, suprimir otros y modificar los campos especialidad y/o curso.

- **Codificación COBOL:**

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID. ACTUALIZACION-ALUMNOS.
*
 ENVIRONMENT DIVISION.
 INPUT-OUTPUT SECTION.
 FILE-CONTROL.
     SELECT OPTIONAL ALUMNOS
       ASSIGN DISK
       ORGANIZATION INDEXED
       ACCESS RANDOM
       RECORD KEY NUM-MAT.
*
 DATA DIVISION.
 FILE SECTION.
 FD  ALUMNOS.
 01  REG-ALUMNO.
     05  NUM-MAT       PIC X(10).
     05  NOMBRE        PIC X(30).
     05  ESPECIALIDAD PIC X(11).
     05  CURSO         PIC 9.
 WORKING-STORAGE SECTION.
 01  VARIABLES.
     05  OPERACION     PIC X.
     05  ESPERA        PIC X.
     05  FIN           PIC X.
*
 PROCEDURE DIVISION.
 PROCESO.
     OPEN I-O ALUMNOS
     MOVE "N" TO FIN
     PERFORM UNTIL FIN = "S" OR = "s"
       PERFORM PRESENTAR-MENU
       PERFORM SELECCIONAR-OPERACION
```

```
           PERFORM REALIZAR-OPERACION
         END-PERFORM
         CLOSE ALUMNOS
         STOP RUN.
*
 PRESENTAR-MENU.
         DISPLAY " " ERASE
         DISPLAY " Actualización del archivo de alumnos"
         DISPLAY " ------------------------------------"
         DISPLAY "        1. Alta."
         DISPLAY "        2. Baja."
         DISPLAY "        3. Modificación."
         DISPLAY "        4. Terminar."
         DISPLAY " Escriba el número de la operación a realizar: "
           NO ADVANCING.
*
 SELECCIONAR-OPERACION.
         ACCEPT OPERACION
         PERFORM UNTIL OPERACION > "0" AND < "5"
           DISPLAY " Operación inexistente, elija otra: " NO ADVANCING
           ACCEPT OPERACION
         END-PERFORM.
*
 REALIZAR-OPERACION.
         EVALUATE OPERACION
           WHEN "1" PERFORM DAR-ALTA
           WHEN "2" PERFORM DAR-BAJA
           WHEN "3" PERFORM MODIFICAR
           WHEN "4" PERFORM CONFIRMAR-FIN
         END-EVALUATE.
*
 DAR-ALTA.
         DISPLAY " " ERASE
         DISPLAY " Alta de alumno"
         DISPLAY " -------------"
         DISPLAY " Escriba el número de matricula: " NO ADVANCING
         ACCEPT NUM-MAT
         READ ALUMNOS
           INVALID KEY PERFORM COMPLETAR-ALTA
           NOT INVALID PERFORM ERROR-ALTA
         END-READ.
*
 COMPLETAR-ALTA.
         DISPLAY " Nombre: " NO ADVANCING
         ACCEPT NOMBRE
         DISPLAY " Especialidad (ELECTRONICA, etc.): " NO ADVANCING
         ACCEPT ESPECIALIDAD
         DISPLAY " Curso (1 a 5): " NO ADVANCING
         ACCEPT CURSO
         WRITE REG-ALUMNO.
*
 ERROR-ALTA.
         DISPLAY
           " Error: ya existe un alumno con ese número de matricula."
         DISPLAY " Pulse una tecla para continuar: " NO ADVANCING
         ACCEPT ESPERA.
*
 DAR-BAJA.
         DISPLAY " " ERASE
         DISPLAY " Baja de alumno"
```

```
           DISPLAY " -------------"
           DISPLAY " Escriba el número de matricula: " NO ADVANCING
           ACCEPT NUM-MAT
           DELETE ALUMNOS
             INVALID KEY PERFORM ERROR-BAJA
           END-DELETE.
      *
       ERROR-BAJA.
           DISPLAY
             " Error: no existe alumno con ese número de matricula."
           DISPLAY " Pulse una tecla para continuar: " NO ADVANCING
           ACCEPT ESPERA.
      *
       MODIFICAR.
           DISPLAY " " ERASE
           DISPLAY " Modificación de alumno"
           DISPLAY " ---------------------"
           DISPLAY " Escriba el número de matricula: " NO ADVANCING
           ACCEPT NUM-MAT
           READ ALUMNOS
             INVALID KEY PERFORM ERROR-MODIFICACION
             NOT INVALID PERFORM COMPLETAR-MODIFICACION
           END-READ.
      *
       COMPETAR-MODIFICACION.
           DISPLAY " Nombre: ", NOMBRE
           DISPLAY " Nueva especialidad (<RET> para no modificar): "
             NO ADVANCING
           ACCEPT ESPECIALIDAD UPDATE
           DISPLAY " Nuevo curso (<RET> para no modificar): "
             NO ADVANCING
           ACCEPT CURSO UPDATE
           REWRITE REG-ALUMNO.
      *
       ERROR-MODIFICACION.
           DISPLAY
             " Error: No existe alumno con ese número de matricula."
           DISPLAY " Pulse una tecla para continuar: " NO ADVANCING
           ACCEPT ESPERA.
      *
       CONFIRMAR-FIN.
           DISPLAY "¿Está seguro de que quiere terminar (S/N)? "
             NO ADVANCING
           ACCEPT FIN.
```

2. *Se dispone de un archivo secuencial denominado ARTICULOS que contiene los datos de los artículos existentes en un determinado almacén. Sus registros tienen los siguientes campos:*

```
CODIGO DE ARTICULO de tipo numérico entero
DESCRIPCION de 52 caracteres
PRECIO UNITARIO de tipo numérico real
EXISTENCIAS de tipo numérico entero
```

Dicho archivo se desea actualizar con los movimientos habidos a lo largo de un cierto periodo de tiempo, los cuales se han registrado en los tres siguientes archivos secuenciales:

ALTAS.DAT, con la misma estructura que el anterior, contiene nuevos artículos que hay que añadir al primero.

BAJAS.DAT. cuyos registros contienen únicamente el campo CODIGO DE ARTICULO y corresponden a artículos que ya no se tienen en el almacén y cuyos registros han de ser eliminados del primer archivo.

MODIF.DAT. con los campos CODIGO DE ARTICULO y EXISTENCIAS. corresponde a cambios en la cantidad almacenada de algunos artículos. lo cual hay que reflejar en el primer archivo.

Los cuatro archivos están clasificados ascendentemente por su primer campo. y los valores de este campo no se repiten en ningún archivo ni en los tres de movimientos entre si.

El programa que realiza la actualización creará los archivos:

ARTINUEVO.DAT. archivo actualizado.

ERRORES.DAT con los registros correspondientes a las tres situaciones anómalas posibles· alta de un registro ya existente y baja o modificación de un registro no existente.

- **Codificación en Pascal:**

```
program ACTUALIZ (ARTICULOS, ALTAS, BAJAS, MODIF, ARTINUEVO, ERRORES);
(**)
const
  CMAX = maxint;
(**)
type
  CADENA_52 = packed array [1..52] of char;
  TIPO_ERROR = (ALTA, BAJA, MODI);
  REG_ARTI  = record
                COD_ART: integer;
                DESCRIP: CADENA_52;
                PRECIO : real;
                EXISTEN: integer
              end;
  REG_BAJAS = record
                COD_ART : integer
              end;
  REG_MODIF = record
                COD_ART: integer;
                EXISTEN: integer
              end;
  REG_ERROR = record
                COD_ART : integer;
                case ERROR : TIPO_ERROR of
                  ALTA : (DESCRIP: CADENA_52;
                          PRECIO : real;
                          EXISTEN: integer);
                  BAJA : ();
                  MODI : (EXISTEN: integer)
              end;
  FICH_ARTI  = file of REG_ARTI;
  FICH_BAJAS = file of REG_BAJAS;
  FICH_MODIF = file of REG_MODIF;
  FICH_ERROR = file of REG_ERROR;
(**)
var
  ARTICULOS, ALTAS, ARTINUEVO: FICH_ARTI;
  BAJAS : FICH_BAJAS;
  MODIF : FICH_MODIF;
  ERRORES : FICH_ERROR;
  R_ARTI, R_ALTA, R_NUEVO : REG_ARTI;
  R_BAJA : REG_BAJAS;
```

```
  R_MODI : REG_MODIF;
  R_ERROR : REG_ERROR;
(**)
procedure ABRIR_ARCHIVOS
  begin (* ABRIR_ARCHIVOS *)
    assign (ARTICULOS, 'ARTICULOS.DAT');
    reset (ARTICULOS);
    assign (ALTAS, 'ALTAS.DAT');
    reset (ALTAS);
    assign (BAJAS, 'BAJAS.DAT');
    reset (BAJAS);
    assign (MODIF, 'MODIF.DAT');
    reset (MODIF);
    assign (ARTINUEVO, 'ARTINUEVO.DAT');
    rewrite (ARTINUEVO);
    assign (ERRORES, 'ERRORES.DAT');
    rewrite (ERRORES)
  end;  (* ABRIR_ARCHIVOS *);
(**)
procedure LEER_ARTI;
  begin (* LEER_ARTI *)
    if eof (ARTICULOS)
      then R_ARTI.COD_ART := CMAX
      else read (ARTICULOS, R_APTI)
  end;  (* LEER_ARTI *)
(**)
procedure LEER_ALTA;
  begin (* LEER_ALTA *)
    if eof (ALTAS)
      then R_ALTA.COD_ART := CMAX
      else read (ALTAS, R_ALTA)
  end;  (* LEER_ALTA *)
(**)
procedure LEER_BAJA;
  begin (* LEER_BAJA *)
    if eof (BAJAS)
      then R_BAJA.COD_ART := CMAX
      else read (BAJAS, R_BAJA)
  end;  (* LEER_BAJA *)
(**)
procedure LEER_MODI;
  begin (* LEER_MODI *)
    if eof (MODIF)
      then R_MODI.COD_ART := CMAX
      else read (MODIF, R_MODI)
  end;  (* LEER_MODI *)
(**)
procedure LEER_PRIMEROS;
  begin (* LEER_PRIMEROS *)
    LEER_ARTI;
    LEER_ALTA;
    LEER_BAJA;
    LEER_MODI
  end;  (* LEER_PRIMEROS *)
(**)
function FIN_ARCHIVOS : boolean;
  begin (* FIN_ARCHIVOS *)
    FIN_ARCHIVOS := (R_ARTI.COD_ART = CMAX) and
                    (R_ALTA.COD_ART = CMAX) and
                    (R_BAJA.COD_ART = CMAX) and
```

```
                    (R_MODI.COD_ART = CMAX)
  end;  (* FIN_ARCHIVOS *)
(**)
function ALTA_MENOR : boolean;
  begin (* ALTA_MENOR *)
    ALTA_MENOR := (R_ALTA.COD_ART < R_BAJA.COD_ART) and
                  (R_ALTA.COD_ART < R_MODI.COD_ART)
  end;  (* ALTA_MENOR *)
(**)
function BAJA_MENOR : boolean;
  begin (* BAJA_MENOR *)
    BAJA_MENOR := (R_BAJA.COD_ART < R_ALTA.COD_ART) and
                  (R_BAJA.COD_ART < R_MODI.COD_ART)
  end;  (* BAJA-MENOR *)
(**)
procedure COPIAR_ARTI;
  begin (* COPIAR_ARTI *)
    R_NUEVO := R_ARTI;
    write (ARTINUEVO, R_NUEVO)
  end;  (* COPIAR_ARTI *)
(**)
procedure VER_ALTA;
(**)
  procedure DAR_ALTA;
    begin (* DAR_ALTA *)
      R_NUEVO := R_ALTA;
      write (ARTINUEVO, R_ALTA)
    end;  (* DAR_ALTA *)
(**)
  procedure ERROR_ALTA;
    begin (* ERROR_ALTA *)
      R_ERROR.COD_ART := R_ALTA.COD_ART;
      R_ERROR.ERROR   := ALTA;
      R_ERROR.DESCRIP := R_ALTA.DESCRIP;
      R_ERROR.PRECIO  := R_ALTA.PRECIO;
      R_ERROR.EXISTEN := R_ALTA.EXISTEN;
      write (ERRORES, R_ERROR)
    end;  (* ERROR_ALTA *)
  (**)
  begin (* VER_ALTA *)
    if R_ALTA.COD_ART < R_ARTI.COD_ART
      then begin
             DAR_ALTA;
             LEER_ALTA
           end
      else if R_ALTA.COD_ART = R_ARTI.COD_ART
             then begin
                    ERROR_ALTA;
                    LEER_ALTA
                  end
             else begin
                    COPIAR_ARTI;
                    LEER ARTI
                  end
  end;  (* VER_ALTA *)
(**)
procedure VER_BAJA;
  (**)
  procedure ERROR_BAJA;
    begin (* ERROR_BAJA *)
```

```
      R_ERROR.COD_ART := R_BAJA.COD_ART;
      R-ERROR.ERROR := BAJA;
      write (ERRORES, R_ERROR)
    end;  (* ERROR_BAJA *)
  (**)
  begin (* VER_BAJA *)
    if R_BAJA.COD_ART < R_ARTI.COD_ART
      then begin
             ERROR_BAJA;
             LEER_BAJA
           end
      else if R_BAJA.COD_ART = R_ARTI.COD_ART
             then begin
                  (* DAR BAJA *)
                    LEER_BAJA;
                    LEER_ARTI
                  end
             else begin
                    COPIAR_ARTI;
                    LEER_ARTI
                  end
  end;  (* VER_BAJA *)
(**)
procedure VER_MODI;
(**)
  procedure MODIFICAR;
    begin (* MODIFICAR *)
      R_NUEVO := R_ARTI;
      R_NUEVO.EXISTEN := R_MODI.EXISTEN;
      write (ARTINUEVO, R_NUEVO)
    end;  (* MODIFICAR *)
  (**)
  procedure ERROR_MODI;
    begin (* ERROR_MODI *)
      R_ERROR.COD_ART := R_MODI.COD_ART;
      R_ERROR.ERROR := MODI;
      R_ERROR.EXISTEN_M := R_MODI.EXISTEN;
      write (ERRORES, R_ERROR)
    end;  (* ERROR_MODI *)
  (**)
  begin (* VER_MODI *)
     if R_MODI.COD_ART < R_ARTI.COD_ART
       then begin
              ERROR_MODI;
              LEER_MODI
            end
       else if R_MODI.COD_ART = R_ARTI.COD_ART
              then begin
                     MODIFICAR;
                     LEER_MODI;
                     LEER_ARTI
                   end
              else begin
                     COPIAR_ARTI;
                     LEER_ARTI
                   end
  end;  (* VER_MODI *)
(**)
procedure CERRAR_ARCHIVOS;
  begin (* CERRAR_ARCHIVOS *)
```

```
    close (ARTICULOS);
    close (ALTAS);
    close (BAJAS);
    close (MODIF);
    close (ARTINUEVO);
    close (ERRORES)
  end;  (* CERRAR_ARCHIVOS *)
(**)
begin (* ACTUALIZ *)
  ABRIR_ARCHIVOS;
  LEER_PRIMEROS;
  write not FIN_ARCHIVOS do
    if ALTA_MENOR
      then VER_ALTA
      else if BAJA_MENOR
             then VER_BAJA
             else VER_MODI; (* MODI MENOR *)
  CERRAR_ARCHIVOS
end.  (* ACTUALIZ *)
```

EJERCICIOS PROPUESTOS

1. Programa que permite consultas a un archivo directo PERSONAS, de 1000 registros, por el contenido de cualquiera de sus campos o subcampos. Los campos son:

```
NOMBRE alfanumérico
DNI alfanumérico
DOMICILIO, con los subcampos
  CALLE alfanumérico
  NUMERO numérico entero
  POBLACION alfanumérico
  PROVINCIA alfanumérico
  TELEFONO alfanumérico
```

   La búsqueda se realizará secuencialmente, y en caso de existir repeticiones del valor buscado, se obtendrá el primer registro que lo contenga.

2. Se dispone de dos archivos secuenciales, F1 y F2, ambos con los campos:

```
CLAVE alfanumérico
RESTO DE CAMPOS
```

   F1 y F2 están ordenados ascendentemente por el campo CLAVE, que es único en cada archivo, existiendo registros comunes a uno y otro. Programa que obtiene el archivo F, intersección de F1 y F2, conteniendo los registros comunes una sola vez.

3. Programa que, dados F1 y F2 del ejercicio anterior, obtiene F3, ordenado por el campo CLAVE, con los registros de ambos que no son comunes.

4. Se tiene el archivo secuencial ARTICULOS, cuyos registros contienen los campos: CODIGO DE ARTICULO, PRECIO y EXISTENCIAS, ordenado ascendentemente por el primer campo.

   Se tiene también el archivo secuencial MOVIMIENTOS, con los campos: TIPO,

CODIGO DE ARTICULO, PRECIO y EXISTENCIAS, ordenado ascendentemente por el mismo campo.

El campo TIPO indica la clase de movimiento: «A» para altas, «M» para modificaciones y «B» para bajas (en este último caso sólo se utilizará el valor del campo CODIGO DE ARTICULO).

Programa que genera el archivo NARTICULOS como actualización del archivo ARTICULOS con el de MOVIMIENTOS.

CAPITULO 11

Estructuras de datos dinámicas

11.1. INTRODUCCION

Las estructuras de datos internas son aquellas que residen en la memoria central de la computadora. De entre ellas, las que hemos usado hasta ahora, como las tablas, pertenecen a la clase denominada estructuras de datos estáticas, las cuales se caracterizan porque su tamaño y estructura se fijan en tiempo de compilación y permanecen inalterables durante todo el tiempo que dure la ejecución del programa o subprograma en que han sido declaradas

Para un determinado tipo de problemas en los que el número de datos a tratar varía durante su proceso, las estructuras estáticas no son adecuadas porque limitan el número máximo de datos a almacenar. Por otra parte, si este número de datos decrece, se sigue ocupando el total de la memoria reservada, aunque no se utilice.

Algunos lenguajes de programación, como el Pascal, proporcionan una nueva clase de estructuras de datos para resolver esta cuestión. Estas, denominadas **estructuras de datos dinámicas**, se caracterizan porque **su tamaño y estructura pueden ser modificados en tiempo de ejecución**, y la única limitación respecto a su tamaño máximo viene impuesta por el tamaño de la memoria interna de la computadora.

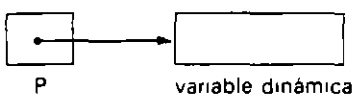
Mediante el uso de un tipo de variables denominadas **punteros** se pueden crear variables dinámicas en cualquier momento de la ejecución de un programa y eliminarlas cuando ya no sean necesarias, liberando el espacio de memoria que ocupaban. Esto permite un mejor aprovechamiento de la memoria interna, que es un recurso escaso y limitado.

11.2. PUNTEROS Y VARIABLES DINAMICAS

A diferencia de las variables estáticas, las variables dinámicas no tienen nombre. Para crearlas, acceder a ellas y eliminarlas se utiliza otra variable (estática) denominada puntero.

- **Puntero:** Es una variable utilizada para almacenar la dirección de memoria o posición de una variable (dinámica) de un tipo determinado.
- **Variable dinámica:** Es una variable simple o estructura de datos sin nombre creada en tiempo de ejecución.

Para representar gráficamente un puntero, que almacena una dirección de memoria, se dibujará una flecha que sale del puntero y llega a la variable dinámica apuntada.



La declaración de punteros y sus operaciones básicas se hace de la siguiente manera:

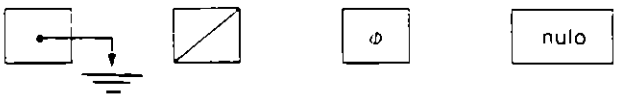
Declaración: `NOMBRE-PUNTERO es puntero a TIPO-ELEMENTO`

Esta declaración incluida en el entorno de un programa o subprograma crea una variable de tipo puntero mediante la cual podremos crear variables dinámicas y acceder a ellas en el algoritmo.

Inicialización: `NOMBRE-PUNTERO ← nulo`

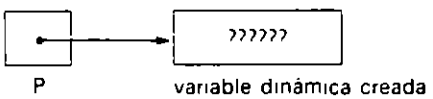
Se asigna la constante «nulo» a un puntero de cualquier tipo para indicar que no apunta a ninguna variable. Obsérvese que es diferente el hecho de que su valor sea nulo del hecho de que su valor sea indeterminado, lo que ocurre con toda variable declarada de cualquier tipo a la que no se haya asignado ningún valor.

Para representarlo gráficamente se utiliza alguna de las siguientes figuras:



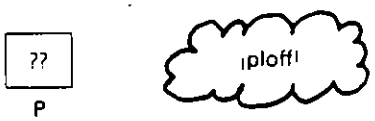
Creación de una variable dinámica: `RESERVAR (NOMBRE-PUNTERO)`

Mediante este procedimiento se crea una variable dinámica que está apuntada por el puntero.



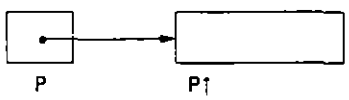
Eliminación de una variable dinámica: `LIBERAR (NOMBRE-PUNTERO)`

Este procedimiento hace que la variable apuntada deje de existir, liberando el espacio de memoria que ocupaba.



Acceso a una variable dinámica: `NOMBRE-PUNTERO↑`

Con el nombre del puntero, seguido de una flecha, se referencia o accede a la variable dinámica apuntada que, como ya se dijo, no tiene nombre. La variable así referenciada puede intervenir en toda operación o expresión de las permitidas para una variable estática de su mismo tipo base (asignación, lectura, escritura, comparación, etc.).



Además de las vistas, con los punteros sólo se pueden realizar otras dos operaciones: comparar dos punteros mediante el operador de igualdad para ver si apuntan a lo mismo y asignar el valor de un puntero a otro.

La expresión lógica P = Q vale CIERTO si ambos punteros apuntan a la misma variable dinámica o a nulo y FALSO en cualquier otro caso.

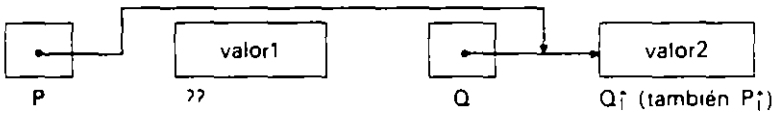
Es importante distinguir bien la diferencia entre las dos siguientes asignaciones, en las que intervienen dos punteros del mismo tipo base:

$$P \leftarrow Q$$
$$P\uparrow \leftarrow Q\uparrow$$

En la primera, la dirección almacenada en Q se copia en P, es decir, se hace que el puntero P apunte a la misma variable dinámica apuntada por Q, o a nulo si fuese el caso. Sin embargo, en la segunda asignación los punteros no cambian de valor (siguen apuntando a la misma variable dinámica que apuntaba cada uno) y el efecto producido consiste en copiar el valor almacenado en la variable apuntada por el puntero Q en la variable apuntada por el puntero P.

Antes:

Después de $P \leftarrow Q$:



Antes:

Después de $P\uparrow \leftarrow Q\uparrow$:

- **Codificación Pascal:**

A continuación se describe cómo están implementadas en este lenguaje las operaciones antes descritas.

Declaración: `P: ^TIPO_ELEMENTO`
Inicialización: `P:= nil`
Creación de una variable dinámica: `new(P)`
Eliminación de una variable dinámica: `dispose(P)`
Acceso a una variable dinámica: `P^`
Asignaciones: `P := Q; P^ := Q^`

11.3. LISTAS

Una **lista** es una estructura de datos interna consistente en una secuencia lógica de elementos del mismo tipo.

Se denomina **nodo** a cada uno de los elementos de la lista. Pueden ser datos simples o estructuras de datos, principalmente registros. Si se almacenan siguiendo algún criterio de ordenación, denominaremos **clave de ordenación** al campo utilizado para ello.

Ejemplo: *Lista de los huéspedes de un hotel ordenada alfabéticamente.*

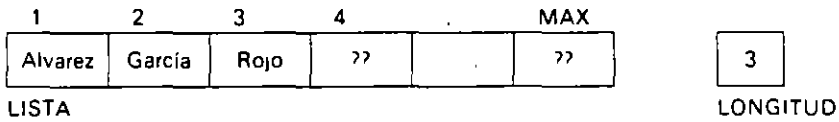
El acceso a los nodos de la lista, excepto al primero, se realiza siempre a partir del nodo anterior, y las operaciones más frecuentes son **consultar, añadir** y **suprimir** nodos, además de recorrerla secuencialmente para realizar un mismo tratamiento sobre todos ellos.

Como estructura abstracta, la lista es una estructura de datos dinámica, pues el número de nodos que la forman puede aumentar y disminuir durante su proceso. No obstante, para su implementación se puede utilizar una estructura de datos estática como la tabla, lo que impondrá algunas limitaciones en su manejo en cuanto al número máximo de nodos que puede contener, además de que la ocupación en memoria será constante.

11.3.1. LISTAS DENSAS

Se denomina **lista densa** o **contigua** a aquella en que sus nodos ocupan posiciones consecutivas de la memoria.

La forma más natural de realizar este tipo de listas es por medio de la estructura de datos tabla, para la que habrá que fijar una dimensión suficientemente grande para almacenar la máxima cantidad de datos previsibles y, puesto que normalmente se tendrán menos datos que el máximo fijado, se le asociará una variable numérica que contenga en cada momento el número de nodos, siendo accesible únicamente la subtabla formada por las componentes desde la primera hasta la longitud registrada.



- **Pseudocódigo:**

A continuación figura la declaración de una lista y las operaciones más habituales que se realizan sobre ella.

```
** Declaración

    LISTA es tabla(MAX) de TIPO-ELEMENTO
    LONGITUD es numérica entera
    P, I son numéricas enteras
    ELEMENTO es variable de TIPO-ELEMENTO

** Crear una lista vacía:

    LONGITUD ← 0
```

```
** Comprobación de lista vacia

     si LONGITUD = 0 entonces ...

** Comprobación de lista completa

     si LONGITUD = MAX entonces ...

** Consultar la posición de un elemento (búsqueda).
** Se supone la lista no vacia.

     P ← 1
     mientras LISTA(P) <> ELEMENTO y P < LONGITUD hacer
       P ← P + 1
     finmientras
     si LISTA(P) = ELEMENTO
       entonces "encontrado en la posición P"
       sino     "no pertenece a la lista"
     finsi

** Insertar ELEMENTO en la posición P
** El valor de P está comprendido entre 1 y LONGITUD
** Se supone la lista no completa

     LONGITUD ← LONGITUD + 1
     para I de LONGITUD a P + 1 con incremento -1 hacer
       LISTA(I) ← LISTA(I - 1)
     finpara
     LISTA(P) ← ELEMENTO

** Suprimir nodo en la posición P
** El valor de P está comprendido entre 1 y LONGITUD
** Se supone la lista no vacia

     LONGITUD ← LONGITUD - 1
     para I de P a LONGITUD hacer
       LISTA(I) ← LISTA(I + 1)
     finpara

** Tratar todos los nodos de la lista (recorrido secuencial)

     para I de 1 a LONGITUD hacer
       "tratar LISTA(I)"
     finpara
```

- **Codificación Pascal:**

Se codificarán mediante subprogramas (procedimiento o función) cada una de las operaciones necesarias.

```
(* Declaraciones *)
     TYPE
       TIPO_LONGITUD = 0..MAX;
       TIPO_POSICION = 1..MAX;
       TIPO_LISTA    =    RECORD
                            NODO: ARRAY [TIPO_POSICION] OF TIPO_ELEMENTO;
                            LONGITUD: TIPO_LONGITUD
                          END;
     VAR
       LISTA:       TIPO_LISTA;
```

```
      POSICION:    TIPO_POSICION;
      ELEMENTO:    TIPO_ELEMENTO;
      ENCONTRADO: BOOLEAN;

(* Creación de una lista vacia *)
    PROCEDURE CREAR_LISTA (VAR LISTA: TIPO_LISTA);
    BEGIN
      WITH LISTA DO LONGITUD := 0
    END;

(* Comprobación de lista vacia *)
    FUNCTION LISTA_VACIA (LISTA: TIPO_LISTA): BOOLEAN;
    BEGIN
      WITH LISTA DO LISTA_VACIA := LONGITUD = 0
    END;

(* Comprobación de lista completa *)
    FUNCTION LISTA_COMPLETA (LISTA: TIPO_LISTA): BOOLEAN;
    BEGIN
      WITH LISTA DO LISTA_COMPLETA := LONGITUD = MAX
    END;

(* Consultar la posición de un elemento *)
(* Se supone la lista no vacia *)
    PROCEDURE CONSULTAR_ELEMENTO (LISTA:          TIPO_LISTA;
                                  ELEMENTO:       TIPO_ELEMENTO;
                                  AR POSICION:    TIPO_POSICION;
                                  AR ENCONTRADO: BOOLEAN);
    BEGIN
      WITH LISTA DO BEGIN
        POSICION := 1;
        WHILE (NODO[POSICION] <> ELEMENTO) AND (POSICION < LONGITUD)
          DO POSICION := POSICION + 1;
        ENCONTRADO := NODO[POSICION] = ELEMENTO
      END
    END;
(* El valor de POSICION sólo es significativo si ENCONTRADO = TRUE *)

(* Insertar ELEMENTO en el lugar POSICION *)
(* El valor de POSICION está comprendido entre 1 y LONGITUD + 1 *)
(* Se supone la lista no completa *)
    PROCEDURE INSERTAR_ELEMENTO (VAR LISTA: TIPO_LISTA;
                                 ELEMENTO: TIPO_ELEMENTO;
                                 POSICION: TIPO_POSICION);
      VAR I: TIPO_POSICION;
    BEGIN
      WITH LISTA DO
        BEGIN
          LONGITUD := LONGITUD + 1;
          FOR I := LONGITUD DOWNTO POSICION + 1 DO
            LISTA[I] := LISTA[I - 1];
          LISTA[POSICION] := ELEMENTO
        END
    END;

(* Suprimir el elemento del lugar POSICION *)
(* El valor de POSICION está comprendido entre 1 y LONGITUD *)
(* Se supone la lista no vacia *)
    PROCEDURE SUPRIMIR_ELEMENTO (VAR LISTA: TIPO_LISTA;
                                 POSICION: TIPO_POSICION);
```

```
    VAR I: TIPO_POSICION;
  BEGIN
    WITH LISTA DO
      BEGIN
        LONGITUD := LONGITUD - 1;
        FOR I := POSICION TO LONGITUD DO LISTA[I] := LISTA[I + 1]
      END
  END;

(* Tratar todos los elementos de la lista *)
  PROCEDURE TRATAR_LISTA (VAR LISTA: TIPO_LISTA);
    VAR I: TIPO_POSICION;
  BEGIN
    WHIT LISTA DO
      FOR I := 1 TO LONGITUD DO
        TRATAR_NODO (NODO[I])
  END;
  (* Si la lista es vacia este procedimiento no hace nada *)
```

- **Codificación COBOL:**

Las operaciones se codificarán mediante procedimientos. Las declaraciones necesarias figurarán en la WORKING-STORAGE SECTION.

```
* ** Declaraciones
 01  TABLA_LISTA.
     05  LISTA      PIC tipo-elemento
           OCCURS long-max INDEXED BY I.
     05  LONGITUD   PIC 9(5).
 01  VARIABLES.
     05  POSICION   PIC 9(5).
     05  AUX        PIC 9(5).
     05  ELEMENTO   PIC tipo-elemento.
     05  ENCONTRADO PIC XX.

* **
 CREAR-LISTA.
     MOVE 0 TO LONGITUD.

* **
 CONSULTAR-ELEMENTO.
     SET I TO 1
     SEARCH LISTA
       AT END
         MOVE "NO" TO ENCONTRADO
       WHEN LISTA(I) = ELEMENTO
         SET POSICION TO I
         MOVE "SI" TO ENCONTRADO
     END-SEARCH.

* **
 INSERTAR-ELEMENTO.
     ADD 1 TO LONGITUD GIVING AUX
     ADD 1 TO LONGITUD
     PERFORM VARYING I FROM LONGITUD BY -1 UNTIL I = POSICION
       MOVE LISTA(I - 1) TO LISTA(I)
     END-PERFORM
     MOVE ELEMENTO TO LISTA(POSICION).

* **
 SUPRIMIR-ELEMENTO.
```

```
        SUBTRACT 1 FROM LONGITUD
        PERFORM VARYING I FROM POSICION BY 1 UNTIL I > LONGITUD
          MOVE LISTA(I + 1) TO LISTA(I)
        END-PERFORM.

      * **
       TRATAR-LISTA.
        PERFORM VARYING I FROM 1 BY 1 UNTIL I > LONGITUD
          PERFORM TRATAR-NODO
        END-PERFORM.
```

11.3.2. LISTAS ENLAZADAS

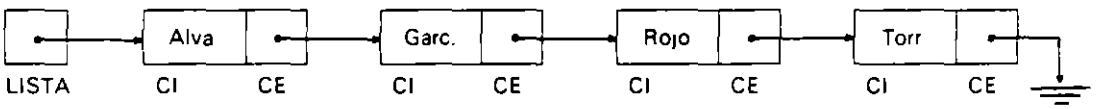
Se denomina **lista enlazada** o **encadenada** a aquella en que la secuencia lógica de sus elementos viene determinada por un campo de enlace que posee cada uno de los nodos para conectarse con el que le sigue.

Por tanto, sus nodos pueden ocupar posiciones no consecutivas de la memoria. A cada nodo se accede por medio del enlace del nodo anterior y existe un enlace especial, externo a la lista, para acceder a su primer nodo.

El acceso a una lista enlazada puede compararse a lo que hacemos cuando necesitamos realizar alguna gestión con la administración: Llegamos al primer mostrador del organismo correspondiente (el de información), de aquí nos envían al siguiente y de éste a otro (que no tienen por qué estar próximos), y así sucesivamente hasta que logramos completar la gestión.

Para la implementación de esta estructura de datos se utilizan los punteros, aunque también se podría hacer mediante tablas imponiendo las restricciones de tamaño y procesamiento ya comentadas.

En una lista enlazada, realizada mediante punteros, cada nodo tendrá un campo de información y otro de enlace al siguiente nodo, y existirá un puntero externo que apuntará al primer nodo.



LISTA: Puntero externo que permite el acceso a la lista.
CI: Campo de información.
CE: Campo de enlace.

Todos los nodos apuntan al siguiente nodo, excepto el último, que apunta a «nulo», y todos los nodos son apuntados por el anterior, excepto el primero, que es apuntado por el puntero externo.

- **Pseudocódigo:**

Se utilizan la declaración y las operaciones de punteros enunciadas en el Apartado 11.2.

```
** Declaración

   Lista es puntero a NODO
   NODO es registro compuesto de
             INFO es TIPO-ELEMENTO
             SIG  es puntero a NODO
          finregistro
   ELEMENTO es variable de TIPO-ELEMENTO
```

```
** Crear una lista vacia

     LISTA ← nulo

** Comprobacion de lista vacia

     si LISTA = nulo entonces ...

** No existe limitación en cuanto al tamaño máximo de la lista y, por tanto,
** no es necesario comprobarlo.

** Consultar ELEMENTO: Se obtendrá un puntero ACTUAL al nodo que contiene
** ese valor, si está en la lista.
** Se supone la lista no vacia

     ACTUAL ← LISTA
     mientras INFO de ACTUAL↑ <> ELEMENTO y
              SIG  de ACTUAL↑ <> nulo hacer
       ACTUAL ← SIG de ACTUAL↑
     finmientras
     si INFO de ACTUAL↑ = ELEMENTO
       entonces "encontrado"
       sino     "no pertenece a la lista"
     finsi

** Insertar ELEMENTO: Se inserta a continuación del nodo apuntado por
** ANTERIOR, si este puntero vale nulo significa que ha de insertarse al
** comienzo.

     RESERVAR (AUX) ** se crea un nuevo nodo
     INFO de AUX↑ ← ELEMENTO
     si ANTERIOR = nulo
              ** la lista está vacía o se inserta al comienzo
       entonces SIG de AUX↑ ← LISTA
                LISTA ← AUX
       sino   ** la inserción es por el medio o al final
                SIG de AUX↑ ← SIG de ANTERIOR↑
                SIG de ANTERIOR↑ ← AUX
     finsi

** Suprimir ELEMENTO: Se suprime el nodo apuntado por el puntero ACTUAL.
** El puntero ANTERIOR, necesario para esta operación, señala al nodo
** anterior o a nulo si el nodo a suprimir es el primero de la lista.
** Se supone la lista no vacía.

     si ANTERIOR = nulo
       entonces LISTA ← SIG de ACTUAL↑
       sino SIG de ANTERIOR↑ ← SIG de ACTUAL↑
     finsi
     LIBERAR (ACTUAL) ** libera el espacio del nodo suprimido

     ** Tratar la lista secuencialmente

          ACTUAL ← LISTA
          mientras ACTUAL <> nulo hacer
            "tratar INFO de ACTUALP"
            ACTUAL ← SIG de ACTUAL↑
          finmientras
```

- **Codificación Pascal:**

```
(* Definiciones y declaraciones *)
     TYPE ...
       TIPO_ENLACE = ^TIPO_NODO;
       TIPO_NODO   = RECORD
                       INFO: TIPO_ELEMENTO;
                       SIG:  TIPO_ENLACE
                     END;
       TIPO_LISTA  = TIPO_ENLACE;
     VAR ...
       LISTA:       TIPO_ELEMENTO;
       ELEMENTO:    TIPO_ELEMENTO;
       ENCONTRADO: BOOLEAN;

(* Creación de una lista vacia *)
     PROCEDURE CREAR_LISTA (VAR LISTA: TIPO_LISTA);
     BEGIN
       LISTA := nil
     END;

(* Comprobación de lista vacia *)
     FUNCTION LISTA_VACIA (LISTA: TIPO_LISTA): BOOLEAN;
     BEGIN
       LISTA_VACIA := LISTA = nil
     END;

(* Consultar ELEMENTO *)
(* Se obtendrá un puntero ACTUAL al nodo que contiene ese valor, si está en
   la lista. Si no está la variable ENCONTRADO valdrá false *)
(* Se supone la lista no vacia *)
     PROCEDURE CONSULTAR_ELEMENTO (LISTA:          TIPO-LISTA;
                                   ELEMENTO:       TIPO_ELEMENTO;
                                   VAR ACTUAL:     TIPO_ENLACE;
                                   VAR ENCONTRADO: BOOLEAN);
     BEGIN
       ACTUAL := LISTA;
       WHILE (ACTUAL^.INFO <> ELEMENTO) AND (ACTUAL^.SIG <> nil)
           DO ACTUAL := ACTUAL^.SIG;
       ENCONTRADO := ACTUAL^.INFO = ELEMENTO
     END;
(* El valor de ACTUAL sólo es significativo si ENCONTRADO = true *)

(* Insertar ELEMENTO *)
(* Se inserta a continuación del nodo apuntado por ANTERIOR *)
(* Si ANTERIOR = nil significa que ha de insertarse al comienzo *)
     PROCEDURE INSERTAR_ELEMENTO (VAR LISTA: TIPO_LISTA;
                                  ELEMENTO:  TIPO_ELEMENTO;
                                  ANTERIOR:  TIPO_ENLACE);
       VAR AUX: TIPO_ENLACE;
     BEGIN
       new(AUX);
       AUX^.INFO := ELEMENTO;
       IF ANTERIOR = nil
         THEN BEGIN (* lista vacia o inserción al comienzo *)
                AUX^.SIG := LISTA;
                LISTA := AUX
              END
         ELSE BEGIN (* inserción por el medio o al final *)
                AUX^.SIG := ANTERIOR^.SIG;
```

```
                ANTERIOR^.SIG := AUX
              END
        END;

(* Suprimir un elemento *)
(* Se suprime el nodo apuntado por ACTUAL. El puntero ANTERIOR, necesario
   para esta operación, señala al nodo anterior o a nil si el nodo a
   suprimir es el primero de la lista *)
(* Se supone la lista no vacía *)
    PROCEDURE SUPRIMIR_ELEMENTO (VAR LISTA: TIPO_LISTA;
                                 ANTERIOR, ACTUAL : TIPO_ENLACE);
    BEGIN
      IF ANTERIOR = nil
        THEN LISTA := ACTUAL^.SIG
        ELSE ANTERIOR^.SIG := ACTUAL^.SIG;
      dispose (ACTUAL) (* se libera el espacio del nodo suprimido *)
    END;

(* Tratar todos los elementos de la lista *)
    PROCEDURE TRATAR_LISTA (LISTA: TIPO_LISTA);
      VAR ACTUAL: TIPO_ENLACE;
    BEGIN
      ACTUAL := LISTA;
      WHILE ACTUAL <> nil DO
        BEGIN
          TRATAR_ELEMENTO (ACTUAL^.INFO);
          ACTUAL := ACTUAL^.SIG
        END
    END;
```

11.4. PILAS

Una **pila** es una lista en la que las inserciones, consultas y supresiones pueden hacerse únicamente por un extremo denominado **cima**.

Se las denomina listas **LIFO**, siglas de la expresión inglesa *Last In First Out* (último en entrar, primero en salir), y se encuentran múltiples ejemplos de esta estructura en la vida diaria: Una pila de platos, una fila de coches que aparcan en un callejón sin salida (para poder sacar uno cualquiera de ellos es necesario sacar previamente todos los que han aparcado después de él).

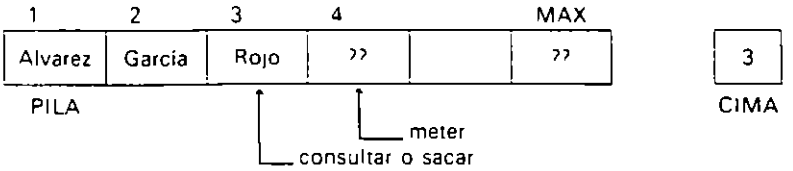
Esta estructura es muy útil y muy utilizada en programación de sistemas, por ejemplo para registrar las llamadas a subprogramas junto con sus parámetros, y, en general, siempre que se quiera recuperar una secuencia de elementos en el orden inverso al que fueron obtenidos.

Para su implementación se pueden utilizar tablas o punteros, de forma similar a como se hizo con las listas, teniendo en cuenta las características de acceso de esta estructura.

11.4.1. IMPLEMENTACION MEDIANTE TABLAS

Una pila quedará implementada mediante una tabla (PILA) cuya dimensión limita el máximo número de elementos que puede contener junto con una variable numérica (CIMA) que sirve para saber en cada momento el número de elementos almacenados y también el índice del último que se metió. Además habrá que programar las operaciones meter, consultar y sacar.

Solamente estará permitido consultar o eliminar el elemento de la componente señalada por el valor de CIMA, siempre que éste no sea 0, y añadir un nuevo elemento en la componente siguiente, siempre que quede espacio en la tabla.



- **Pseudocódigo:**

```
** Declaraciones

     PILA es tabla(MAX) de TIPO-ELEMENTO
     CIMA es numérica entera
     ELEMENTO es variable TIPO-ELEMENTO

** Crear una pila vacía

     CIMA ← 0

** Comprobación de pila vacía

     si CIMA = 0 entonces ...

** Comprobación de pila completa

     si CIMA = MAX entonces ...

** Meter un elemento
** Se supone que la pila no está completa

     CIMA ← CIMA + 1
     PILA(CIMA) ← ELEMENTO

** Consultar un elemento
** Se supone que la pila no está vacía
** El único elemento que se puede consultar es el último que entró

     ELEMENTO ← PILA(CIMA)

** Sacar un elemento
** Se supone que la pila no está vacía
** El último valor metido se copia en la variable ELEMENTO y
** después se le elimina de la pila

     ELEMENTO ← PILA(CIMA)
     CIMA ← CIMA - 1

** Tratar todos los elementos de la pila
** Después de este tratamiento la pila queda vacía

     mientras CIMA > 0 hacer
       "tratar PILA(CIMA)"
       CIMA ← CIMA - 1
     finmientras
```

- **Codificación Pascal:**

```
(* Declaraciones *)
CONST
```

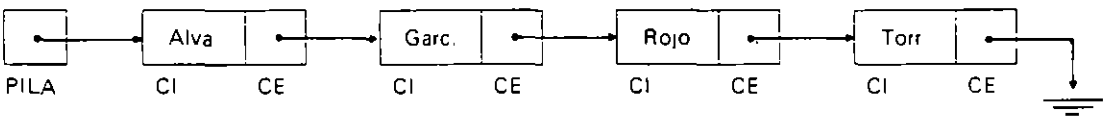
```
  MAX = máximo-número-permitido-de-elementos;
TYPE
  TIPO_CIMA = 0..MAX;
  TIPO_PILA = RECORD
                INFO: ARRAY [1..MAX] OF TIPO_ELEMENTO;
                CIMA: TIPO_CIMA
              END;
VAR
  PILA: TIPO_PILA;
  ELEMENTO: TIPO_ELEMENTO;

(* Crear una pila vacía *)
PROCEDURE CREAR_PILA (VAR PILA: TIPO_PILA);
BEGIN
  WITH PILA DO CIMA := 0
END;

(* Comprobación de pila vacía *)
FUNCTION PILA-VACIA (PILA: TIPO_PILA): BOOLEAN;
BEGIN
  WITH PILA DO PILA_VACIA := CIMA = 0
END;

(* Comprobación de pila completa *)
FUNCTION PILA_COMPLETA (PILA: TIPO_PILA): BOOLEAN;
BEGIN
  WITH PILA DO PILA_COMPLETA := CIMA = MAX
END;

(* Meter un elemento *)
(* Se supone que la pila no está completa *)
PROCEDURE METER (VAR PILA: TIPO_PILA; ELEMENTO: TIPO_ELEMENTO);
BEGIN
  WITH PILA DO BEGIN
    CIMA := CIMA + 1;
    INFO[CIMA] := ELEMENTO
  END
END;

(* Consultar un elemento *)
(* Se supone que la pila no está vacía *)
(* La pila no se modifica *)
PROCEDURE CONSULTAR (PILA: TIPO_PILA; VAR ELEMENTO: TIPO_ELEMENTO);
BEGIN
  WITH PILA DO
    ELEMENTO := INFO[CIMA]
END;

(* Sacar un elemento *)
(* Se supone que la pila no está vacía *)
PROCEDURE SACAR (VAR PILA: TIPO_PILA; VAR ELEMENTO: TIPO_ELEMENTO);
BEGIN
  WITH PILA DO BEGIN
    ELEMENTO := INFO[CIMA];
    CIMA := CIMA - 1
  END
END;

(* Tratar todos los elementos de la pila *)
(* La pila quedará vacía *)
```

```
PROCEDURE TRATAR_PILA (VAR PILA: TIPO_PILA);
BEGIN
  WITH PILA DO
    WHILE CIMA > 0 DO BEGIN
      TRATAR_ELEMENTO (INFO[CIMA]);
      CIMA := CIMA - 1
    END
END;
```

La codificación en COBOL se deja al lector como ejercicio.

11.4.2. IMPLEMENTACION MEDIANTE PUNTEROS

Se representa con una lista enlazada en la cual todas las inserciones. consultas o supresiones se realizan al comienzo de la misma por medio del puntero PILA que señala la cima de la pila.



PILA: Puntero externo que señala la cima de la pila.
CI: Campo de información.
CE: Campo de enlace.

- **Pseudocódigo:**

```
** Declaraciones

     PILA es puntero a NODO
     NODO es registro compuesto de
               INFO es TIPO-ELEMENTO
               SIG es puntero a NODO
             finregistro
       ELEMENTO es variable de TIPO-ELEMENTO

** Crear una pila vacia

     PILA ← nulo

** Comprobación de pila vacia

     si PILA = nulo entonces ...

** Meter un elemento
     RESERVAR (AUX)
     INFO de AUX↑ ← ELEMENTO
     SIG de AUX↑ ← PILA
     PILA ← AUX

** Consultar pila
** Se obtendrá el valor del elemento de la cima
** Se supone la pila no vacia

     ELEMENTO ← INFO de PILA↑

** Sacar un elemento
```

```
** Se supone la pila no vacia
** El valor de la cima suprimido queda copiado en ELEMENTO

    ELEMENTO ← INFO DE PILA↑
    AUX ← PILA
    PILA ← SIG de PILA↑
    LIBERAR (AUX)

** Tratar todos los elementos de la pila
** Después de este tratamiento la pila queda vacia

    mientras PILA <> nulo hacer
      "tratar INFO DE PILA↑"
       AUX ← PILA
       PILA ← SIG de PILA↑
       LIBERAR (AUX)
    finmientras
```

La codificación en lenguaje Pascal de estos procedimientos se deja al lector como ejercicio.

11.5. COLAS

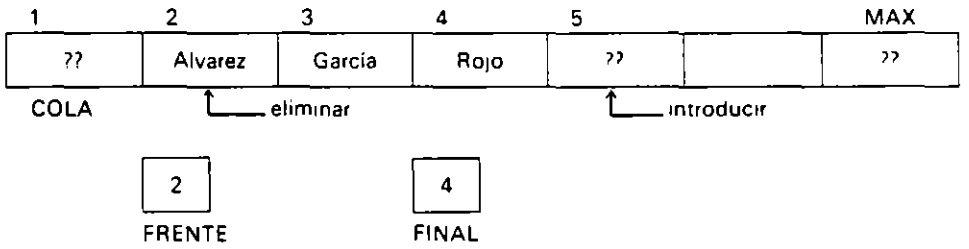
Una **cola** es una lista en la que las inserciones se realizan por uno de sus extremos, llamado FINAL, y las supresiones por el otro, llamado FRENTE.

Se denominan listas **FIFO**, siglas de *First In First Out* (primero en entrar, primero en salir), y se corresponden con los múltiples ejemplos existentes en la realidad de colas de espera: Una cola de personas a la entrada de un cine, una cola de automoviles en un túnel de lavado automático, etc.

En programación de sistemas son de gran aplicación para establecer la asignación de recursos compartidos por varios procesos o usuarios (colas de trabajos de impresión para una determinada impresora, colas de petición de acceso a un determinado archivo, etc.).

11.5.1. IMPLEMENTACION MEDIANTE TABLAS CIRCULARES

Una cola se puede representar mediante una tabla (COLA) y dos variables numéricas (FRENTE y FINAL) para indicar las componentes primera y última. Por ejemplo, si se añaden elementos de izquierda a derecha, se irá incrementando la variable FINAL, y para suprimirlos se hará por el extremo opuesto, es decir, incrementando la variable FRENTE.

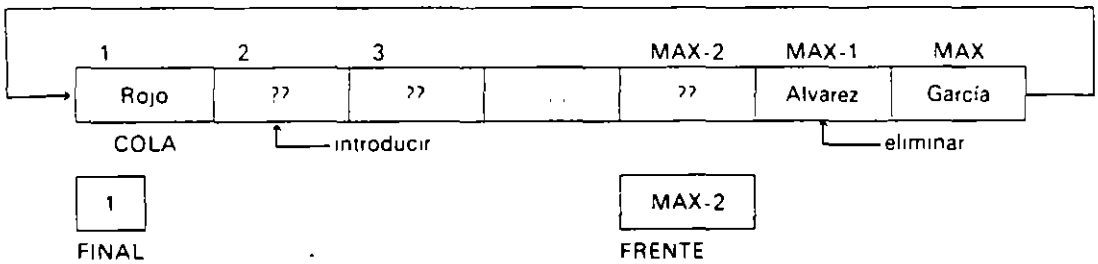


Esta implementación presenta el inconveniente de que puede ocurrir que la variable FINAL llegue al valor máximo de la tabla MAX, con lo cual no se pueden seguir añadiendo elementos a la cola, aun cuando queden posiciones libres a la izquierda de la posición FRENTE.

Una solución a ello sería mantener fijo a 1 el valor de FRENTE, haciendo un desplazamiento de una posición para todas las componentes ocupadas cada vez que se haga una supresión. Esta solución tiene el inconveniente del tiempo necesario para los desplazamientos.

La solución que consideramos mejor, en cuanto a aprovechamiento de memoria y tiempo de proceso, consiste en la utilización de una tabla circular, esto es, una tabla en la que se considera que la componente primera (1) sigue a la componente última (MAX). De esta manera, sin realizar desplazamientos de las componentes, cuando la variable FINAL ha alcanzado el valor MAX y se desean añadir nuevos elementos, se hará a continuación, es decir, en las componentes primera y sucesivas, siempre que estén libres.

Se tendrá una cola vacía siempre que FRENTE = FINAL y, para distinguir esta situación de la cola completa, consideraremos que la variable FRENTE señala siempre la posición libre anterior al primer elemento de la cola.



El incremento de las variables FRENTE y FINAL, que hay que realizar cada vez que se produzca una supresión o inserción, se hará de la siguiente manera:

```
si VARIABLE = MAX
   entonces VARIABLE ← 1
   sino     VARIABLE ← VARIABLE + 1
finsi
```

O de esta otra manera equivalente, utilizando el operador mod que calcula el resto entero por defecto:

```
VARIABLE ← VARIABLE mod MAX + 1
```

- **Pseudocódigo:**

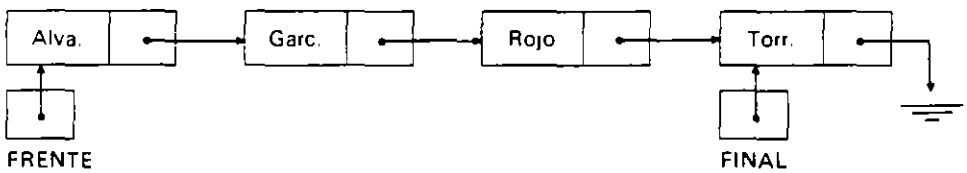
```
** Declaraciones

     COLA es tabla(MAX) de TIPO-ELEMENTO
     FRENTE, FINAL son numéricas enteras
     ELEMENTO es variable TIPO-ELEMENTO

** Crear una cola vacia

     FRENTE ← MAX
     FINAL  ← MAX

** Comprobación de cola vacia

     si FRENTE = FINAL entonces ...
```

```
** Comprobación de cola completa

     si FRENTE = FINAL mod MAX + 1 entonces ...

** Introducir un elemento
** Se supone que la cola no está completa

     FINAL ← FINAL mod MAX + 1
     COLA(FINAL) ← ELEMENTO

** Consultar un elemento
** Se supone que la cola no está vacia
** El único elemento que se puede consultar es el primero que entró

     ELEMENTO ← COLA(FRENTE mod MAX + 1)

** Eliminar un elemento
** Se supone que la cola no está vacia

     FRENTE ← FRENTE mod MAX + 1

** Tratar todos los elementos
** Después de este tratamiento la cola queda vacia

     mientras FRENTE <> FINAL hacer
       "tratar COLA(FRENTE mod MAX + 1)"
       FRENTE ← FRENTE mod MAX + 1
     finmientras
```

Las codificaciones en COBOL y Pascal se dejan al lector como ejercicio.

11.5.2. IMPLEMENTACION MEDIANTE PUNTEROS

Una cola se representará mediante una lista enlazada en la que se utilizarán dos punteros, FRENTE y FINAL, para señalar los nodos frente y final de la cola.



- **Pseudocódigo:**

```
** Declaraciones
     NODO es registro compuesto de
               INFO es TIPO-ELEMENTO
               SIG  es puntero a NODO
             finregistro
     FRENTE, FINAL son punteros a NODO
     ELEMENTO es variable de TIPO-ELEMENTO

** Crear una cola vacia

     FRENTE ← nulo
     FINAL  ← nulo
```

```
** Comprobación de cola vacia

     si FRENTE = nulo entonces ...

** Introducir un elemento

     RESERVAR (AUX)
     INFO de AUX↑ ← ELEMENTO
     SIG  de AUX↑ ← nulo
     si FINAL = nulo
       entonces FRENTE ← AUX
       sino SIG de FINAL↑ ← AUX
     finsi
     FINAL ← AUX

** Consultar un elemento
** Se supone que la cola no está vacia

     ELEMENTO ← INFO de FRENTE↑
       '
** Eliminar un elemento
** Se supone que la cola no está vacia

     AUX ← FRENTE
     FRENTE ← SIG de FRENTE↑
     si FRENTE = nulo
       entonces FINAL ← nulo
     finsi
     LIBERAR (AUX)

** Tratar todos los elementos de la cola
** Después de este tratamiento la cola queda vacia

     mientras FRENTE <> nulo hacer
        AUX ← FRENTE
        FRENTE ← SIG de FRENTE↑
       "tratar INFO de AUX↑ "
        LIBERAR (AUX)
     finmientras
```

La codificación en Pascal se deja al lector como ejercicio.

11.6. ARBOLES

El concepto más general de estructura enlazada es el de grafo, plexo o multilista, consistente en un conjunto de nodos y un conjunto de enlaces entre ellos Un caso particular de esta estructura es el árbol, una estructura dinámica que por su gran utilidad en el ámbito de la programación expondremos a continuación, aunque sin entrar en detalle debido a su gran extensión y complejidad.

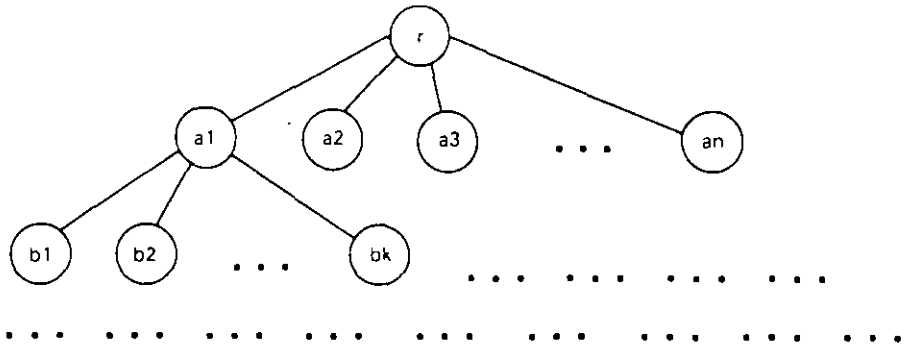
Esta estructura se usa principalmente para representar aquellos datos que mantienen una relación jerárquica entre sí, como pueden ser un árbol genealógico, una estructura jerárquica de almacenamiento de archivos, etc.

11.6.1. CONCEPTOS Y DEFINICIONES

Un **árbol A** es una estructura que consiste en un conjunto finito de elementos llamados **nodos**, tales que:

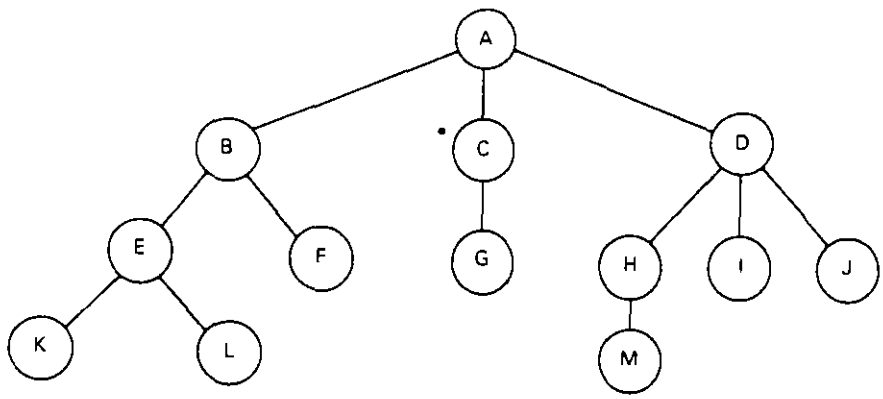
a) A es vacío, o

b) A contiene un nodo especial r, llamado **raíz** de A, y los restantes nodos de A forman n conjuntos disjuntos $A_1, A_2, \ldots, A_n$, que son asimismo árboles de raíces $a_1, a_2, \ldots, a_n$



El número de subárboles de un nodo se denomina **grado del nodo.** El **grado de un árbol** es el máximo grado de todos sus nodos. Los nodos de grado cero se denominan **hojas o nodos terminales.** Las raíces de los subárboles de un nodo x son los **hijos** de x, a x se le llama **padre**. Los hijos del mismo padre se dice que son **hermanos**. Se denominan **antecesores** de un nodo a todos los nodos que están en el camino que va desde la raíz al nodo. Una **rama** es un camino desde la raíz a una hoja. El **nivel de un nodo** se define así: el nivel de la raíz es 1; si un nodo tiene nivel n, sus hijos tienen nivel $n + 1$. La **altura** o **profundidad** de un árbol es el máximo nivel de todos sus nodos. Se denomina **bosque** al conjunto formado por dos o más árboles.

Ejemplo:



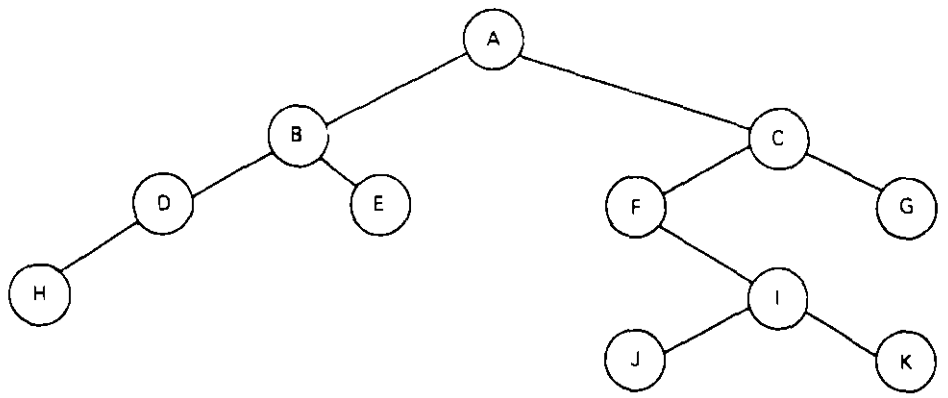
Raíz: A
Hojas: K, L, F, G, M, I, J
Antecesores de L: A, B, E
Rama de M: A, D, H, M
Hijos de D: H, I, J (hermanos)
Grado de A = 3
Grado de C = 1
Grado de M = 0 (hoja)
Grado del árbol = 3
Nivel de A = 1
Nivel de C = 2
Nivel de M = 4
Altura del árbol = 4

Los árboles *n*-arios (de grado *n*) no se suelen utilizar en la práctica debido a la complejidad de su definición y su manejo; por otra parte, es bastante fácil transformarlos en árboles binarios que mantengan la jerarquía y principales propiedades, siendo estos últimos más fáciles de tratar.

11.6.2. ARBOLES BINARIOS

Son árboles de grado 2, es decir, cada uno de sus nodos puede tener dos hijos como máximo, los cuales, si existen, se denominan **hijo izquierdo** e **hijo derecho**.

Ejemplo:

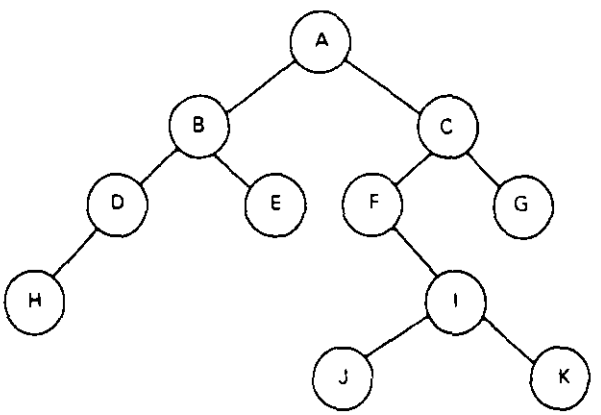


Se dice que un árbol binario es **equilibrado** si para todos sus nodos la altura de sus dos subárboles se diferencia en uno como máximo.

Un árbol binario **completo** es aquel en que todos sus nodos, excepto las hojas, tienen exactamente dos hijos.

Para la representación de árboles binarios se pueden utilizar listas enlazadas con dos punteros en cada nodo, o tablas, teniendo en cuenta que será necesario utilizar una dimensión suficiente para almacenar todos sus nodos.

	INFO	HI	HD
1	A	2	3
2	B	4	5
3	C	6	7
4	D	8	0
5	E	0	0
6	F	0	9
7	G	0	0
8	H	0	0
9	I	10	11
10	J	0	0
11	K	0	0



Representación de un árbol binario mediante una tabla.

11.6.3. REPRESENTACION MEDIANTE TABLAS

Una primera representación consiste en utilizar una tabla en la que cada componente contiene un campo de información y dos campos numéricos para indicar la posición de las componentes en que están sus hijos izquierdo y derecho, respectivamente (el valor 0 indica que no tiene hijo). Véase figura en página anterior.

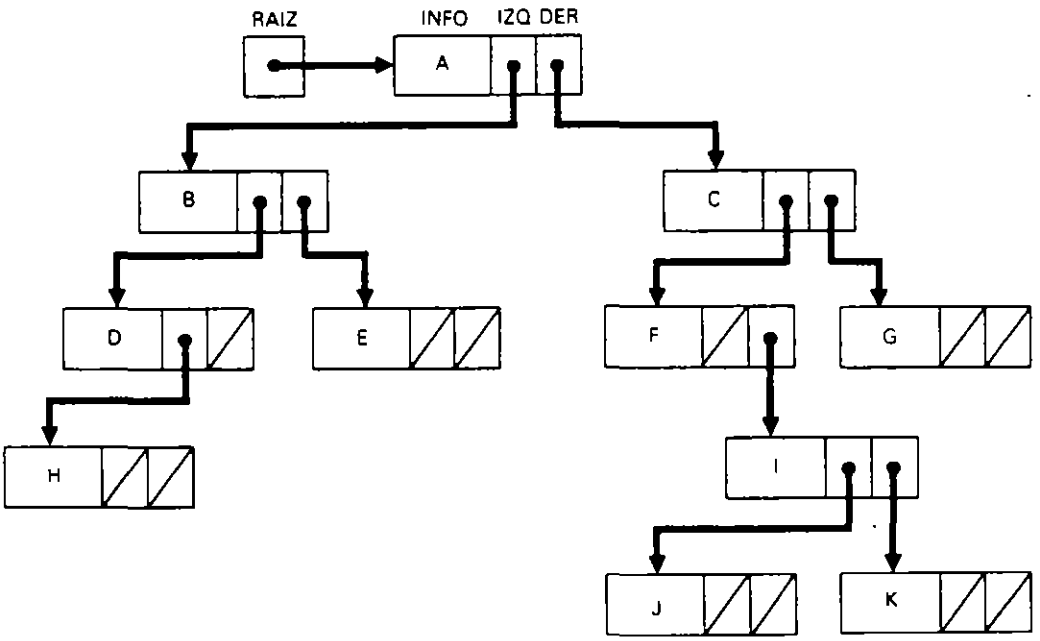
Otra posible representación consiste en utilizar una tabla en la que solamente se almacena la información. La raiz ocupará la posición 1, y para cada nodo situado en la posición i, sus hijos izquierdo y derecho ocuparán las posiciones $2*i$ y $2*i+1$, respectivamente.

Teniendo en cuenta que para un árbol binario de altura n se necesitarán $2^n - 1$ componentes (máximo número de nodos = árbol lleno), el árbol anterior se puede representar de la siguiente manera:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
A	B	C	D	E	F	G	H	-	-	-	-	I	-	-	-	-	-	-	-	-	-	-	-	-	J	K	-	-	-	-

11.6.4. REPRESENTACION MEDIANTE PUNTEROS

Cada nodo del árbol contendrá un campo de información y dos punteros para enlazarse con sus hijos izquierdo y derecho. El acceso al árbol se realiza mediante un puntero externo al nodo raiz.



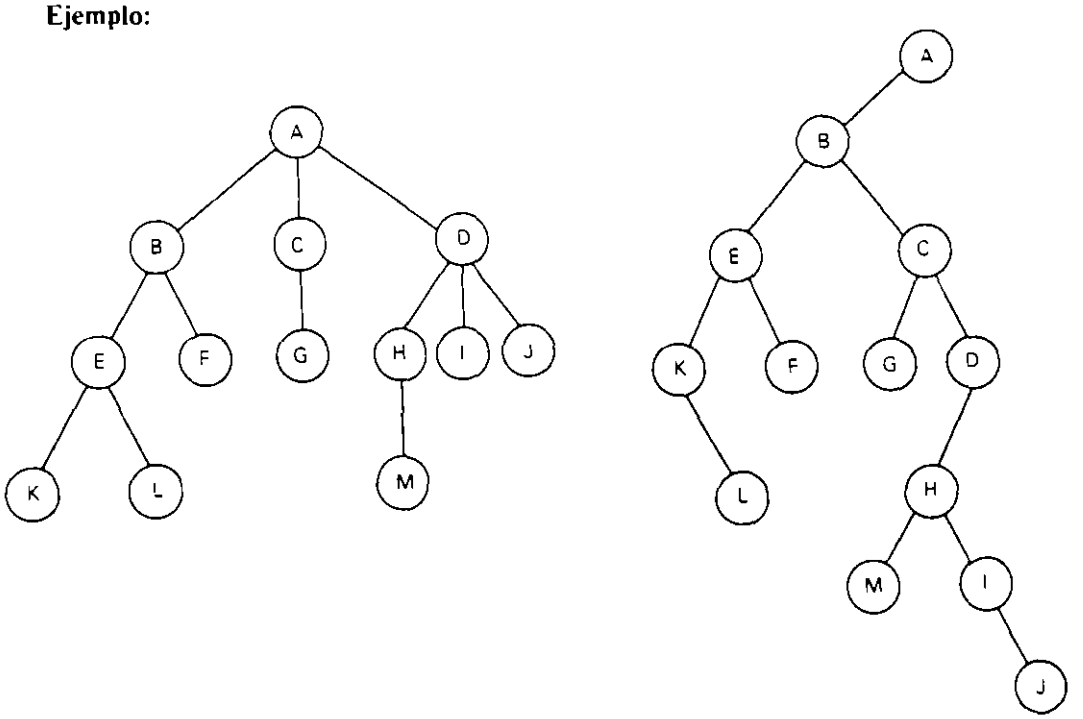
11.6.5. TRANSFORMACION DE ARBOLES N-ARIOS EN BINARIOS

Un árbol n-ario se representa mediante un árbol binario de la siguiente manera:

1. La raiz del árbol binario es la raiz del árbol n-ario.
2. Dado un nodo cualquiera x del árbol n-ario, le corresponde un nodo k en el árbol binario tal que:

- El hijo izquierdo de k: k_i, es el hijo mayor (hijo más a la izquierda) del nodo x en el árbol n-ario.
- El hijo derecho de k: k_d es el hermano siguiente al nodo x en el árbol n-ario

Ejemplo:



11.6.6. RECORRIDOS DE UN ARBOL BINARIO

Hay muchos procesos y algoritmos habituales que utilizan la estructura de árbol binario, y una de las operaciones que más frecuentemente hay que realizar sobre ellos es recorrerlos, de forma que se consulten todos los nodos, pero solamente una vez. Para ello basta tratar cada nodo y sus subárboles de la misma forma.

Los principales recorridos son **preorden**, **inorden** y **postorden**, que se definen a continuación.

- **Recorrido en preorden (RID).**

1. Se consulta la raíz.
2. Se recorre el subárbol izquierdo en preorden.
3. Se recorre el subárbol derecho en preorden.

- **Recorrido en inorden (IRD).**

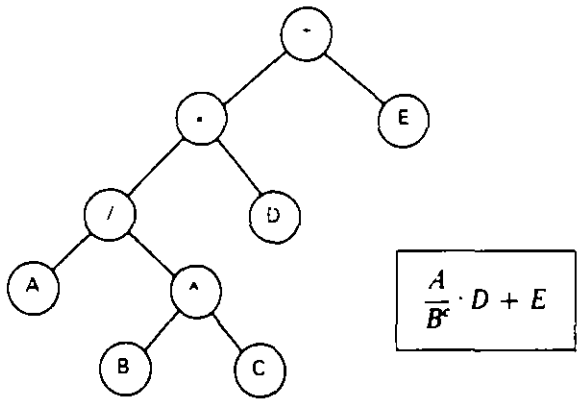
1. Se recorre el subárbol izquierdo en inorden.
2. Se consulta la raíz.
3. Se recorre el subárbol derecho en inorden.

- **Recorrido en postorden (IDR).**

1. Se recorre el subárbol izquierdo en postorden.
2. Se recorre el subárbol derecho en postorden.
3. Se consulta la raíz.

Los nombres preorden, inorden y postorden provienen de la correspondencia natural que existe entre los recorridos y la obtención de las notaciones prefija, infija y postfija de una expresión representada mediante un árbol binario.

Ejemplo: *Sea la expresión numérica representada mediante el siguiente árbol binario.*



$$\frac{A}{B^C} \cdot D + E$$

Preorden: + * / A ^ B C D E
Inorden: A / B ^ C * D + E
Postorden: A B C ^ / D * E +

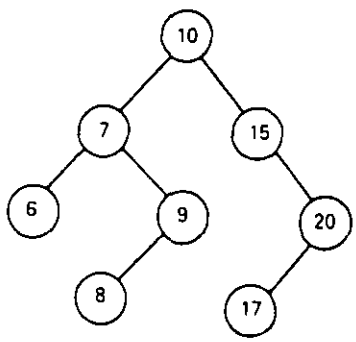
11.6.7. ARBOLES BINARIOS DE BUSQUEDA

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 definición de árbol binario de búsqueda se puede formalizar de la siguiente manera:

1. Todo subárbol de un árbol binario de búsqueda es un árbol binario de búsqueda.
2. Para todo nodo i del subárbol izquierdo de un árbol binario de búsqueda de raíz r se cumple que $i < r$.
3. Para todo nodo d del subárbol derecho de un árbol binario de búsqueda de raíz r se cumple que $r < d$.

Ejemplo:



La utilidad de este árbol se refiere a la eficiencia en la búsqueda de un nodo, ya que en la comparación con uno cualquiera de ellos podemos despreciar la mitad del subárbol correspondiente (igual que se hace en la búsqueda binaria en una tabla).

Asimismo, por medio de árboles binarios de búsqueda, se obtienen los algoritmos de ordenación más rápidos.

En estos árboles la secuencia ordenada de sus nodos se obtiene mediante un recorrido en inorden.

EJERCICIO RESUELTO

1. *Programa para gestionar la lista de personas alojadas en un hotel, con las siguientes condiciones.*

El hotel dispone de 200 habitaciones, numeradas de 1 a 200.

Para cada solicitud de alojamiento de una persona se le asigna la habitación libre de número menor.

El programa debe incluir, mediante un menú, las siguientes opciones:

— Ocupar una habitación.
— Dejar la habitación.
— Consultar el número de habitación ocupada por un huésped.
— Consultar el nombre del ocupante de una habitación.
— Visualizar la lista de todos los huéspedes.
— Visualizar la lista de las habitaciones libres.

La estructura de datos elegida para la representación de la ocupación es la de lista densa, con los campos NOMBRE del huésped y NUMERO de habitación.

Para el control de las habitaciones libres se utiliza un vector, LIBRES, de 200 componentes de tipo booleano.

- **Pseudocódigo:**

```
Programa GESTION-DE-OCUPACION
Entorno:
  HUESPEDES es tabla (200) de
    registro compuesto de
      NOMBRE es alfanumérico
      NUMERO es numérico entero
    finregistro
  NUMERO-HUESPEDES es numérica entera
  LIBRES es tabla (200) booleana
  HUESPED es alfanumérica
  OPERACION, NUM-HAB, POSICION, I son numéricas enteras
  ENCONTRADO es booleana
  FIN es alfanumérica
* *
Algoritmo:
  INICIALIZAR-OCUPACION
  FIN ← "N"
  mientras FIN <> "S" y FIN <> "s" hacer
    PRESENTAR-MENU
    SELECCIONAR-OPERACION
    REALIZAR-OPERACION
  finmientras
Finprograma
```

```
Subprograma INICIALIZAR-OCUPACION
Algoritmo:
  ** Crear lista vacia
  NUMERO-HUESPEDES ← 0
  ** Inicializar vector de habitaciones libres
  para NUM-HAB de 1 a 200 hacer
    LIBRES (NUM-HAB) ← CIERTO
  finpara
  Finsubprograma
* *
Subprograma PRESENTAR-MENU
Algoritmo:
  escribir "    Gestion de la ocupación del hotel"
  escribir "    ------------------------------"
  escribir " 1. Ocupar habitación."
  escribir " 2. Dejar habitación."
  escribir " 3. Consultar por nombre de huésped."
  escribir " 4. Consultar por número de habitación."
  escribir " 5. Visualizar lista de huespedes."
  escribir " 6. Visualizar lista de habitaciones libres."
  escribir " 7. Terminar el programa."
  escribir "              Elija la operación a realizar: "
Finsubprograma
* *
Subprograma SELECCIONAR-OPERACION
Algoritmo:
  leer OPERACION
  mientras OPERACION < 1 o OPERACION > 7 hacer
    escribir " Operación inexistente, elija otra: "
    leer OPERACION
  finmientras
Finsubprograma
* *
Subprograma REALIZAR-OPERACION
Algoritmo:
  opción OPERACION de
    1 hacer OCUPAR-HABITACION
    2 hacer LIBERAR-HABITACION
    3 hacer CONSULTAR-NOMBRE
    4 hacer CONSULTAR-NUMERO
    5 hacer LISTAR-HUESPEDES
    6 hacer LISTAR-LIBRES
    7 hacer CONFIRMAR-FINAL
  finopción
Finsubprograma
* *
Subprograma OCUPAR-HABITACION
Algoritmo:
  si NUMERO-HUSPEDES = 200
    entonces
      escribir "No es posible por ocupación total."
    sino
      escribir " Escriba el nombre del nuevo huésped: "
      leer HUESPED
      BUSCAR-HABITACION
      escribir " Se le asigna la habitación núm. ", NUM-HAB
      NUMERO-HUESPEDES ← NUMERO-HUESPEDES + 1
      NOMBRE (NUMERO-HUESPEDES) ← HUESPED
      NUMERO (NUMERO-HUESPEDES) ← NUM-HAB
      LIBRES(NUM-HAB) ← FALSO
```

```
    finsi
Finsubprograma
* *
Subprograma BUSCAR-HABITACION
Algoritmo:
  NUM-HAB ← 1
  mientras no LIBRES (NUM-HAB) hacer
    NUM-HAB ← NUM-HAB + 1
  finmientras
Finsubprograma
* *
Subprograma LIBERAR-HABITACION
Algoritmo:
  si NUMERO-HUSPEDES = 0
    entonces
      escribir "No es posible, no hay ningún huésped"
    sino
      escribir " Escriba el nombre del huésped: "
      leer HUESPED
      BUSCAR-HUESPED
      si ENCONTRADO
        entonces
          escribir " Queda libre la habitación núm. ",
            NUMERO (POSICION)
          LIBRES (NUMERO (POSICION)) ← CIERTO
          NUMERO-HUESPEDES ← NUMERO-HUESPEDES - 1
          para I de POSICION a NUMERO-HUESPEDES hacer
            HUESPEDES (I) ← HUESPEDES (I + 1)
          finpara
        sino
          escribir "Ese nombre no está en la lista de huéspedes"
      finsi
  finsi
Finsubprograma
* *
Subprograma BUSCAR-HUESPED
Algoritmo:
  POSICION ← 1
  mientras NOMBRE (POSICION) <> HUESPED y
           POSICION < NUMERO-HUESPEDES hacer
    POSICION ← POSICION + 1
  finmientras
  ENCONTRADO ← NOMBRE (POSICION) = HUESPED
Finsubprograma
* *
Subprograma CONSULTAR-NOMBRE
Algoritmo:
  si NUMERO-HUESPEDES = 0
    entonces
      escribir "No es posible, no hay ningún huésped"
    sino
      escribir " Escriba el nombre del huésped: "
      leer HUESPED
      BUSCAR-HUESPED
      si ENCONTRADO
        entonces
          escribir "Su habitación es la núm. ", NUMERO (POSICION)
        sino
          escribir "Ese nombre no está en la lista de huéspedes"
      finsi
```

```
  finsi
Finsubprograma
* *
Subprograma CONSULTAR-NUMERO
Algoritmo:
  si NUMERO-HUESPEDES = 0
    entonces
      escribir "No es posible, no hay ningún huésped"
    sino
      escribir " Escriba el número de habitación: "
      leer NUM-HAB
      BUSCAR-NUM-HAB
      si ENCONTRADO
        entonces
          escribir "Está alojado el Sr. ", NOMBRE (POSICION)
        sino
          escribir "Esa habitación no está ocupada"
    finsi
  finsi
Finsubprograma
* *
Subprograma BUSCAR-NUM-HAB
Algoritmo:
  POSICION ← 1
  mientras NUMERO (POSICION) <> NUM-HAB y
           POSICION < NUMERO-HUESPEDES hacer
    POSICION ← POSICION + 1
  finmientras
  ENCONTRADO ← NUMERO (POSICION) = NUM-HAB
Finsubprograma
* *
Subprograma LISTAR-HUESPEDES
Algoritmo:
  si NUMERO-HUESPEDES = 0
    entonces
      escribir "No es posible, no hay ningún huésped"
    sino
      escribir "Nombre          Número de habitación"
      escribir "------          --------------------"
      para I de 1 a NUMERO-HUESPEDES hacer
        escribir NOMBRE (I), "     ", NUMERO (I)
      finpara
  finsi
Finsubprograma
* *
Subprograma LISTAR-LIBRES
Algoritmo:
  si NUMERO-HUESPEDES = 200
    entonces
      escribir "No hay ninguna habitación libre"
    sino
      escribir "Las habitaciones libres son las siguientes: "
      para NUM-HAB de 1 a 200 hacer
        si LIBRES (NUM-HAB de 1 a 200 hacer
          entonces escribir NUM-HAB, ", "
        finsi
      finpara
  finsi
Finsubprograma
* *
```

```
Subprograma CONFIRMAR-FINAL
Algoritmo:
  escribir "¿Está seguro de que quiere terminar (S/N)? ".
  leer FIN
Finsubprograma
```

EJERCICIOS PROPUESTOS

1. Modificar el programa del ejercicio resuelto para que la lista de huéspedes se mantenga en todo momento clasificada alfabéticamente por nombre.

2. Escribir el programa del ejercicio resuelto utilizando la estructura de lista enlazada.

3. Dadas dos listas enlazadas de números enteros, L1 y L2, clasificadas ascendentemente, escribir un subprograma que obtenga la mezcla de las dos listas en otra lista enlazada L.

4. Escribir un subprograma para mezclar las dos listas del ejercicio anterior en la primera haciendo desaparecer la segunda

5. Dada la lista enlazada de números enteros L, clasificada ascendentemente, escribir un subprograma para eliminar los nodos repetidos, liberando la memoria correspondiente

6. Escribir un programa para decidir si una frase es o no palíndroma (sin considerar los espacios en blanco).

   Ejemplo: «DABALE ARROZ A LA ZORRA EL ABAD».

7. Dos pilas se dicen complementarias si se representan en una misma tabla (cada una comienza en cada uno de sus extremos). Si consideramos que el total de elementos entre las dos es como máximo N, escribir las declaraciones y procedimientos para su implementación.

8. Una cola doble o deque (*Double Queue*) es una lista en la que se permiten inserciones, consultas y supresiones por ambos extremos. Escribir su implementación mediante una tabla circular.

9. Escribir un deque mediante una lista doblemente enlazada (con un puntero al elemento siguiente y otro al anterior).

10. Aplicar la generalización de los recorridos sobre un árbol binario para obtener la secuencia de nodos del árbol *n*-ario del Apartado 11.6.1.

    El recorrido en inorden se generaliza como sigue:

    — Se recorre el primer subárbol de la raíz en inorden.
    — Se consulta la raíz.
    — Se recorren en inorden los subárboles de la raíz 2.°, 3.°, etc.

11. Escribir el árbol binario correspondiente a la expresión

    ```
    ((A + B) + C * (D + E) + F) * (G + H)
    ```

12. Obtener, a partir del árbol del ejercicio anterior, las notaciones prefija y postfija de la expresión representada.

13. Escribir un árbol binario de búsqueda con los elementos:

18, 1, 19, 5, 8, 28, 42, 36, 23, 44, 25, 34 (colóquese 28 en la raiz).

CAPITULO 12

Tablas de decisión

12.1. INTRODUCCION

Se denomina **tabla de decisión** a una **representación tabular de la lógica de un problema, en el que se presentan variadas situaciones y diferentes alternativas para cada una de ellas.**

Aparecieron a finales de los años cincuenta y su utilización fue muy extensa en la década de los sesenta. Actualmente su uso ha quedado bastante reducido, aunque su estudio es interesante, pues muestra cómo se puede abordar un determinado tipo de problemas desde el punto de vista lógico.

Una tabla de decisión es, por tanto:

— Una herramienta para el análisis de problemas.
— Un elemento de comunicación dentro de la jerarquía informática.
— Una representación de problemas que facilita la codificación de los mismos.
— Un instrumento que facilita la detección de errores u omisiones.

Existen programas específicos para tablas de decisión que contienen instrucciones preparadas para la resolución de este tipo de representación de problemas.

A nivel de programación, una tabla de decisión consiste en la representación de un programa o parte del mismo, de tal forma que a partir de unos datos de entrada determinados realiza una acción de acuerdo con las condiciones del problema de que se trate.

Dos objetivos importantes en la programación de computadoras, que se deben tener siempre en cuenta, son: diseñar programas con ocupación mínima de memoria y que su tiempo de ejecución sea también el mínimo posible. El proceso de resolución de una tabla de decisión lleva a conseguir los objetivos anteriores a la vez que ayuda a clarificar el problema.

Las tablas de decisión nos proporcionan, a partir del planteamiento de un problema, la posibilidad de su codificación en un lenguaje de programación, de tal manera que la solución obtenida es la óptima.

12.2. ESTRUCTURA DE UNA TABLA DE DECISION

Una tabla de decisión es una representación en la que se distinguen cuatro zonas:

— **Condiciones.** Consta de un vector columna donde figuran las condiciones que intervienen en el problema. Si se establece *a priori* un orden de importancia de las mismas, las más importantes deben figurar en la parte superior y las de menor importancia en su parte inferior.

— **Acciones.** Consta de un vector columna en el que aparecen las acciones a realizar. Si en algún caso, para un estado determinado de las condiciones, se realizan varias acciones y éstas se tienen que ejecutar en un orden preestablecido, figurarán en ese orden, de arriba a abajo.

— **Entrada de condiciones.** Es una matriz de tantas filas como condiciones y columnas como situaciones distintas se puedan presentar.

— **Salida de acciones.** Matriz en la que figuran tantas filas como acciones y columnas como situaciones distintas se pueden presentar.

CONDICIONES	ENTRADA DE CONDICIONES
ACCIONES	SALIDA DE ACCIONES

Por ejemplo, supongamos que tenemos tres condiciones que se evalúan con «se cumple» (S) o «no se cumple» (N) y tres acciones que se ejecutarán según la siguiente representación:

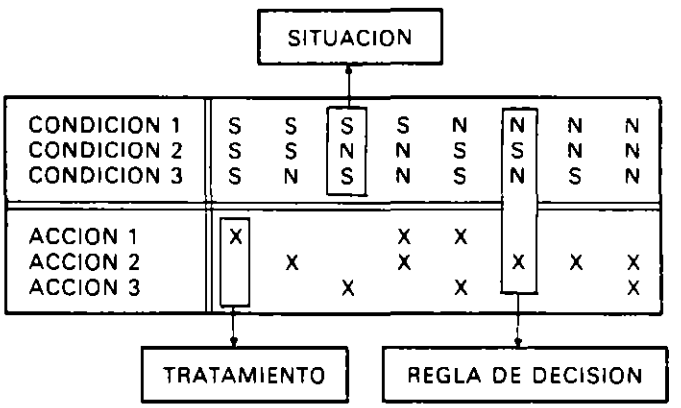
CONDICION 1	S	S	S	S	N	N	N	N
CONDICION 2	S	S	N	N	S	S	N	N
CONDICION 3	S	N	S	N	S	N	S	N
ACCION 1	X			X	X			
ACCION 2		X		X		X	X	X
ACCION 3			X		X			X

Cada columna de la matriz que representa la entrada de condiciones, correspondiente a un determinado estado de las mismas, se denomina **situación**.

Cada columna de la matriz que representa la salida de acciones se denomina **tratamiento**.

Las acciones que componen un tratamiento son las que corresponden a las aspas (X) que hay en la columna. Estas se ejecutan, en caso de haber más de un aspa, de arriba a abajo, salvo que se especifique otro orden mediante subíndices en las mismas (X_1, X_2, etc.).

Definimos como **regla de decisión** el conjunto formado por una **situación** y su **tratamiento** correspondiente.



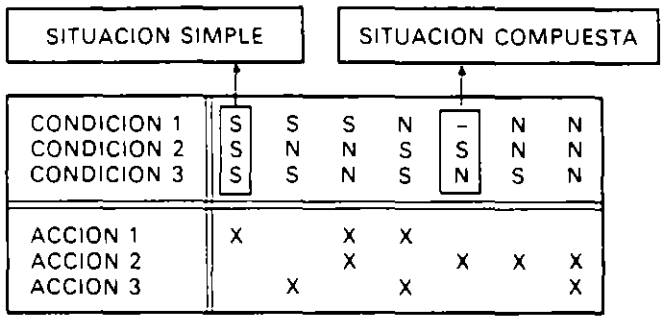
Las situaciones pueden ser de dos tipos:

— **Simples.** Son aquellas en las que todas las condiciones han de ser evaluadas.

— **Compuestas.** Son aquellas en las que una o más condiciones no precisan ser evaluadas. Se dice entonces que la situación tiene una indiferencia en esas condiciones y se representa mediante un guión (—).

Una situación compuesta equivale a tantas situaciones simples como el resultado de multiplicar el número de valores de cada condición que es indiferente en esa situación.

Ejemplo:



La situación compuesta de la anterior tabla de decisión (T.D.) equivale a dos situaciones simples por tener dos posibles valores la CONDICION 1 en la cual posee una indiferencia.

-		S N
S	equivale a	S S
N		N N

Se dice que una regla es simple o compuesta según lo sea la situación que contiene.

12.3. TIPOS DE REGLAS

- **Reglas AND.** Son aquellas en las que se tienen que dar «todos» los valores de su situación. Sus indiferencias se representan mediante un guión (−).
- **Reglas OR.** En ellas es suficiente con que se tenga «alguno» de los valores de su situación. Sus indiferencias se representan mediante un asterisco (*), siendo necesario que haya al menos una, puesto que una regla sin indiferencia es del tipo AND.
- **Reglas ELSE.** Es una regla, única en una T.D., en la que no se evalúa «ninguna» condición. Agrupa todas las reglas no contempladas explícitamente en la T.D.

Ejemplo: *Tabla de decisión que expresa las acciones a realizar por un joven en edad próxima al servicio militar, suponiendo que los condicionantes son los expresados en la misma, así como las acciones a realizar:*

Es voluntario	*	N	E
Ha sido sorteado	*	N	L
Casado, con hijos	S	N	S
Está incapacitado	S	N	E
Exento del servicio	X		
Esperar reemplazo		X	
Incorporarse a filas			X

La primera regla de la T.D. es del tipo OR (lo que se expresa con uno o más asteriscos en sus indiferencias), indicando que si se cumple alguna de las condiciones «Casado, con hijos» o «Está incapacidado», se está «Exento del servicio».

La segunda es del tipo AND, indicando que si todas las condiciones resultan negativas, entonces se ha de «Esperar reemplazo».

La tercera es del tipo ELSE, indicando que en cualquier otra situación es preciso «Incorporarse a filas».

La anterior T.D., desarrollada completamente y utilizando reglas del tipo AND, es:

Es voluntario	S	S	S	S	S	S	S	S	N	N	N	N	N	N	N	N
Ha sido sorteado	S	S	S	S	N	N	N	N	S	S	S	S	N	N	N	N
Casado, con hijos	S	S	N	N	S	S	N	N	S	S	N	N	S	S	N	N
Está incapacitado	S	N	S	N	S	N	S	N	S	N	S	N	S	N	S	N
Exento de servicio	X	X	X		X	X	X		X	X	X		X	X	X	
Esperar reemplazo																X
Incorporarse a filas				X				X				X				

Las reglas OR no suelen utilizarse en la práctica, por lo que serán sustituidas por otras AND equivalentes en caso de tenerlas.

12.4. CLASIFICACION DE LAS TABLAS DE DECISION

Las T D. se clasifican, según el número de valores que pueden tomar sus condiciones, en:

- **Tablas de decisión binarias.**
- **Tablas de decisión múltiples.**
- **Tablas de decisión mixtas.**

Si una T.D. entre sus acciones tiene referencias a otra T.D., se denomina **tabla de decisión abierta**, y en caso contrario, **tabla de decisión cerrada.**

12.4.1. TABLAS DE DECISION BINARIAS

Son aquellas cuyas condiciones sólo pueden tomar dos valores. También se denominan **limitadas**.

Estos valores, en general, serán sí (S) y no (N), aunque pueden ser otros valores binarios como, por ejemplo, blanco (B) y negro (N).

Ejemplo: *Se dispone de una aplicación para representar gráficos con computadora por pantalla o impresora, dependiendo de las características del equipo (monitor monocromo (M) o color (C), tarjeta de gráficos (S/N) e impresora gráfica (S/N)).*

La siguiente T.D. binaria expresa las diferentes posibilidades de ejecutar la aplicación.

Tipo de monitor	M	M	M	M	C	C	C	C
Tarjeta gráfica	S	S	N	N	S	S	N	N
Impresora gráfica	S	N	S	N	S	N	S	N
Visualizar pantalla	X	X			X	X		
Sacar por impresora	X		X		X			
Equipo insuficiente				X				
Contradicción							X	X

Las dos últimas situaciones son contradicciones (no se pueden presentar), no obstante deben figurar en la tabla, por lo cual se ha añadido una última acción que expresa tales contradicciones.

12.4.2. TABLAS DE DECISION MULTIPLES

Son aquellas en que todas sus condiciones pueden tomar más de dos valores. También se denominan **ampliadas**.

Ejemplo: *Una empresa, en la confección de la nómina, incluye un plus sobre el sueldo de sus empleados, dependiendo de su estado civil (S/C/V/D) y de su nivel de estudios (primarios (P), medios (M) y superiores (S)).*

La siguiente T.D múltiple refleja los diferentes pluses aplicados.

Estado civil	S	S	S	C	C	C	V	V	V	D	D	D
Nivel estudios	P	M	S	P	M	S	P	M	S	P	M	S
Plus de primera						X			X			
Plus de segunda			X		X			X			X	X
Plus de tercera		X		X			X			X		
Plus de cuarta	X											

12.4.3. TABLAS DE DECISION MIXTAS

Son aquellas en que intervienen condiciones binarias y múltiples.

Ejemplo: *Un centro de estudios que prepara opositores a banca distribuye sus cursos según las siguientes condiciones de sus alumnos: nivel de estudios (P/M/S), mayoría de edad (S/N) y experiencia (S/N).*

La siguiente T.D mixta refleja las diferentes posibilidades de realización de cursos.

Nivel estudios	P	P	P	P	M	M	M	M	S	S	S	S
Mayor de edad	S	S	N	N	S	S	N	N	S	S	N	N
Experiencia	S	N	S	N	S	N	S	N	S	N	S	N
Directivos									X			
Técnicos administrativos					X		X		X	X		
Auxiliares administrativos					X	X	X	X		X		
Botones	X	X	X	X		X		X				
Contradicción											X	X

12.5. PROCESO DE RESOLUCION DE UNA TABLA DE DECISION

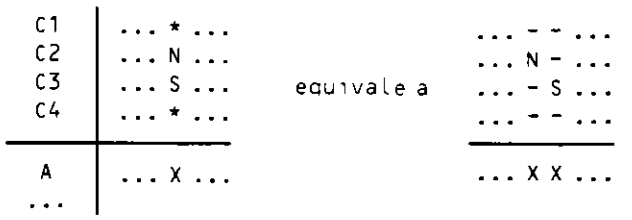
Una T.D. surge del análisis de un problema en el que se han de aplicar diferentes tratamientos para las distintas situaciones que se puedan presentar. El proceso de resolución de la misma puede partir desde el propio enunciado del problema o a partir de una T.D. previamente confeccionada en alguna fase anterior del análisis.

En el resto del capítulo se estudian T.D. binarias con reglas AND exclusivamente, por lo que previamente a iniciar el proceso de resolución de la misma se han de sustituir las reglas OR por reglas AND y las condiciones múltiples por binarias.

- **Transformación de reglas OR en reglas AND:**

Cada regla OR que se evalúa en *n* condiciones equivale a *n* reglas AND en las que cada una tiene valor en una de las condiciones evaluadas en la OR y en el resto de condiciones indiferencias.

Ejemplo:



- **Tranformación de condiciones múltiples en binarias:**

Cada condición múltiple con *n* valores equivale a *n* condiciones binarias, consistentes en transformar cada valor en una condición.

Ejemplo: *La condición «Nivel de estudios», con los valores primarios (P), medios (M) y superiores (S) equivale a las siguientes condiciones:*

— Estudios primarios (S/N).
— Estudios medios (S/N).
— Estudios superiores (S/N).

Los pasos que se deben seguir en la resolución de una T.D. binaria con reglas AND son los siguientes:

— **Eliminar redundancias.**
— **Comprobar que es completa.**
— **Simplificar.**
— **Ordenar por importancia de reglas y condiciones.**

12.5.1. REDUNDANCIAS

Una T.D. tiene redundancias si existe una o más situaciones repetidas. El primer paso a realizar con una T.D. es eliminarlas.

Existen dos tipos de redundancias:

— **Redundancias congruentes.** Son las que tienen el mismo tratamiento. Se eliminan de la T.D. sin más.
— **Redundancias incongruentes.** Son las que tienen diferente tratamiento, lo cual implica que hay una contradicción que se ha de resolver revisando el enunciado del problema.

12.5.2. COMPLETITUD

Una T.D. es completa si contiene todas las situaciones posibles.

Para la resolución correcta del problema es necesario que la T.D. sea completa.

- **Contar situaciones**

Una T.D. con *n* condiciones tiene:

$$\text{Número de situaciones posibles} = 2^n$$

Se comprobará que el número de situaciones simples existentes en la T.D. coincide con el de situaciones posibles, teniendo en cuenta que una situación compuesta con *n* indiferencias equivale a 2^n situaciones simples.

Ejemplo: *Comprobar si es completa la siguiente T.D. en la que no existen redundancias.*

CONDICION 1	S	S	S	N
CONDICION 2	S	-	N	-
CONDICION 3	S	N	S	-
ACCION 1		X		X
ACCION 2			X	
ACCION 3	X	X		

El número de situaciones posibles es 2^n, luego:

NSP=2*2*2=8

El número de situaciones existentes es:

La primera situación es simple 1
La segunda tiene una indiferencia, por tanto equivale a dos situaciones simples 2
La tercera situación es simple 1
La cuarta tiene dos indiferencias, por tanto equivale a 2 * 2 situaciones simples 4

NSE = 8

La tabla es completa por coincidir el número de situaciones posibles (NSP) con el número de situaciones existentes (NSE).

- **Desarrollo completo de la T.D.**

Consiste en sustituir todas las situaciones compuestas por las correspondientes situaciones simples (en el caso de existir indiferencias) y comprobar que hay 2^n situaciones.

Ejemplo: *El desarrollo completo de la T.D. del ejemplo anterior es:*

CONDICION 1	S	S	S	S	N	N	N	N
CONDICION 2	S	S	N	N	S	S	N	N
CONDICION 3	S	N	N	S	S	N	S	N
ACCION 1		X	X		X	X	X	X
ACCION 2				X				
ACCION 3	X	X	X					

12.5.3. SIMPLIFICACION

Una vez eliminadas las redundancias y comprobada la completitud de la T.D. se procede a simplificarla para obtener el máximo número de indiferencias posibles.

El proceso de simplificación consiste en agrupar cada par de reglas sustituyéndolo por otra con una nueva indiferencia, siempre que se cumplan las siguientes condiciones:

— Ambas reglas han de tener igual tratamiento.
— Sus situaciones deben coincidir en los valores de todas las condiciones, excepto en una.

La regla resultante del agrupamiento tendrá:

— El mismo tratamiento.
— Igual valor en todas las condiciones en que coincidian éstos.
— Una indiferencia en la condición en que no coincidian.

Ejemplos:

a)

C1	S S
C2	N N
C3	N S
A1	
A2	X X
A3	

equivale a

C1	S
C2	N
C3	–
A1	
A2	X
A3	

b)

C1	N S
C2	– –
C3	S S
A1	X X
A2	
A3	X X

equivale a

C1	–
C2	–
C3	S
A1	X
A2	
A3	X

El proceso de agrupamiento de reglas se puede generalizar a más de dos simultáneamente mediante la denominada **regla del paraguas**, que consiste en el establecimiento de ciclos cerrados uniendo pares de reglas que cumplen las condiciones antes enunciadas.
Los agrupamientos posibles de reglas con este método son:

2 simples en una compuesta con una indiferencia.
4 simples en una compuesta con 2 indiferencias.
8 simples en una compuesta con 3 indiferencias.
...
2^n simples en una compuesta con n indiferencias.

Ejemplo: *Las tablas siguientes son equivalentes:*

C1	S	S	S	S	S	S	S	S	N	N	N	N	N	N	N	N
C2	S	S	S	S	N	N	N	N	S	S	S	S	N	N	N	N
C3	S	S	N	N	S	S	N	N	S	S	N	N	S	S	N	N
C4	S	N	S	N	S	N	S	N	S	N	S	N	S	N	S	N
A1			X						X	X	X	X	X			
A2	X	X		X	X	X	X	X					X	X		
A3				X					X	X	X	X		X	X	X

C1	S	S	S	S	N	N	N	N
C2	-	S	S	N	S	N	N	N
C3	S	N	N	N	-	S	S	N
C4	-	S	N	-	-	S	N	-
A1		X			X	X		
A2	X		X	X		X	X	
A3			X		X		X	X

Si al utilizar esta regla, con un grupo de situaciones de igual tratamiento, no se encuentra un ciclo cerrado, la simplificación óptima se consigue tomando el mayor número de arcos disjuntos para agrupar reglas.

12.5.4. ORDENACION POR IMPORTANCIAS

El último paso de la resolución de una T.D. consiste en ordenarla colocando las reglas más importantes hacia la izquierda y las condiciones más importantes hacia arriba.

El objetivo es conseguir que la transformación en programa de la T.D. sea la óptima.

La importancia de una regla (I.R.) viene dada por el número de reglas simples a que equivale.

La importancia de una condición (I.C.) es la suma de las importancias de las reglas en las que la condición es evaluada.

En el método diseñado por S L. Pollack se denomina **Dash Count** (D.C.) de una condición a la suma de las importancias de las reglas en las que la condición no es evaluada, es decir, la diferencia entre el número de situaciones posibles de la T.D. (2^n) y la importancia de la condición.

Una T.D. así ordenada contendrá hacia arriba las condiciones con menos indiferencias, y hacia la izquierda las reglas con más indiferencias.

Ejemplo: *Ordenar las siguiente T.D.*

Importancia de reglas

	4	1	1	2	4	1	1	2			
C1	-	S	N	-	-	S	N	-	4		12
C2	-	S	S	N	S	N	N	N	12	Importancia	4
C3	S	S	S	S	N	N	N	N	16	de condiciones	0
C4	S	N	N	N	-	S	S	N	12		4
A1		X			X	X					
A2	X		X	X		X	X				
A3	X			X	X			X			

D C

- Ordenación por reglas.

I.R	4	4	2	2	1	1	1	1
C1	-	-	-	-	S	N	S	N
C2	-	S	N	N	S	S	N	N
C3	S	N	S	N	S	S	N	N
C4	S	-	N	N	N	N	S	S
A1		X			X		X	
A2	X		X			X	X	X
A3	X	X	X	X				

- Ordenación por condiciones.

									I.C.	δ
C3	S	N	S	N	S	S	N	N	16	0
C2	-	S	N	N	S	S	N	N	12	1
C4	S	-	N	N	N	N	S	S	12	1
C1	-	-	-	-	S	N	S	N	4	0
A1		X			X		X			
A2	X		X			X	X	X		
A3	X	X	X	X						

En aquellos casos en que la I.C. de dos condiciones coincide se calcula un segundo coeficiente δ que se obtiene mediante el valor absoluto (sin signo) de la diferencia entre la cantidad de valores «S» y de valores «N». La condición con mayor δ superará en importancia a la otra. En la tabla anterior puede verse que C2 y C4 son de idéntica importancia por coincidir la I.C. y δ, por tanto se pueden colocar en cualquier orden.

Esta última T.D. no tiene redundancias, es completa, está simplificada y ordenada, con lo cual se ha concluido todo el proceso.

12.6. PASO A ORDINOGRAMA

Para confeccionar el ordinograma partimos de la T.D. resultante del proceso anterior.

El ordinograma será un anidamiento de alternativas que se puede realizar de dos maneras:

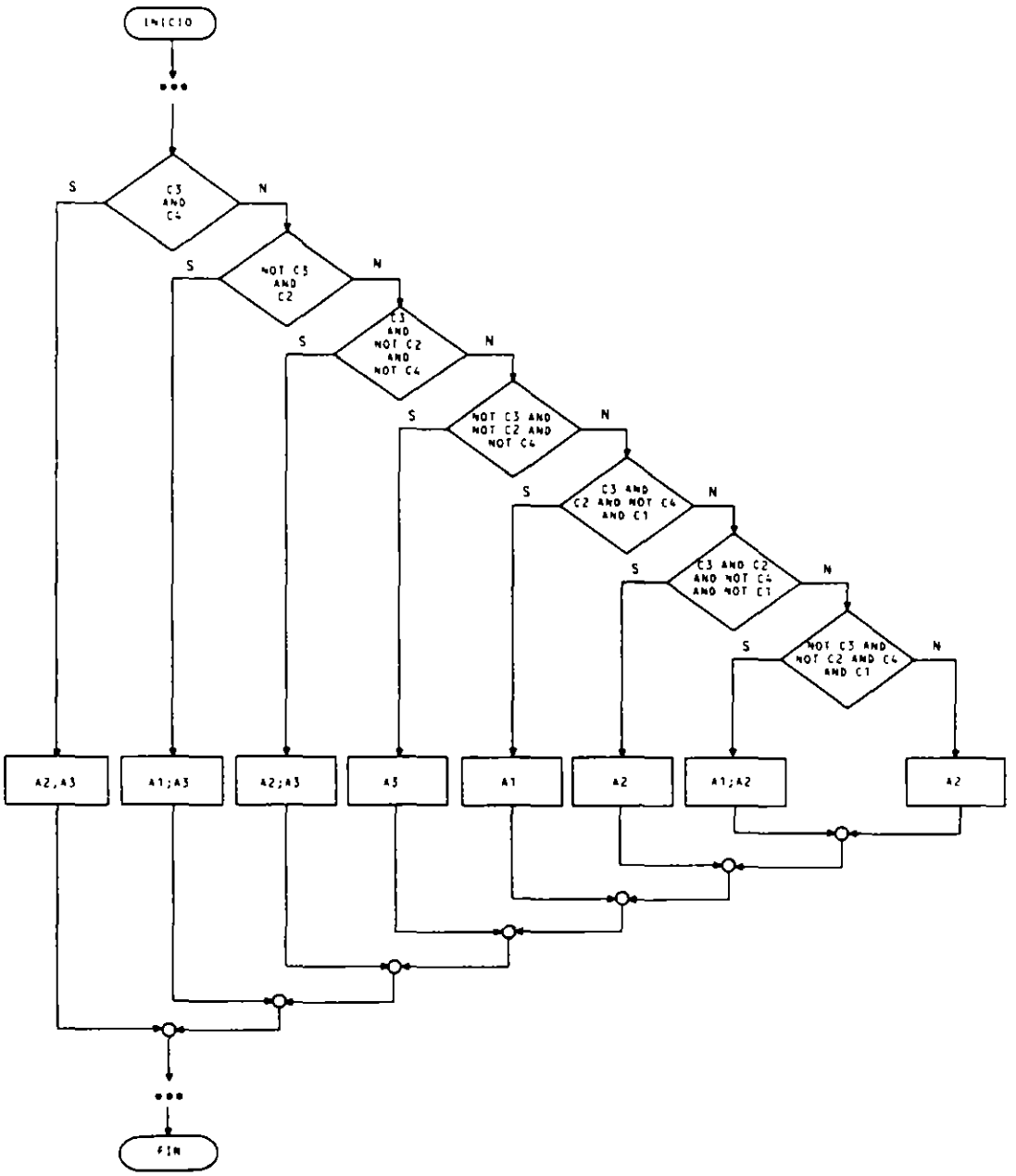
- **Ordinograma mediante operadores AND.**

Se anidan las reglas de izquierda a derecha, asociando cada situación a su tratamiento.

Ejemplo: *Ordinograma (AND) de la T.D. siguiente:*

C3	S	N	S	N	S	S	N	N
C2	-	S	N	N	S	S	N	N
C4	S	-	N	N	N	N	S	S
C1	-	-	-	-	S	N	S	N
A1		X			X		X	
A2	X		X			X	X	X
A3	X	X	X	X				

- **Ordinograma:**



- **Ordinograma mediante condiciones simples** (Método de Pollack)

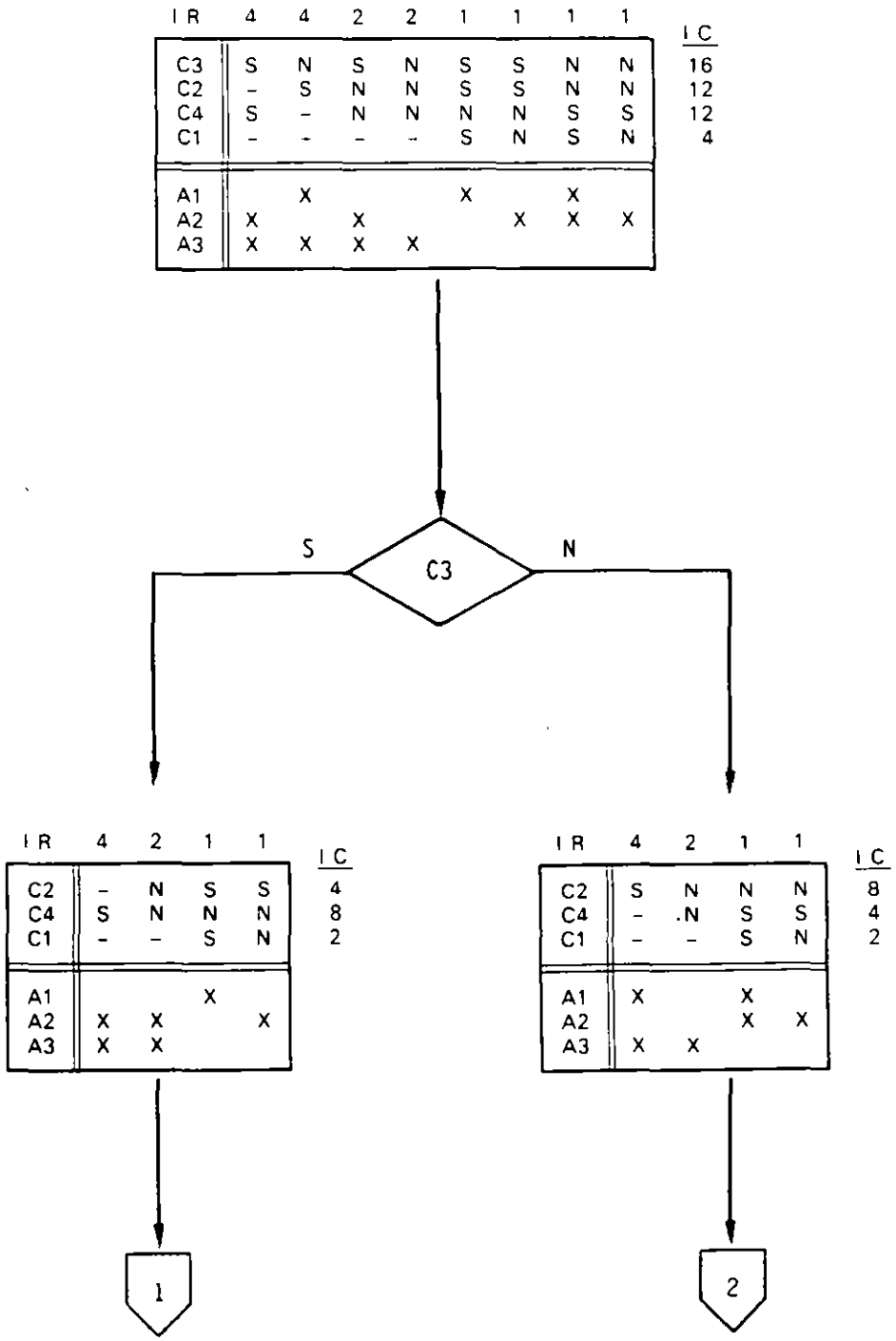
Se inicia el ordinograma con la primera condición, dividiendo la T.D. en dos, correspondientes a los dos valores de la condición tomada, que ya no figura en las nuevas.

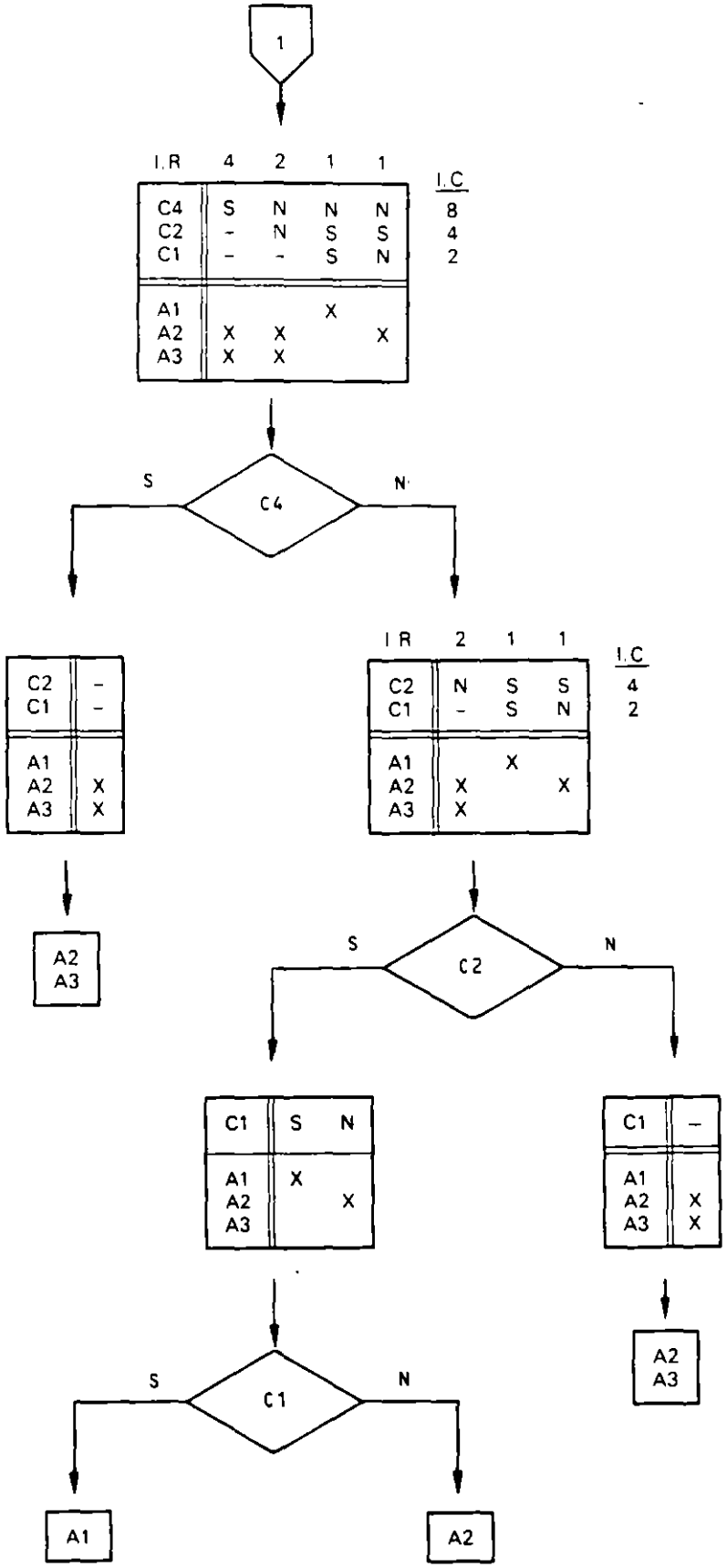
Cada tabla resultante se ordena de nuevo, y se repite el mismo proceso que representa la continuación del ordinograma por la rama correspondiente al valor de la condición que se tomó.

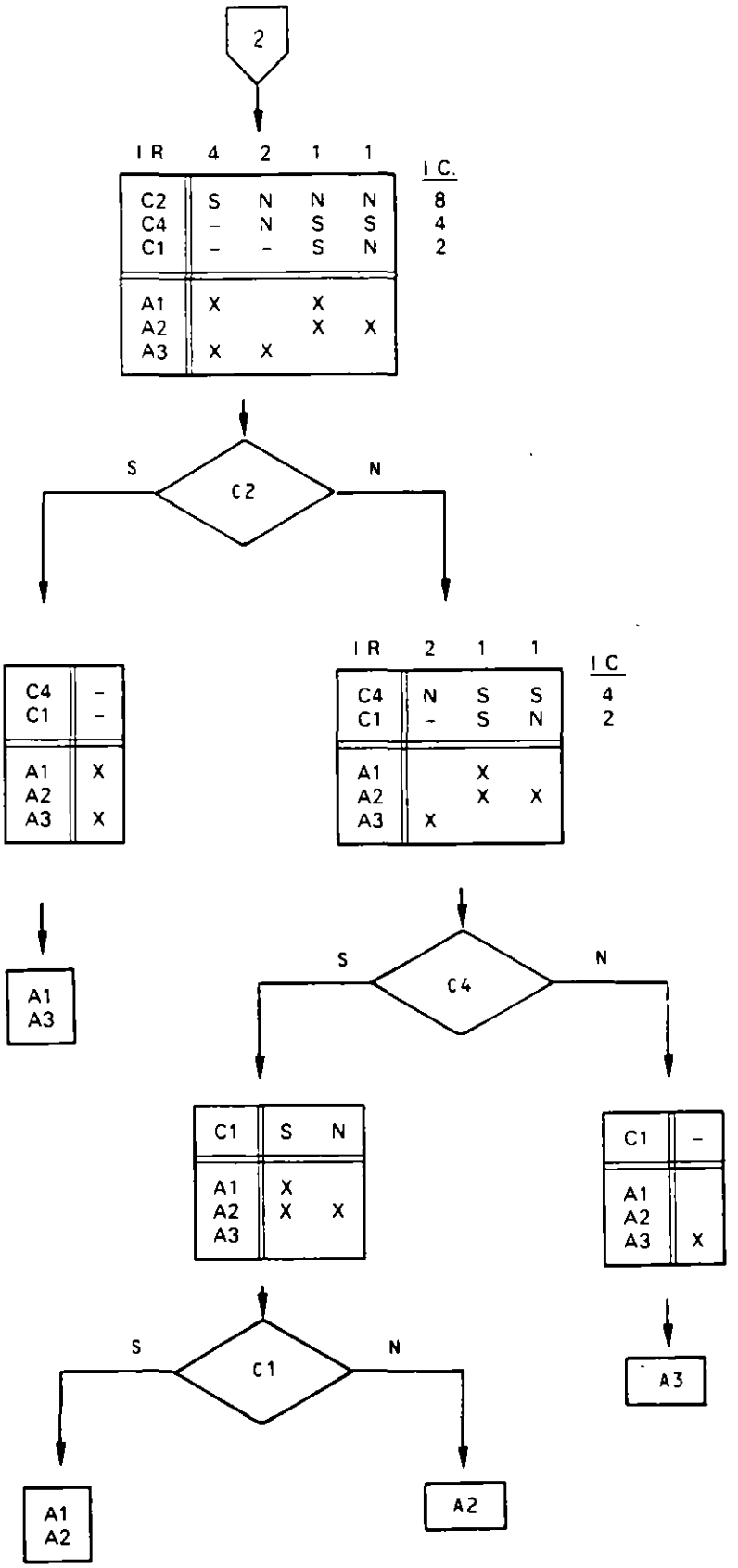
El proceso termina al llegar por cada una de las ramas a una T.D. que tenga:

— Todo indiferencias. En este caso se conecta a la rama el tratamiento de la única regla que puede haber.
— Una sola condición con sus dos valores. Se toma la condición y se conectan cada uno de los tratamientos a las ramas establecidas por los dos valores de la condición.

Ejemplo: *Construcción del ordinograma mediante condiciones simples de la T.D. siguiente*

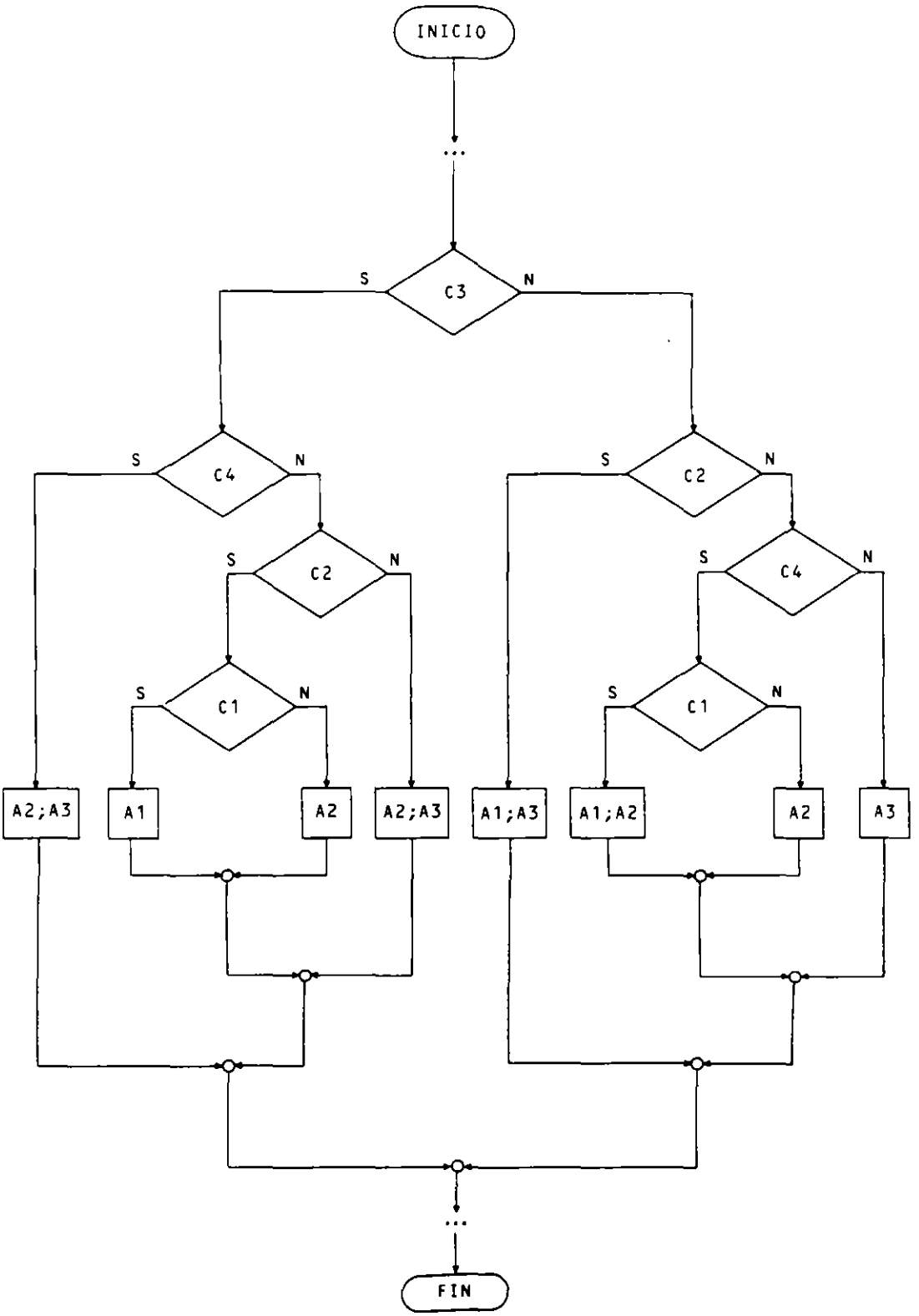



1
I.R 4 2 1 1
I.C
C4 S N N N 8
C2 - N S S 4
C1 - - S N 2
A1 X
A2 X X X
A3 X X
S
C4
N
C2 -
C1 -
A1
A2 X
A3 X
A2
A3
I.R 2 1 1
I.C
C2 N S S 4
C1 - S N 2
A1 X
A2 X X
A3 X
S
C2
N
C1 S N
A1 X
A2 X
A3
C1 -
A1
A2 X
A3 X
A2
A3
S
C1
N
A1
A2


2
I R 4 2 1 1
I C.
C2 S N N N 8
C4 - N S S 4
C1 - - S N 2
A1 X X
A2 X X
A3 X X
S
C2
N
C4 -
C1 -
A1 X
A2
A3 X
I R 2 1 1
I C
C4 N S S 4
C1 - S N 2
A1 X
A2 X X
A3 X
A1
A3
S
C4
N
C1 S N
A1 X
A2 X X
A3
C1 -
A1
A2
A3 X
S
C1
N
A1
A2
A3

- **Ordinograma:**

Resultado global del proceso anterior.



EJERCICIOS RESUELTOS

1. *Resolver y pasar a ordinograma la siguiente T D. Si hay alguna redundancia incongruente eliminar la regla que tenga un tratamiento de mayor número. Si falta alguna situación ponerle como tratamiento A4.*

C1	N	S	S	N	–	S	N
C2	–	S	–	S	S	N	N
C3	S	S	N	–	N	S	–
C4	S	–	S	–	N	–	–
A1	X			X			X
A2		X				X	
A3			X				
A4					X		

Primer paso: Estudiar redundancias.

Se desarrolla la T.D. y se examinan visualmente las posibles redundancias.

C1	N	N	S	S	S	S	N	N	N	N	S	N	S	S	N	N	N	N
C2	S	N	S	S	S	N	S	S	S	S	S	S	N	N	N	N	N	N
C3	S	S	S	S	N	N	S	S	N	N	N	N	S	S	S	S	N	N
C4	S	S	S	N	S	S	S	N	S	N	N	N	S	N	S	N	S	N
A1	X	X					X	X	X	X					X	X	X	X
A2			X	X									X	X				
A3					X	X												
A4											X	X						
	1	2					1				3		3			2		

Existen las redundancias congruentes marcadas con los números 1 y 2; se eliminan quitando una de las reglas repetidas.

Existe la redundancia incongruente marcada con el número 3; según el enunciado se ha de eliminar la regla de la derecha por tener tratamiento superior.

Segundo paso: Estudiar completitud.

Se ordena la tabla anterior por situaciones para examinar si falta alguna:

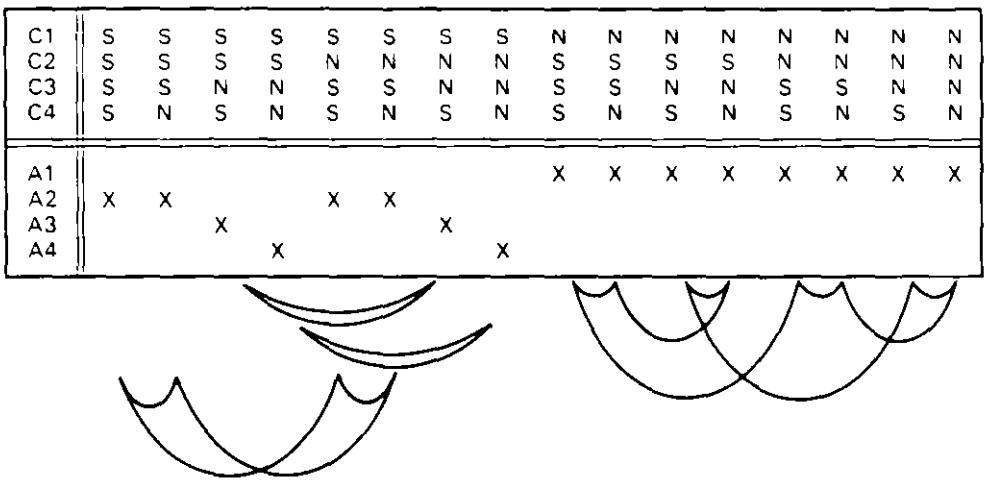
C1	S	S	S	S	S	S	S	N	N	N	N	N	N	N	N	
C2	S	S	S	S	N	N	N	S	S	S	S	N	N	N	N	
C3	S	S	N	N	S	S	N	S	S	N	N	S	S	N	N	
C4	S	N	S	N	S	N	S	S	N	S	N	S	N	S	N	
A1								X	X	X	X	X	X	X	X	
A2	X	X			X	X										
A3			X				X									
A4				X												

Se observa que la tabla es incompleta por faltar la situación «S N N N». Según el enunciado se debe incluir con el tratamiento A4.

C1	S	S	S	S	S	S	S	S	N	N	N	N	N	N	N	N
C2	S	S	S	S	N	N	N	N	S	S	S	S	N	N	N	N
C3	S	S	N	N	S	S	N	N	S	S	N	N	S	S	N	N
C4	S	N	S	N	S	N	S	N	S	N	S	N	S	N	S	N
A1									X	X	X	X	X	X	X	X
A2	X	X			X	X										
A3			X				X									
A4				X				X								

Tercer paso: Simplificar.

Se establecen todos los arcos posibles, según la regla del paraguas, y se buscan ciclos cerrados



La tabla simplificada que se obtiene es:

C1	S	S	S	N
C2	–	–	–	–
C3	S	N	N	–
C4	–	S	N	–
A1				X
A2	X			
A3		X		
A4			X	

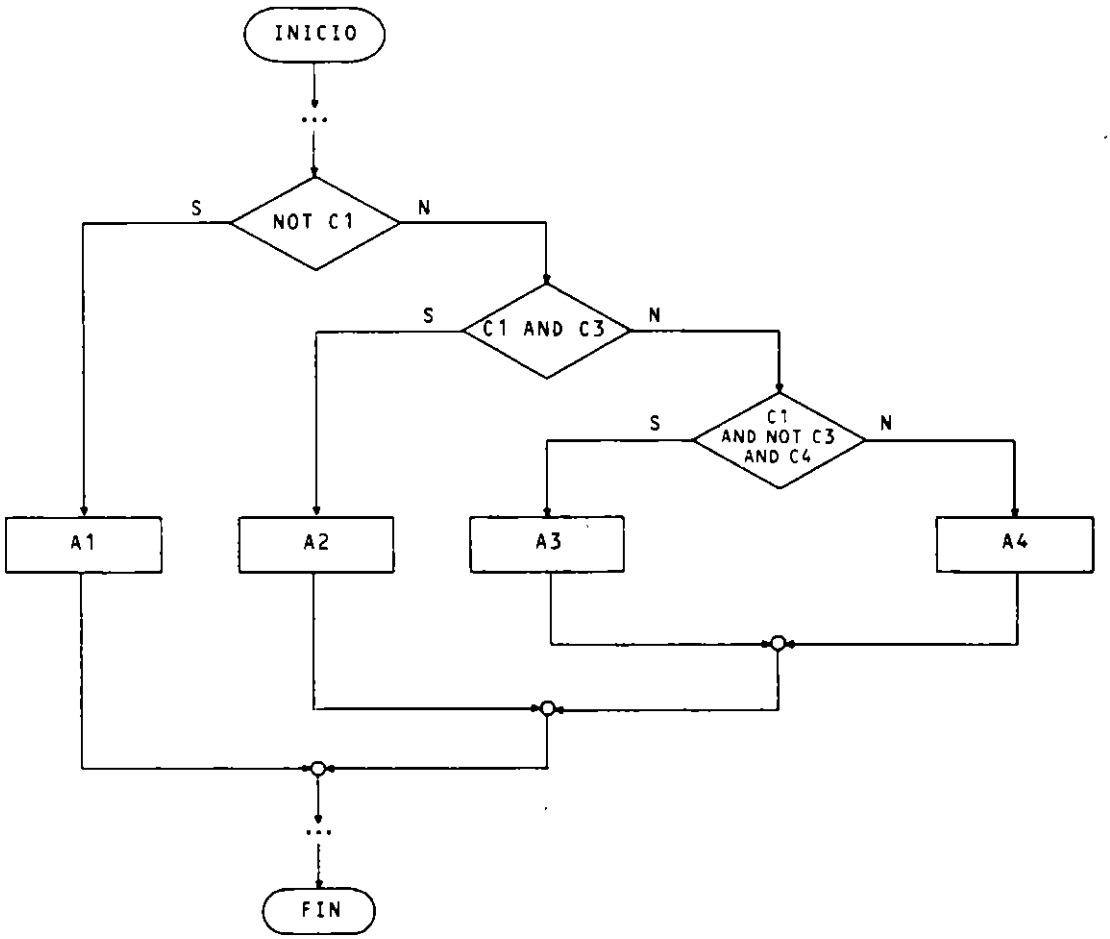
Cuarto paso: Ordenar por importancias.

I R	8	4	2	2	I C.
C1	N	S	S	S	16
C3	–	S	N	N	8
C4	–	–	S	N	4
C2	–	–	–	–	0
A1	X				
A2		X			
A3			X		
A4				X	

La condición C2 puede ser eliminada por tener importancia 0, es decir, no necesita ser evaluada.

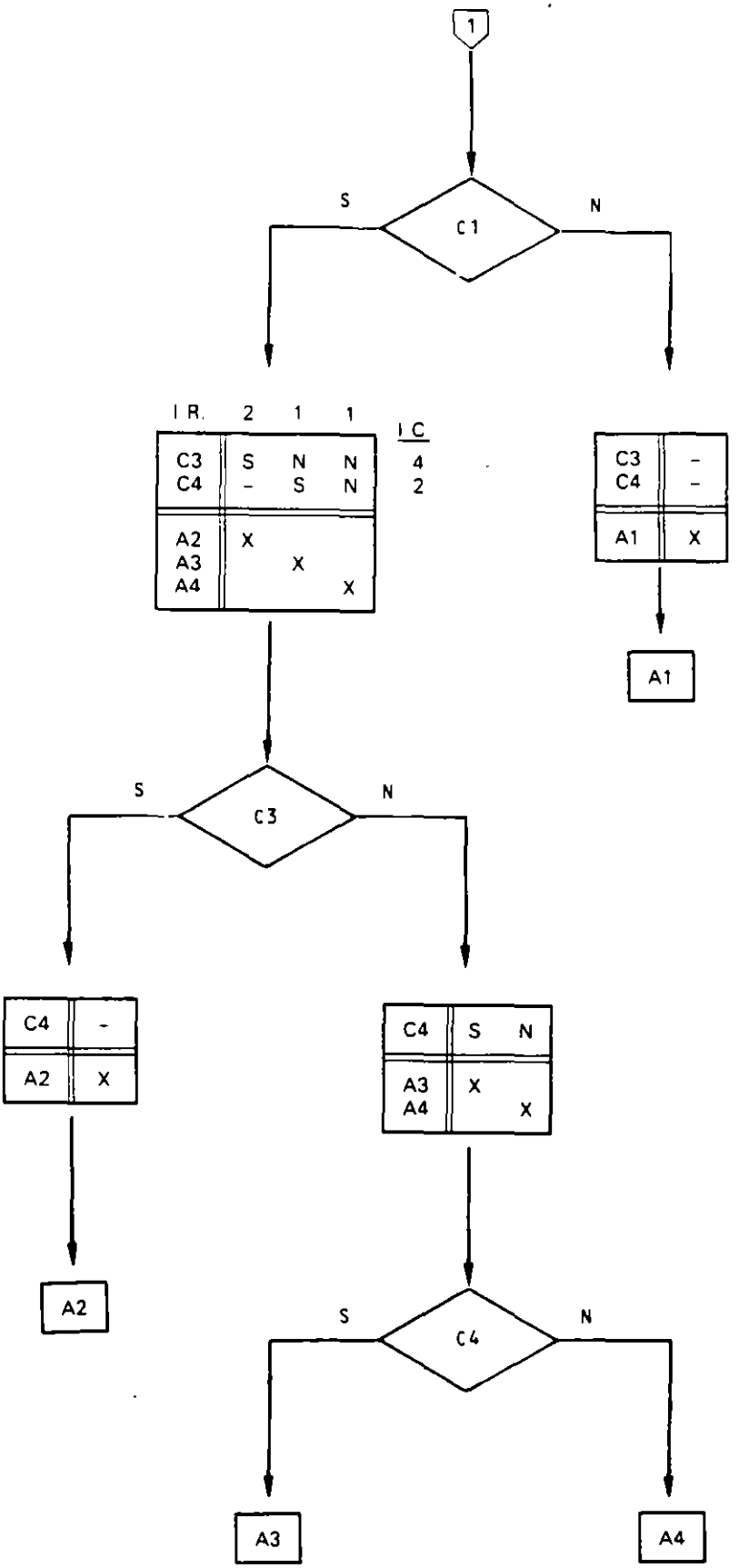
Paso a ordinograma:

Utilizando el primer método, mediante operadores AND, se tiene:

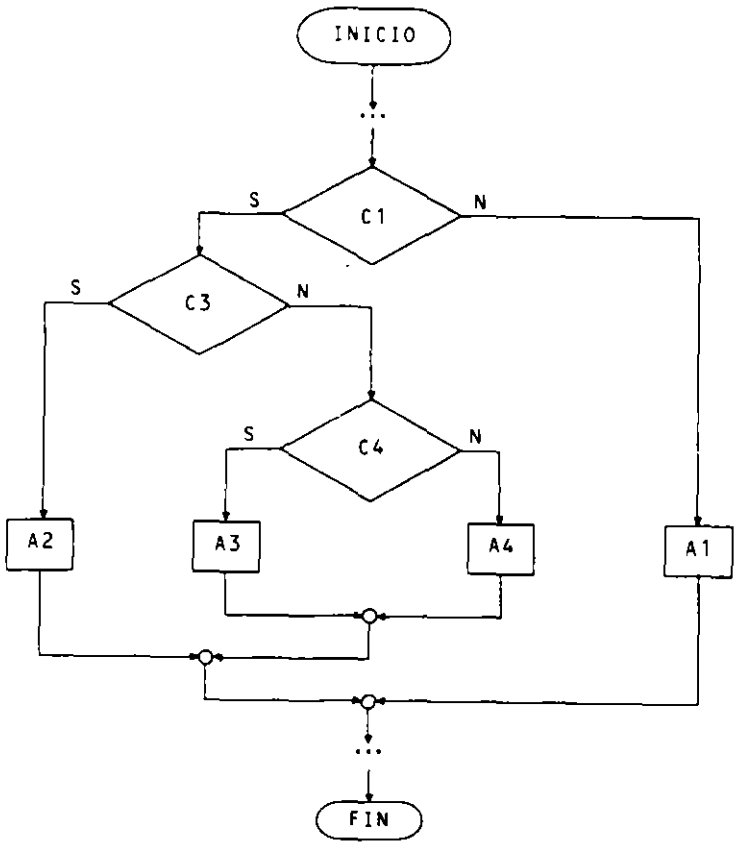


Por el segundo método, mediante condiciones simples, se obtiene:

I R	8	4	2	2	I C.
C1	N	S	S	S	16
C3	-	S	N	N	8
C4	-	-	S	N	4
A1	X				
A2		X			
A3			X		
A4				X	


1
S
C1
N
I R. 2 1 1
I C
C3 S N N 4
C4 - S N 2
A2 X
A3 X
A4 X
C3 -
C4 -
A1 X
A1
S
C3
N
C4 -
A2 X
A2
C4 S N
A3 X
A4 X
S
C4
N
A3
A4

El ordinograma resultante es:



2. *Un número natural del 0 al 15 puede ser representado en base 2 utilizando cuatro dígitos binarios* (b_3, b_2, b_1, b_0).

Ejemplo: *El número 12 se representa.*

1	1	0	0
↑	↑	↑	↑
b_3	b_2	b_1	b_0

Consideremos los cuatro dígitos binarios de un número como las entradas de las condiciones de una tabla de decisión donde el 1 es equivalente al S y el 0 al N.

Se quiere obtener como salida un mensaje que refleje si el número es o no primo.

Plantear, resolver y pasar a ordinograma la T.D. correspondiente.

Consideramos el 0 como número primo.

A partir del enunciado planteamos la tabla:

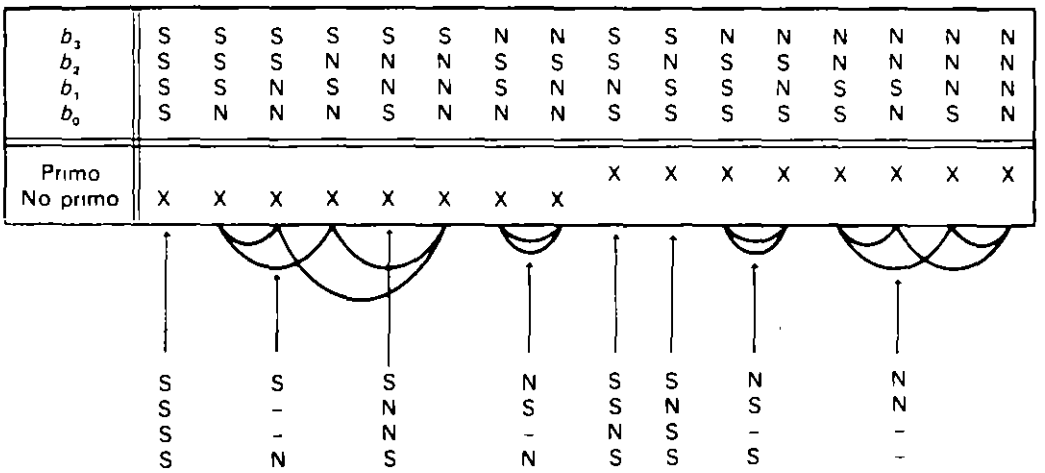
b_3	S	S	S	S	S	S	S	S	N	N	N	N	N	N	N	N
b_2	S	S	S	S	N	N	N	N	S	S	S	S	N	N	N	N
b_1	S	S	N	N	S	S	N	N	S	S	N	N	S	S	N	N
b_0	S	N	S	N	S	N	S	N	S	N	S	N	S	N	S	N
Primo			X		X				X		X		X	X	X	X
No primo	X	X		X		X	X	X		X		X				
	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Como hemos planteado las 16 situaciones posibles con su tratamiento, la tabla obtenida es completa, no tiene redundancias ni contradicciones.

Para facilitar la simplificación agrupamos las reglas por su tratamiento.

b_3	S	S	S	S	S	S	N	N	S	S	N	N	N	N	N	N
b_2	S	S	S	N	N	N	S	S	S	N	S	S	N	N	N	N
b_1	S	S	N	S	N	N	S	N	N	S	S	N	S	S	N	N
b_0	S	N	N	N	S	N	N	N	S	S	S	S	S	N	S	N
Primo									X	X	X	X	X	X	X	X
No primo	X	X	X	X	X	X	X	X								
	15	14	12	10	9	8	6	4	13	11	7	5	3	2	1	0

Simplificación:



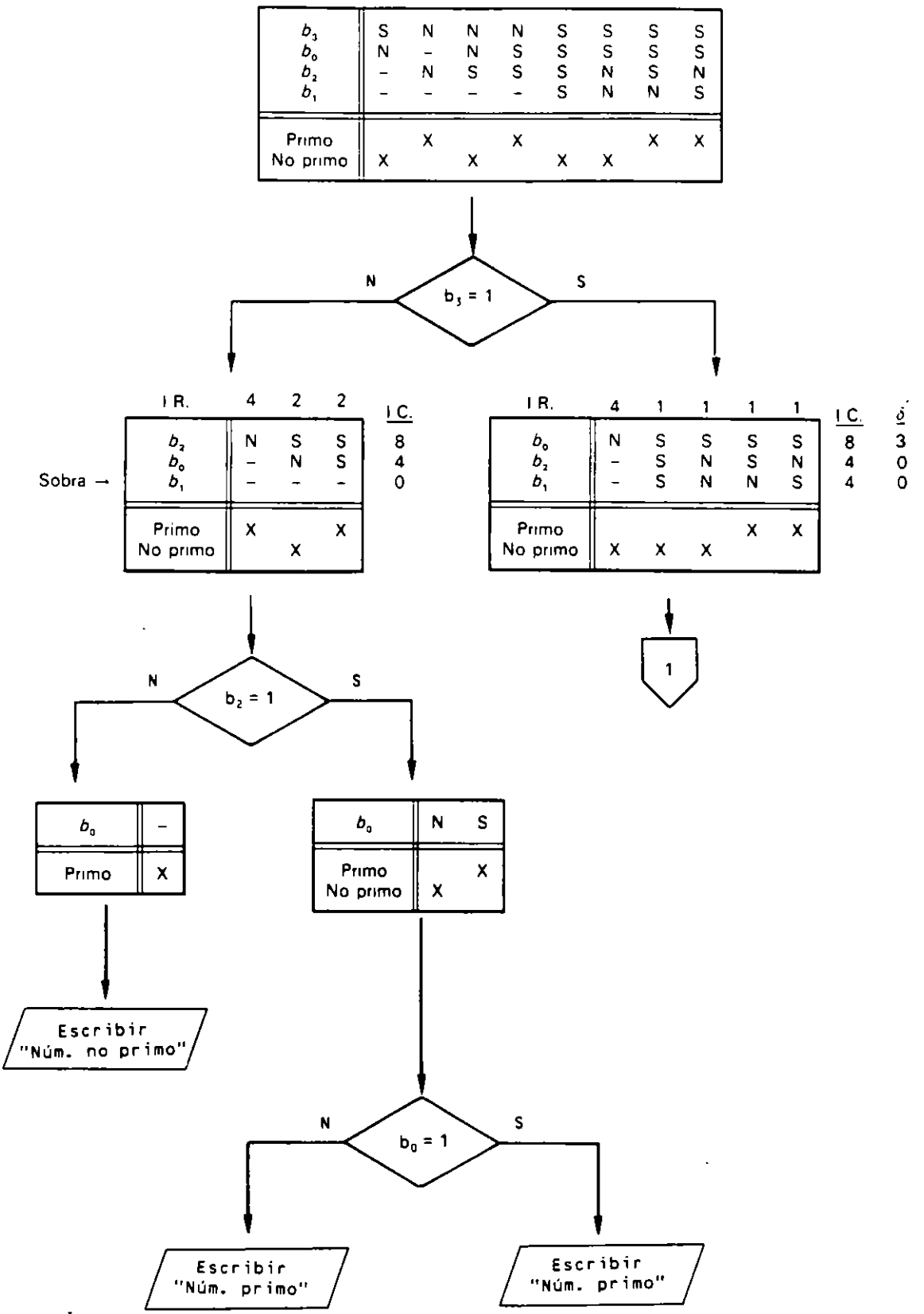
La tabla simplificada queda:

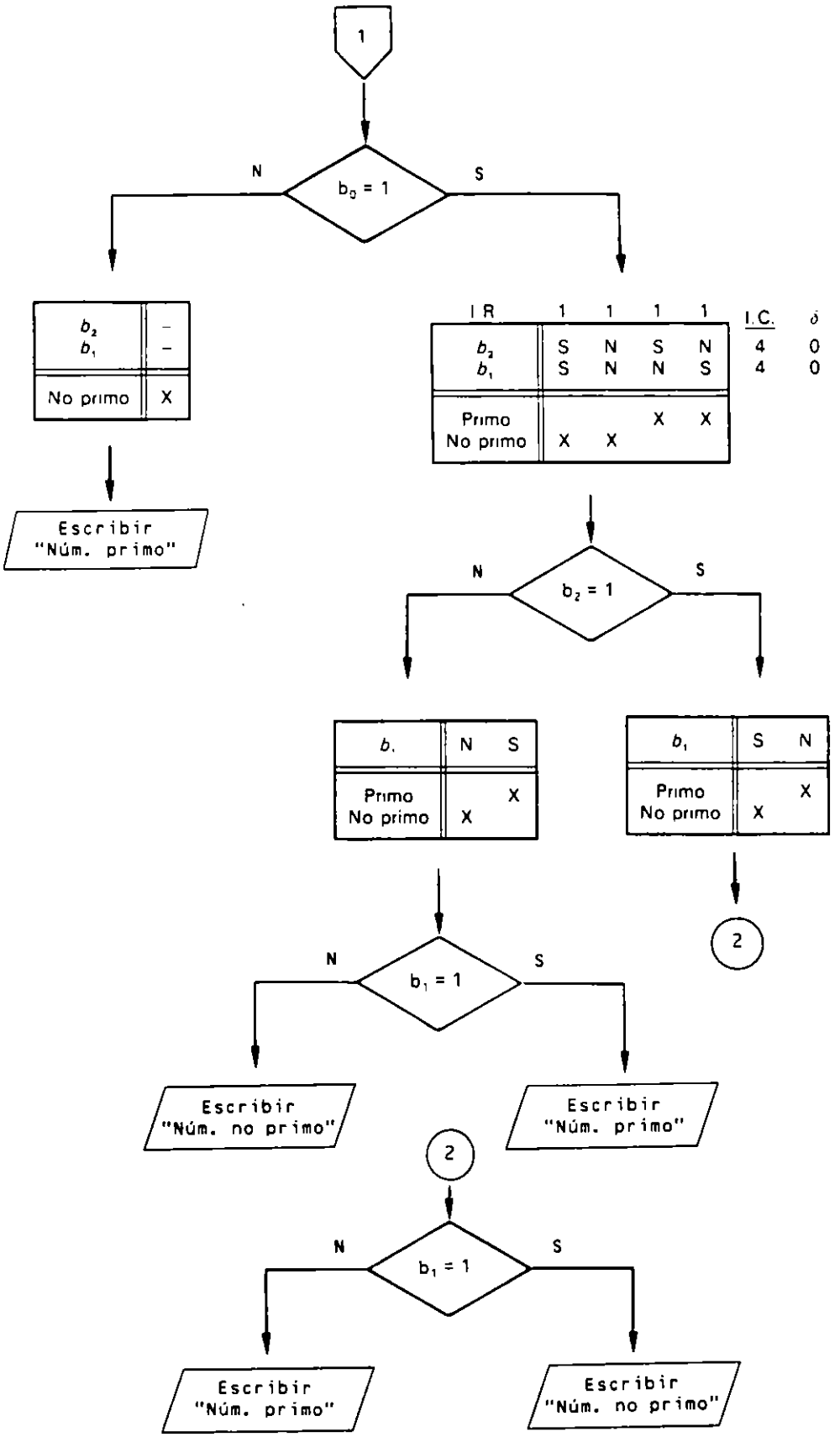
I.R.	1	4	1	2	1	1	2	4	I.C	S
b_3	S	S	S	N	S	S	N	N	16	2
b_2	S	–	N	S	S	N	S	N	12	1
b_1	S	–	N	–	N	S	–	–	4	0
b_0	S	N	S	N	S	S	S	–	12	3
Primo					X	X	X	X		
No primo	X	X	X	X						

La tabla simplificada ordenada por importancias es:

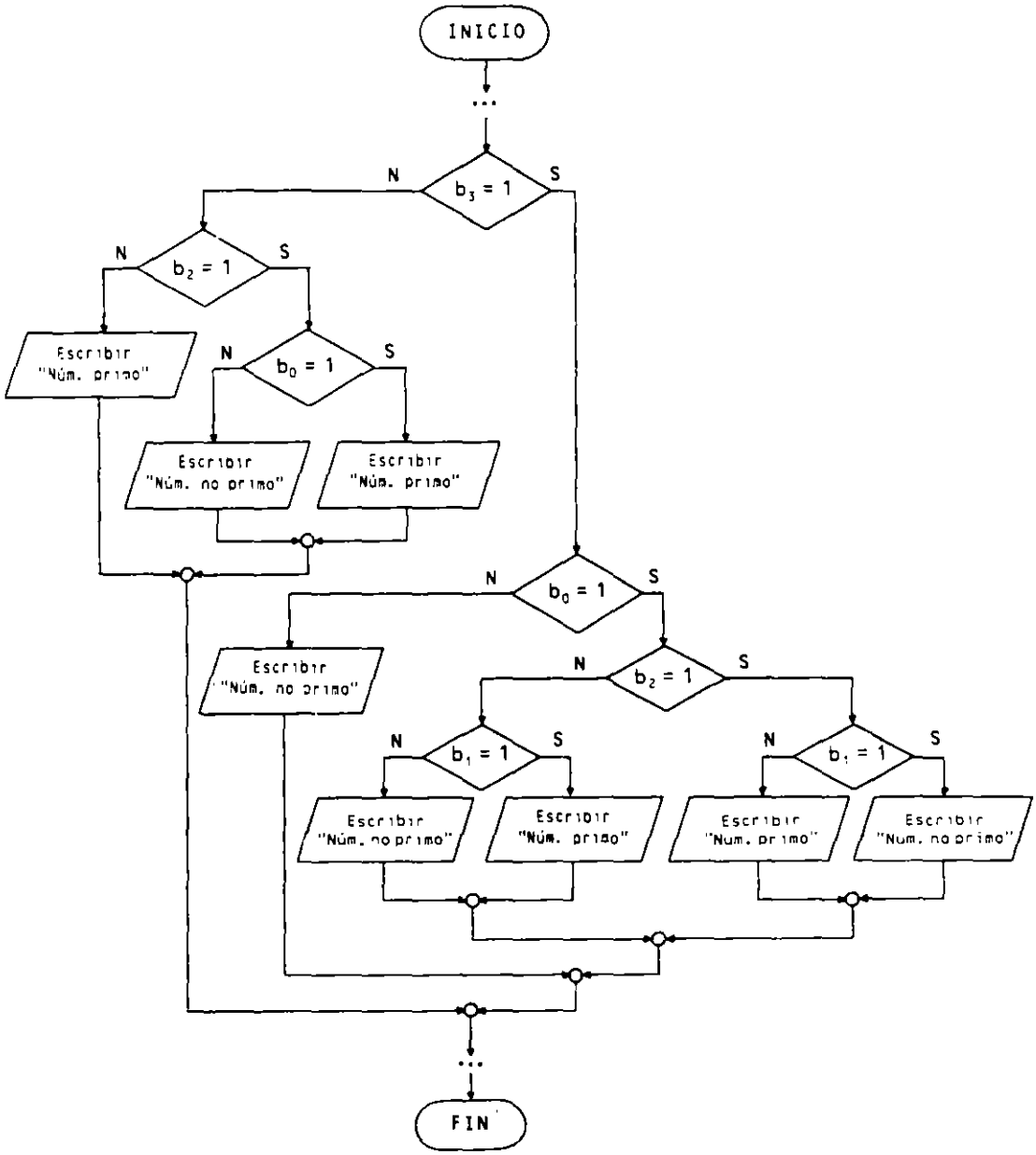
I.R.	4	4	2	2	1	1	1	1	I.C	S
b_3	S	N	N	N	S	S	S	S	16	2
b_0	N	–	N	S	S	S	S	S	12	3
b_2	–	N	S	S	S	N	S	N	12	1
b_1	–	–	–	–	S	N	N	S	4	0
Primo		X		X			X	X		
No primo	X		X		X	X				

Desarrollamos el método de Pollack para la conversión de la tabla en ordinograma.




1
N
$b_0 = 1$
S
b_2 b_1 | –
No primo | X
Escribir "Núm. primo"
I R 1 1 1 1 I.C. δ
b_2 S N S N 4 0
b_1 S N N S 4 0
Primo X X
No primo X X
N
$b_2 = 1$
S
b_2 | N S
Primo | X
No primo | X
b_1 | S N
Primo | X
No primo | X
2
N
$b_1 = 1$
S
Escribir "Núm. no primo"
Escribir "Núm. primo"
2
N
$b_1 = 1$
S
Escribir "Núm. primo"
Escribir "Núm. no primo"

El ordinograma del proceso anterior es:



EJERCICIOS PROPUESTOS

1. Una compañía de seguros tiene establecidas las siguientes tarifas para sus dos tipos de póliza de seguro de automóviles:

   — Tarifa base para seguro obligatorio 5.000
   — Tarifa base para seguro todo riesgo 45.000

Estas tarifas pueden incrementarse o reducirse según la situación de cada cliente:

— Si el vehículo es industrial se aumenta 1.7 veces el precio de la tarifa base.
— Si el asegurado es mayor de 55 años, su vehículo no es industrial y la póliza concertada es de seguro obligatorio se incrementa la tarifa base en un 8.5 %.

Diseñar una T D., resolverla y pasarla a ordinograma.

Si una situación es contradictoria se le aplicara en la T D. un tratamiento de error.

2. En un curso de programación de computadoras se estudian cuatro lenguajes:

— C.
— COBOL.
— Pascal.
— FORTRAN.

La nota final del curso depende de una práctica que se realiza en cada uno de los lenguajes, que se califican con APTO o NO APTO (S/N), según las siguientes normas de puntuación:

— Apto en los cuatro lenguajes, SOBRESALIENTE.
— Apto en COBOL, Pascal y uno de los otros dos, NOTABLE.
— Apto en COBOL y Pascal solamente, BIEN.
— Apto en FORTRAN, C y uno de los otros dos, BIEN.
— Apto en COBOL y uno de entre FORTRAN o C, SUFICIENTE.
— Apto en Pascal y uno entre FORTRAN o C, SUFICIENTE.
— En el resto de situaciones, INSUFICIENTE.

Diseñar, resolver y pasar a ordinograma la T D. correspondiente

3. Una familia consta de cuatro hijos de 9, 8, 7 y 6 años de edad. Los padres les regalan una computadora y un juego donde el factor más importante es el de la edad de los jugadores.

Las normas del juego son:

— Es necesario que la suma de las edades de los que quieran jugar sea mayor que la suma de las edades de los que no quieran.
— En el caso de que la suma de las edades de los que quieran jugar sea igual a la suma de las edades de los que no quieran hacerlo se hará lo que desee el de mayor edad.

Además se imponen las siguientes restricciones:

— No pueden jugar si sólo desea hacerlo uno de los cuatro hijos.
— No pueden jugar si sólo desean hacerlo los hijos de edades impares.

Los datos de entrada son la decisión de cada hijo de querer jugar o no. La salida será la determinación de si jugarán o no (se debe entender que el juego necesita cuatro jugadores).

Diseñar, resolver y pasar a ordinograma la T.D. correspondiente.

CAPITULO 13

Metodologías de desarrollo de programas

13.1. INTRODUCCION

En los capítulos anteriores hemos estudiado una serie de herramientas y estilos de programación que, por una parte, son básicos y necesarios, pero, por otra, podrían ser insuficientes, dadas las necesidades actuales en el desarrollo de software.

El estado actual en el diseño de programas exige no sólo la realización de buenos programas, sino que éstos cada vez puedan hacerse en menos tiempo, estén bien documentados y sean fáciles de mantener.

Los **inconvenientes** más esenciales que se nos presentan son los siguientes:

— Cada programador tiene su propio estilo de programación.
— Los cambios y modificaciones de los programas, es decir, su mantenimiento, resulta a veces complicado.
— Excesivo tiempo en el desarrollo de un programa.

Por todo ello, desde hace unos años han venido sucediéndose una serie de estudios en los que se ha tratado de modelizar el diseño de los programas con el fin de automatizar de alguna forma su desarrollo.

Las **ventajas** que pueden apreciarse con estos nuevos métodos de diseño son:

— Para todos los programadores existe un mismo estilo o forma de desarrollar un programa.
— La documentación que acompaña a un programa es más sencilla de entender por todos.
— Se facilita enormemente el mantenimiento de los programas.
— Aparecen herramientas de desarrollo y codificación automática.

13.2. METODOLOGIA JACKSON

En 1975 se publicó el libro *Principles of Program Design*, donde el autor, *Michael A. Jackson*, describe el método de **Programación Estructurada de Jackson** *(JSP)* como método de diseño de programas con aplicación a la mayoría de los lenguajes de programación.

13.2.1. INTRODUCCION

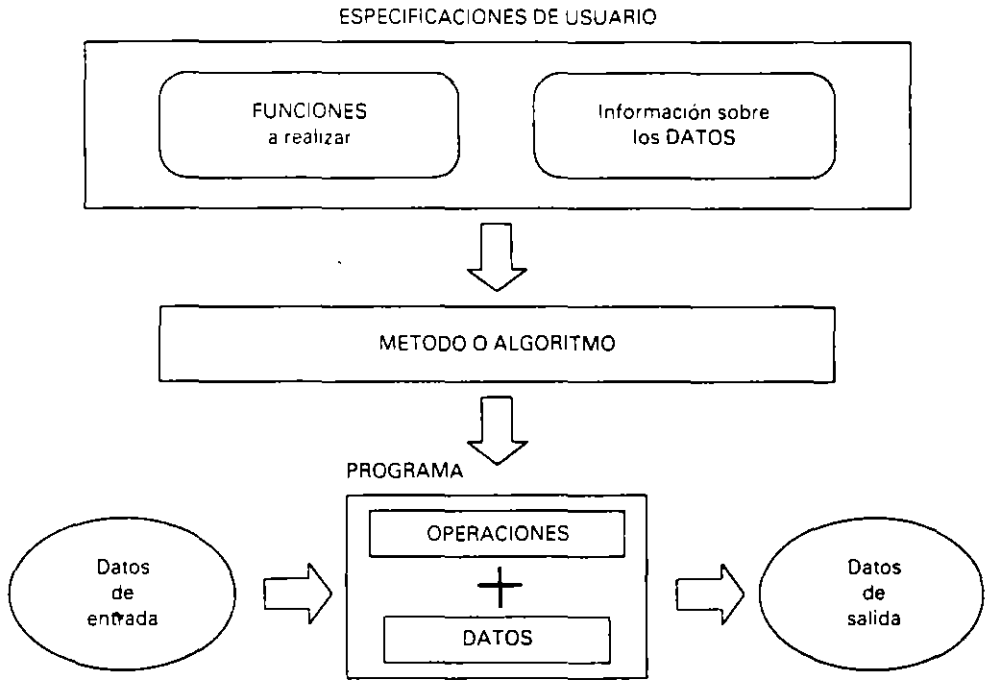
La Metodología de Jackson es un proceso sistemático para la realización de un programa, utilizando la técnica del diseño descendente (*Top-Down*), cuyo producto final es el programa escrito en el pseudocódigo propio del método, de tal forma que su

codificación posterior en cualquier lenguaje de programación es relativamente sencilla, bien manualmente o de forma automática.

Para el diseño de un programa, en general, se parte de unas especificaciones proporcionadas por el usuario en las cuales se definen y detallan:

- Las funciones a realizar y su naturaleza.
- Los datos a manejar y su naturaleza.

El resultado final será un programa o conjunto de programas capaces de resolver el problema solicitado. El siguiente gráfico nos muestra, esquemáticamente, el proceso de creación de un programa.



Los **objetivos** pretendidos en la realización de programas, siguiendo esta metodología, son los siguientes:

- Obtención de programas fiables. Fiabilidad.
- Obtención de programas sencillos. Simplicidad.
- Facilidad en la puesta a punto de los programas.
- Facilidad en el mantenimiento de los programas.

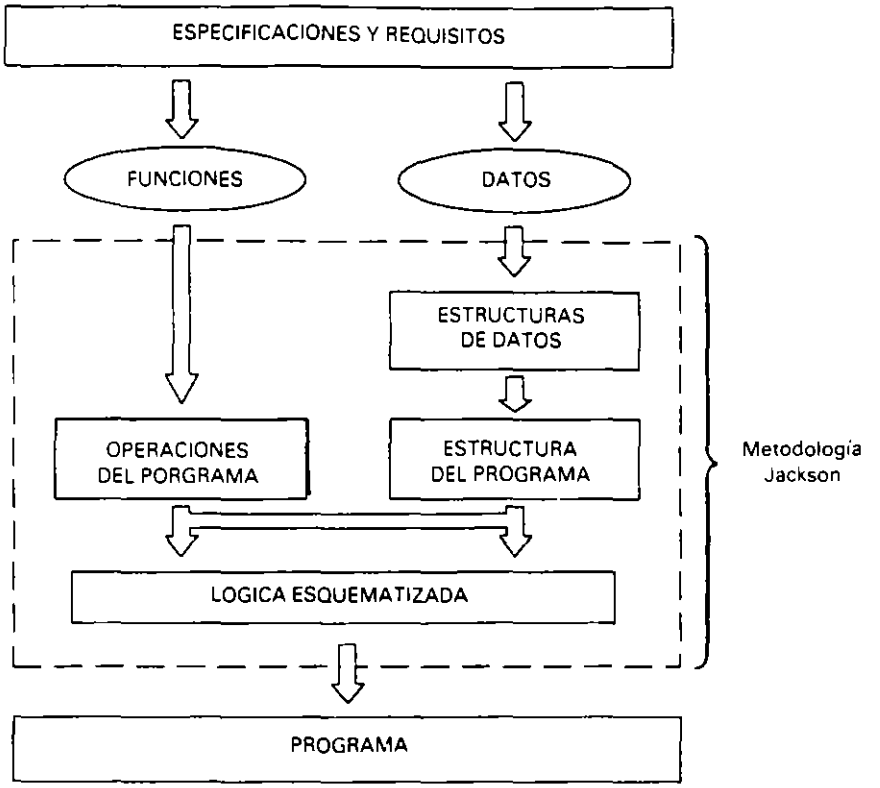
El punto de partida en la Metodología Jackson es el de la definición de los datos que vamos a manejar, tanto en la entrada como en la salida, y como consecuencia del tratamiento requerido en estos datos se obtiene el programa.

Desarrollar la estructura de un programa sobre la estructura de los datos que va a manejar ofrece:

- Una base consistente para la definición del programa.
- Las operaciones se ubicarán en el programa de forma lógica.
- El programa será fiel reflejo del problema a resolver.

Esta metodología permite la realización de los programas siguiendo las directrices de la programación estructurada, puesto que la representación de los datos se basa en las estructuras básicas: secuencial, alternativa y repetitiva, admitiendo además un sencillo diseño descendente y modular.

Los pasos esenciales en la Metodología Jackson se muestran en el siguiente esquema.



Como podemos observar, la determinación de las **funciones** y los **datos** a manejar se obtienen de las especificaciones y requisitos del usuario. Las **estructuras de datos** (tanto de entrada como de salida) se forman a partir de los datos. Mediante determinación de **relaciones** y **correspondencias** entre las distintas estructuras de datos que intervengan se obtiene la **estructura del programa**. De las funciones a realizar se deducen las **operaciones** que serán necesarias en el programa. A la estructura del programa se le **asignan** las operaciones necesarias y el resultado se escribe en una notación propia similar a un pseudocódigo denominado **lógica esquematizada**.

La lógica esquematizada se convertirá posteriormente en un programa codificado en el lenguaje que corresponda de forma manual o utilizando herramientas de conversión que lo harán automáticamente.

13.2.2. FUNDAMENTOS BASICOS. ENTIDADES

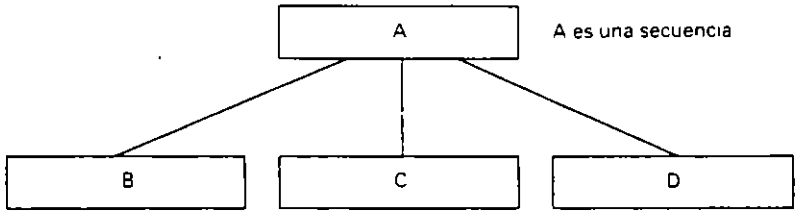
La Metodología Jackson utiliza una representación gráfica basada en el concepto de **entidad**. Una entidad es un elemento simple o compuesto que se representa a sí mismo (simple) o representa a un conjunto que mantiene una cierta relación (compuesto).

Una **entidad simple** se representa por medio de un rectángulo horizontal como el de la siguiente figura:

Las entidades compuestas se utilizan tanto para representar datos (estructuras de datos) como para elementos de programa (estructuras de control).

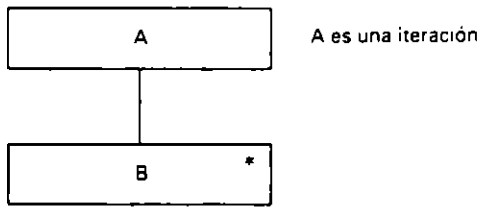
Las relaciones entre las entidades pueden ser de tres tipos:

- **Secuencia.** Una entidad se compone de un conjunto de entidades que mantienen un cierto orden. En la figura, una entidad A se compone de B, seguido de C, seguido de D.



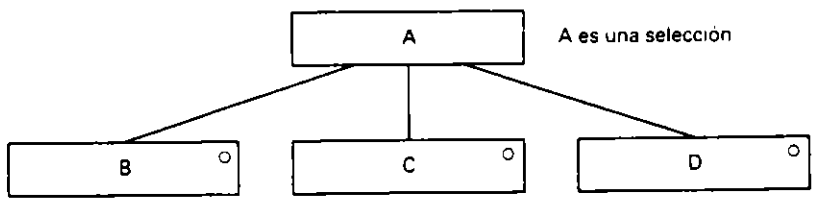
En una secuencia sólo puede haber elementos de secuencia y no de iteración o selección.

- **Iteración.** Una entidad se compone de la repetición de otra un número entero de veces que puede ser 0 o más. En la figura, A se compone de B, 0 veces o más.



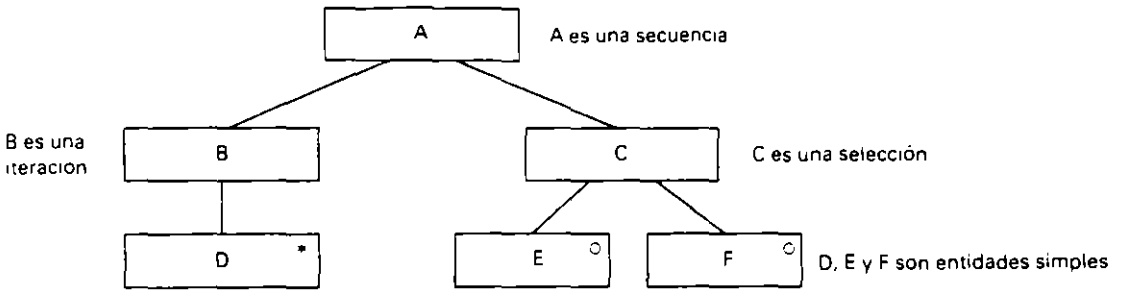
El "*" indica orden de multiplicidad y sólo puede haber un elemento repetitivo.

- **Selección.** Una entidad se compone de una, y sólo una, de entre varias alternativas posibles. En la figura, A está compuesta por B o C o D.



El carácter "o" significa o este o aquel o ..., y sólo puede haber en la selección elementos alternativos.

Ejemplo: *Una entidad A se compone de B y C; por su parte, B es la repetición un número indeterminado de veces de otra entidad D, y C puede ser o E o F. Representar la correspondiente estructura de entidades.*



13.2.3. FASES DE LA METODOLOGIA JACKSON

La Metodología de Jackson desarrolla un programa en cinco fases o pasos que se realizan consecutivamente. Estas fases son las siguientes:

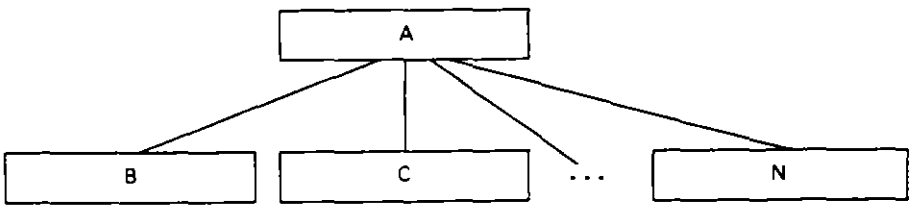
1.ª **Definir las estructuras de datos.** Consiste en una representación jerárquica de los datos definidos en las especificaciones del problema.
2.ª **Encontrar correspondencias entre las estructuras de datos.** Determinar qué entidades se pueden procesar simultáneamente, es decir, qué elementos ocurren el mismo número de veces y en la misma secuencia.
3.ª **Formar la estructura del programa.** Trata de reunir todas las estructuras de datos basándose en las correspondencias obtenidas para configurar la estructura del programa.
4.ª **Listar y asignar las operaciones y condiciones a realizar.** Consiste en listar las operaciones a realizar asignándolas a la estructura del programa y determinar las condiciones de alternativas y repeticiones.
5.ª **Escribir la lógica esquematizada.** Descripción del programa en el pseudocódigo propio de la metodología a partir del árbol obtenido en la fase anterior.

Veamos cada una de las fases con más detalle:

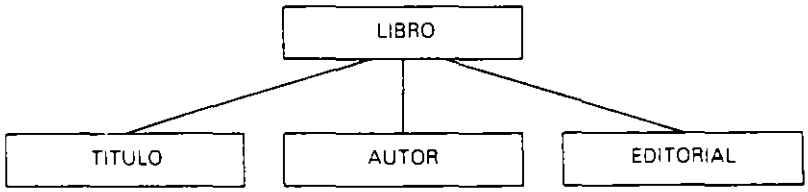
- **Definir las estructuras de datos. (1.ª fase).**

Consiste en analizar las especificaciones del problema y representar las entidades de datos que intervengan tanto en la entrada como en la salida.

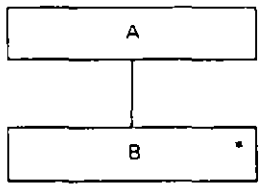
— Si una información o dato se compone de una secuencia de otros se representará:



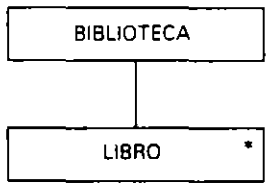
Ejemplo: *La información LIBRO se compone de TITULO más el AUTOR más la EDITORIAL. Se representará por:*



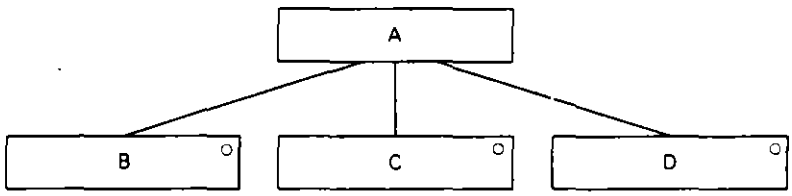
— Si una información o dato se compone de una repetición de otra un número de veces 0 o más, tendremos:



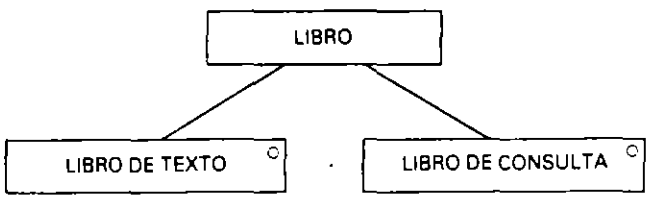
Ejemplo: *Una BIBLIOTECA se compone de una serie de libros, es decir, se repite la entidad LIBRO un número indeterminado de veces. Lo representaremos de la siguiente manera:*



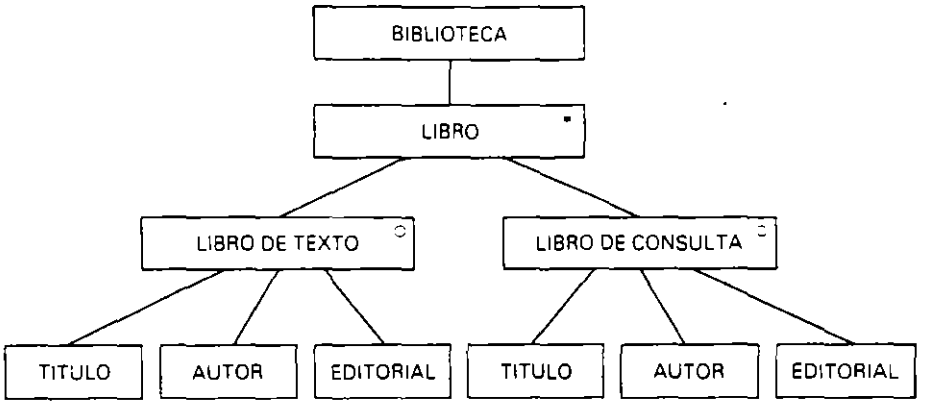
— Si una información o dato puede ser uno determinado entre varios, la representación será:



Ejemplo: *Un LIBRO de una biblioteca puede ser LIBRO DE TEXTO o LIBRO DE CONSULTA. Se representará:*

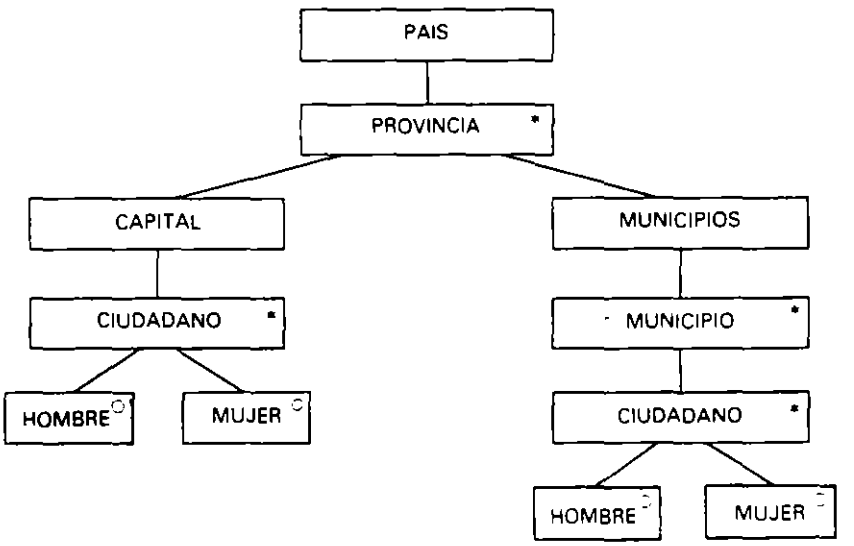


Ejemplo: *Supongamos una biblioteca con libros. Cada libro puede ser libro de texto o libro de consulta, y, además, los datos de cada uno de ellos son, el título, el autor y la editorial. Representar la estructura de datos de la biblioteca.*

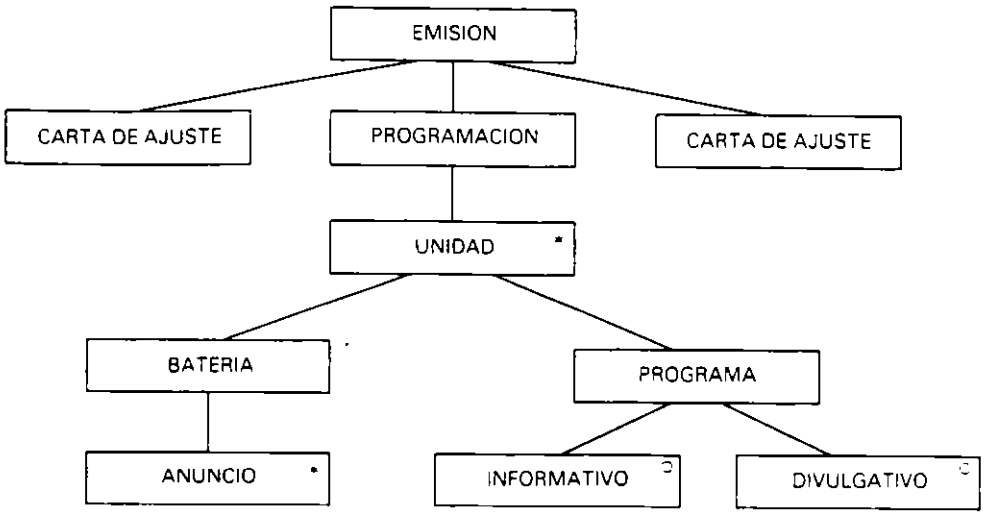


■ Ejemplos de representación de estructuras de datos

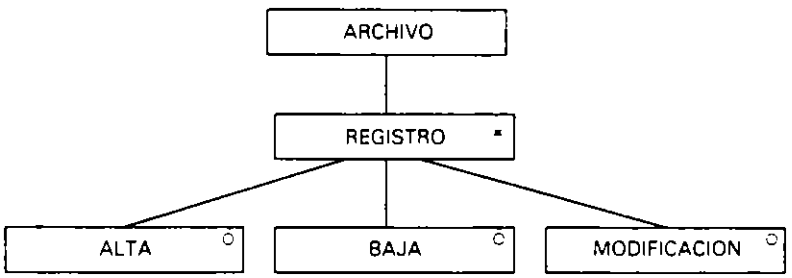
1. *Representar la estructura de los ciudadanos de un país donde cada provincia tiene una capital y un conjunto de municipios en los que, además, cada ciudadano puede ser hombre o mujer.*



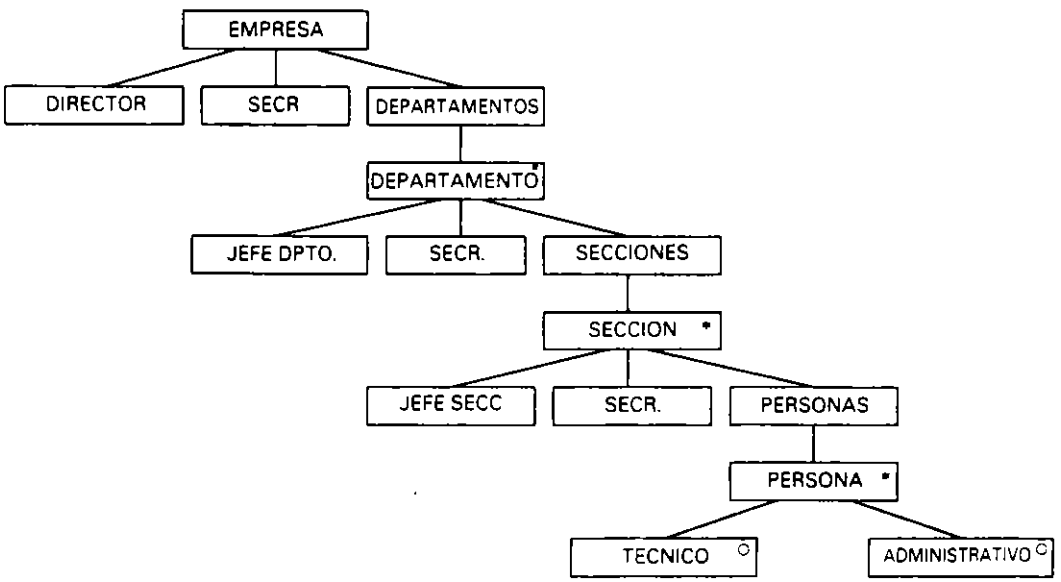
2. *La emisión diaria de una cadena de televisión, en un día, empieza y termina con una carta de ajuste. Entre ellas se emiten una serie de programas que pueden ser informativos o divulgativos y van precedidos todos ellos por una batería de anuncios publicitarios. Representar la emisión de un día.*



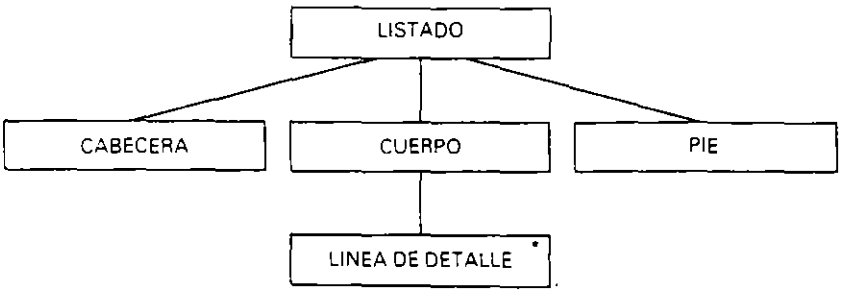
3. *Un archivo de movimientos tiene registros de tres tipos: altas, bajas y modificaciones. Representar su estructura.*



4. *Una empresa tiene un director general y está dividida en departamentos, cada departamento tiene un jefe, y se divide en secciones, cada una de las cuales posee un jefe de sección y un conjunto de personas que pueden ser técnicos o administrativos. Además, el director y todos los jefes de departamentos o secciones tienen una secretaria. Representar la estructura del personal de la empresa.*



5. *Un listado que se produce en un proceso tiene en primer lugar una cabecera, a continuación un conjunto de líneas de detalle, en lo que se denomina cuerpo del listado, y por último un pie de listado donde aparecen datos globalizados. Representar su estructura.*



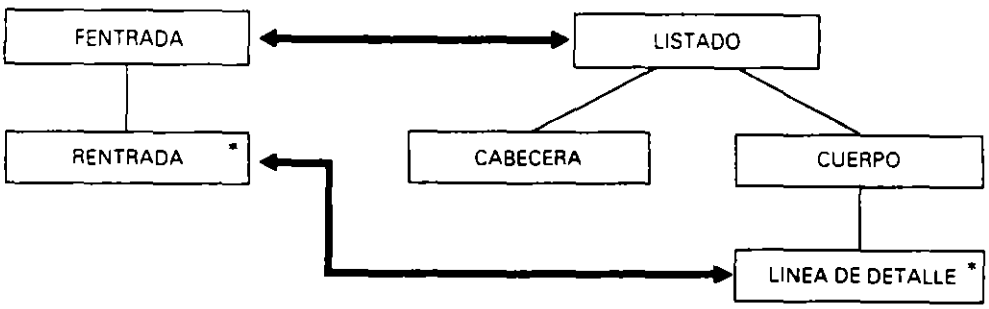
- **Encontrar correspondencias entre las estructuras de datos (2.ª fase).**

Representadas todas las estructuras de datos que intervienen en un proceso se buscan las posibles correspondencias que existan entre entidades de dos estructuras distintas.

Para que exista correspondencia entre dos entidades de distintas estructuras de datos se tienen que cumplir las siguientes condiciones:

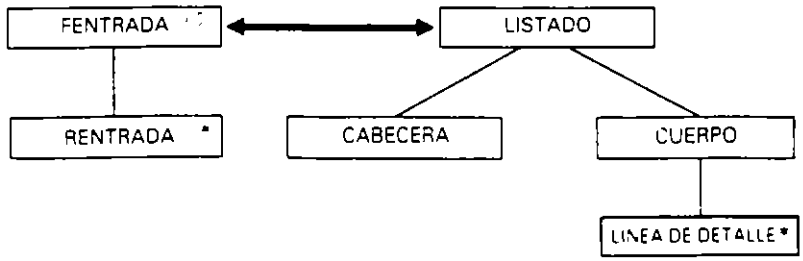
— Ambas entidades deben ocurrir el mismo número de veces.
— Deben poder procesarse cada pareja de entidades (una de cada estructura) al mismo tiempo, es decir, se pueden procesar en paralelo y alternativamente las ocurrencias de dichas entidades.

Ejemplo: *Supongamos un programa que maneja un archivo de entrada, FENTRADA, con información en sus registros RENTRADA de un conjunto de personas, y necesitamos obtener un listado con una determinada cabecera y en cada línea de detalle datos elaborados sobre las mencionadas personas. Las estructuras de datos y las correspondencias son:*



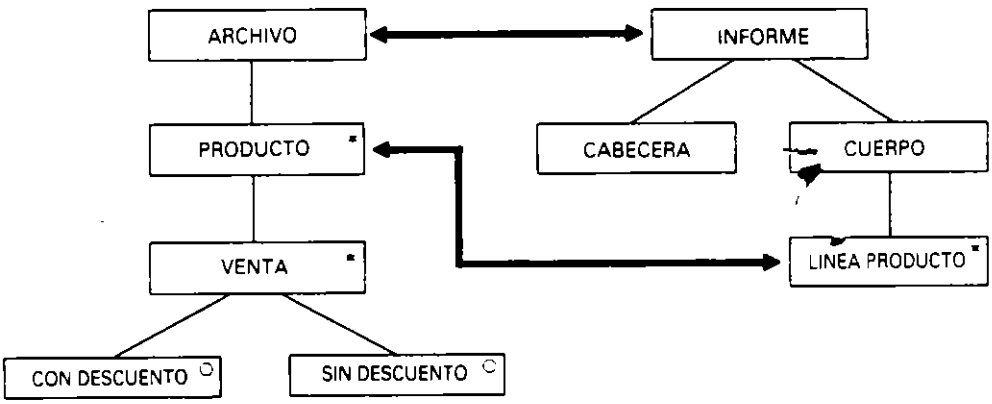
Obsérvese que entre el archivo y el listado existe correspondencia porque ambos se presentan una vez, y al procesar la única ocurrencia del archivo se procesa también el listado. De igual forma, entre el registro y la línea de detalle existe correspondencia puesto que aparece una línea por cada persona y se pueden procesar simultáneamente, es decir, cada registro leído nos permite obtener cada línea de detalle.

Ejemplo: *Supongamos la misma situación del ejemplo anterior en la que el archivo se encuentra desordenado y se desea obtener el listado con la información ordenada según alguno de sus datos. Representar las estructuras de datos y encontrar las correspondencias.*



Obsérvese que la correspondencia entre el archivo y el listado se mantiene, pero ya no existe correspondencia entre los registros y las líneas de detalle, puesto que aunque sí aparecen el mismo número de veces, no se pueden procesar en paralelo.

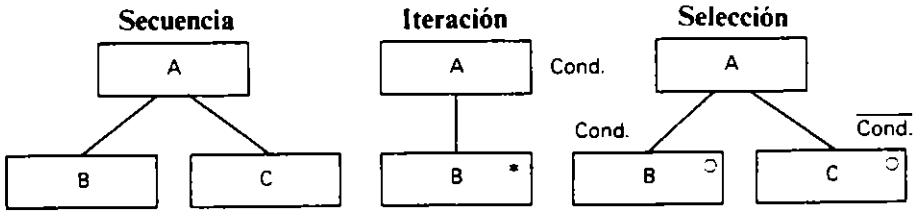
Ejemplo: *Se dispone de un archivo con registros sobre las ventas realizadas de los productos de una empresa durante un periodo de tiempo. Los registros se encuentran clasificados por código del producto (que es un dato del registro), y pueden existir varias ventas de un mismo producto (estarán de forma consecutiva en el archivo). Las ventas, además, pueden ser de dos tipos: con o sin descuento. Se quiere obtener, a partir del archivo anterior, un informe compuesto por una cabecera y una línea de detalle por cada producto. Representar las estructuras de datos y encontrar las correspondencias.*



- **Formar la estructura del programa (3.ª fase).**

Se trata de construir la estructura del programa a partir de las estructuras de datos y sus elementos correspondientes y no correspondientes.

Para ello se utiliza la representación ya estudiada de entidades donde, en lugar de aparecer entidades de datos, aparecerán entidades de programa, o lo que es lo mismo, estructuras de control.



Por cada entidad de datos que exista, el programa debe contener una entidad que la contenga. Por ello:

- Si un programa trata una sola estructura de datos, la estructura del programa coincide con ella.
- Si un programa procesa más de una estructura de datos, su estructura será una combinación de las mismas.
- Todas las estructuras de datos deben estar totalmente contenidas en la estructura del programa, sin que sean alteradas las relaciones entre las entidades de cada una de ellas.

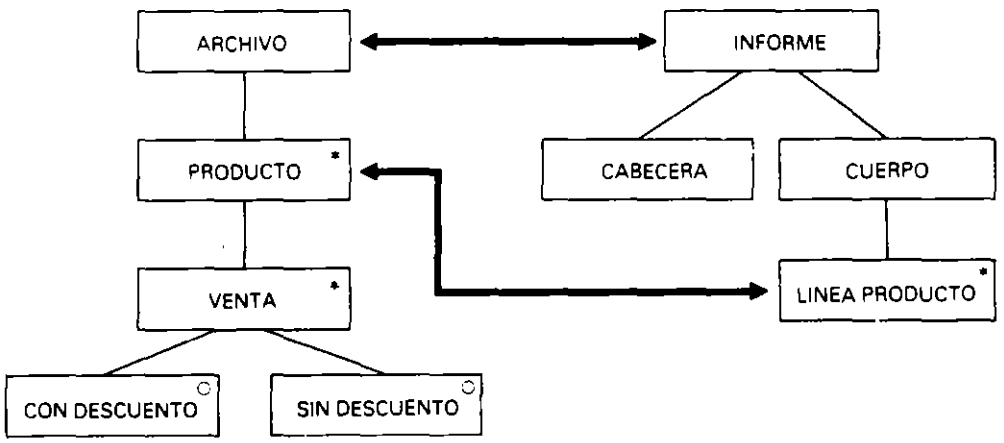
Las reglas de construcción de la estructura del programa son:

Regla 1. Se crea una componente del programa por cada correspondencia encontrada entre las entidades de las estructuras de datos, manteniendo las relaciones jerárquicas que existan entre ellas.

Regla 2. Añadir una componente de proceso a la estructura del programa por cada entidad de datos que no posea correspondencia en las estructuras de datos de entrada, manteniendo la relación entre sus elementos.

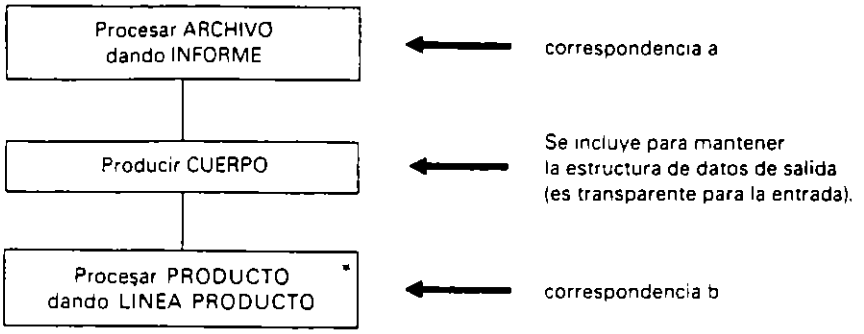
Regla 3. Añadir una componente de proceso a la estructura del programa por cada entidad de datos que no posea correspondencia en las estructuras de datos de salida, manteniendo sus relaciones y asegurando que los datos necesarios para producir la salida se encuentran presentes.

Ejemplo: *Basándonos en el ejercicio anterior, donde las estructuras de datos y las correspondencias son las siguientes:*

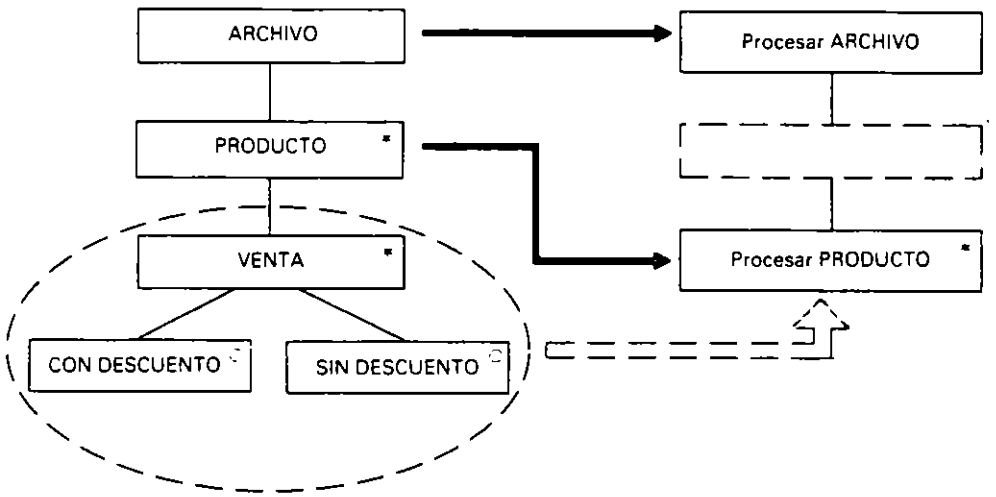


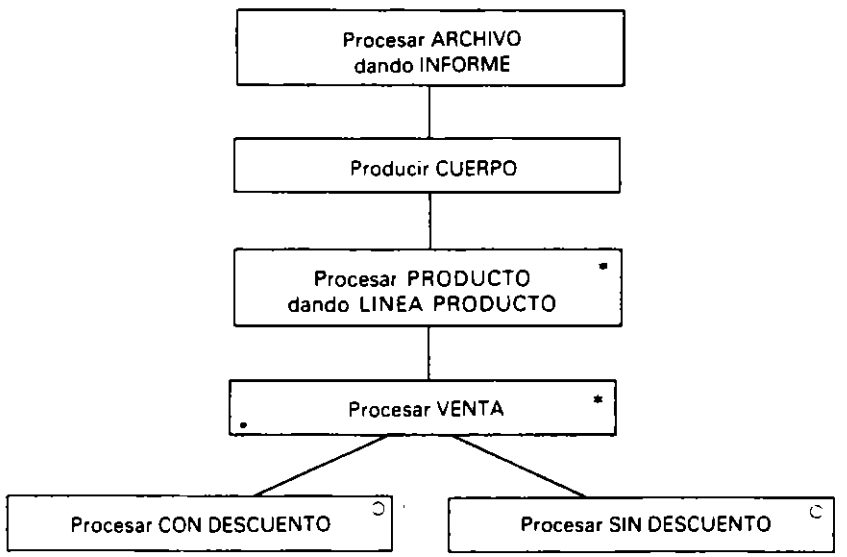
Construir la estructura del programa:

Regla 1. Se crea una componente de programa a partir de cada correspondencia, donde se escribe la palabra **Procesar** para las entidades de entrada y **dando** para las de salida. Además, si existe alguna entidad intermedia entre las que son correspondientes, tanto en la entrada como en la salida, también se lleva a la estructura del programa para mantener la jerarquía entre las que son correspondientes. Estas últimas entidades llevan la palabra **Procesar** si son de entrada y **Producir** si son de salida. En el ejemplo, el resultado de este paso es:

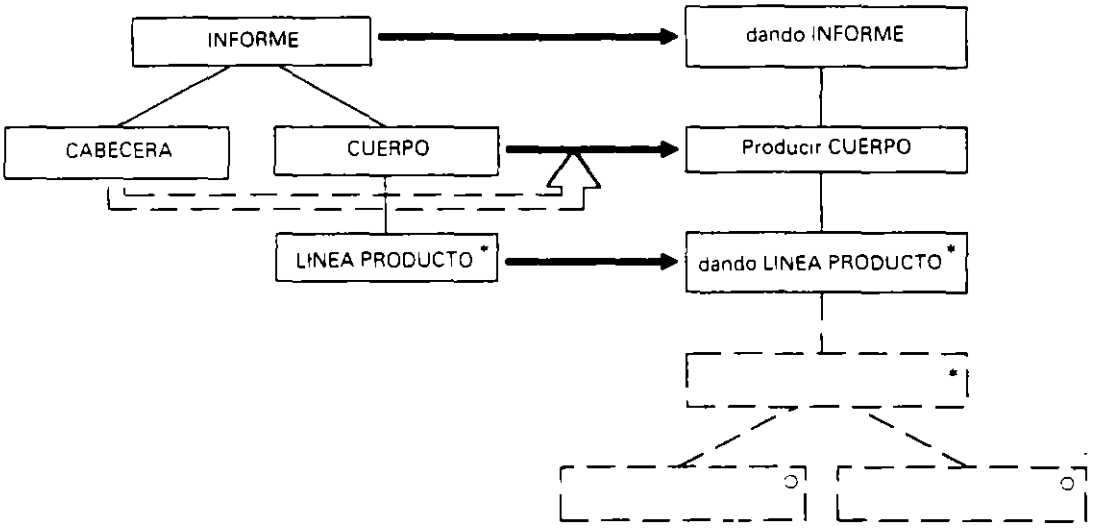


Regla 2. Se añade a la estructura del programa el resto de entidades de la estructura de datos de entrada conformando entidades de programa.

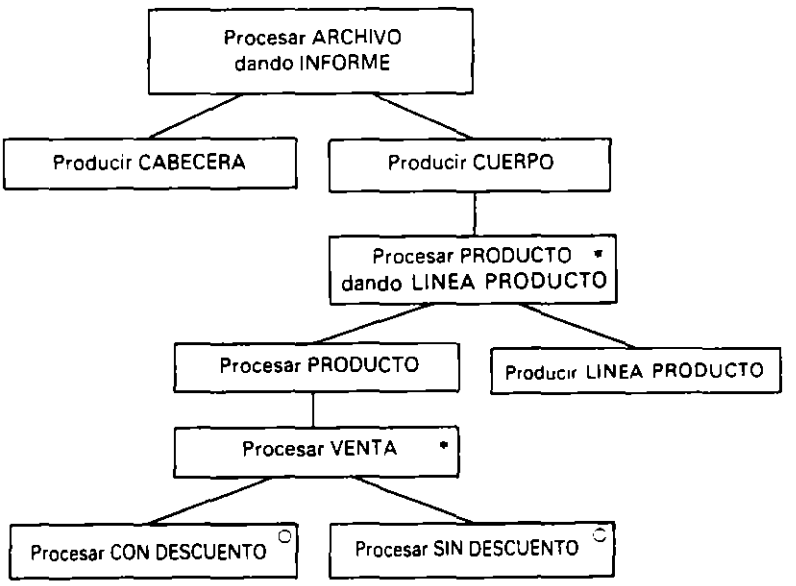




Regla 3. Se añade a la estructura del programa el resto de entidades de la estructura de datos de salida, configurando entidades de programa, manteniendo la jerarquía y asegurando que los datos necesarios para la salida se encuentran presentes.



Si suponemos que para producir la línea de detalle es necesario tener procesado al completo cada producto, para asegurar que los datos de salida están presentes en su momento, será necesario secuenciar el procesar PRODUCTO y dar LINEA.



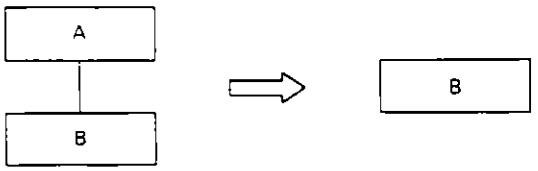
Esta última representación es la estructura del programa, en la que aparecen todas las estructuras de control y falta la inclusión de operaciones y condiciones que se incorporan en la fase siguiente.

En general se debe **verificar la estructura del programa**, una vez construida, mediante un proceso inverso al de su construcción. Partiendo de la estructura del programa se

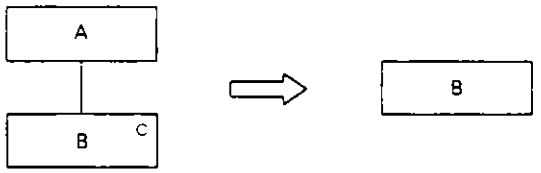
tienen que obtener las distintas estructuras de datos de la primera fase. El proceso es el siguiente:

— Para cada estructura de datos se hacen desaparecer las componentes de la estructura del programa que no están en la estructura de datos.
— La estructura resultante se reduce hasta obtener la estructura de datos teniendo en cuenta que:

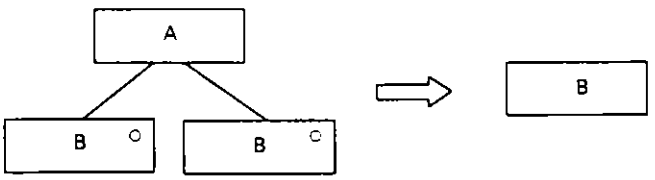
- Una secuencia de un elemento se reduce a él mismo



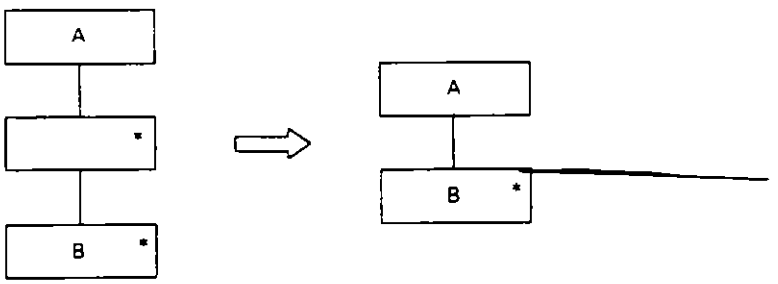
- Una selección de una sola opción es ella misma.



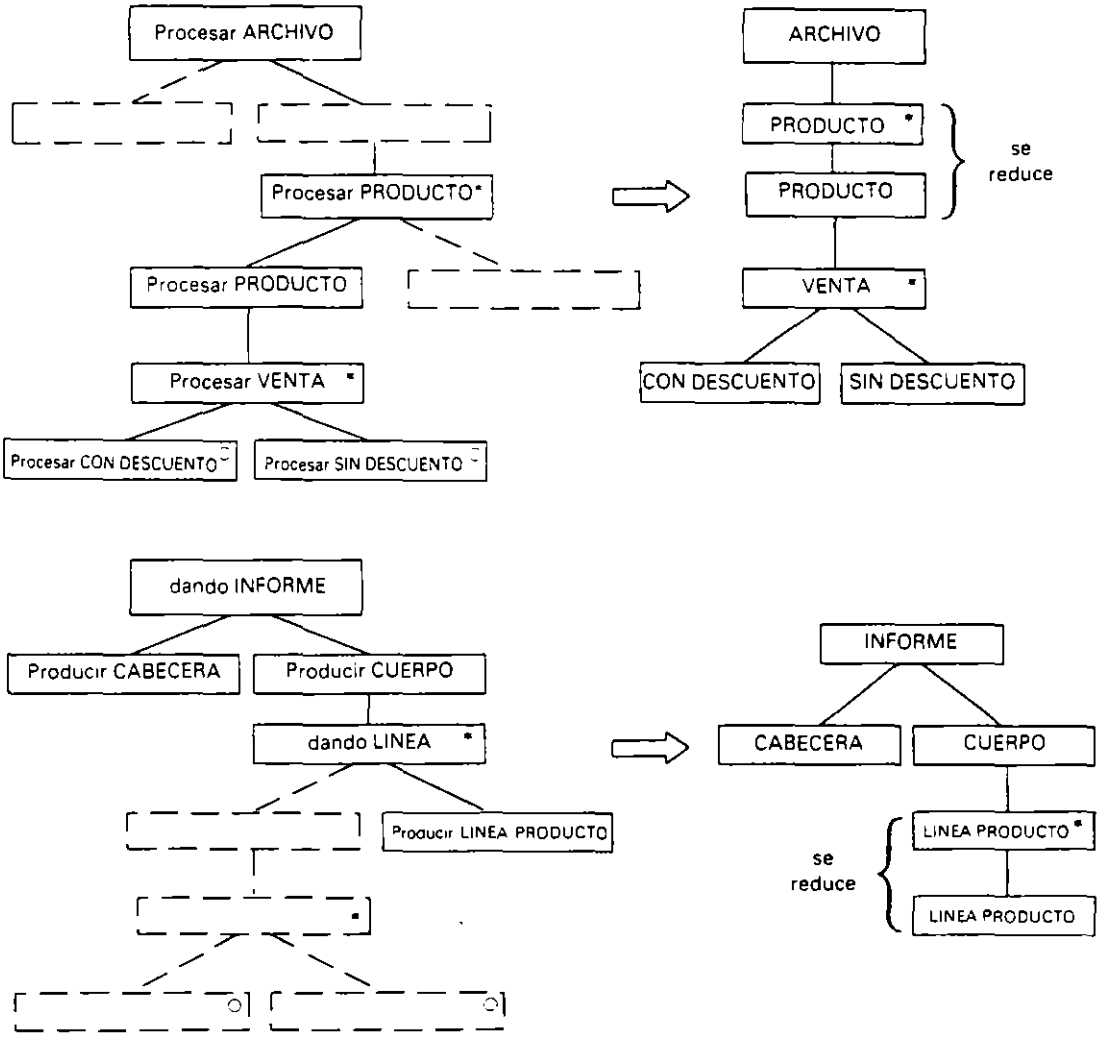
- Una selección con varias ramas iguales se reduce a una de ellas.



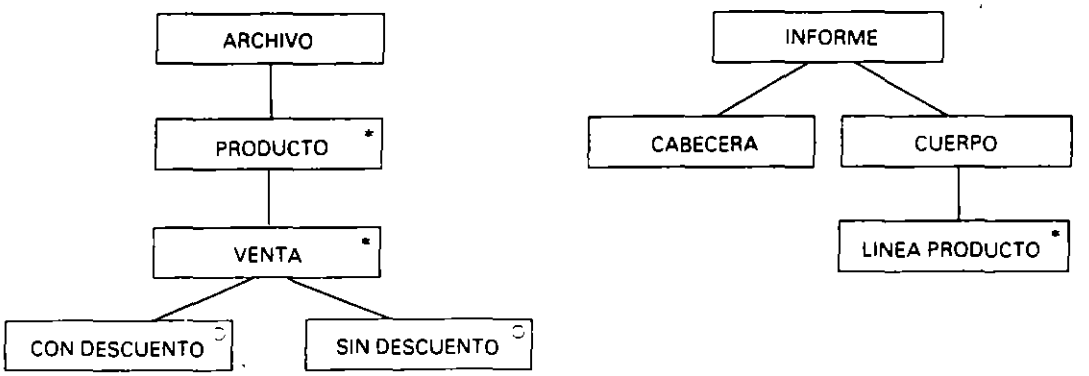
- Una repetición de repeticiones se reduce a una única repetición.



Ejemplo: *Verificamos la estructura del programa del ejercicio anterior.*



Como puede verse, se obtienen de la estructura del programa las estructuras de datos originales, por tanto **queda validada la estructura del programa.**



- **Listar y asignar las operaciones y condiciones a realizar (4.ª fase).**

En esta fase se incorpora a cada estructura del programa que configure una selección o una iteración su correspondiente condición, que será obtenida de las especificaciones del problema.

Por otra parte, se listan las operaciones elementales (instrucciones primitivas) que van a constituir el proceso, y, por último, se asignan a la estructura del programa en el lugar que les corresponda. Una operación puede estar asignada en más de un lugar dentro del programa.

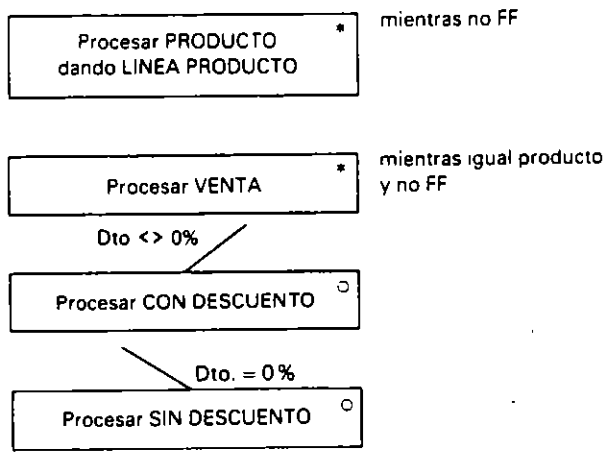
El método más utilizados para listar operaciones es el de confeccionar una **lista base** (*Checklist*), donde aparecen operaciones en grupos, y, a partir de esta lista general, se confecciona la particular de cada proceso. El *checklist* puede ser confeccionado a gusto de cada diseñador.

El *checklist* que utilizaremos en este tema para los ejercicios es el siguiente:

1. Inicio y fin.
2. Apertura de archivos.
3. Cierre de archivos.
4. Inicializaciones.
5. Acumulaciones.
6. Contabilizaciones.
7. Operaciones en general.
8. Lectura y escritura de registros.
9. Confección de listados.
10. ...

Ejemplo: *En el ejercicio anterior supongamos que en cada registro tenemos el código del producto, el importe de la venta y el descuento, y, por otra parte, las líneas de detalle del listado nos indican el código de la pieza y el total vendido de la misma.*

Las condiciones de este proceso son las siguientes:



La lista de operaciones es:

1.1 Inicio.
1.2 Fin.
2.1 Abrir ARCHIVO.
3.1 Cerrar ARCHIVO.
4.1 Inicializar ACUMULADOR.

5.1 Acumular VENTA.
7.1 Retener CODIGO.
7.2 Calcular DESCUENTO.
7.3 Restar DESCUENTO de VENTA.
8.1 Leer REGISTRO.
9.1 Escribir CABECERA.
9.2 Escribir LINEA PRODUCTO.

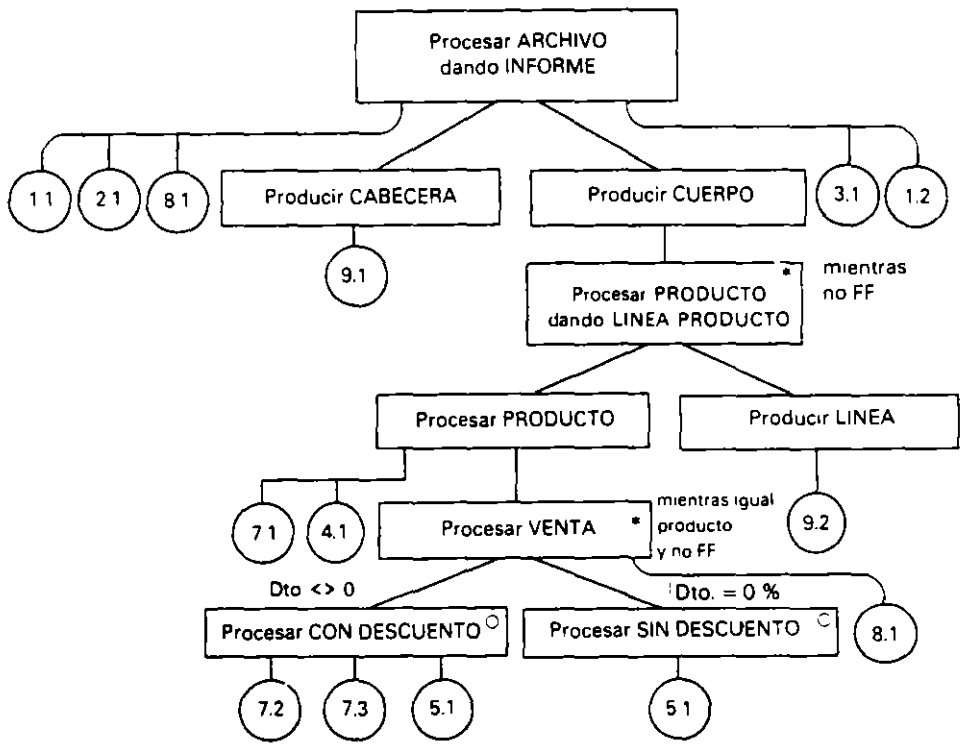
Como puede verse, la lista de operaciones se hace por grupos, numerando las operaciones dentro de cada uno de ellos.

Número grupo. Número de operación en el grupo

La lectura de registros la hacemos por el método de lectura adelantada, lo que supone:

— Leer el primer registro al abrir el archivo.
— Leer el siguiente registro tras el proceso del anterior.

La asignación de operaciones a la estructura del programa se hace por lógica, que se obtiene del propio proceso.

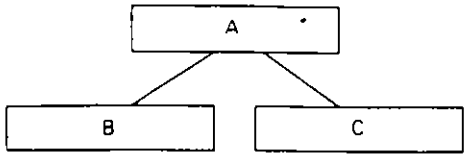


- **Escribir la lógica esquematizada (5.ª fase).**

Esta fase se basa en la escritura de la estructura del programa más las operaciones en un pseudocódigo propio e independiente de los lenguajes de programación. Para ello se recorre el árbol del programa en preorden, es decir, de arriba abajo y de izquierda a derecha.

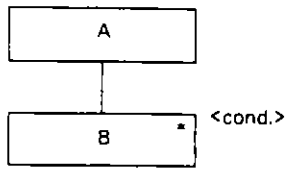
Las transformaciones a realizar son:

- Secuencia:



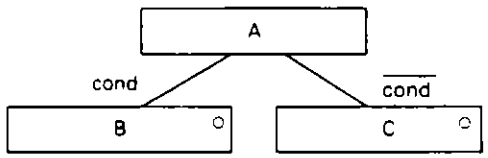

```
A  sec
   B
   C
A  end
```

- Iteración:




```
A  iter <cond.>
   B
A  end
```

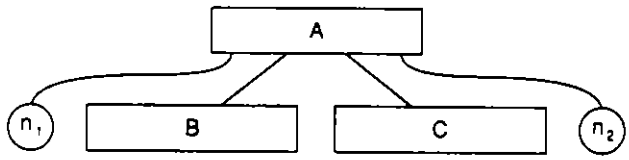
- Selección:




```
A  sel <cond >
   B
A  or <cond >
   C
A  end
```

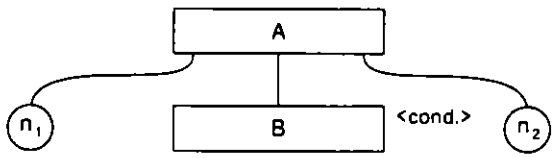
A cada una de estas estructuras, además, se añaden las operaciones asignadas a cada entidad de programa.

- Secuencia:



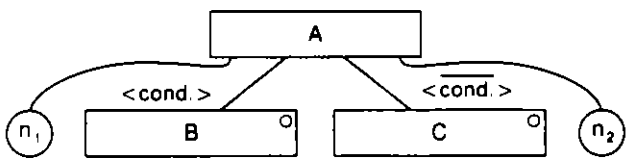

```
A  sec
   n1
   B
   C
   n2
A  end
```

- Iteración:



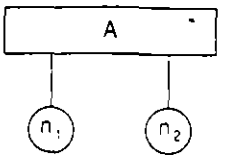

```
n1
A  iter <cond >
   B
A  end
n2
```

- Selección:




```
n1
A  sel <cond >
   B
A  or <cond >
   C
A  end
n2
```

Por último, si a una entidad simple se le asignan operaciones, su transformación es:



A sec
n_1
n_2
A end

Ejemplo: *La lógica esquematizada del ejercicio anterior es:*

Se realiza la lectura del árbol obtenido en la fase 4, obteniendo:

```
PARCHIVOINFORME sec
  Inicio
  Abrir ARCHIVO
  Leer REGISTRO
  PCABECERA sec
    Escribir CABECERA
  PCABECERA end
  PCUERPO iter mientras no FF
     PPRODUCTOLINEAPRODUCTO sec
       Retener CODIGO
       Inicializar ACUMULADOR
       PPRODUCTO iter mientras igual producto y no FF
         PVENTA sel Dto <> 0%
           PCONDESCUENTO sec
             Calcular DESCUENTO
             Restar DESCUENTO de VENTA
             Acumular VENTA
           PCONDESCUENTO end
         PVENTA or Dto = 0%
           PSINDESCUENTO sec
             Acumular VENTA
           PSINDESCUENTO end
         PVENTA end
         Leer REGISTRO
       PPRODUCTO end
       PLINEAPRODUCTO sec
         Escribir LINEAPRODUCTO
       PLINEAPRODUCTO end
     PPRODUCTOLINEAPRODUCTO
  PCUERPO end
  Cerrar ARCHIVO
  Fin
PARCHIVOINFORME end
```

Obsérvese que el nombre asignado a cada entidad de programa es el conjunto de sus letras mayúsculas sin espacios intermedios.

13.2.4. ERRORES E INVALIDEZ

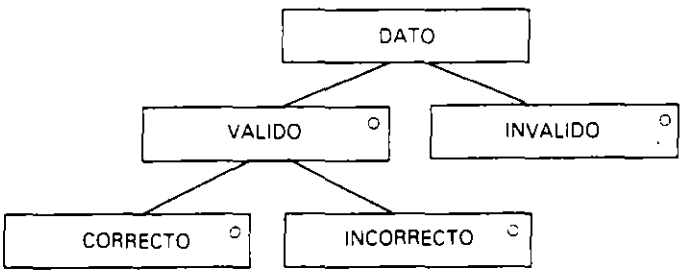
Un programa debe saber reconocer datos erróneos o datos inválidos, puesto que, si éstos se presentan, los resultados obtenidos no serán los deseados.

Un dato es erróneo cuando su naturaleza no es la esperada, es decir, necesitamos un dato numérico y aparece uno alfanumérico. Por otra parte, un dato es inválido cuando

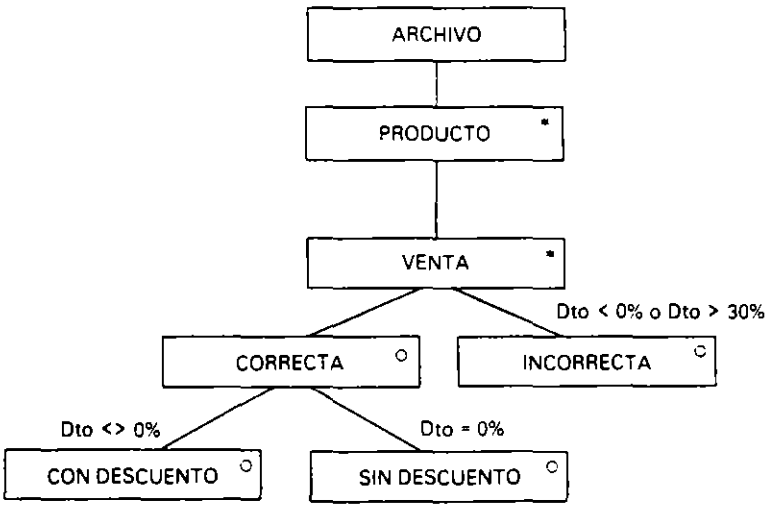
el programa no tiene operaciones previstas para tratarlo. Por ejemplo, si vamos a calcular si un número es par o impar y nos aparece el número 2 5.

Por tanto, cuando se quiere evitar en un programa la aparición de este tipo de datos y su proceso incorrecto, se deben incluir en las especificaciones del problema los controles y mensajes para tratarlos, y todas las situaciones que se puedan presentar se deberán incluir en las estructuras de datos para que aparezcan en la estructura del programa las entidades correspondientes, donde podremos asignar las operaciones para el tratamiento de dichos datos incorrectos.

En la siguiente figura se puede ver cómo quedará la estructura de un dato para controlar su validez.



Supongamos en el ejercicio anterior que queremos controlar en el programa los errores en el descuento, donde sabemos que éste puede oscilar entre el 0 y el 30 %.



13.2.5. COLLATING

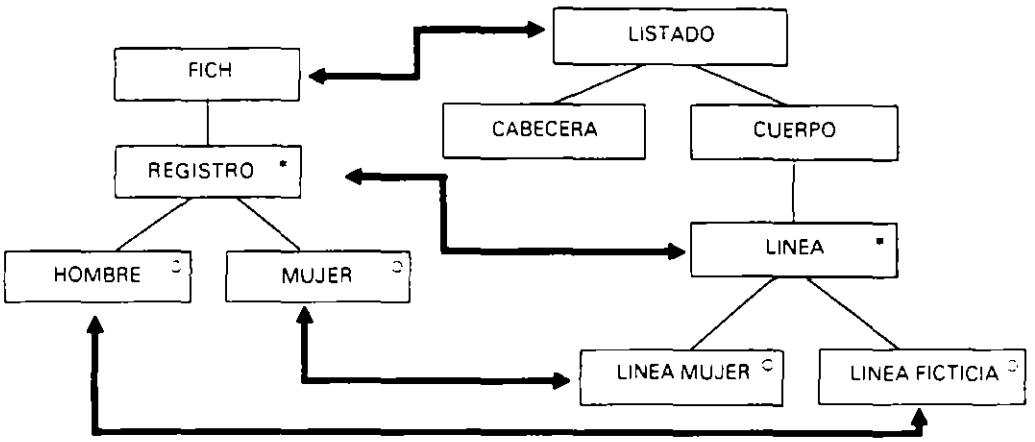
En aquellos programas, donde tanto entre las entradas como entre la entrada y salida, las estructuras de datos tengan correspondencias entre elementos de forma no completa, es decir, se pueden procesar en paralelo de forma sincronizada, pero no tienen igual número de elementos, se puede utilizar la técnica del collating (intercalación). Consiste en imaginar que existen elementos ficticios para igualar el número de elementos entre las

distintas estructuras de datos y así conseguir las correspondencias necesarias para la realización del tratamiento sincronizado de las estructuras.

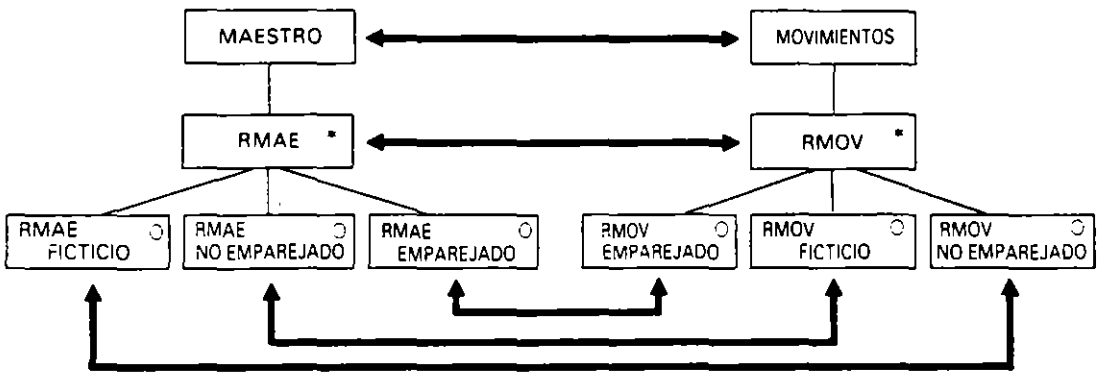
Se utilizan en estos casos los conceptos de entidad emparejada y entidad no emparejada.

Ejemplo: *Supongamos un archivo FICH cuyos registros contienen información de personas de ambos sexos y un proceso de obtención de un listado donde además de una cabecera se desea obtener una línea de detalle con información de los registros pertenecientes a las mujeres.*

Imaginaremos que en el listado también existen líneas de detalle para los hombres. Estas líneas son ficticias y podemos imaginar que son líneas en blanco sin avance del papel (por ejemplo).



Ejemplo: *Supongamos un archivo MAESTRO y otro de MOVIMIENTOS con sus registros típicos. Añadir registros ficticios utilizando el concepto de registros emparejados y no emparejados para provocar correspondencias.*

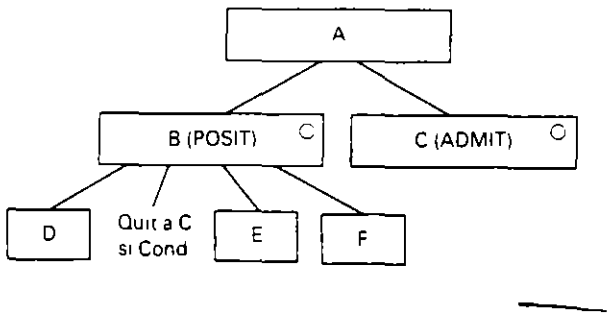


13.2.6. BACKTRACKING

En ocasiones aparecen en un programa entidades selectivas donde no puede cerciorarse cuál de ellas es la que se debe ejecutar, puesto que dicha selección depende de los propios datos que se van a procesar en las propias ramas. Estos casos suelen estar relacionados con tratamientos de validación de datos

Para estos casos, la Metodología de Jackson posee el *backtracking*, consistente en establecer una selección de dos ramas donde no se puede discernir cuál de ellas es la que se deba ejecutar. Se elige una de ellas, y si al ejecutarse se comprueba que ésta no es la correcta, automáticamente se salta a la otra rama. Cuando se efectúa el salto a la otra rama, esta se ejecuta de forma completa.

Gráficamente, el backtracking puede representarse de la siguiente forma: A es un backtracking, B es la entidad más probable (POSIT) y C es la entidad que se ejecutará si B no es correcta (ADMIT), y Cond es la condición que, si se cumple, nos indica que debemos ir a C (QUIT).




```
A  posit
   B  sec
      D
      Quit a C si Cond
      E
      F
   B  end
A  admit
   C
A  end
```

Los pasos para realizar un backtracking son:

— Configurar una selección simple (dos alternativas).
— Determinar la configuración y condiciones de la estructura, teniendo en cuenta que pueden aparecer efectos laterales por interrumpir a medias un proceso iniciado.
— En la lógica esquematizada cambiar sel por posit, or por admit y colocar una línea quit en cada elemento de posit que sea necesario.

En ocasiones también se utiliza el backtracking en estructuras repetitivas de forma similar a la selección.

13.2.7. COLISION DE ESTRUCTURAS

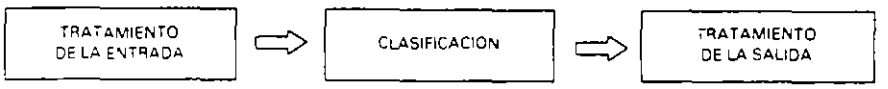
En determinados programas nos podemos encontrar con la ausencia de correspondencias en la segunda fase. Esto nos dificulta la construcción de la estructura del programa. Veamos cómo podemos actuar frente a este tipo de problemas.

Cuando no existen correspondencias entre las estructuras de datos aparece lo que se denomina una colisión entre estructuras, que puede resolverse de la siguiente forma:

— Identificar cada estructura sin correspondencias.
— Diseñar para cada una de ellas un programa simple mediante las cinco fases de la Metodología Jackson.
— Combinar los distintos programas simples en un solo.

Existen tres tipos de colisiones entre estructuras:

- **De orden.** Se presenta cuando los datos ocurren el mismo número de veces pero en distinto orden. Se soluciona intercalando entre el tratamiento de cada estructura un programa de clasificación para obtener la secuencia de salida.



- **De límite o frontera.** Las entidades de datos en niveles altos y bajos se corresponden, pero no existe correspondencia en los niveles intermedios. Se soluciona intercalando entre ambas estructuras otra estructura que haga de puente entre ellas.

- **De interrelación.** Se produce cuando las entidades relacionadas entre sí, que ocurren el mismo número de veces, no tienen una correspondencia 1 a 1. Pueden no tener una disponibilidad a la par. Se trata de un caso similar a la colisión de orden y puede ser solucionado de la misma manera.

13.2.8. INVERSION DE PROGRAMAS

Los problemas de colisión de estructuras de límite o frontera se solucionan con un archivo intermedio. La inversión de programas pretende eliminar la generación de este archivo intermedio, convirtiendo los programas simples que tratan las estructuras en rutinas, actuando uno como principal y otro como subrutina de éste, invirtiéndolo.

Esta técnica sobrepasa los niveles pretendidos en este libro, y tanto ella como las restantes colisiones no son tratadas con mayor profundidad. Recomendamos a los lectores, para profundizar en ellas, dirigirse a los libros que de forma monográfica tratan el tema y que se relacionan en la bibliografía.

Ejercicio

a) Una empresa tiene varios distribuidores, cada uno de ellos comercializa una serie de productos de los que, a lo largo de un mes, realiza una serie de ventas que se encuentran registradas en un archivo VENTAS, en el que cada registro refleja una operación de venta de un producto a un cliente. La estructura de un registro es:

DISTRIBUIDOR	PRODUCTO	CANTIDAD	PRECIO UNITARIO

El archivo se encuentra ordenado por distribuidor, y dentro de cada distribuidor por producto, pudiéndose dar el caso de que para un mismo distribuidor aparezcan varias ventas de un mismo producto.

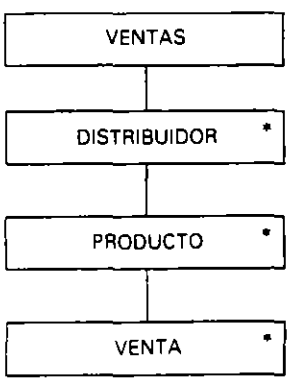
Se quiere obtener, a partir del archivo, un informe por distribuidor y producto con el siguiente formato:

```
                RESUMEN DE VENTAS EN EL MES

Distribuidor: XXXXXXXXXXXXX
      Producto                    Importe
-------------------------------------------
XXXXXXXXXXXXXXXXXX                 XXXXXX
      ...                           ...
XXXXXXXXXXXXXXXXXX                 XXXXXX
                  -------------------------
          Total distribuidor:      XXXXXXX

             . . .

Distribuidor: XXXXXXXXXXXXX
      Producto                    Importe
-------------------------------------------
XXXXXXXXXXXXXXXXXX                 XXXXXX
      ...                           ...
XXXXXXXXXXXXXXXXXX                 XXXXXX
                  -------------------------
          Total distribuidor:      XXXXXXX
                  =========================
             TOTAL GENERAL:        XXXXXXXX
```

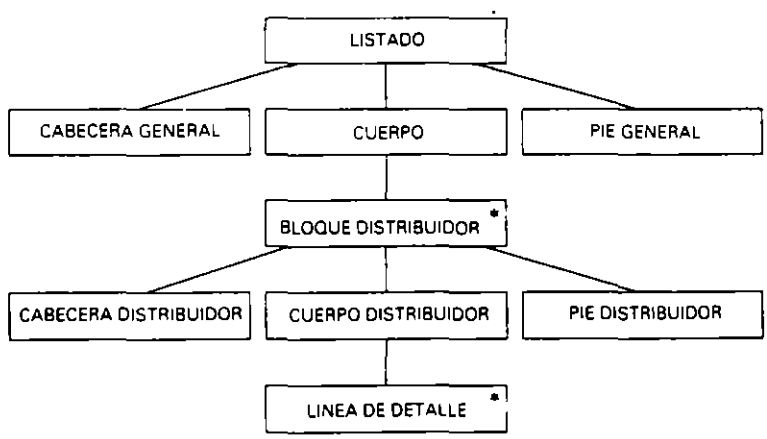
Cada línea de detalle representará el total vendido en el mes de cada producto y por cada distribuidor.

1.º Definir las estructuras de datos.

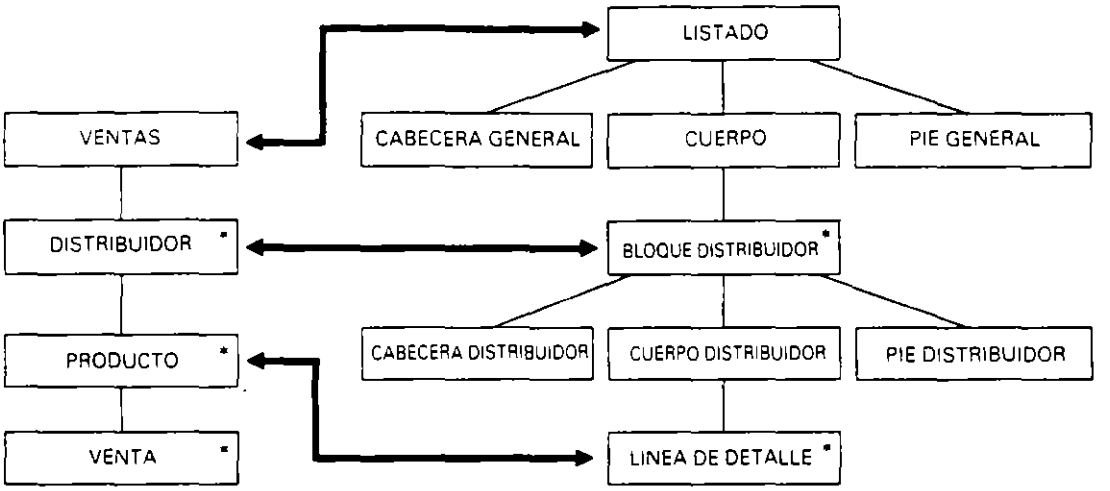
- **Archivo:**



- **Listado:**



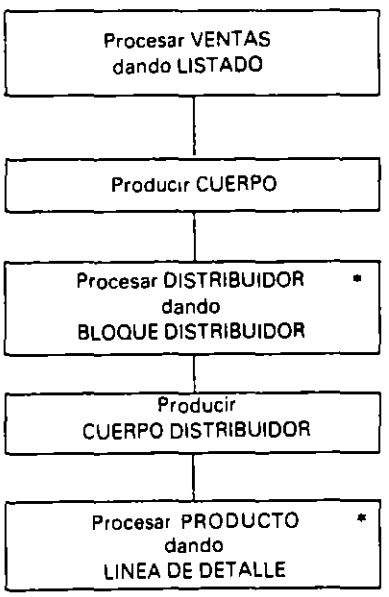
2.° Encontrar correspondencias entre las estructuras de datos.



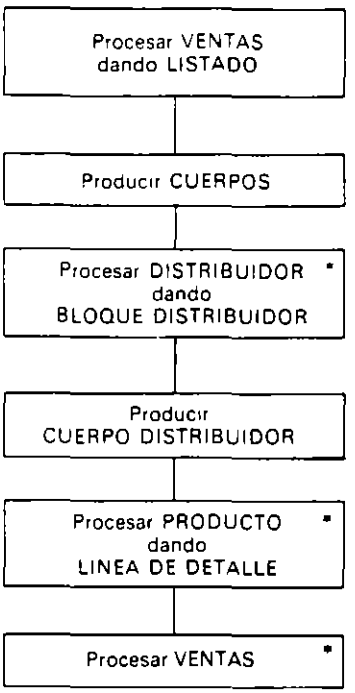
Como se puede ver, existen tres correspondencias claras entre VENTAS y LISTADO (ocurren una vez) entre DISTRIBUIDOR y BLOQUE DISTRIBUIDOR (ocurren igual número de veces y se pueden procesar en paralelo) y entre PRODUCTO y LINEA DE DETALLE (ocurren igual número de veces y se pueden procesar en paralelo)

3.° Formar la estructura del programa.

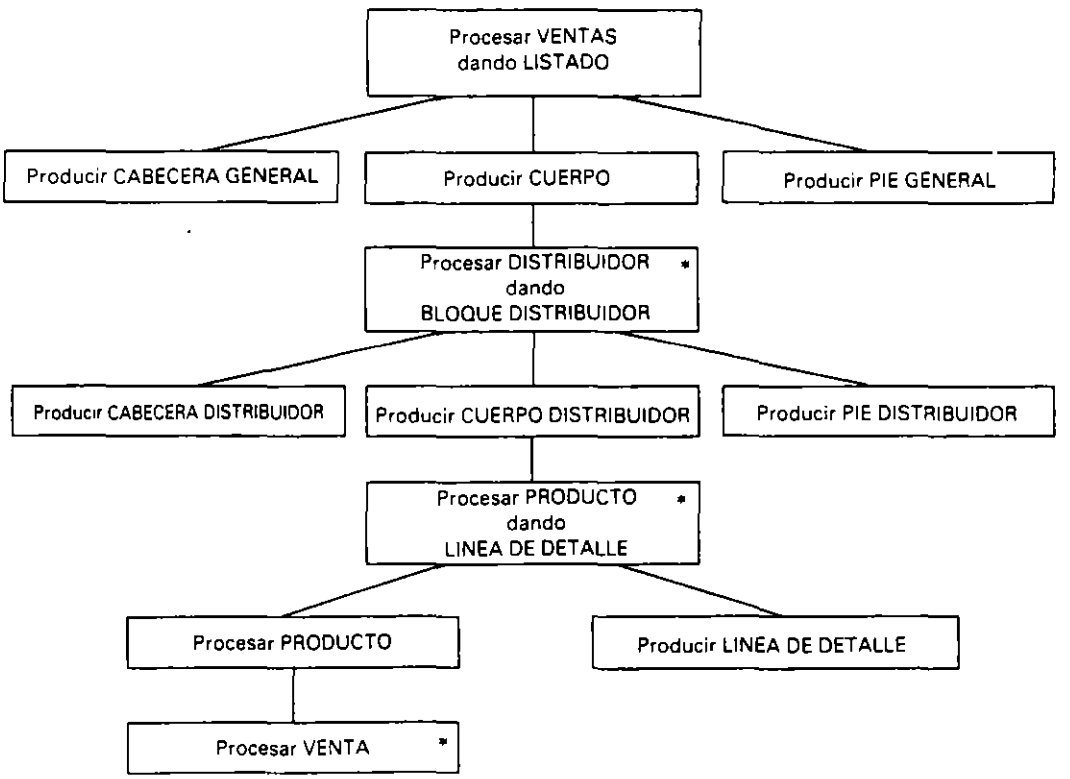
Regla 1. Crear una componente de programa por cada correspondencia, manteniendo el orden jerárquico entre ellas.



Regla 2. Se añade el resto de la estructura de entrada.



Regla 3. Se añade el resto de la estructura de salida asegurando que los datos están presentes en el momento apropiado.

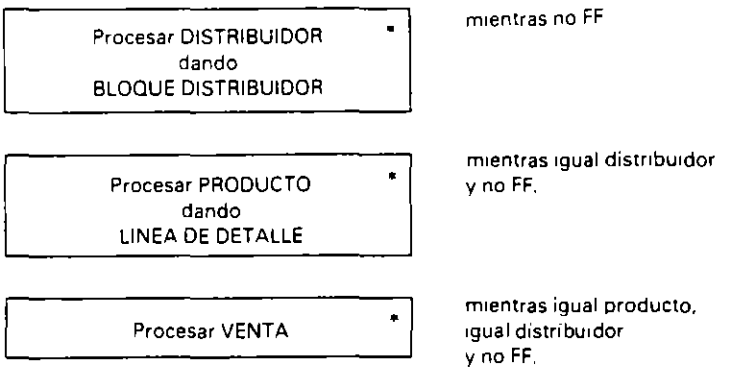


Con lo cual hemos obtenido la estructura del programa, donde puede observarse que la entidad Procesar PRODUCTO dando LINEA DE DETALLE se descompone en la secuencia Procesar PRODUCTO y Producir LINEA DE DETALLE, puesto que es necesario el proceso completo de un producto para escribir su correspondiente línea.

Antes de continuar con la siguiente fase, en general, es conveniente verificar la estructura del programa que por extensión se deja como ejercicio para el lector.

4.° Listar y asignar las operaciones y condiciones.

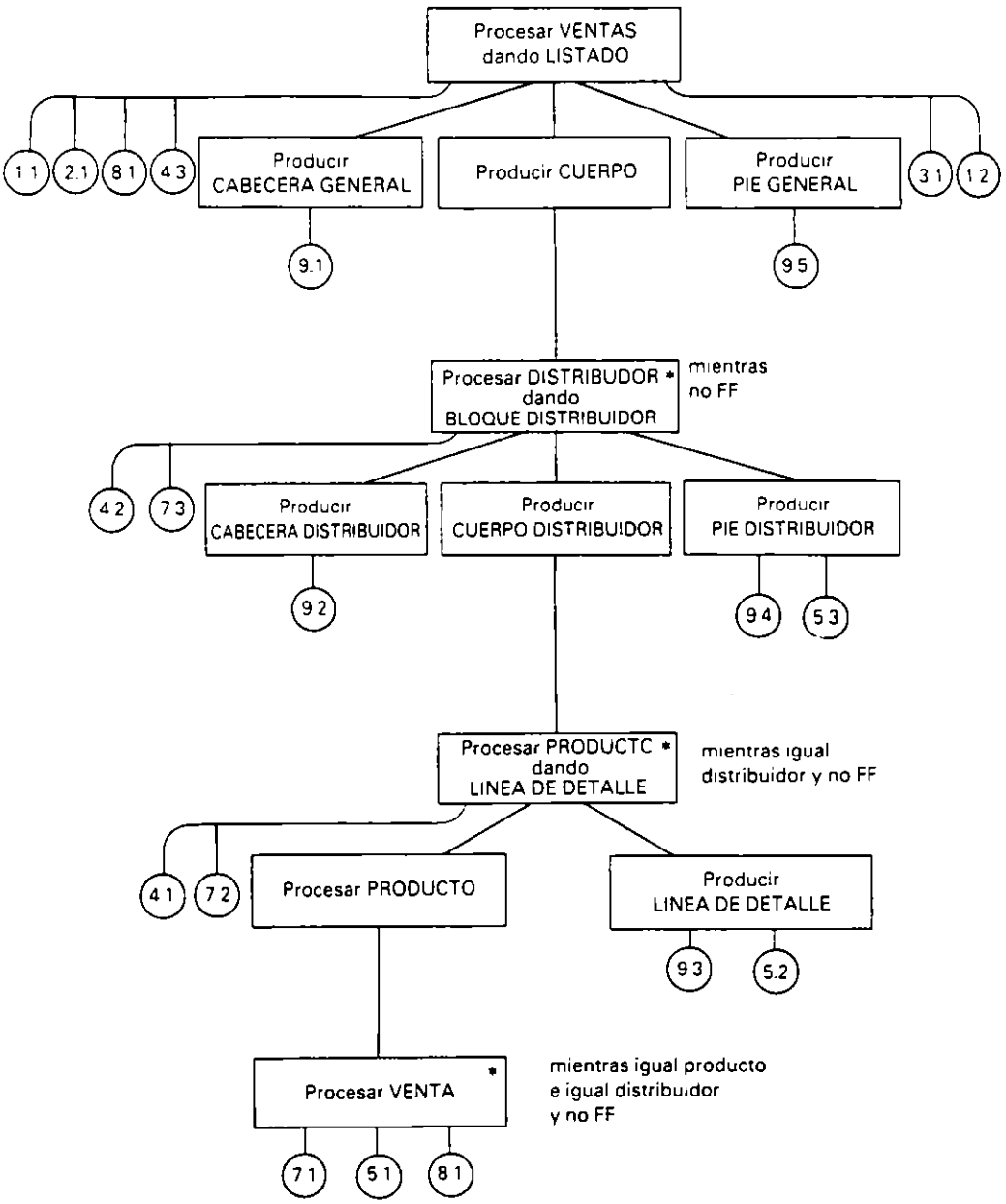
- Las condiciones en este proceso son las siguientes:



- La lista de operaciones es la siguiente:

1.1 Inicio.
1.2 Fin.
2.1 Abrir VENTAS.
3.1 Cerrar VENTAS.
4.1 Inicializar TOTAL PRODUCTO.
4.2 Inicializar TOTAL DISTRIBUIDOR.
4.3 Inicializar TOTAL GENERAL.
5.1 Acumular IMPORTE VENTA en TOTAL PRODUCTO.
5.2 Acumular TOTAL PRODUCTO en TOTAL DISTRIBUIDOR.
5.3 Acumular TOTAL DISTRIBUIDOR en TOTAL GENERAL.
7.1 Calcular IMPORTE VENTA.
7.2 Retener PRODUCTO.
7.3 Retener DISTRIBUIDOR.
8.1 Leer REGISTRO.
9.1 Escribir CABECERA GENERAL.
9.2 Escribir CABECERA DISTRIBUIDOR.
9.3 Escribir LINEA DE DETALLE.
9.4 Escribir PIE DISTRIBUIDOR.
9.5 Escribir PIE GENERAL.

- La asignación de operaciones y condiciones quedará:



5.º Escribir la lógica esquematizada.

```
PVENTASLISTADO sec
  Inicio
  Abrir VENTAS
  Leer REGISTRO
  Inicializar TOTAL GENERAL
  PCABECERAGENERAL sec
    Escribir CABECERA GENERAL
  PCABECERAGENERAL end
```

```
  PCUERPO iter mientras no FF
    PDISTRIBUIDORBLOQUEDISTRIBUIDOR sec
      Inicializar TOTAL DISTRIBUIDOR
      Retener DISTRIBUIDOR
      PCABECERADISTRIBUIDOR sec
        Escribir CABECERA DISTRIBUIDOR
      PCABECERADISTRIBUIDOR end
      PCUERPODISTRIBUIDOR iter mientras igual distribuidor y no FF
        PPRODUCTOLINEADEDETALLE sec
          Inicializar TOTAL PRODUCTO
          Retener PRODUCTO
          PPRODUCTO iter mientras igual producto e igual distribuidor y no FF
            PVENTA sec
              Calcular IMPORTE VENTA
              Acumular IMPORTE VENTA en TOTAL PRODUCTO
              Leer REGISTRO
            PVENTA end
          PPRODUCTO end
          PLINEADEDETALLE sec
            Escribir LINEA DE DETALLE
            Acumular TOTAL PRODUCTO en TOTAL DISTRIBUIDOR
          PLINEADEDETALLE end
        PPRODUCTOLINEADEDETALLE end
      PCUERPODISTRIBUIDOR end
      PPIEDISTRIBUIDOR sec
        Escribir PIE DISTRIBUIDOR
        Acumular TOTAL DISTRIBUIDOR en TOTAL GENERAL
      PPIEDISTRIBUIDOR end
    PDISTRIBUIDORBLOQUEDISTRIBUIDOR end
  PCUERPO end
  PPIEGENERAL sec
    Escribir PIE GENERAL
  PPIE GENERAL end
  Cerrar VENTAS
  Fin
PVENTASLISTADO end
```

13.3. METODOLOGIA WARNIER

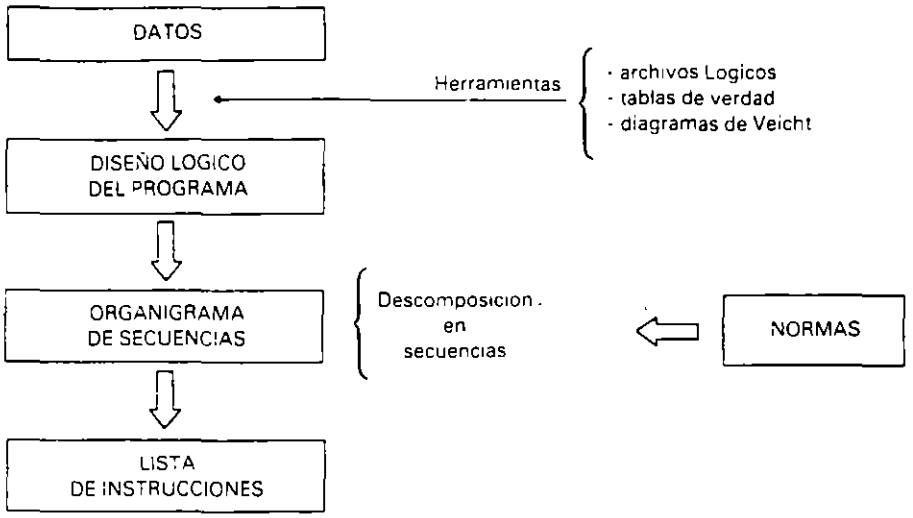
La Metodología Warnier fue creada por Jean D. Warnier en 1975, y se dio a conocer en el libro *Entrainement a la Programmation. Construction des Programmes.*

Es una metodología estructurada, formada por un conjunto de normas y herramientas que nos permiten solucionar un problema de forma clara y sencilla.

Se basa fundamentalmente en la jerarquía de los datos, tanto de entrada como de salida, siendo estos últimos los que de forma directa ejercen una influencia esencial en la definición y control del programa.

13.3.1. INTRODUCCION

El desarrollo de un programa siguiendo la Metodología Warnier sigue un conjunto de fases que parten de los datos del problema hasta la definición completa del mismo. El esquema de todo el proceso viene representado en el siguiente gráfico:



Las herramientas utilizadas son:

— Definicion de los archivos lógicos.
— Utilización de tablas de verdad.
— Utilización de los diagramas de Veicht.

Las tablas de verdad y los diagramas de Veicht se utilizan para optimizar estructuras, y su estudio se sale de las pretensiones de este libro.

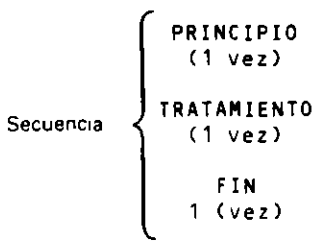
13.3.2. ESTRUCTURAS BASICAS

Esta metodología representa las estructuras de dos formas diferentes:

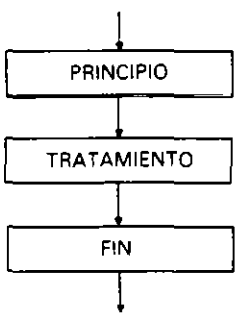
— **Cuadro de descomposición de secuencias.** Representación basada en el uso de llaves donde desde el principio (parte superior) hasta el fin (parte inferior) aparecen los elementos integrantes de la estructura y su número de ocurrencias.
— **Organigrama de secuencias.** Organigrama que utiliza los símbolos normalizados con un estilo propio. Empieza con un principio y termina con un fin.

El elemento esencial en el que se basa el método es la **secuencia lógica** compuesta por un conjunto de sentencias o estructuras que se ejecutan el mismo número de veces y en el mismo orden. Su representación es la siguiente:

Cuadro de descomposición



Organigrama de secuencias



• **Estructura alternativa.** Es aquella que empieza con una condición y permite dos o más caminos alternativos de los que se ejecutará uno y siempre uno.

Alternativa doble

Cuadro de descomposición

Alternativa doble

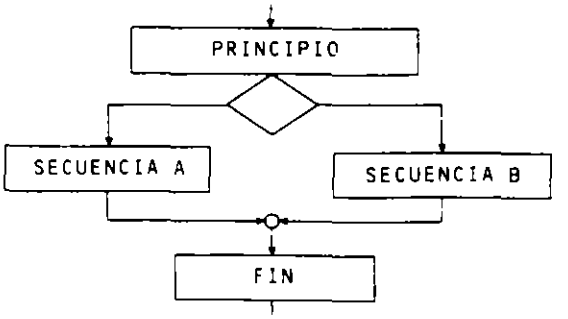
PRINCIPIO
(1 vez)

SECUENCIA A
(0 - 1 veces)

SECUENCIA B
(0 - 1veces)

FIN
1(vez)

Organigrama de secuencias



Esta estructura admite que una de sus ramas sea vacía.

Alternativa simple

Cuadro de descomposición

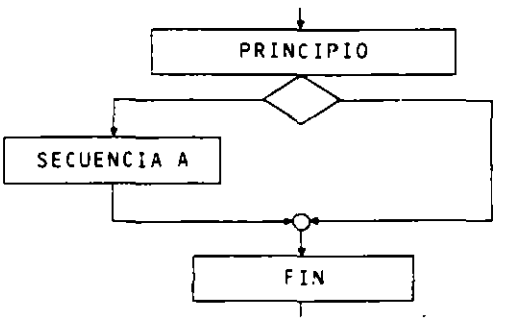
Alternativa

PRINCIPIO
(1 vez)

SECUENCIA A
(0 - 1 vez)

FIN
(1 vez)

Organigrama de secuencias



Estructura repetitiva. Es aquella que permite la repetición de su rango un número indeterminado de veces.

Cuadro de descomposición

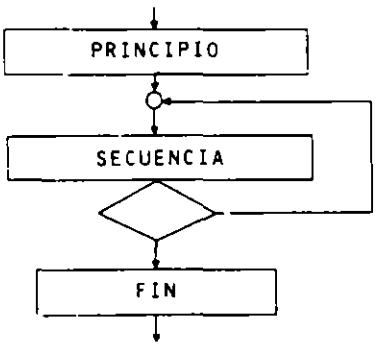
Repetitiva

PRINCIPIO
(1 vez)

SECUENCIA
(n veces)

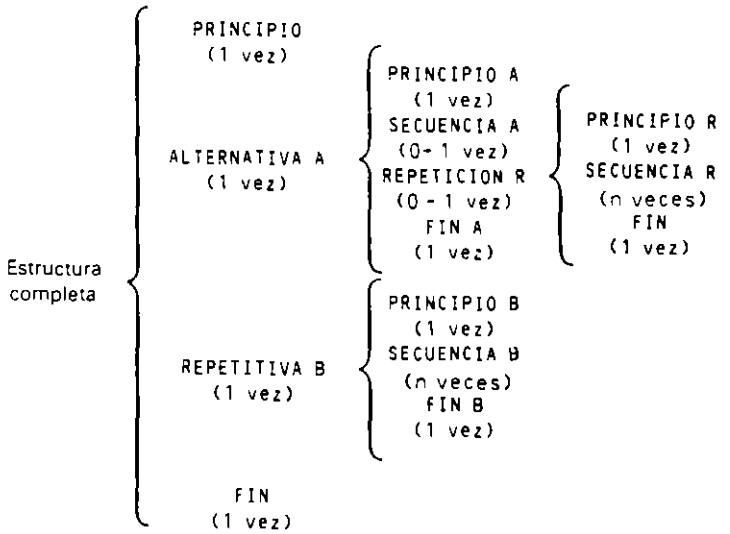
FIN
(1 vez)

Organigrama de secuencias

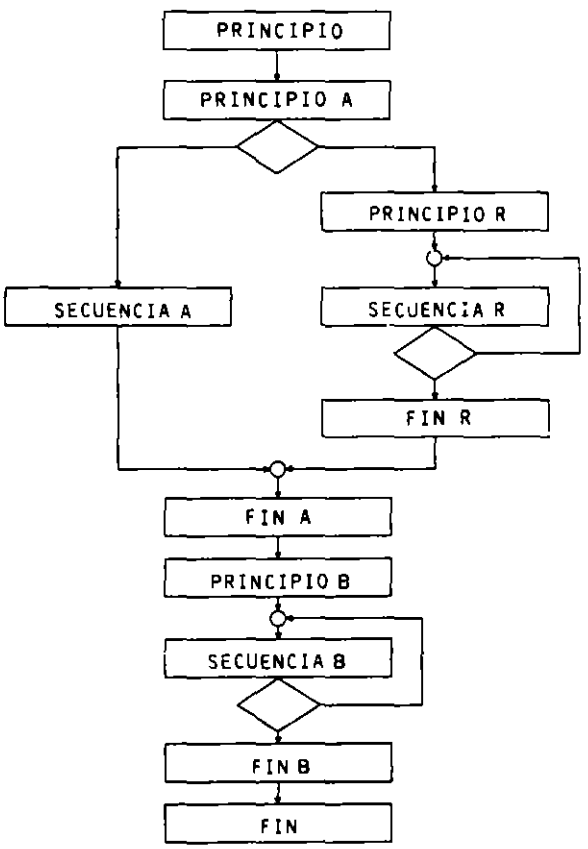


• **Estructuras complejas.** Son aquellas que están compuestas por una combinación de las anteriores. Empiezan con un principio del conjunto; cada estructura alternativa o repetitiva lleva su principio y fin, y, por último, aparece un fin del conjunto. Veamos un ejemplo.

Cuadro de descomposicion



Organigrama de secuencias



13.3.3. FASES DE LA METODOLOGIA WARNIER.

Las fases que se siguen para desarrollar un programa son:

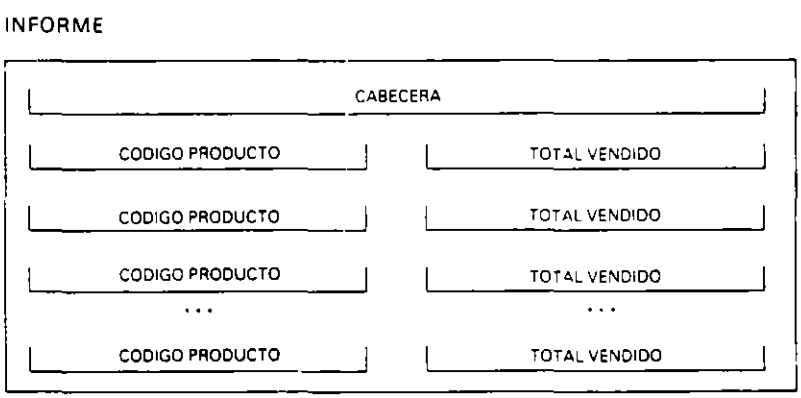
1.ª Estudio de los datos de salida. Trata de crear el archivo lógico de salida (ALS).
2.ª Estudio de los datos de entrada. Trata de crear el archivo lógico de entrada (ALE), teniendo en cuenta la organización de los datos de salida y los posibles datos intermedios.
3.ª Hacer el cuadro de descomposición de secuencias.
4.ª Dibujar el organigrama de secuencias de Warnier.
5.ª Construir la lista de instrucciones y asignarlas en el organigrama de secuencias.
6.ª Diseñar el juego de datos de ensayo y analizar los resultados.

13.3.4. ESTUDIO DE LOS DATOS DE SALIDA

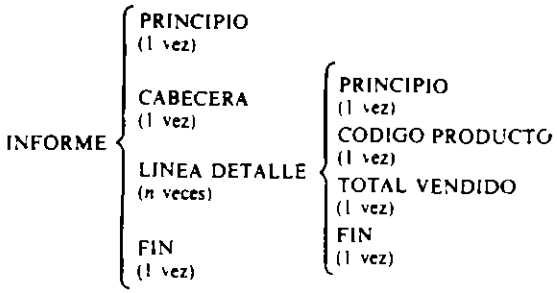
Consiste en la creación del cuadro de descomposición de los archivos físicos de salida, listados y cualquier otra estructura de datos que configure la salida del programa. Todo el conjunto de estructuras físicas configura lo que se denomina **archivo lógico de salida** ALS y en él no sólo aparecen los archivos y estructuras físicas, sino también sus relaciones.

El cuadro de descomposición de una estructura de datos lleva en cada elemento el número de ocurrencias del mismo, y en el caso de varias ocurrencias 0-1 veces, éstas pueden ser conjuntas + (se pueden dar simultáneamente) o disjuntas ⊕ (sólo se puede dar una de ellas).

Ejemplo: *Supongamos un proceso en el que se desea obtener un informe cuya estructura es la siguiente:*



Como puede verse, se trata de un informe con una cabecera seguida de una línea de detalle por cada producto. El cuadro de descomposición será el siguiente:

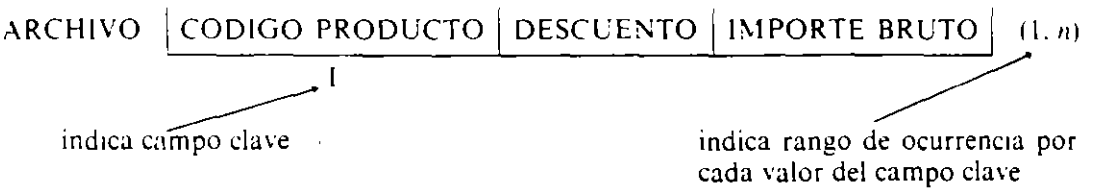


Nota: Entre las fases 2.ª y 3.ª se debe hacer un estudio de los datos intermedios necesarios que no pertenecen a la entrada ni a la salida.

13.3.5. ESTUDIO DE LOS DATOS DE ENTRADA

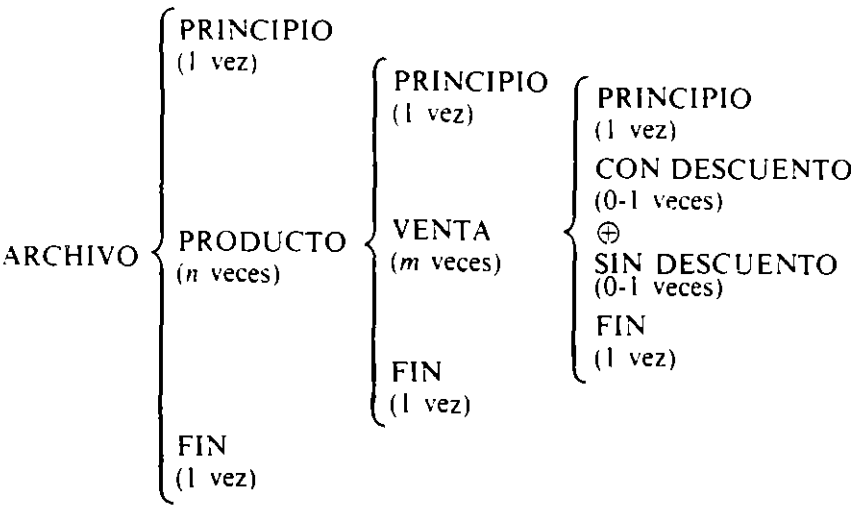
En esta fase se crean los cuadros de descomposición de los archivos o estructuras de datos físicos que configuren la entrada del programa que, junto con sus relaciones, formarán lo que se denomina **archivo lógico de entrada**. Para su confección es necesario tener presente la estructura del archivo lógico de salida obtenido en la fase anterior y los posibles datos intermedios que no pertenecen a la entrada ni a la salida.

Ejemplo: *Supongamos que para obtener el informe del ejemplo anterior disponemos de un archivo cuyos registros tienen la siguiente estructura.*



El archivo se encuentra ordenado por código y cada registro representa una venta realizada que puede ser con descuento (>0 %) o sin descuento (0 %).

El cuadro de descomposición será el siguiente:

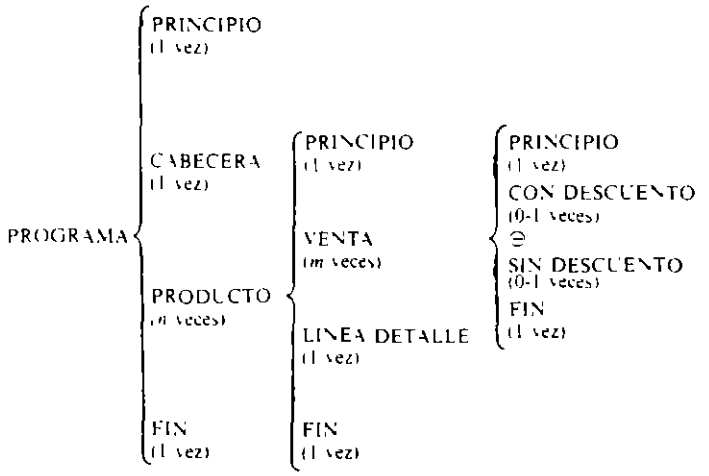


13.3.6. CUADRO DE DESCOMPOSICION DE SECUENCIAS

En esta fase se desarrolla la lógica del programa de forma descendente, construyéndose el cuadro de descomposición de secuencias a partir del archivo lógico de entrada.

Los elementos de las estructuras de datos del archivo lógico de entrada se transforman en elementos de programa, de tal forma que las secuencias de datos se transforman en procesos secuenciales de los mismos, las repeticiones en procesos repetitivos, las alternativas de datos en procesos alternativos y si existen elementos del tipo 0-1 veces con la conexión +, es decir, que puedan darse simultáneamente, se construirá una tabla de verdad y un diagrama de Veitch para obtener la estructura correspondiente.

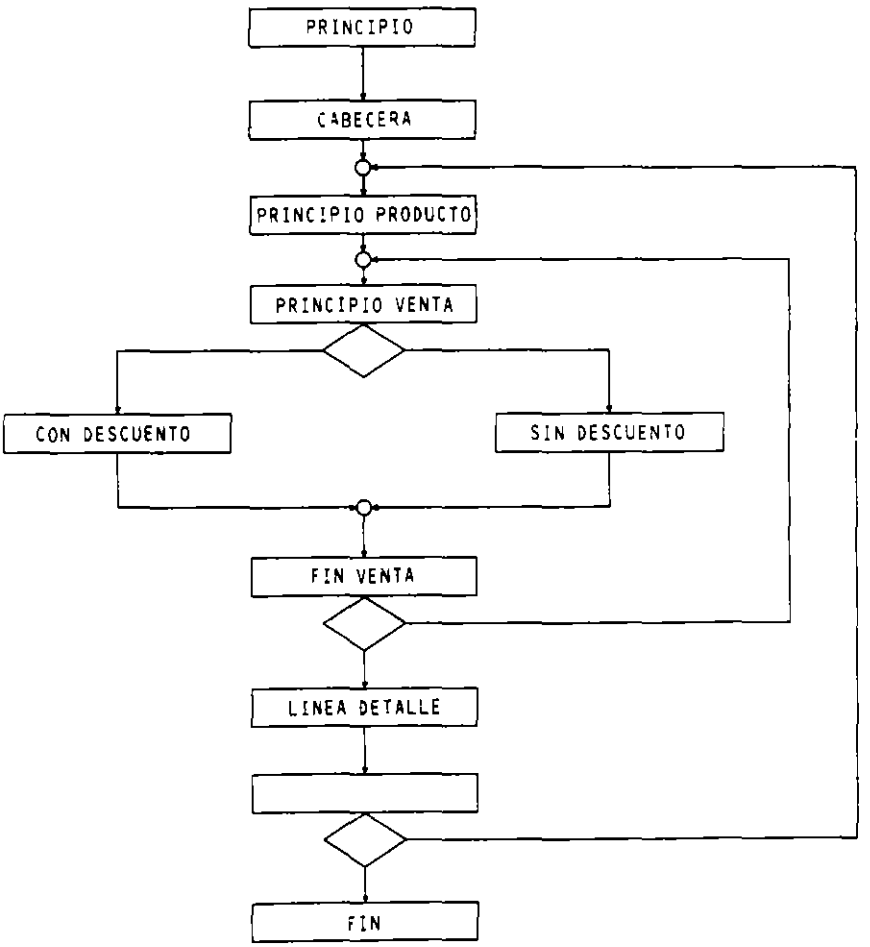
Ejemplo: *El cuadro de descomposición del programa de los ejemplos anteriores es:*



13.3.7. ORGANIGRAMA DE SECUENCIAS DE WARNIER

En esta fase pasamos del cuadro de descomposición de secuencias obtenido en la fase anterior al correspondiente organigrama de secuencias de Warnier en un proceso simple de traducción.

Ejemplo: *El organigrama de secuencias del ejemplo anterior es:*



13.3.8. LISTA DE INSTRUCCIONES Y ASIGNACION DE LAS MISMAS

Una vez construido el organigrama de secuencias del programa es necesario listar las instrucciones de que consta y asignarlas dentro de la estructura en aquellos lugares donde deban ejecutarse. Para ello se realizarán los siguientes puntos:

— Confeccionar la lista de instrucciones.
— Asignar las instrucciones en las secuencias.
— Determinar la posición de cada instrucción en cada secuencia.

Para confeccionar la lista de instrucciones se agrupan éstas en dos secciones:

— **Instrucciones de tratamiento.** Son las instrucciones primitivas cuyo cometido es realizar una operación de forma inmediata. Por ejemplo, la lectura de un dato, la inicialización de una variable, la realización de un determinado cálculo, etc.
— **Instrucciones de estructura.** Son instrucciones de control del programa: alternativas y repeticiones que deben estar reflejadas ya en la estructura del programa, en el cuadro de descomposición de secuencias y en el organigrama de secuencias.

En el tratamiento secuencial de archivos suele seguirse una regla común para la lectura y tratamiento de sus registros, que se denomina **lectura adelantada**, consistente en:

— Se lee el primer registro inmediatamente después de la apertura del archivo en cuestión.
— Cada siguiente registro se lee al terminar el proceso del registro anterior.

Para la lista de las instrucciones de tratamiento se utiliza una tabla como la siguiente:

Secuencia	Instrucción
Num secuencia	Lista de instrucciones

Donde a cada secuencia se le asigna un número comenzando por el 10, y de 10 en 10, y a continuación se describen en la columna de la derecha la o las instrucciones de que consta.

Por último se asigna a cada secuencia del organigrama de secuencias su número correspondiente, se numeran las condiciones y se listan éstas.

Ejemplo: *La lista de instrucciones y asignación del ejemplo anterior es:*

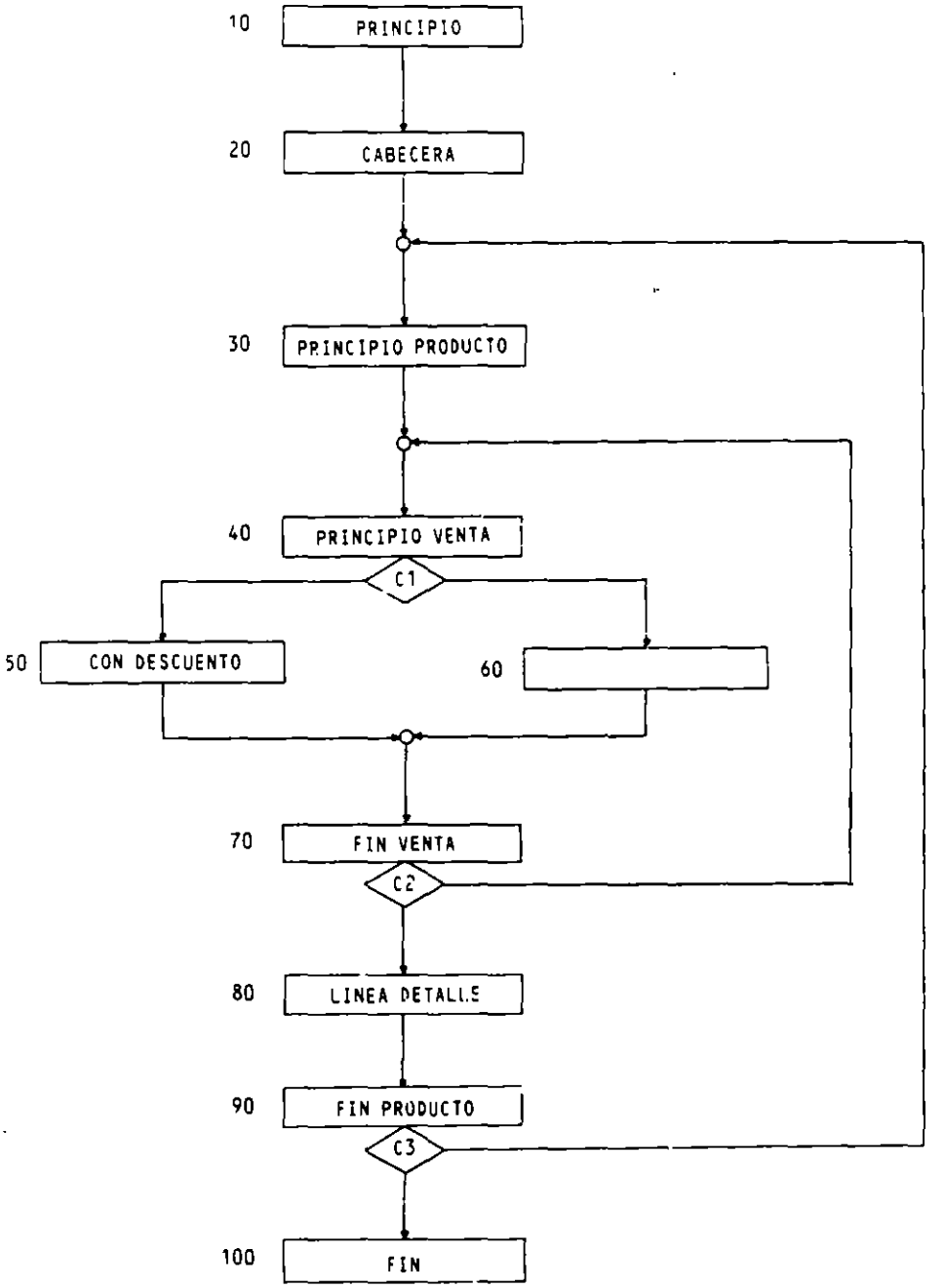
Secuencia	Instrucción
10	Abrir ARCHIVO Leer VENTA
20	Escribir CABECERA
30	Retener CODIGO PRODUCTO Inicializar ACUMULADOR
40	Continuar
50	Calcular DESCUENTO Restar DESCUENTO de IMPORTE BRUTO Acumular VENTA
60	Acumular VENTA
70	Leer VENTA
80	Escribir CODIGO PRODUCTO y ACUMULADOR
90	Continuar
100	Cerrar ARCHIVO

En general. la lista de instrucciones se hace especificándolas en el lenguaje en el que posteriormente va a ser codificado el programa. En nuestros ejemplos utilizaremos una notación genérica sin referirnos a ningún lenguaje concreto.

Las condiciones a asignar son las siguientes:

C1 DESCUENTO < > 0%.
C2 mientras igual CODIGO PRODUCTO y no FF
C3 mientras no FF

Asignando operaciones y condiciones tendremos:



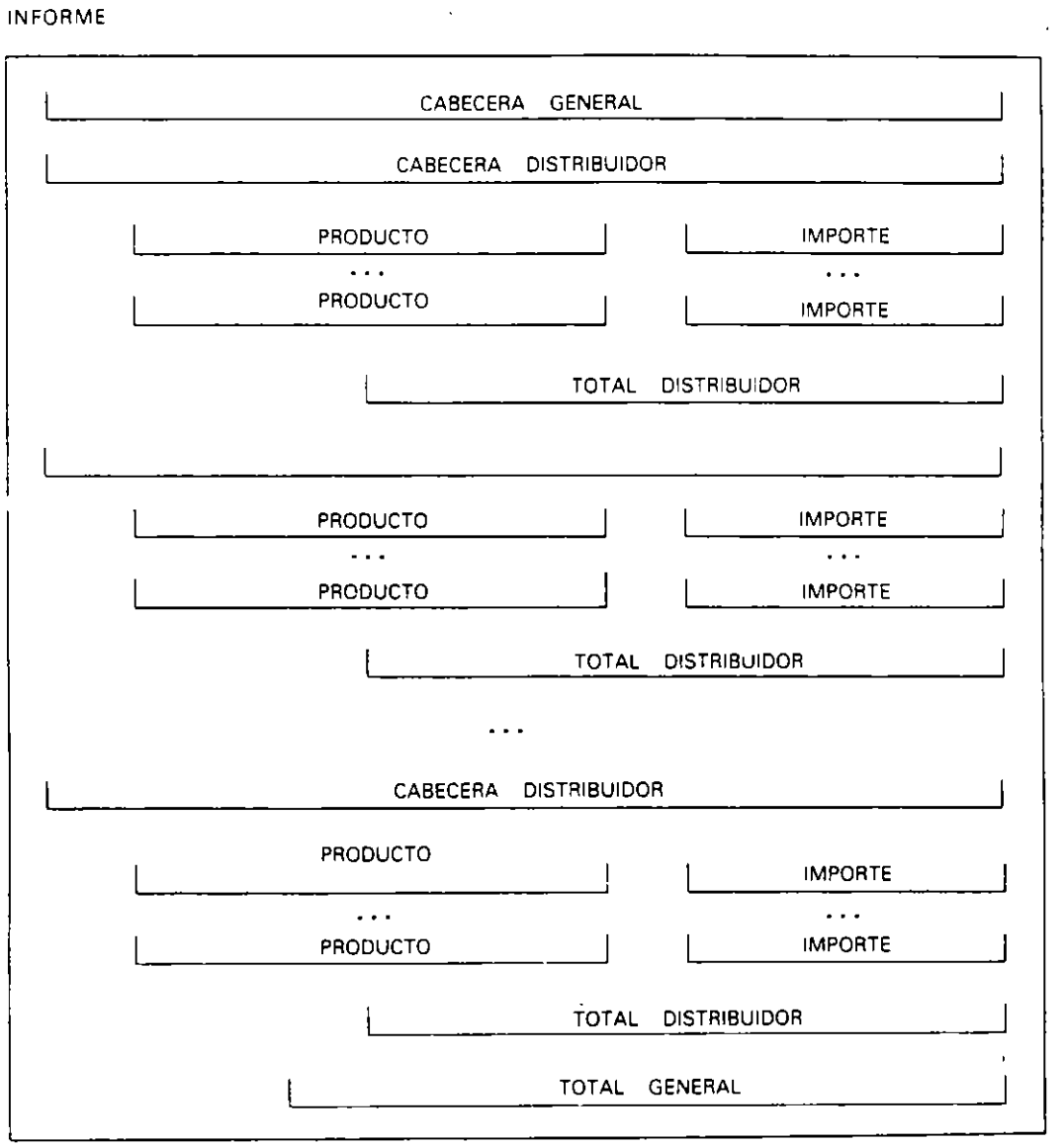
La fase del diseño del juego de datos de ensayo y análisis de los resultados se salen de la extensión de este libro, que solo ha pretendido presentar de forma breve y sencilla la metodología de Warnier.

En esta metodología el tratamiento de errores, la intercalación (collating), el Backtracking, las colisiones entre estructuras y la inversión de programas se tratan de manera similar a la Metodología de Jackson.

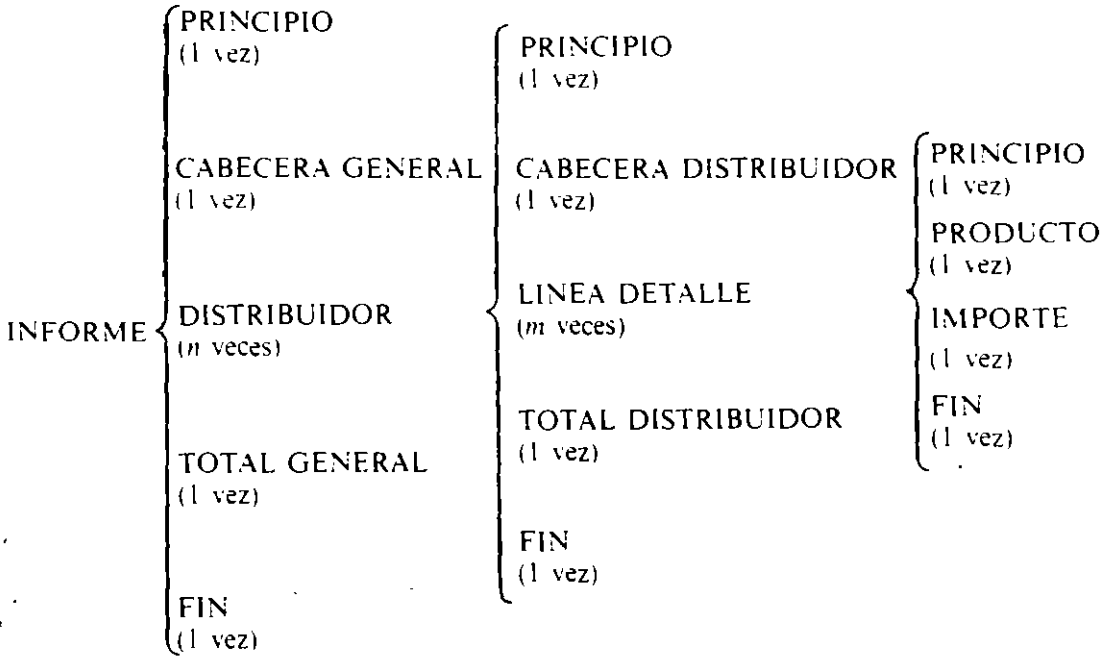
Ejercicio

Desarrollar por Warnier el ejercicio del punto final del estudio de la Metodología Jackson

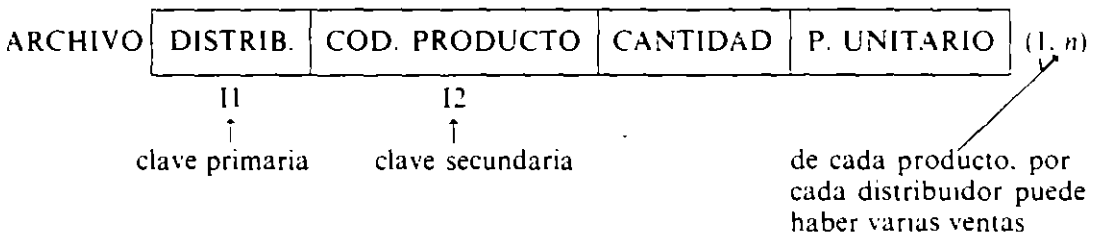
1.º *Estudio de los datos de salida. La salida en este proceso es un informe con la siguiente estructura:*



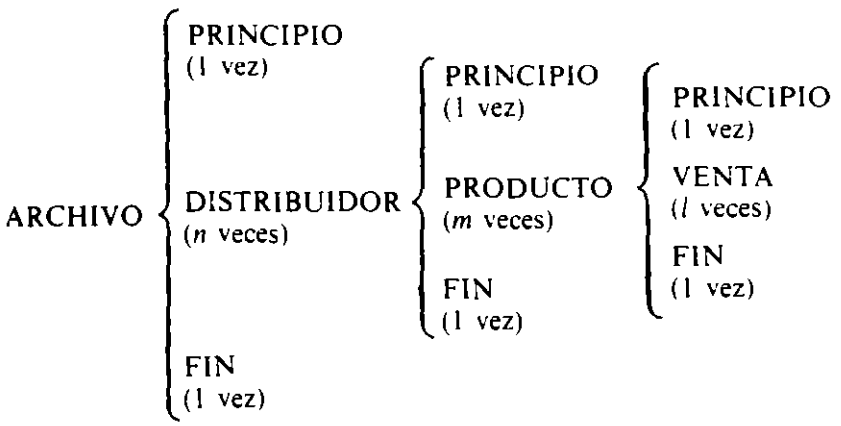
El cuadro de descomposición del archivo lógico de salida es el siguiente:



2.° *Estudio de los datos de entrada. El archivo de entrada tiene la siguiente estructura.*

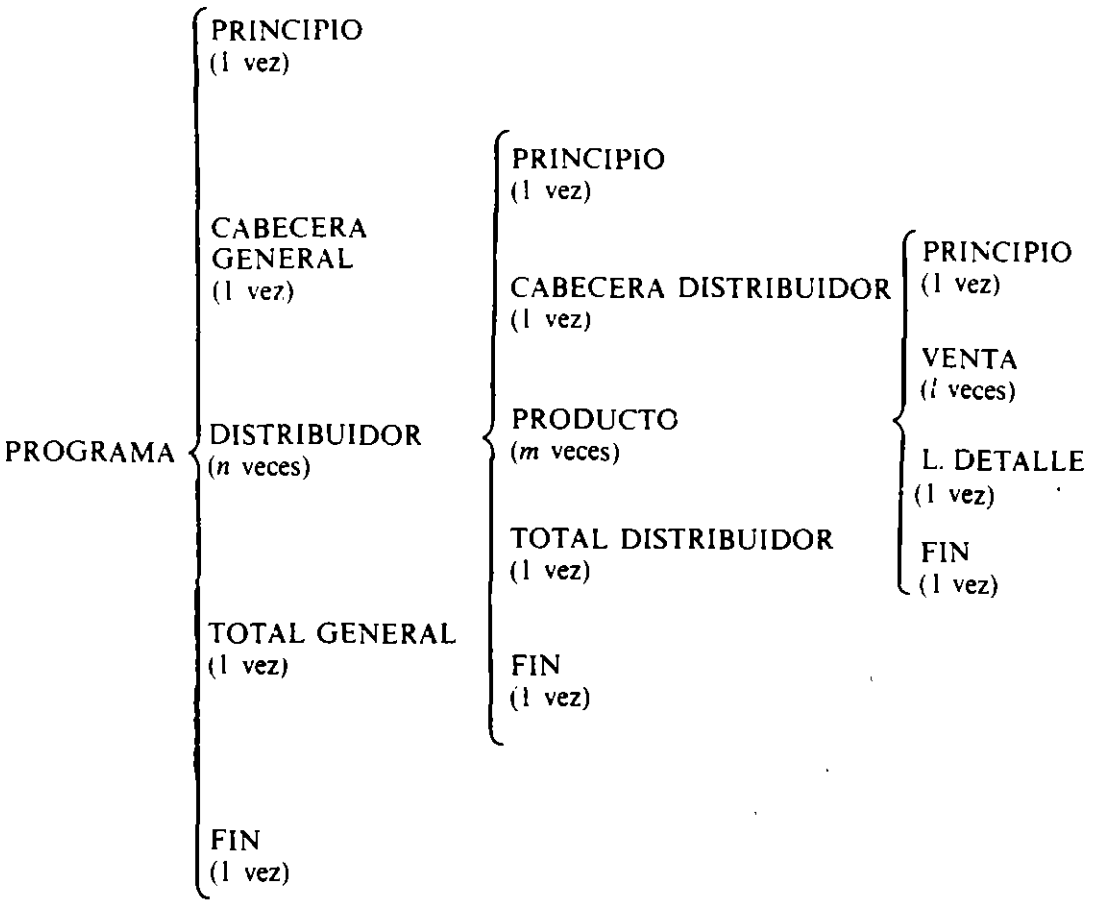


El archivo se encuentra ordenado por distribuidor y dentro de cada distribuidor por producto; por tanto el cuadro de descomposición será el siguiente:

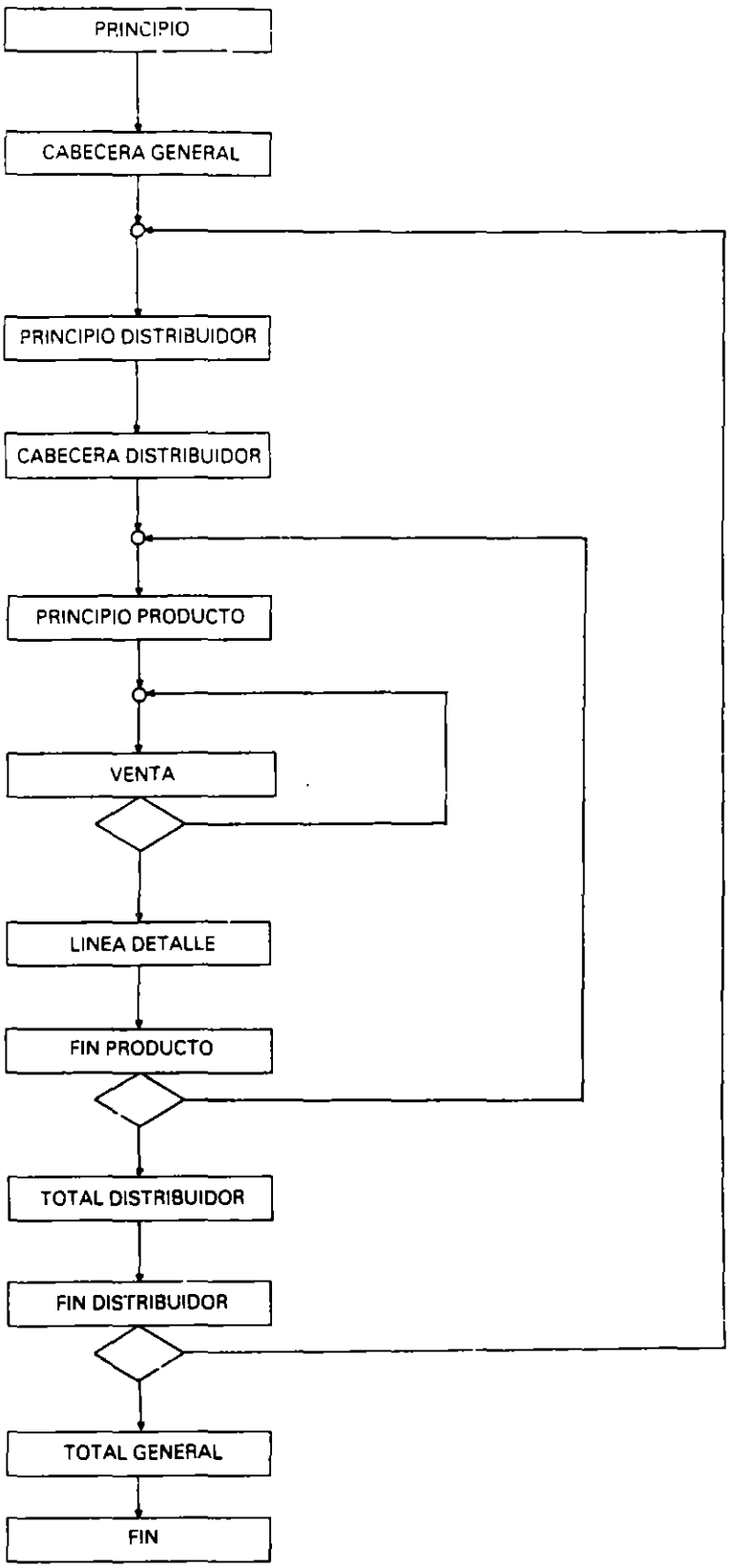


De los puntos 1.° y 2.° puede deducirse que las estructuras de entrada y salida mantienen una correspondencia directa sin que exista ningún tipo de colisión.

3.° *Cuadro de descomposición de secuencias. Conjuntando la entrada con la salida, obtenemos la estructura del programa.*



4.° *Organigrama de secuencias. Se obtiene a partir del cuadro de descomposición.*



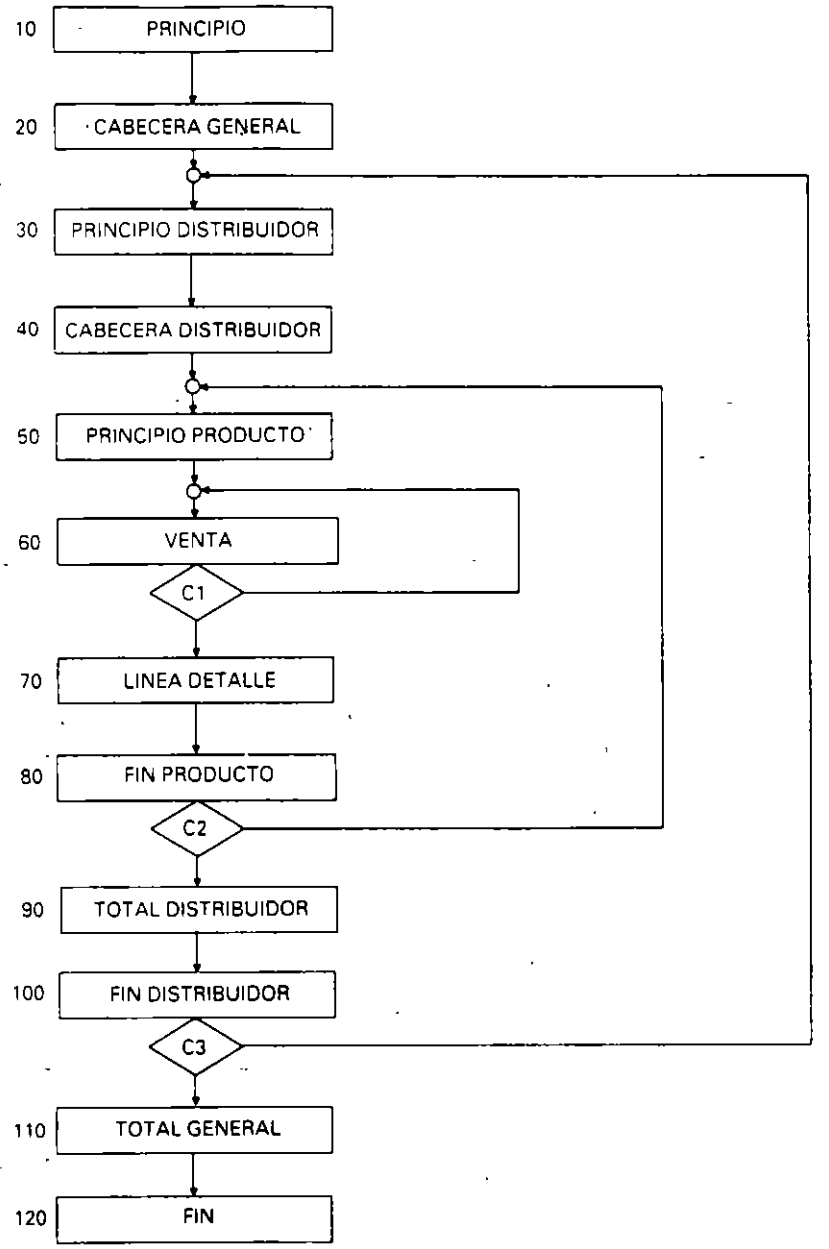
5.° *Lista de instrucciones y asignación. En este proceso, la lista de instrucciones es la siguiente:*

Secuencia	Instrucción
10	Abrir ARCHIVO Leer VENTA Inicializar TOTAL GENERAL
20	Escribir CABECERA GENERAL
30	Retener DISTRIBUIDOR Inicializar TOTAL DISTRIBUIDOR
40	Escribir CABECERA DISTRIBUIDOR
50	Retener CODIGO PRODUCTO Inicializar TOTAL PRODUCTO
60	Calcular IMPORTE VENTA Acumular IMPORTE VENTA en TOTAL PRODUCTO Leer VENTA
70	Escribir CODIGO PRODUCTO y TOTAL PRODUCTO Acumular TOTAL PRODUCTO en TOTAL DISTRIBUIDOR
80	Continuar
90	Escribir TOTAL DISTRIBUIDOR Acumular TOTAL DISTRIBUIDOR en TOTAL GENERAL
100	Continuar
110	Escribir TOTAL GENERAL
120	Cerrar ARCHIVO

Las condiciones serán:

C1 mientras igual DISTRIBUIDOR e igual PRODUCTO y no FF.
C2 mientras igual DISTRIBUIDOR y no FF.
C3 mientras no FF.

Asignando operaciones y condiciones quedará:



13.4. OTRAS METODOLOGIAS

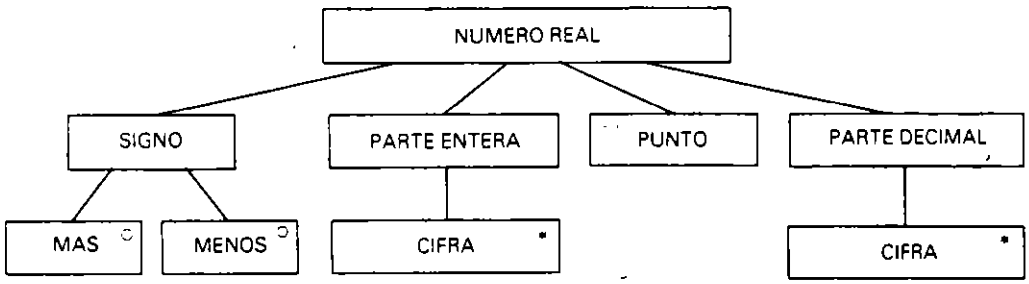
Existen otras metodologías para el desarrollo de programas, como la de Bertini, que no ha sido desarrollada en este libro por falta de espacio. Sus fundamentos y procedimientos son similares a las dos estudiadas en este capítulo, sin que existan ventajas ni inconvenientes notables entre ellas.

En el mercado actual existe otro tipo de metodologías que abarcan no sólo el desarrollo del programa, sino todo el ciclo de vida de una aplicación (análisis, programación, pruebas, mantenimiento, ...) que salen de la extensión de este libro Merise, Yourdon, etc.).

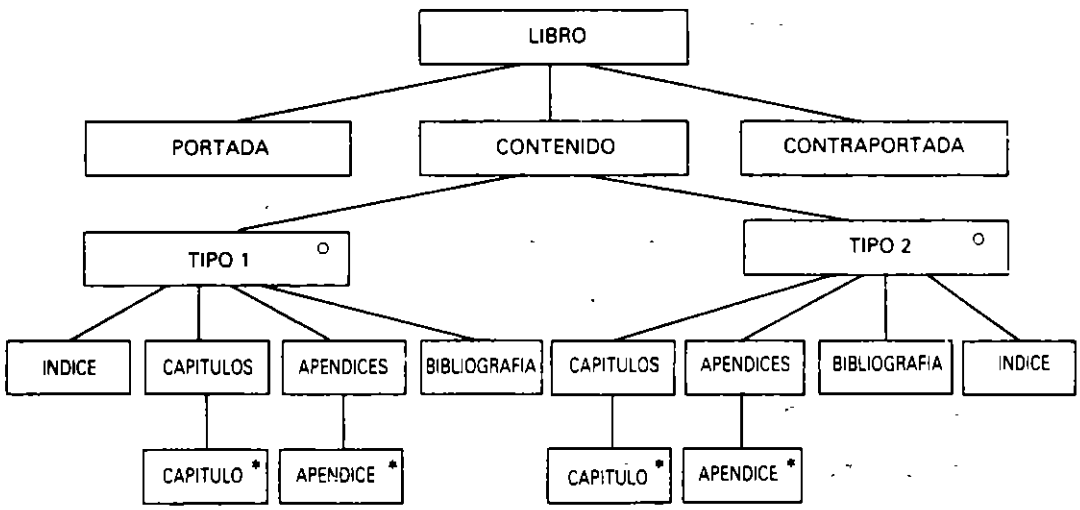
EJERCICIOS RESUELTOS

1. *Representar por Jackson la estructura de un número real con signo, parte entera, punto decimal y parte decimal, todo ello a nivel de caracteres.*

   Se considera como un carácter el signo, cada cifra y el punto decimal.



2. *Los libros técnicos que se editan en la actualidad tienen la siguiente estructura: Una portada y una contraportada al principio y final, respectivamente; entre ambas aparecen una serie de capítulos que pueden venir seguidos de algún apéndice. Siempre aparece detrás de los posibles apéndices la bibliografía. Por último poseen un índice que puede venir al principio o al final del contenido del libro. Representar la estructura por Jackson.*



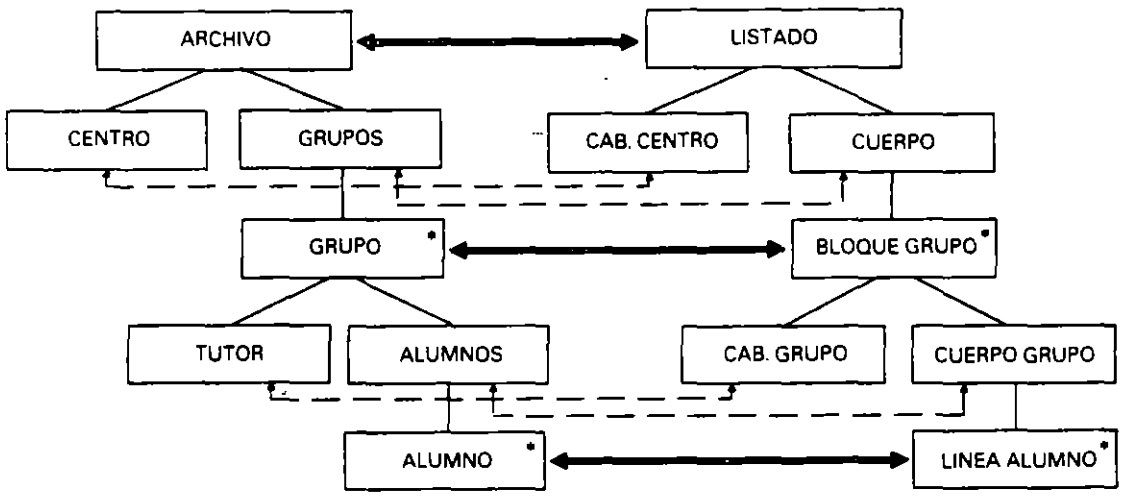
3. *En un Centro de Enseñanza se dispone de un archivo en el que el primer registro contiene información sobre el Centro (entre otros datos, el nombre del Centro). El resto de registros tiene información sobre los tutores y alumnos de los distintos grupos. Estos registros contienen, entre otros datos, el nombre y el grupo al que pertenecen tanto los tutores como los alumnos. El archivo se encuentra ordenado por grupos y cada grupo contiene en su primer registro la información del tutor.*

Se quiere obtener un listado cuya estructura es:

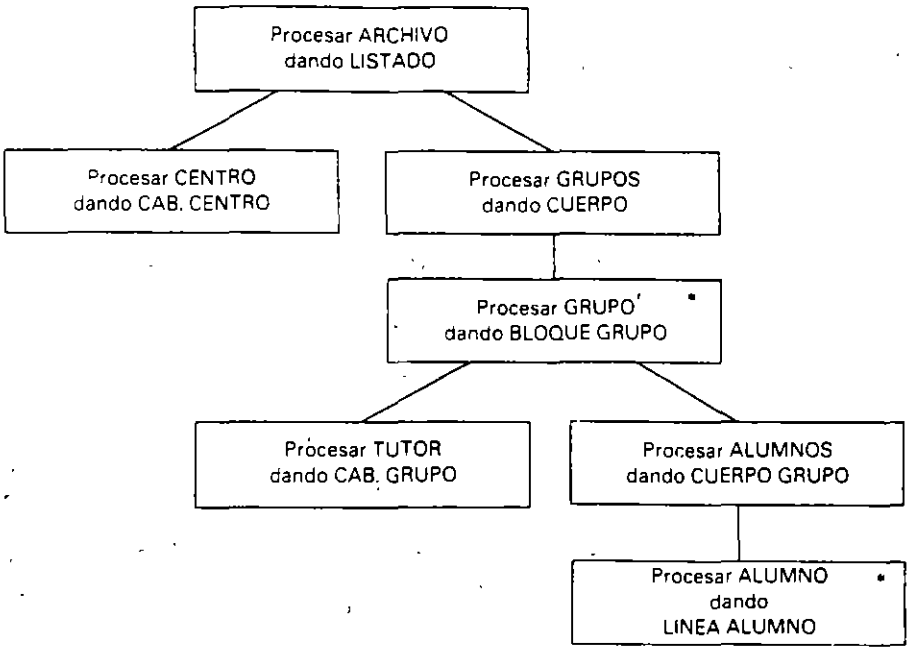
```
Nombre del centro: XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
Grupo: XXX  Tutor D. XXXXXXXXXXXXXXXXXXXXXXXXXXXXX
      Número              Alumno
      ------    --------------------------
        1       XXXXXXXXXXXXXXXXXXXXXXXX
        2       XXXXXXXXXXXXXXXXXXXXXXXX
       ...                 ...
        n       XXXXXXXXXXXXXXXXXXXXXXXX
Grupo: XXX  Tutor D. XXXXXXXXXXXXXXXXXXXXXXXXXXXXX
      Número              Alumno
      ------    --------------------------
        1       XXXXXXXXXXXXXXXXXXXXXXXX
        2       XXXXXXXXXXXXXXXXXXXXXXXX
       ...                 ...
        m       XXXXXXXXXXXXXXXXXXXXXXXX
                 ...
```

Desarrollar el programa siguiendo la Metodología de Jackson.

1.° y 2.°. Estructuras de datos y correspondencias.

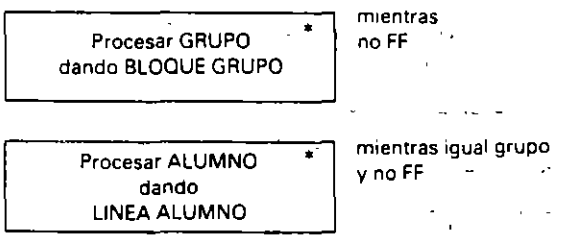


3.º Estructura del programa.



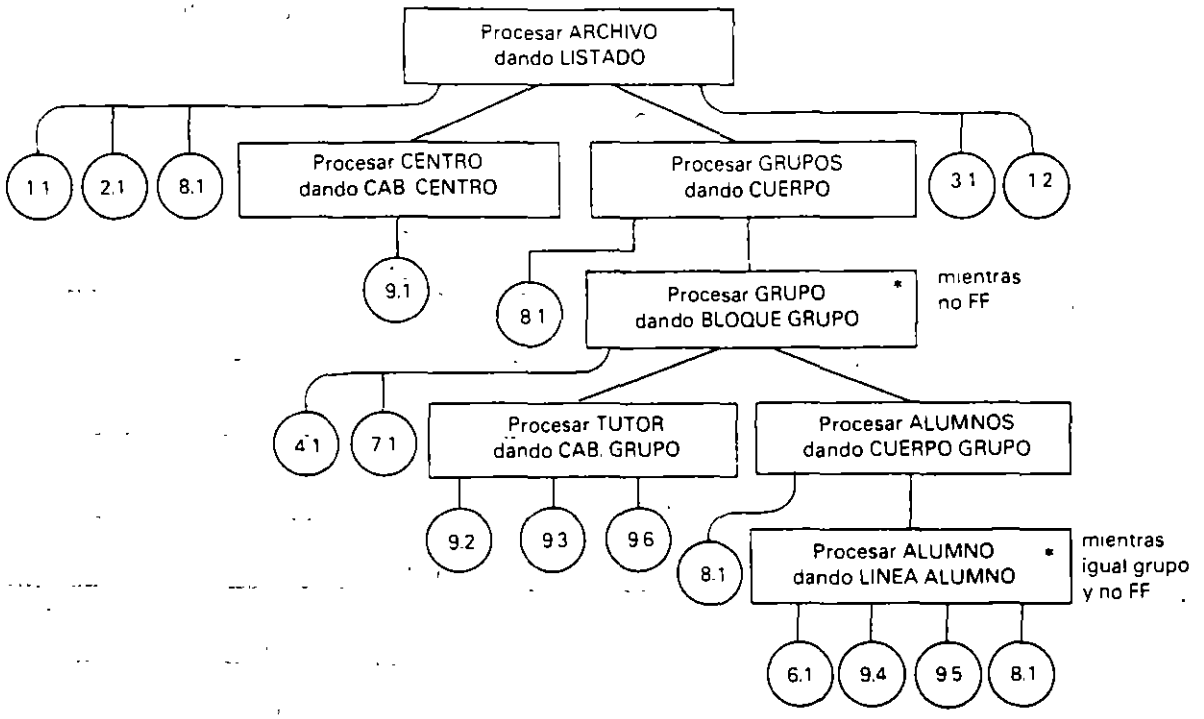
4.º Lista y asignación de operaciones y condiciones.

- Condiciones:



- Operaciones:

1.1 Inicio
1.2 Fin
2.1 Abrir ARCHIVO
3.1 Cerrar ARCHIVO
4.1 Inicializar CONTADOR de alumnos de un grupo
6.1 Contabilizar ALUMNO
7.1 Retener GRUPO
8.1 Leer REGISTRO
9.1 Escribir NOMBRE CENTRO
9.2 Escribir GRUPO
9.3 Escribir NOMBRE TUTOR
9.4 Escribir CONTADOR
9.5 Escribir NOMBRE ALUMNO
9.6 Escribir CABECERA ALUMNOS

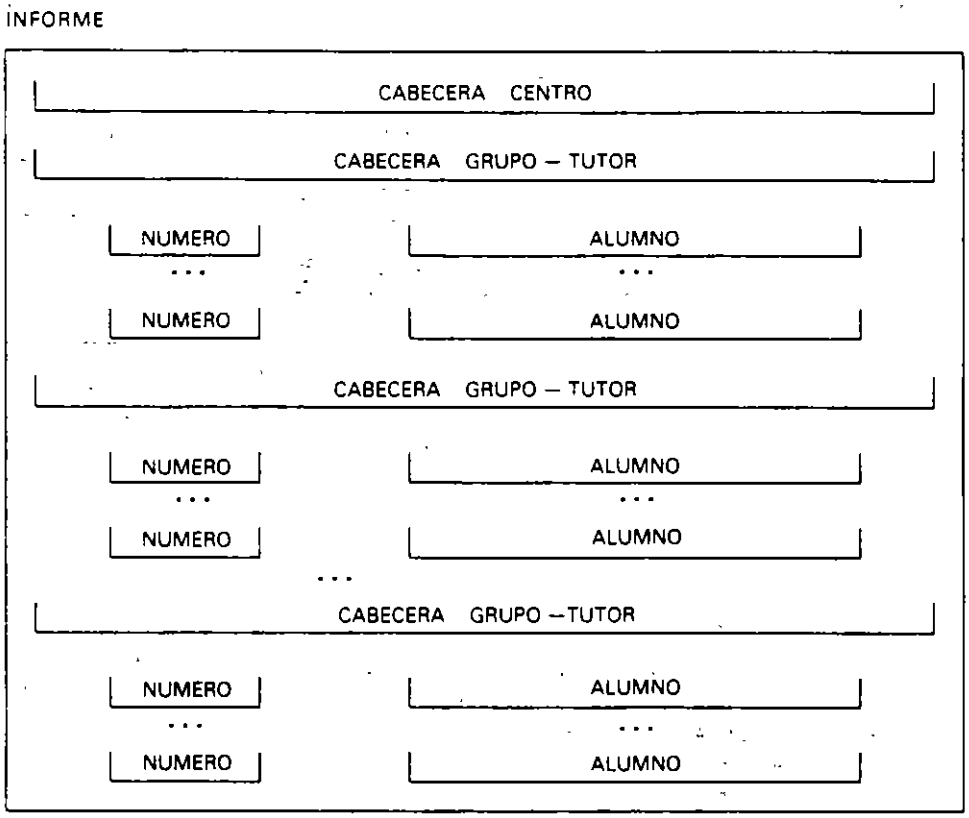


5.º Lógica esquematizada

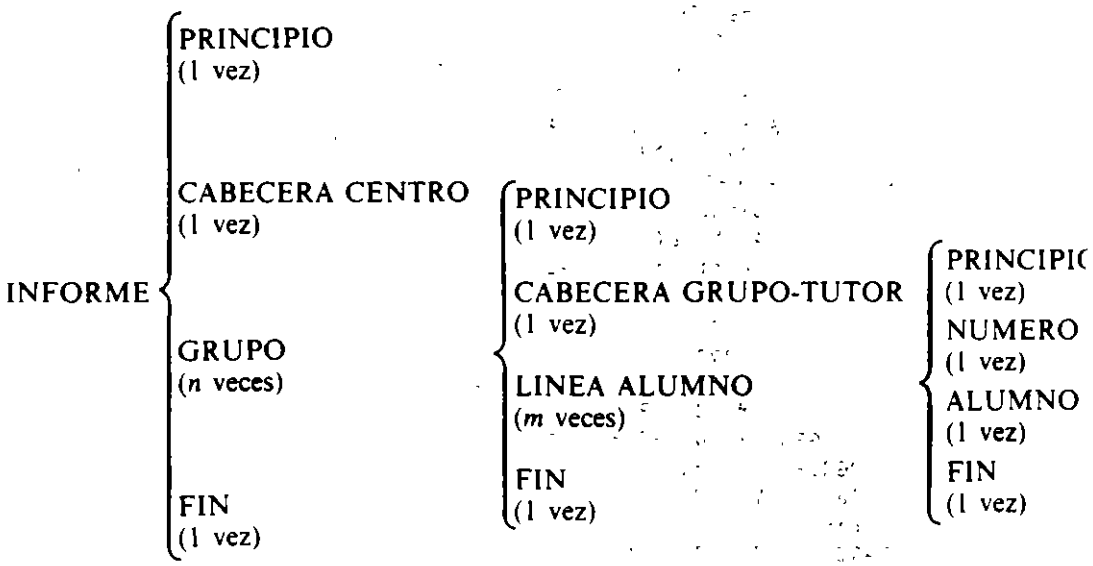
```
PARCHIVOLISTADO sec
  Inicio
  Abrir ARCHIVO
  Leer REGISTRO
  PCENTROCABCENTRO sec
    Escribir NOMBRE CENTRO
  PCENTROCABCENTRO end
  Leer REGISTRO
  PGRUPOSCUERPO iter mientras no FF
    PGRUPOBLOQUEGRUPO sec
      Inicializar CONTADOR
      Retener GRUPO
      PTUTORCABGRUPO sec
        Escribir GRUPO
        Escribir NOMBRE TUTOR
        Escribir CABECERA ALUMNOS
      PTUTORCABGRUPO end
      Leer REGISTRO
      PALUMNOSCUERPOGRUPO iter mientras igual grupo y no FF
        PALUMNOLINEAALUMNO sec
          Contabilizar ALUMNO
          Escribir CONTADOR
          Escribir NOMBRE ALUMNO
          Leer REGISTRO
        PALUMNOLINEAALUMNO end
      PALUMNOSCUERPOGRUPO end
    PGRUPOBLOQUEGRUPO end
  PGRUPOSCUERPO end
  Cerrar ARCHIVO
  Fin
PARCHIVOLISTADO end
```

4. *Desarrollar el ejercicio anterior siguiendo la Metodologia Warnier.*

1.º Estudio de los datos de salida. El informe que realiza este proceso tiene la siguiente estructura:

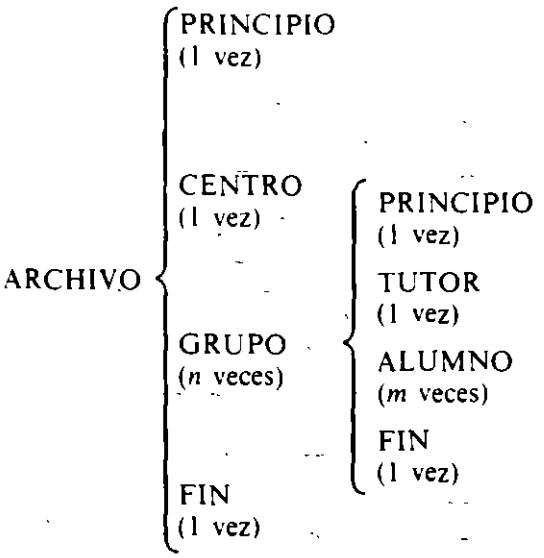


El cuadro de descomposición de la salida es:

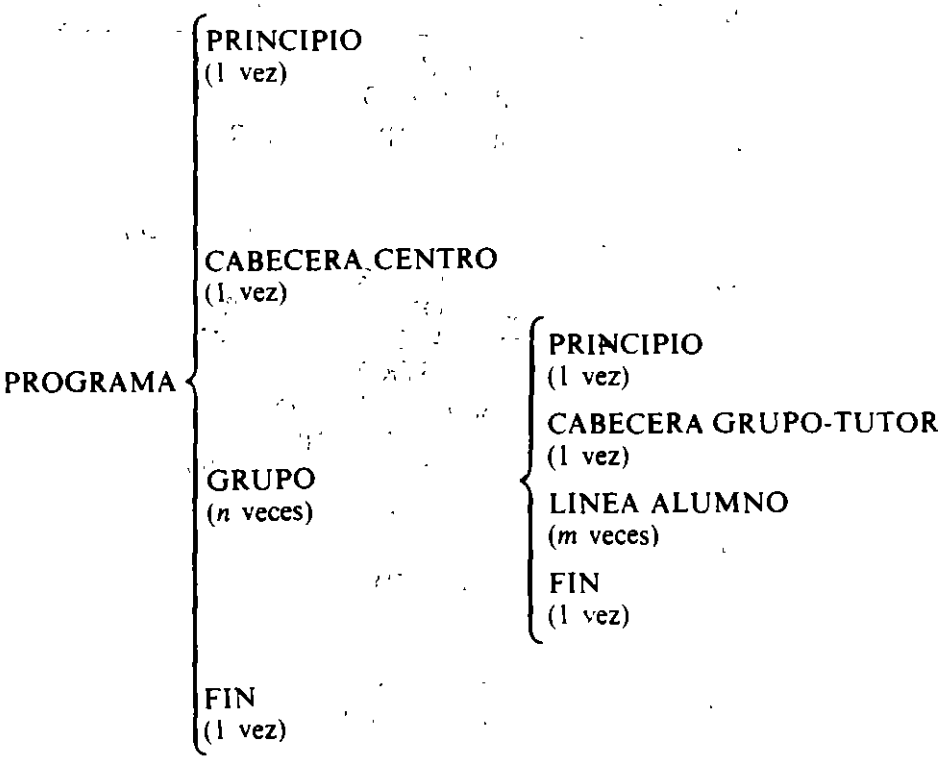


2.º Estudio de los datos de entrada.

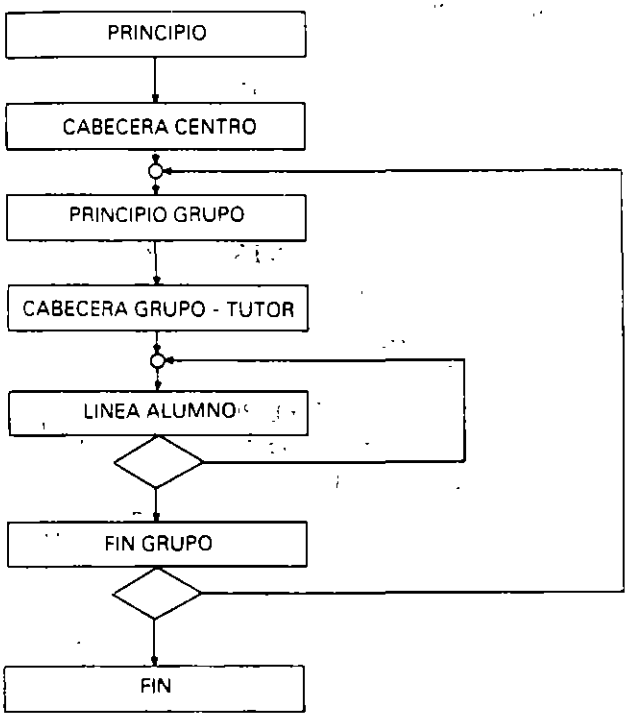
El cuadro de descomposición del archivo de entrada es el siguiente:



3.º Los archivos lógicos de salida y entrada tienen una correspondencia directa y no poseen ningún tipo de colisión; por tanto, el cuadro de descomposición de secuencias del programa es el siguiente:



4.° Organigrama de secuencias.



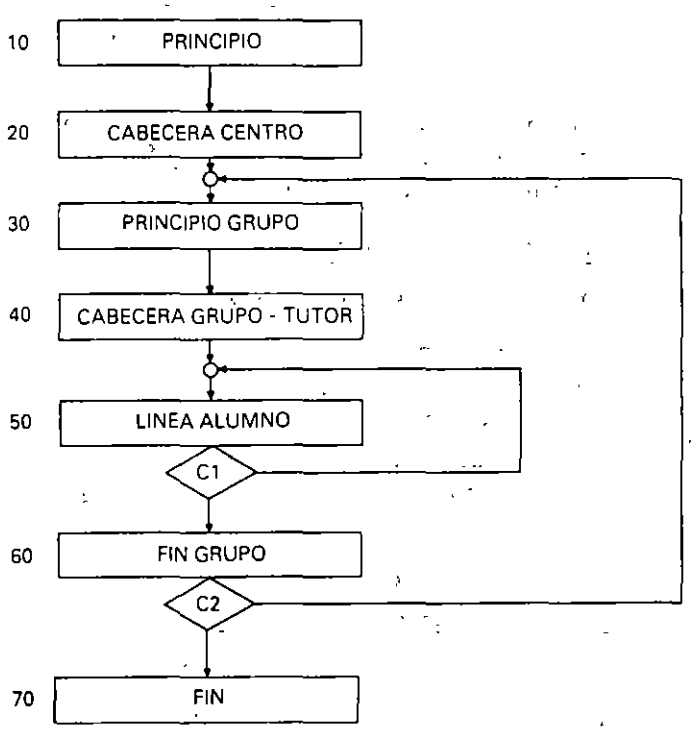
5.° Lista de instrucciones y asignación.

Secuencia	Instrucción
10	Abrir ARCHIVO Leer REGISTRO
20	Escribir NOMBRE CENTRO
30	Leer REGISTRO Retener GRUPO Inicializar CONTADOR ALUMNOS
40	Escribir NOMBRE GRUPO Escribir NOMBRE TUTOR Escribir CABECERA ALUMNOS Leer REGISTRO
50	Contabilizar ALUMNO Escribir CONTADOR Escribir NOMBRE ALUMNO Leer REGISTRO
60	Continuar
70	Cerrar ARCHIVO

Las condiciones son las siguientes:

C1 mientras igual GRUPO y no FF
C2 mientras no FF

Asignando operaciones y condiciones tendremos:



EJERCICIOS PROPUESTOS

1. Una empresa posee diversos almacenes en los que en cada uno de ellos existe un documento en el que se indica todo el material existente en el mismo. Cada almacén se divide en secciones, existiendo igualmente un documento con el contenido total de cada sección. Las secciones se dividen en galerías (pasillo largo), donde se encuentra un conjunto de piezas indicadas en un directorio situado al principio de cada galería.

   Las galerías contienen un número indeterminado de huecos (en los estantes correspondientes), donde en cada uno de ellos se depositan piezas de un solo tipo. Cada hueco tiene una etiqueta con las características y número de piezas que posee, asimismo cada pieza posee una etiqueta de identificación.

   Representar por Jackson la estructura de informaciones descrita.

2. Un centro de enseñanza tiene un archivo con información de todos sus alumnos, estando la misma clasificada por curso y sexo.

   Cada curso tiene un delegado que puede ser chico o chica y figura al principio de su curso.

   Representar el archivo siguiendo la técnica de Jackson.

3. Una empresa de servicios informáticos tiene una red de técnicos para la reparación de equipos. En todas las reparaciones se aplica una tarifa de 8500 ptas. hora más 3000 pesetas en concepto de desplazamiento, y siempre se aplica un incremento del 13 %

de IVA. Posee un archivo en el que se encuentran registradas todas las reparaciones que se han realizado durante un mes con el siguiente formato:

TECNICO	CLIENTE	DIRECCION	DURACION	FECHA

El archivo está ordenado por técnico y en cada uno de ellos por fecha, y pueden existir varios registros para un mismo técnico y fecha.

Se quiere obtener un listado según el siguiente formato:

```
          RESUMEN MENSUAL DE REPARACIONES

TECNICO: XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
      Cliente          Duración       Importe
 XXXXXXXXXXXXXXXXXX     XX.X          XXXXXXX
        ...             ...            ...
 XXXXXXXXXXXXXXXXXX     XX.X          XXXXXXX
                    ---------------------------
                    Total Técnico.....:XXXXXXXX (Incluido desplaz. e IVA)

                          . . .

TECNICO: XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
      Cliente          Duración       Importe
 XXXXXXXXXXXXXXXXXX     XX.X          XXXXXXX
        ...             ...            ...
 XXXXXXXXXXXXXXXXXX     XX.X          XXXXXXX
                    ---------------------------
                    Total Técnico.....:XXXXXXXX

     IMPORTE TOTAL MES (IVA incluido): XXXXXXXXXXX
```

Desarrollar el programa siguiendo la Metodología de Jackson y la Metodología de Warnier.

Indice analítico

Acción, 207
Acciones, 39
Acumuladores, 53
Algoritmo, 3, 40, 71
Algoritmos de resolución, 5
Altas, 237
Alternativa, 6
Análisis fase de, 2
Aplicación informática, 2
  ciclo de vida, 2
Arbol binario:
  completo, 323
  de búsqueda, 326
  equilibrado, 323
  recorridos, 325
  recorridos en inorden, 325
  recorridos en postorden, 325
  recorridos en preorden, 325
Arbol *n*-ario, 324
Arboles, 321
Arboles binarios, 323
Archivo lógico:
  de entrada, 392
  de salida, 391
Archivos,
  actualización de, 239, 270, 280
  actualización interactiva de, 286
  borrado de, 239
  caracteristica de los, 236
  clasificación de, 239, 277
  clasificación de raíz, 279
  clasificación por mezcla directa, 277
  clasificación por mezcla equilibrada, 278
  concatenación, 239
  consulta, 239
  copia, 239
  creación, 239
  de constantes, 237
  de maniobra, 237
  de movimientos, 237
  de situación, 237
  desordenados, búsqueda, 271
  directos, actualización, 283
  directos, lectura-escritura, 250
  fusión de, 239
  históricos, 237
  indexados, actualización, 285
  indexados, lectura-escritura, 255
  instrucciones para manejo, 239
  maestros, 237
  métodos de tratamiento de, 270
  mezcla con registro centinela, 274
  mezcla controlada por clave máxima, 275
  mezcla controlada por fin de archivo, 276
  mezcla de, 239, 270, 274
  operaciones sobre, 239
  ordenados, búsqueda, 271
  organización de, 237
  partición de, 239, 272
  partición en secuencias, 273
  partición por contenido, 272
  permanentes, 236
  reorganización de, 238
  ruptura de secuencia, 289
  secuenciales, actualización, 280
  secuenciales, creación, 240
  secuenciales, lectura, 240
  sincronización de, 270, 292
Area de excedentes, 238
Area de índices, 238
Area primaria, 238
Asignación, instrucciones de, 42, 66

Bajas, 237
Bloques, 203
Búsqueda, 270
Búsqueda, algoritmos de, 132
Búsqueda binaria, 137
Búsqueda, clave, 270
Búsqueda dicotómica, 137
Búsqueda lineal, 132
Búsqueda lineal en un vector; 133
Búsqueda lineal en un vector ordenado, 134
Búsqueda lineal en una matriz, 135

Campo clave, 236
Campos, 235
Checklist, 374
Cima, 314
Clasificación, 270
Clasificación, algoritmos de, 132
Clave de ordenación, 277, 306
Clave de ruptura, 289
Clave de sincronización, 292
Codificación, fase de, 3
Codificación, hoja de, 3
Código autodocumentado, 7
Colas, 318
Comentarios, 6, 51, 70

Compilación, fase de, 4
Compiladores, 4
Completitud, 338
Computadora, 1
Condición, 46
Conmutadores, 54
Constantes, 10
Contador asociado, 46
Contadores, 52
Control, instrucciones de, 43, 66
Cuadro de descomposición de secuencias, 388, 392

Dash count, 341
Datos, 39
  de ensayo, 5
  de prueba, 4
  de entrada de, 40
  salida de, 41
Declaración, instrucciones de, 41
Declaraciones, bloque de, 39
Depuración, 4
Diagrama propio, 204
Diagramas de flujo, 15
Diseño, 5
  descendente, 174
  top-down, 174, 207
Documentación de los programas, 6
  interna, 6

Edición, fase de, 4
Editor, 4
Eficiencia, 5
Entidad simple, 361
Entidades, 361
Entorno, 70, 71
Entrada, instrucción de, 41, 66
Error, mensaje de, 4
Errores, 4
  de compilación, 4
  de ejecución, 5
  de especificación, 5
  de lógica, 5
  sintácticos, 4
Especificación, 2
Especificaciones, 5
Especificaciones del análisis, 7
Estructura alternativa, 208
  de datos, 8, 93
  de datos dinámica, 93, 304
  de datos estática, 93
  de datos externa, 93, 235
  de datos interna, 93
  hasta (UNTIL), 211
  iterar (LOOP), 211
  mientras (WHILE), 210
  para (FOR), 209
  repetitiva, 209
  secuencial, 208
Estructuras básicas, 207, 388
Explotación, fase de, 4
Expresiones, 10
  alfanuméricas, 11
  booleanas, 11
  numéricas, 10

Factor de bloqueo, 236
Fiabilidad, 5
Ficheros, 234
Final de archivo; comprobación, 246
Funciones, 51

Identificadores, 7
Importancia de una condición, 341
  de una regla, 341
Inserción directa, ordenación por, 139
Instalación, 5
Instrucción alternativa doble, 44, 67
  alternativa múltiple, 45, 67
  alternativa simple, 45, 67
  iterar (LOOP), 48, 69
  mientras (WHILE), 46, 68
  para (FOR), 48, 69
  repetir (REPEAT), 47, 68
Instrucciones, 39
  alternativas, 43
  bloque de, 39
  compuestas, 51, 69
  primitivas, 41
  repetitivas, 46, 68
Intercambio con incrementos decrecientes, ordenación por, 151
Intercambio directo con test, ordenación por, 146
Intercambio directo, ordenación por, 143, 147
Intérpretes, 4
Interruptores, 54
Iteración, 362

Legibilidad, 5
Lenguaje:
  ensamblador, 57
  máquina, 57
Lenguajes:
  convencionales, 57
  conversacionales, 57
  de alto nivel, 57
  de bajo nivel, 57
  estructurados, 57
  funcionales, 57
  imperativos, 57
  interactivos, 57
  orientados a batch, 57
Líneas de flujo, 19, 23

Linker, 4
Listas, 306
  contiguas, 307
  densas, 307
  encadenadas, 311
  enlazadas, 311
Literales, 9
Lógica esquematizada, 361, 375

Mantenimiento, fase de, 4
Manual de mantenimiento, 7
  de usuario, 7
Matrices, 97, 108
Método de Bertini, 218
  de Chapin, 224
  de Jackson, 215
  de la baraja, 139
  de la burbuja, 143
  de la sacudida, 147
  de Nassi/Shneiderman, 224
  de Pollack, 341, 343
  de Tabourier, 221
  de Warnier, 212
  Shell, 151
Metodología Jackson, 359
  backtraking, 380
  colisiones, 381
  collating, 378
  errores e invalidez, 377
  fases de la, 363
  inversión de programas, 381
Metodología Warnier, fases de la, 391
Modificabilidad, 5
Modificaciones, 237
Módulos, 6, 203
Montador, 4

Nodo, 306, 321
Normalización, 24

Objetos, 7, 39, 70
  globales, 179
  locales, 179
Operador NO, 11
Operador O, 12
Operador Y, 12
Operadores, 11
Operadores alfanuméricos, 11
  aritméticos, 11
  lógicos, 11
  orden de evaluación de, 12
  paréntesis, 11
  relacionales, 11
Ordenación de tablas, 138
  por importancias, 341
Ordenador, 1
Ordinograma, 3, 15, 21
Ordinograma, plantilla de, 24
Organigrama, 15, 16
  de secuencias, 388, 393
  aleatoria, 238
  directa, 238
  secuencial, 237
  secuencial indexada, 238

Palabra reservada, 4
Parámetros, 180
  actuales, 180
  formales, 180
  paso de, 182
  paso por referencia, 182
  paso por valor, 182
Párrafos, 51
Pilas, 314
Poliedros, 102, 108
Portabilidad, 5
Proceso, 40
  batch, 39
Programa, 3, 71
  fuente, 4
  objeto, 4
  principal, 175
  propio, 204
Programación estructurada, 6, 203
  estructurada, herramientas de, 207
  estructurada, técnicas de, 203
  fase de, 3
  lenguaje de, 57, 65, 77
  metodología de la, 6
  modular, 6
Programas, 5
  alternativos, 56
  cíclicos, 56
  de diseño, 56
  de gestión, 56
  de inteligencia artificial, 57
  de simulación, 56
  educativos, 56
  equivalencia de, 205
  lineales, 56
  metodologías de desarrollo de, 359
  técnico-científicos, 56
Prueba de ejecución, fase de, 4
Pseudocódigo, 3, 65
Puntero, 304

Rango, 46
Recursividad, 184
Recursos abstractos, 207
Redundancias, 338
  congruentes, 338
  incongruentes, 338
Registro:
  buffer, 236
  consulta de un, 239

físico (bloque), 236
inserción de un, 239
lógico (registro), 235
modificación de un, 239
Registro, supresión de un, 239
Registros, 235
Regla de decisión, 334
de decisión compuesta, 334
de decisión simple, 334
Reglas AND, 335
ELSE, 335
OR, 335
Requisitos de calidad, 5
Resultados, salida de, 40
Rupturas de secuencia, 270
instrucciones de, 50

Salida, instrucción de, 42, 66
Salto condicional, instrucción de, 51
incondicional, instrucción de, 50
Secuencia, 6, 66, 362
lógica, 388
Selección, 362
Selección directa, ordenación por, 141
Símbolo de comentario, 24
Símbolos de conexión, 23
de decisión, 23
de operación, 22
de proceso, 19
de soporte, 17
Simplificación, 339
Sistema informático, 2
Situación, 334
Sort, 270
Subcampos, 235
Subprogramas, 51, 174, 175
declaración de, 175
externos, 177
externos, 4
internos, 176
llamadas a, 175
Subrutinas, 51
Switches, 54

Tabla, 93
componentes de una, 93
dimensión de una, 93
índices de una, 93
longitud de una, 93
tamaño de una, 93
tipo de una, 93
tratamiento secuencial de una, 110
Tablas:
bidimensionales, 97, 108
de decisión, 333
de decisión, acciones, 334
de decisión binarias, 336
de decisión, clasificación de las, 336
de decisión, condiciones, 333
de decisión, entrada de condiciones, 334
de decisión, estructura de una, 333
de decisión, mixtas, 337
de decisión múltiples, 337
de decisión, salida de acciones, 334
de verdad, 11
multidimensionales, 102, 108
unidimensionales, 95, 107
Teorema de la estructura, 203, 204, 205
Tipo booleano, 9
Tipo carácter, 9
Tipo numérico entero, 8
Tipo numérico real, 9
Tipos de datos, 8
Tipos de datos estructurados, 8
Tipos de datos simples, 8
Tipos de reglas, 335
Tratamiento, 334

Variable de control, 48
Variable dinámica, 304
Variables, 10
Variables de enlace, 180
Vectores, 95, 107