CAPÍTULO 10

Ordenación, búsqueda e intercalación

INTRODUCCIÓN

Las computadoras emplean una gran parte de su tiempo en operaciones de *búsqueda*, *clasificación* y *mezcla de datos*. Las operaciones de cálculo numérico y sobre todo de gestión requieren normalmente operaciones de clasificación de los datos: ordenar fichas de clientes por orden alfabético, por direcciones o por código postal. Existen dos métodos de ordenación: *ordenación interna* (de **arrays**, **arreglos**) y *ordenación externa* (archivos). Los arrays se almacenan en la memoria interna o central, de acceso aleatorio y directo, y por ello su gestión es rápida. Los *archivos* se sitúan adecuadamente en dispositivos de almacenamiento externo que son más lentos y basados en dispositivos mecánicos: cintas y discos magnéticos. Las técnicas de ordenación, búsqueda y mezcla son muy importantes y el lector deberá dedicar especial atención al conocimiento y aprendizaje de los diferentes métodos que en este capítulo se analizan.

10.1. INTRODUCCIÓN

Ordenación, búsqueda y, en menor medida, *intercalación* son operaciones básicas en el campo de la documentación y en las que, según señalan las estadísticas, las computadoras emplean la mitad de su tiempo.

Aunque su uso puede ser con vectores (arrays) y con archivos, este capítulo se referirá a vectores.

La *ordenación (clasificación)* es la operación de organizar un conjunto de datos en algún orden dado, tal como creciente o decreciente en datos numéricos, o bien en orden alfabético directo o inverso. Operaciones típicas de ordenación son: lista de números, archivos de clientes de banco, nombres de una agenda telefónica, etc. En síntesis, la ordenación significa poner objetos en orden (orden numérico para los números y alfabético para los caracteres) ascendente o descendente.

Por ejemplo, las clasificaciones de los equipos de fútbol de la liga en la 1.ª división española se pueden organizar en orden alfabético creciente/decreciente o bien por clasificación numérica ascendente/descendente.

Los nombres de los equipos y los puntos de cada equipo se almacenan en dos vectores:

```
equipo [1] = 'Real Madrid'          puntos [1] = 10
equipo [2] = 'Barcelona'            puntos [2] = 14
equipo [3] = 'Valencia'             puntos [3] =  8
equipo [4] = 'Oviedo'               puntos [4] = 12
equipo [5] = 'Betis'                puntos [5] = 16
```

Si los vectores se ponen en orden decreciente de puntos de clasificación:

```
equipo [5] = 'Betis'                puntos [5] = 16
equipo [2] = 'Barcelona'            puntos [2] = 14
equipo [4] = 'Oviedo                puntos [4] = 12
equipo [1] = 'Real Madrid'          puntos [1] = 10
equipo [3] = 'Valencia'             puntos [3] =  8
```

Los nombres de los equipos y los puntos conseguidos en el campeonato de Liga anterior, ordenados de modo alfabético serían:

```
equipo [1] = 'Barcelona'            puntos [1] =  5
equipo [2] = 'Cádiz'                puntos [2] = 13
equipo [3] = 'Málaga'               puntos [3] = 12
equipo [4] = 'Oviedo'               puntos [4] =  8
equipo [5] = 'Real Madrid'          puntos [5] =  4
equipo [6] = 'Valencia'             puntos [6] = 16
```

o bien se pueden situar en orden numérico decreciente:

```
equipo [6] = 'Valencia'             puntos [6] = 16
equipo [2] = 'Cádiz'                puntos [2] = 13
equipo [3] = 'Málaga'               puntos [3] = 12
equipo [4] = 'Oviedo'               puntos [4] =  8
equipo [1] = 'Barcelona'            puntos [1] =  5
equipo [5] = 'Real Madrid'          puntos [5] =  4
```

Los vectores anteriores comienzan en orden alfabético de equipos y se reordenan en orden descendente de "puntos". El listín telefónico se clasifica en orden alfabético de abonados; un archivo de clientes de una entidad bancaria normalmente se clasifica en orden ascendente de números de cuenta. El propósito final de la clasificación es facilitar la manipulación de datos en un vector o en un archivo.

Algunos autores diferencian entre un conjunto o *vector clasificado (sorted)* y *vector ordenado (ordered set)*. Un *conjunto ordenado* es aquel en el que el orden de aparición de los elementos afecta al significado de la estructura completa de datos: puede estar clasificado, pero no es imprescindible. Un *conjunto clasificado* es aquel en que los valores de los elementos han sido utilizados para disponerlos en un orden particular: es, probablemente, un conjunto ordenado, pero no necesariamente.

Es importante estudiar la clasificación por dos razones. Una es que la clasificación de datos es tan frecuente que todos los usuarios de computadoras deben conocer estas técnicas. La segunda es que es una aplicación que se puede describir fácilmente, pero que es bastante difícil conseguir el diseño y escritura de buenos algoritmos.

La clasificación de los elementos numéricos del vector

```
7,  3,  2,  1,  9,  6,  7,  5,  4
```

en orden ascendente producirá

```
1,  2,  3,  4,  5,  6,  7,  7,  9
```

Obsérvese que pueden existir elementos de igual valor dentro de un vector.

Existen muchos algoritmos de clasificación, con diferentes ventajas e inconvenientes. Uno de los objetivos de este capítulo y del Capítulo 11 es el estudio de los métodos de clasificación más usuales y de mayor aplicación.

La *búsqueda* de información es, al igual que la ordenación, otra operación muy frecuente en el tratamiento de información. La búsqueda es una actividad que se realiza diariamente en cualquier aspecto de la vida: búsqueda de palabras en un diccionario, nombres en una guía telefónica, localización de libros en una librería. A medida que la información se almacena en una computadora, la recuperación y búsqueda de esa información se convierte en una tarea principal de dicha computadora.

10.2. ORDENACIÓN

En un vector es necesario, con frecuencia, clasificar u ordenar sus elementos en un orden particular. Por ejemplo, clasificar un conjunto de números en orden creciente o una lista de nombres por orden alfabético.

La clasificación es una operación tan frecuente en programas de computadora que una gran cantidad de algoritmos se han diseñado para clasificar listas de elementos con eficacia y rapidez.

La elección de un determinado algoritmo depende del tamaño del vector o **array** (**arreglo**) a clasificar, el tipo de datos y la cantidad de memoria disponible.

La *ordenación* o *clasificación* es el proceso de organizar datos en algún orden o secuencia específica, tal como creciente o decreciente para datos numéricos o alfabéticamente para datos de caracteres.

Los *métodos de ordenación* se dividen en dos categorías:

- **Ordenación de vectores, tablas (arrays o arreglos).**
- **Ordenación de archivos.**

La ordenación de arrays se denomina también *ordenación interna*, ya que se almacena en la memoria interna de la computadora de gran velocidad y acceso aleatorio. La ordenación de archivos se suele hacer casi siempre sobre soportes de almacenamiento externo, discos, cintas, etc., y, por ello, se denomina también *ordenación externa*. Estos dispositivos son más lentos en las operaciones de entrada/salida, pero, por el contrario, pueden contener mayor cantidad de información.

Ordenación interna: clasificación de los valores de un vector según un orden en memoria central: rápida.
Ordenación externa: clasificación de los registros de un archivo situado en un soporte externo: menos rápido.

EJEMPLO

Clasificación en orden ascendente del vector.

```
7, 3, 2, 1, 9, 6, 7, 5, 4
```

se obtendrá el nuevo vector

```
1, 2, 3, 4, 5, 6, 7, 7, 9
```

Los métodos de clasificación se explicarán aplicados a vectores (arrays unidimensionales), pero se pueden extender a matrices o tablas (arrays o arreglos bidimensionales), considerando la ordenación respecto a una fila o columna.

Los *métodos directos* son los que se realizan en el espacio ocupado por el array. Los más populares son:

- *Intercambio.*
- *Selección.*
- *Inserción.*

10.2.1. Método de intercambio o de burbuja

El algoritmo de clasificación de *intercambio o de la burbuja* se basa en el principio de comparar pares de elementos adyacentes e intercambiarlos entre sí hasta que estén todos ordenados.

Supongamos que se desea clasificar en orden ascendente el vector o lista

```
 50        15        56        14        35        1         12        9
A[1]      A[2]      A[3]      A[4]      A[5]      A[6]      A[7]      A[8]
```

Los pasos a dar son:

1. Comparar `A[1]` y `A[2]`; si están en orden, se mantienen como están, en caso contrario se intercambian entre sí.
2. A continuación se comparan los elementos `2` y `3`; de nuevo se intercambian si es necesario.
3. El proceso continúa hasta que cada elemento del *vector* ha sido comparado con sus elementos adyacentes y se han realizado los intercambios necesarios.

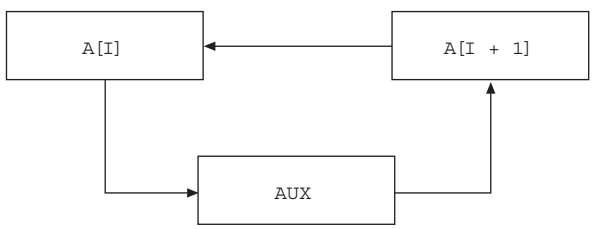
El método expresado en pseudocódigo en el primer diseño es:

```
desde I ← 1 hasta 7 hacer
  si elemento[I] > elemento[I + 1] entonces
    intercambiar (elemento[I], elemento [I + 1])
  fin_si
fin_desde
```

La acción *intercambiar* entre sí los valores de dos elementos `A[I]`, `A[I+1]` es una acción compuesta que contiene las siguientes acciones, considerando una variable auxiliar `AUX`.

```
AUX ← A[I]
A[I] ← A[I+1]
A[I+1] ← AUXI
```

En realidad, el proceso gráfico es



El elemento cuyo valor es mayor sube posición a posición hacia el final de la lista, al igual que las burbujas de aire en un depósito o botella de agua. Tras realizar un recorrido completo por todo el vector, el elemento mencionado habrá subido en la lista y ocupará la última posición. En el segundo recorrido, el segundo elemento llegará a la penúltima, y así sucesivamente.

En el ejercicio citado anteriormente los sucesivos pasos con cada una de las operaciones se muestran en las Figuras 10.1 y 10.2.

vector inicial	*1.ª comp.*	*2.ª comp.*	...				
A[1]	50	15	15	15	15	15	15
A[2]	15	50	50	50	50	50	50
A[3]	56	56	56	14	14	14	14
A[4]	14	14	14	56	35	35	35
A[5]	35	35	35	35	56	1	1
A[6]	1	1	1	1	1	56	12
A[7]	12	12	12	12	12	12	56
A[8]	9	9	9	9	9	9	9

Figura 10.1. Método de la burbuja (paso 1).

Si se efectúa $n - 1$ veces la operación sobre una tabla de n valores se tiene ordenada la tabla. Cada operación requiere $n - 1$ comprobaciones o test y como máximo $n - 1$ intercambios. La ordenación total exigirá un máximo de $(n - 1) * (n - 1) = (n - 1)^2$ intercambios de elementos.

Los estados sucesivos del vector se indican en la Figura 10.2:

Estado inicial	Después de paso 1	Después de paso 2
50	15	15
15	50	14
56	14	35
14	35	1
35	1	12
1	12	9
12	9	50
9	56	56

Figura 10.2. Método de la burbuja (paso 2).

EJEMPLO 10.1

Describir los diferentes pasos para clasificar en orden ascendente el vector

```
72  64  50  23  84  18  37  99  45  8
```

Las sucesivas operaciones en cada uno de los pasos necesarios hasta obtener la clasificación final se muestra en la Tabla 10.1

Tabla 10.1. Pasos necesarios de la ordenación por burbuja

Vector desordenado	**Número de paso**						**Fin de clasificación**		
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
72	64	50	23	23	18	18	18	18	8
64	50	23	50	18	23	23	23	8	18
50	23	64	18	37	37	37	8	23	23
23	72	18	37	50	45	8	37	37	37
84	18	37	64	45	8	45	45	45	45
18	37	72	45	8	50	50	50	50	50
37	84	45	8	64	64	64	64	64	64
99	45	8	72	72	72	72	72	72	72
45	8	84	84	84	84	84	84	84	84
8	99	99	99	99	99	99	99	99	99

Método 1

El algoritmo se describirá, como siempre, con un diagrama de flujo y un pseudocódigo.

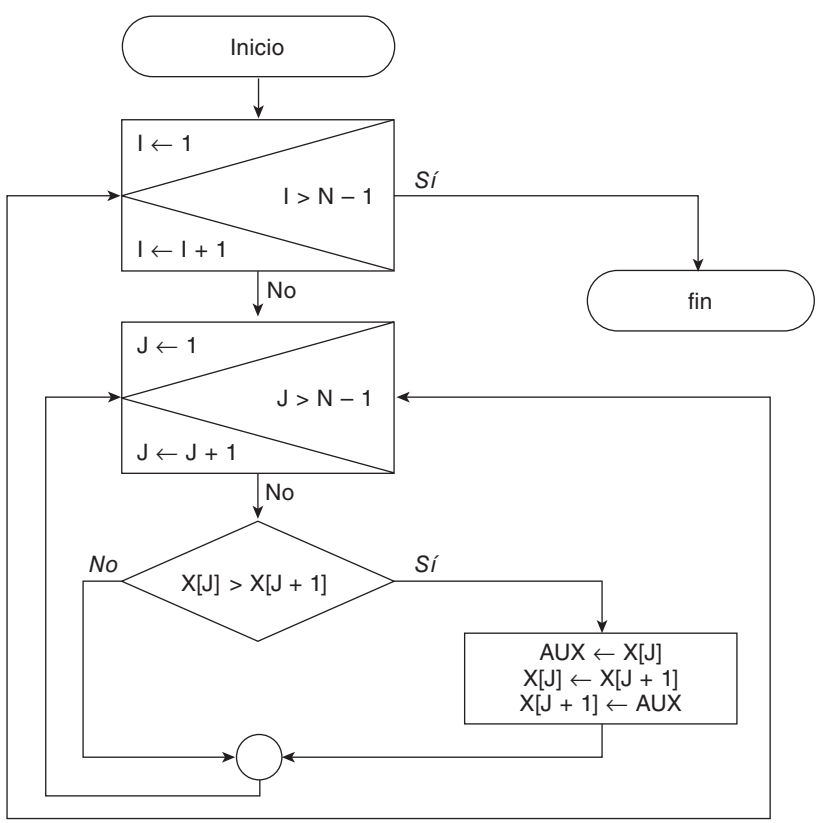
Pseudocódigo

```
algoritmo burbuja1
//incluir las declaraciones precisas//
inicio
  //lectura del vector//
  desde i ← 1 hasta N hacer
    leer(X[I])
  fin_desde
  //clasificación del vector
  desde I ←1 hasta N-1 hacer
    desde J ←1 hasta J ← N-1 hacer
      si X[j] > X[J+1] entonces
      //intercambiar
        AUX ← X[J]
        X[J] ←X[J+1]
        X[J+1] ←AUXI
      fin_si
    fin_desde
  fin_desde
  //imprimir lista clasificada
  desde J ←1 hasta N hacer
    escribir(X[J])
  fin_desde
fin
```

Diagrama de flujo 10.1

Para clasificar el vector completo se deben realizar las sustituciones correspondientes `(N-1) * (N-1)` o bien N^2-2N+1 veces. Así, en el caso de un vector de cien elementos `(N = 100)` se deben realizar casi 10.000 iteraciones.

El algoritmo de clasificación es:



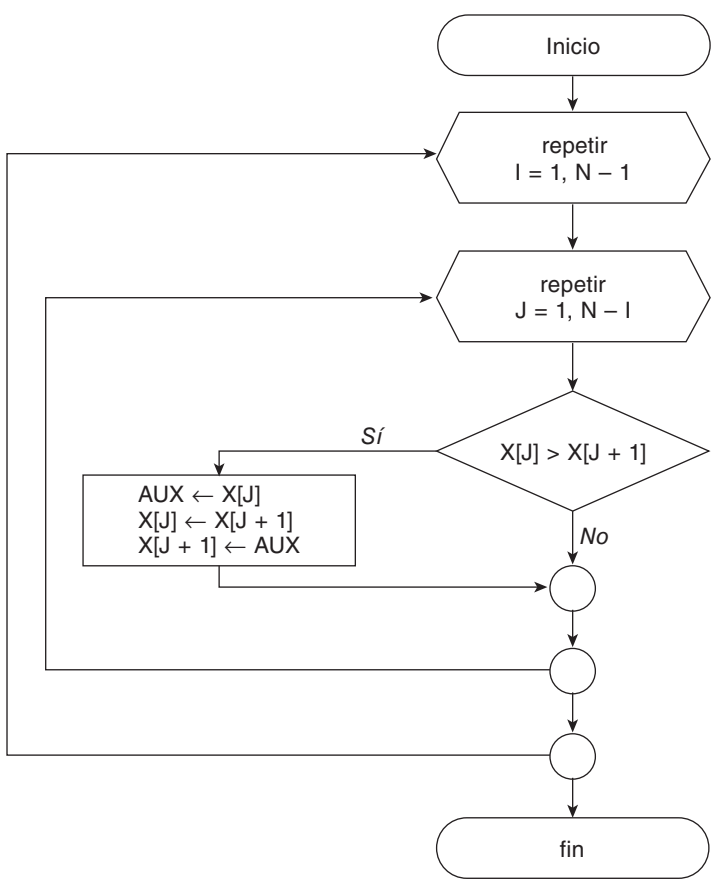
Método 2

Se puede realizar una *mejora en la velocidad de ejecución del algoritmo*. Obsérvese que en el primer recorrido del vector (cuando `I = 1`) el valor mayor del vector se mueve al último elemento `X[N]`. Por consiguiente, en el siguiente paso no es necesario comparar `X[N – 1]` y `X[N]`. En otras palabras, el límite superior del bucle `desde` puede ser `N – 2`. Después de cada paso se puede decrementar en uno el límite superior del bucle `desde`. El algoritmo sería:

Pseudocódigo

```
algoritmo burbuja2
  //declaraciones
inicio
  //...
  desde I ← 1 hasta N-1 hacer
    desde J ← 1 hasta N-I hacer
      si X[J] > X[J+1] entonces
        AUX ← X[J]
        X[J] ← X[J+1]
        X[J+1] ← AUX
      fin_si
    fin_desde
  fin_desde
fin
```

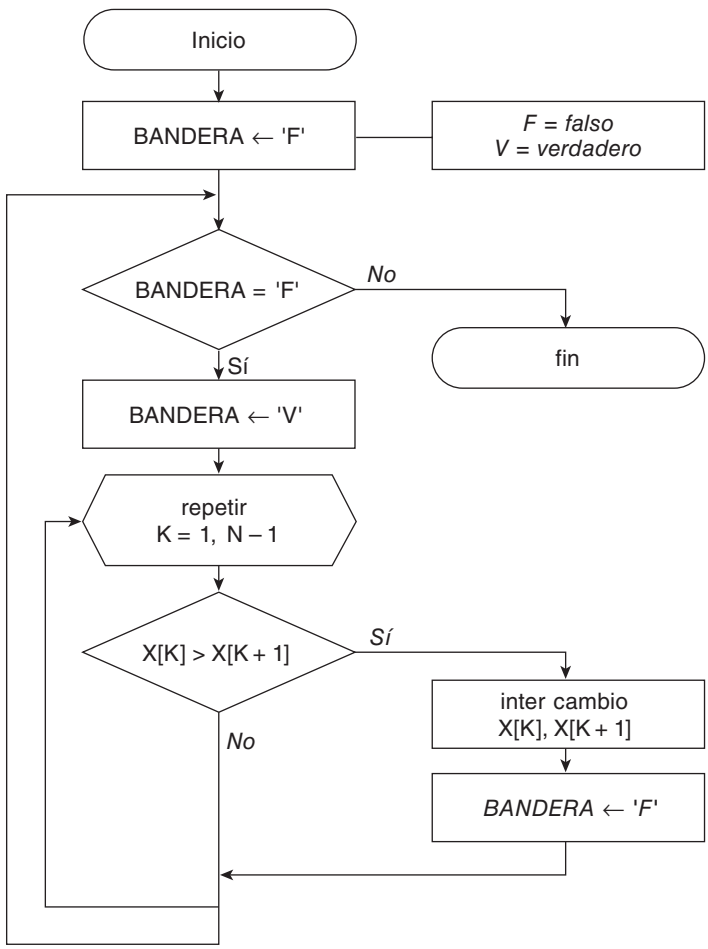
Diagrama de flujo 10.2



Método 3 (uso de una bandera/indicador)

Mediante una **bandera/indicador** o **centinela** (**switch**) o bien una variable lógica, se puede detectar la presencia o ausencia de una condición. Así, mediante la variable `BANDERA` se representa *clasificación terminada* con un valor verdadero y *clasificación no terminada* con un valor falso.

Diagrama de flujo 10.3



Pseudocódigo

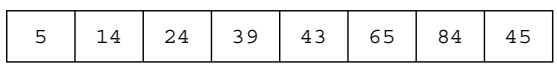
```
algoritmo burbuja 3
  //declaraciones
inicio
  //lectura del vector
  BANDERA ← 'F' //    F, falso; V, verdadero
i ← 1
  mientras (BANDERA = 'F') Y (i < N ) hacer
    BANDERA ← 'V'
    desde K ← 1 hasta N-i hacer
      si X[K] > X[K+1] entonces
        intercambiar(X[K],X[K + 1])
        //llamada a procedimiento intercambio
        BANDERA ← 'F'
      fin_si
```

```
    fin_desde
    i ← i+1
  fin_mientras
fin
```

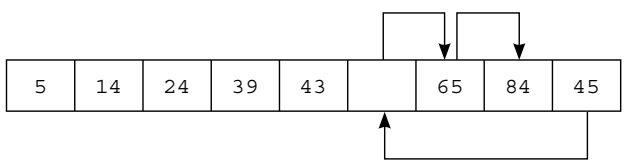
10.2.2. Ordenación por inserción

Este método consiste en insertar un elemento en el vector en una parte ya ordenada de este vector y comenzar de nuevo con los elementos restantes. Por ser utilizado generalmente por los jugadores de cartas se le conoce también por el nombre de *método de la baraja*.

Así, por ejemplo, suponga que tiene la lista desordenada



Para insertar el elemento `45`, habrá que insertarlo entre `43` y `65`, lo que supone desplazar a la derecha todos aquellos números de valor superior a `45`, es decir, saltar sobre `65` y `84`.



El método se basa en comparaciones y desplazamientos sucesivos. El algoritmo de clasificación de un vector `X` para `N` elementos se realiza con un recorrido de todo el vector y la inserción del elemento correspondiente en el lugar adecuado. El recorrido se realiza desde el segundo elemento al *n-ésimo*.

```
desde i ←  2 hasta N hacer
  //insertar X[i] en el lugar
  //adecuado entre X[1]..X[i-1])
fin_desde
```

Esta acción repetitiva —*insertar*— se realiza más fácilmente con la inclusión de un valor centinela o bandera (`SW`).

Pseudocódigo

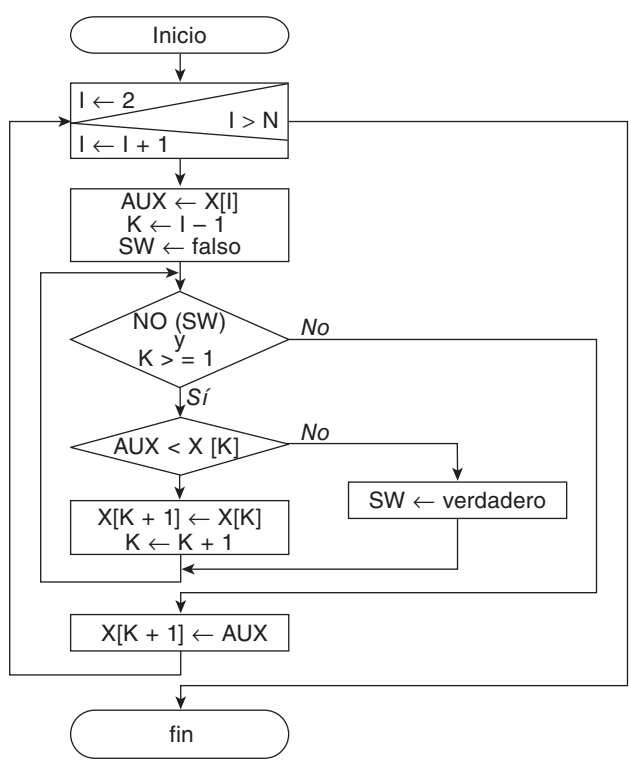
```
algoritmo clas_insercion1
//declaraciones
inicio
  .....
  //ordenacion
  desde I ← 2 hasta N hacer
    AUXI ← X[I]
    K ← I-1
    SW ← falso
    mientras no (SW) y (K >= 1) hacer
      si AUXI < X[K] entonces
        X[K+1] ← X[K]
        K ← K-1
      si_no
        SW ← verdadero
      fin_si
    fin_mientras
```

```
      X[K+1] ← AUXI
    fin_desde
  fin
```

Algoritmo inserción mejorado

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

```
algoritmo clas_insercion_binaria
//declaraciones
inicio
  //...
  desde I ← 2 hasta N hacer
    AUX ← X[I]
    P ← 1    //primero
    U ← I-1  //último
    mientras P <= U hacer
      C ← (P+U) div 2
      si AUX < X[C] entonces
        U ← C-1
      si_no
        P ← C+1
      fin_si
    fin_mientras
    desde K ← I-1 hasta P decremento 1 hacer
      X[K+1] ← X[K]
    fin_desde
      X[P] ← AUX
  fin_desde
fin
```



Número de comparaciones

El cálculo del número de comparaciones $F(n)$ que se realiza en el algoritmo de inserción se puede calcular fácilmente.

Consideraremos el elemento que ocupa la posición X en un vector de n elementos, en el que los $X - 1$ elementos anteriores se encuentran ya ordenados ascendentemente por su clave.

Si la clave del elemento a insertar es mayor que las restantes, el algoritmo ejecuta sólo una comparación; si la clave es inferior a las restantes, el algoritmo ejecuta $X - 1$ comparaciones.

El número de comparaciones tiene por media $X/2$.

Veamos los casos posibles.

Vector ordenado en origen

Comparaciones mínimas

$$(n - 1)$$

Vector inicialmente en orden inverso

Comparaciones máximas

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

ya que

$$(n-1) + (n-2) + ... + 3 + 2 + 1 = \frac{(n-1)n}{2} \qquad \textit{es una progresión aritmética}$$

Comparaciones medias

$$\frac{(n-1) + (n-1)n/2}{2} = \frac{n^2 + n - 2}{4}$$

otra forma de deducirlas sería:

$$Cmedias = \underbrace{\frac{n-1+1}{2} + \frac{n-2+1}{2} + ... + \frac{1+1}{2}}_{n-1 \text{ veces}}$$

y la suma de los términos de una progresión aritmética es:

$$Cmedias = (n-1)\frac{(n/2)+1}{2} = (n-1)\frac{n+2}{4} = \frac{n^2 + 2n - n - 2}{4} = \frac{n^2 + n - 2}{4}$$

10.2.3. Ordenación por selección

Este método se basa en buscar el elemento menor del vector y colocarlo en primera posición. Luego se busca el segundo elemento más pequeño y se coloca en la segunda posición, y así sucesivamente. Los pasos sucesivos a dar son:

1. Seleccionar el elemento menor del vector de n elementos.
2. Intercambiar dicho elemento con el primero.
3. Repetir estas operaciones con los $n - 1$ elementos restantes, seleccionando el segundo elemento; continuar con los $n - 2$ elementos restantes hasta que sólo quede el mayor.

Un ejemplo aclarará el método.

EJEMPLO 10.2

Clasificar la siguiente lista de números en orden ascendente:

```
320  96  16  90  120  80  200  64
```

El método comienza buscando el número más pequeño, 16.

```
320  96  16  90  120  80  200  64
```

La lista nueva será

```
16  96  320  90  120  80  200  64
```

A continuación se busca el siguiente número más pequeño, `64`, y se realizan las operaciones 1 y 2.
La nueva lista sería

```
16  64  320  90  120  80  200  96
```

Si se siguen realizando dos iteraciones se encontrará la siguiente línea:

```
16  64  80  90  120  320  200  96
```

No se realiza ahora ningún cambio, ya que el número más pequeño del vector `V[4], V[5], ..., V[8]` está ya en la posición más a la izquierda. Las sucesivas operaciones serán:

```
16  64  80  90  96  320  200  120
16  64  80  90  96  120  200  320
16  64  80  90  96  120  200  320
```

y se habrán terminado las comparaciones, ya que el último elemento debe ser el más grande y, por consiguiente, estará en la posición correcta.

Desarrollemos ahora el algoritmo para clasificar el vector `V` de `n` componentes `V[1], V[2], ..., V[n]` con este método. El algoritmo se presentará en etapas y lo desarrollaremos con un refinamiento por pasos sucesivos.

La tabla de variables que utilizaremos será:

`I, J`	*enteras* y se utilizan como índices del vector `V`
`X`	*vector* (array unidimensional)
`AUX`	variables auxiliar para intercambio
`N`	número de elementos del vector `V`

Nivel 1

```
inicio
  desde I ← 1 hasta N-1 hacer
     Buscar elemento menor de X[I], X[I+1], ..., X[N] e intercambiar con X[I]
  fin_desde
fin
```

Nivel 2

```
inicio
  I ← 1
  repetir
```

```
    Buscar elemento menor de X[I], X[I+1], ..., X[N] e intercambiar con X[I]
    I ← I+1
  hasta_que I = N
fin
```

La búsqueda e intercambio se realiza `N - 1` veces, ya que `I` se incrementa en 1 al final del bucle.

Nivel 3

Dividamos el bucle repetitivo en dos partes:

```
inicio
  I ←1
  repetir
    Buscar elemento más pequeño X[I], X[I+1], ..., X[N]
    //Supongamos que es X[K]
    Intercambiar X[K] y X[I]
  hasta_que I = N
fin
```

Nivel 4a

Las instrucciones `"buscar"` e `"intercambiar"` se refinan independientemente. El algoritmo con la estructura **repetir** es:

```
inicio
  //...
  I ← 1
  repetir
    AUXI ← X[I]   //AUXI representa el valor más pequeño
    K ← I         //K representa la posición
    J ← I
    repetir
      J ← J+1
      si X[J] < AUXI entonces
        AUXI ← X[J]    //actualizar AUXI
        K ← J          //K, posición
      fin_si
    hasta_que J = N   //AUXI = X[K] es ahora el más pequeño
    X[K] ← X[I]
    X[I] ← AUXI
    I ← I+1
  hasta_que I = N
fin
```

Nivel 4b

El algoritmo con la estructura **mientras**.

```
inicio
  //...
  I ← 1
  mientras I < N hacer
    AUXI ← X[I]
    K ← I
    J ← I
```

```
    mientras J < N hacer
      J ← J+1
      si X[J] < AUXI entonces
        AUXI ← X[J]
        K ← J
      fin_si
    fin_mientras
    X[K] ← X[I]
    X[I] ← AUXI
    I ← I + 1
  fin_mientras
fin
```

Nivel 4c

El algoritmo de ordenación con estructura **desde**.

```
inicio
  //...
  desde I ← 1 hasta N-1 hacer
    AUXI ← X[I]
    K ← I
    desde J ← I+1 hasta N hacer
      si X[J] < AUXI entonces
        AUXI ← X[J]
        K ← J
      fin_si
    fin_desde
    X[K] ← X[I]
    X[I] ← AUXI
  fin_desde
fin
```

10.2.4. Método de Shell

Es una mejora del método de inserción directa que se utiliza cuando el número de elementos a ordenar es grande. El método se denomina "Shell" —en honor de su inventor Donald Shell— y también método de *inserción* con incrementos decrecientes.

En el método de clasificación por inserción cada elemento se compara con los elementos contiguos de su izquierda, uno tras otro. Si el elemento a insertar es más pequeño —por ejemplo—, hay que ejecutar muchas comparaciones antes de colocarlo en su lugar definitivamente.

Shell modificó los saltos contiguos resultantes de las comparaciones por saltos de mayor tamaño y con eso se conseguía la clasificación más rápida. El método se basa en fijar el tamaño de los saltos constantes, pero de más de una posición.

Supongamos un vector de elementos

```
4  12  16  24  36  3
```

en el método de inserción directa, los saltos se hacen de una posición en una posición y se necesitarán cinco comparaciones. En el método de Shell, si los saltos son de dos posiciones, se realizan tres comparaciones.

```
4  12  16  24  36  3
```

El método se basa en tomar como salto `N/2` (siendo `N` el número de elementos) y luego se va reduciendo a la mitad en cada repetición hasta que el salto o distancia vale 1.

Considerando la variable *salto*, se tendría para el caso de un determinado vector X los siguientes recorridos:

Vector X [X[1], X[2], X[3], ..., X[N]]
Vector X1 [X[1], X[1+salto], X[2+salto], ...]
Vector XN [salto1, salto2, salto3, ...]

EJEMPLO 10.3

Deducir las secuencias parciales de clasificación por el método de Shell para ordenar en ascendente la lista o vector

```
6,  1,  5,  2,  3,  4,  0
```

Solución

Recorrido	*Salto*	*Lista reordenada*	*Intercambio*
1	3	2,1,4,0,3,5,6	(6,2), (5,4), (6,0)
2	3	0,1,4,2,3,5,6	(2,0)
3	3	0,1,4,2,3,5,6	Ninguno
4	1	0,1,2,3,4,5,6	(4,2), (4,3)
5	1	0,1,2,3,4,5,6	Ninguno

Sea un vector X

```
X[1], X[2], X[3], ..., X[N]
```

y consideremos el primer salto a dar que tendrá un valor de

$$\frac{N}{2}$$

por lo que para redondear, se tomará la parte entera

```
N DIV 2
```

y se iguala a salto

```
salto = N div 2
```

El algoritmo resultante será:

```
algoritmo shell
const
  n = 50
tipo
  array[1..n] de entero:lista
var
  lista : L
  entero : k, i, j, salto
inicio
  llamar_a llenar(L)              // llenado de la lista
  salto ← N DIV 2
  mientras salto > 0 hacer
    desde i ← (salto + 1) hasta n hacer
      j ← i - salto
```

```
        mientras j > 0 hacer
          k ← j + salto
          si L[j] <= L[k] entonces
            j ← 0
          si_no
            llamar_a intercambio L[j], L[k]
          fin_si
          j ← j - salto
        fin_mientras
      fin_desde
      salto ← ent ((1 + salto)/2)
    fin_mientras
  fin
```

10.2.5. Método de ordenación rápida (*quicksort*)

El método de *ordenación rápida (quicksort)* para ordenar o clasificar un vector o lista de elementos (array) se basa en el hecho de que es más rápido y fácil de ordenar dos listas pequeñas que una lista grande. Se denomina método de ordenación rápida porque, en general, puede ordenar una lista de datos mucho más rápidamente que cualquiera de los métodos de ordenación ya estudiados. Este método se debe a Hoare.

El método se basa en la estrategia típica de *"divide y vencerás" (divide and conquer)*. La lista a clasificar almacenada en un vector o array se divide *(parte)* en dos sublistas: una con todos los valores menores o iguales a un cierto valor específico y otra con todos los valores mayores que ese valor. El valor elegido puede ser cualquier valor arbitrario del vector. En ordenación rápida se llama a este valor *pivote*.

El primer paso es dividir la lista original en dos sublistas o subvectores y un valor de separación. Así, el vector `V` se divide en tres partes:

- Subvector `VI`, que contiene los valores inferiores o iguales.
- El elemento de separación.
- Subvector `VD`, que contiene los valores superiores o iguales.

Los subvectores `VI` y `VD` no están ordenados, excepto en el caso de reducirse a un elemento.
Consideremos la lista de valores.

```
18  11  27  13  9  4  16
```

Se elige un pivote, `13`. Se recorre la lista desde el extremo izquierdo y se busca un elemento mayor que `13` (se encuentra el `18`). A continuación, se busca desde el extremo derecho un valor menor que `13` (se encuentra el `4`).

```
18  11  27  13  9  4  16
```

Se intercambian estos dos valores y se produce la lista

```
4  11  27  13  9  18  16
```

Se sigue recorriendo el vector por la izquierda y se localiza el `27`, y a continuación otro valor bajo se encuentra a la derecha (el `9`). Intercambiar estos dos valores y se obtiene

```
4  11  9  13  27  18  16
```

Al intentar este proceso una vez más, se encuentra que las exploraciones de los dos extremos vienen juntos sin encontrar ningún futuro valor que esté "fuera de lugar". En este punto se conoce que todos los valores a la derecha son mayores que todos los valores a la izquierda del pivote. Se ha realizado una partición en la lista original, que se ha quedado dividida en dos listas más pequeñas:

```
4  11  9  [13]  27  18  16
```

Ninguna de ambas listas está ordenada; sin embargo, basados en los resultados de esta primera partición, se pueden ordenar ahora las dos particiones independientemente. Esto es, si ordenamos la lista

```
4  11  9
```

en su posición, y la lista

```
27  18  16
```

de igual forma, la lista completa estará ordenada:

```
4  9  11  13  16  18  27
```

El procedimiento de ordenación supone, en primer lugar, una partición de la lista.

EJEMPLO 10.4

Utilizando el procedimiento de ordenación rápida, dividir la lista de enteros en dos sublistas para poder clasificar posteriormente ambas listas.

```
50  30  20  80  90  70  95  85  10  15  75  25
```

Se elige como pivote el número `50`.

Los valores `30`, `20`, `10`, `15` y `25` son más pequeños que `50` y constituirán la primera lista, y `80`, `90`, `70`, `95`, `85` y `75` se sitúan en la segunda lista. Se recorre la lista desde la izquierda para encontrar el primer número mayor que `50` y desde la derecha el primero menor que `50`.

```
50  30  20  80  90  70  95  85  10  15  75  25
```

se localizan los dos números `80` y `25` y se intercambian

```
50  30  20  25  90  70  95  85  10  15  75  80
```

A continuación se reanuda la búsqueda desde la derecha para un número menor que `50`, y desde la izquierda para un número mayor de `50`.

```
50  30  20  25  90  70  95  85  10  15  75  80
```

Estos recorridos localizan los números `15` y `90`, que se intercambian

```
50  30  20  25  15  70  95  85  10  90  75  80
```

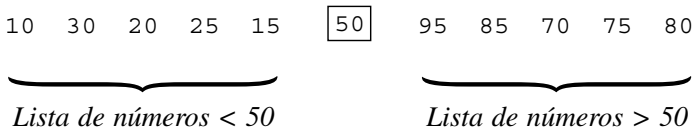
Las búsquedas siguientes localizan `10` y `70`.

```
50  30  20  25  15  70  95  85  10  90  75  80
```

El intercambio proporciona

```
50  30  20  25  15  10  95  85  70  90  75  80
```

Cuando se reanuda la búsqueda desde la derecha para un número menor que `50`, localizamos el valor `10` que se encontró en la búsqueda de izquierda a derecha. Se señala el final de las dos búsquedas y se intercambian `50` y `10`.



Algoritmos

El algoritmo de ordenación rápida se basa esencialmente en un algoritmo de división o partición de una lista. El método consiste en explorar desde cada extremo e intercambiar los valores encontrados. Un primer intento de algoritmo de partición es:

```
algoritmo particion
inicio
  establecer x al valor de un elemento arbitrario de la lista
  mientras division no este terminada hacer
    recorrer de izquierda a derecha para un valor >= x
    recorrer de derecha a izquierda para un valor =< x
    si los valores localizados no estan ordenados entonces
      intercambiar los valores
    fin_si
  fin_mientras
fin
```

La lista que se desea partir es `A[1], A[2], ..., A[n]`. Los índices que representan los extremos izquierdo y derecho de la lista son `L` y `R`. En el refinamiento del algoritmo se elige un valor arbitrario `x`, suponiendo que el valor central de la lista es tan bueno como cualquier elemento arbitrario. Los índices `i, j` exploran desde los extremos. Un refinamiento del algoritmo anterior, que incluye mayor número de detalles es el siguiente:

```
algoritmo particion
  llenar (A)
  i ← L
  j ← R
  x ← A ((L+R) div 2)
  mientras i =< j hacer
    mientras A[i] < x hacer
      i ← i+1
    fin_mientras
    mientras A[j] > x hacer
      j ← j-1
    fin_mientras
    si i =< j entonces
        llamar_a intercambiar (A[i], a[j])
        i ← i+1
        j ← j-1
    fin_si
  fin_mientras
fin
```

En los bucles externos y la sentencia **si** la condición utilizada es `i =< j`. Puede parecer que `i < j` funcionan de igual modo en ambos lugares. De hecho, se puede realizar la partición con cualquiera de las condiciones. Sin embargo, si se utiliza la condición `i < j`, podemos terminar la partición con dos casos distintos, los cuales pueden diferenciarse antes de que podamos realizar divisiones futuras. Por ejemplo, la lista

```
1  7  7  9  9
```

y la condición `i < j` terminará con `i = 3, j = 2` y las dos particiones son `A[L]..A[j]` y `A[i]..A[R]`. Sin embargo, para la lista

```
1  1  7  9  9
```

y la condición `i < j` terminaremos con `i = 3, j = 3` y las dos particiones se solapan.

El uso de la condición `i <= j` produce también resultados distintos para estos ejemplos. La lista

```
1  7  7  9  9
```

y la condición `i <= j` se termina con `i = 3, j = 2` como antes. Para la lista `1, 1, 7, 9, 9` y la condición `i =< j` se termina con `i = 4, j = 2`. En ambos casos las particiones que requieren ordenación posterior son `A[L]..A[j]` y `A[i]..A[R]`.

En los bucles **mientras** internos la igualdad se omite de las condiciones. La razón es que el valor de partición actúe como *centinela* para detectar las exploraciones.

En nuestro ejemplo se ha tomado como valor de partición o pivote el elemento cuya posición inicial es el elemento central. Este no es generalmente el caso. El ejemplo de la clasificación de la lista ya citado

```
50  30  20  80  90  70  95  85  10  15   75  25
```

utilizaba como pivote el primer elemento.

El algoritmo de ordenación rápida en el caso de que el elemento pivote sea el primer elemento se muestra a continuación:

```
algoritmo particion2
//lista a evaluar de 10 elementos
//IZQUIERDO, indice de búsqueda (recorrido) desde la izquierda
//DERECHO, indice de búsqueda desde la derecha

inicio
  llenar (X)
  //inicializar índice para recorridos desde la izquierda y derecha
  IZQUIERDO ← ALTO  //ALTO parametro que indica principio de la sublista
  DERECHO ← BAJO    //BAJO parametro que indica final de la sublista
  A <-  X[1]
  //realizar los recorridos
  mientras IZQUIERDO <= DERECHO hacer
  //búsqueda o recorrido desde la izquierda
    mientras (X[IZQUIERDO] < A) Y (IZQUIERDO < BAJO)
      IZQUIERDO ← IZQUIERDO + 1
    fin_mientras
    mientras X[DERECHO] > A y (DERECHO > ALTO)
      DERECHO ← DERECHO - 1
    fin_mientras
    //intercambiar elemento
    si IZQUIERDO <= DERECHO entonces
      AUXI <-X[IZQUIERDO]
      X[IZQUIERDO] ← X[DERECHO]
      X[DERECHO] ← AUXI
      IZQUIERDO ← IZQUIERDO + 1
      DERECHO ← DERECHO - 1
    fin_si
  fin_mientras
  //fin busqueda; situar elemento seleccionado en su posicion
  si IZQUIERDO < BAJO+1 entonces
    AUXI← X [DERECHO]
    X [DERECHO] ← X [1]
    X [1] ← AUXI
```

```
    si_no
      AUXI ← X [BAJO]
      X [BAJO] ← X[1]
      X[1] ← AUXI
    fin
  fin
```

10.3. BÚSQUEDA

La recuperación de información, como ya se ha comentado, es una de las aplicaciones más importantes de las computadoras.

La *búsqueda (searching)* de información está relacionada con las tablas para consultas *(lookup)*. Estas tablas contienen una cantidad de información que se almacena en forma de listas de parejas de datos. Por ejemplo, un diccionario con una lista de palabras y definiciones; un catálogo con una lista de libros de informática; una lista de estudiantes y sus notas; un índice con títulos y contenido de los artículos publicados en una determinada revista, etc. En todos estos casos es necesario con frecuencia buscar un elemento en una lista.

Una vez que se encuentra el elemento, la identificación de su información correspondiente es un problema menor. Por consiguiente, nos centraremos en el proceso de búsqueda. Supongamos que se desea buscar en el vector `X[1]..X[n]`, que tiene componentes numéricos, para ver si contiene o no un número dado `T`.

Si en vez de tratar sobre vectores se desea buscar información en un archivo, debe realizarse la búsqueda a partir de un determinado campo de información denominado *campo clave*. Así, en el caso de los archivos de empleados de una empresa, el campo clave puede ser el número de DNI o los apellidos.

La búsqueda por claves para localizar registros es, con frecuencia, una de las acciones que mayor consumo de tiempo conlleva y, por consiguiente, el modo en que los registros están dispuestos y la elección del modo utilizado para la búsqueda pueden redundar en una diferencia sustancial en el rendimiento del programa.

El problema de búsqueda cae naturalmente dentro de los dos casos típicos ya tratados. Si existen muchos registros, puede ser necesario almacenarlos en archivos de disco o cinta, externo a la memoria de la computadora. En este caso se llama *búsqueda externa*. En el otro caso, los registros que se buscan se almacenan por completo dentro de la memoria de la computadora. Este caso se denomina *búsqueda interna*.

En la práctica, la búsqueda se refiere a la operación de encontrar la posición de un elemento entre un conjunto de elementos dados: lista, tabla o fichero.

Ejemplos típicos de búsqueda son localizar nombre y apellidos de un alumno, localizar números de teléfono de una agenda, etc.

Existen diferentes algoritmos de búsqueda. El algoritmo elegido depende de la forma en que se encuentren organizados los datos.

La operación de búsqueda de un elemento *N* en un conjunto de elementos consiste en:

- Determinar si *N* pertenece al conjunto y, en ese caso, indicar su posición en él.
- Determinar si *N* no pertenece al conjunto.

Los métodos más usuales de búsqueda son:

- *Búsqueda secuencial o lineal.*
- *Búsqueda binaria.*
- *Búsqueda por transformación de claves (hash).*

10.3.1. Búsqueda secuencial

Supongamos una lista de elementos almacenados en un vector (array unidimensional). El método más sencillo de buscar un elemento en un vector es explorar secuencialmente el vector o, dicho en otras palabras, *recorrer el vector* desde el primer elemento al último. Si se encuentra el elemento buscado, visualizar un mensaje similar a `'Fin de búsqueda'`; en caso contrario, visualizar un mensaje similar a `'Elemento no existe en la lista'`.

En otras palabras, la búsqueda secuencial compara cada elemento del vector con el valor deseado, hasta que éste encuentra o termina de leer el vector completo.

La búsqueda secuencial no requiere ningún registro por parte del vector y, por consiguiente, no necesita estar ordenado. El recorrido del vector se realizará normalmente con estructuras repetitivas.

EJEMPLO 10.5

Se tiene un vector `A` *que contiene* `n` *elementos numéricos* `(n) >= 1(A[1], A[2], A[3], ..., A[n])` *y se desea buscar un elemento dado* `t`. *Si el elemento* `t` *se encuentra, visualizar un mensaje* `'Elemento encontrado'` *y otro que diga* `'posición = '`.

Si existen `n` elementos, se requerirán como media `n/2` comparaciones para encontrar un determinado elemento. En el caso más desfavorable se necesitarán `n` comparaciones.

Método 1

```
algoritmo busqueda_secuencial_1
  //declaraciones
inicio
  llenar (A,n)
  leer(t)
  //recorrido del vector
  desde i ← 1 hasta n hacer
    si A[i] = t entonces
      escribir('Elemento encontrado')
      escribir('en posicion', i)
    fin_si
  fin_desde
fin
```

Método 2

```
algoritmo busqueda_secuencial_2
  //...
inicio
  llenar (A,n)
  leer(t)
  i ← 1
  mientras (A[i] <> t) y (i =< n) hacer
    i ← i + 1
    //este bucle se detiene bien con A[i] = t o bien con i > n
  fin_mientras
  si A[i] = t entonces     //condición de parada
    escribir('El elemento se ha encontrado en la posición', i)
  si_no                    //recorrido del vector terminado
    escribir('El numero no se encuentra en el vector')
  fin_si
fin
```

Este método no es completamente satisfactorio, ya que si `t` no está en el vector A, `i` toma el valor `n + 1` y la comparación

```
A[i] <> t
```

producirá una referencia al elemento `A[n + 1]`, que presumiblemente no existe. Este problema se resuelve sustituyendo `i =< n` por `i < n` en la instrucción **mientras**, es decir, modificando la instrucción anterior **mientras** por

```
mientras (A[i] <> t) y (i < n) hacer
```

Método 3

```
algoritmo busqueda_secuencial_3
  //...
inicio
  llenar (A,n)
  leer(t)
  i ← 1
  mientras (A[i] <> t) y (i < n) hacer
    i ← i+1
    //este bucle se detiene cuando A[i] = t o i >= n
  fin_mientras
  si A[i] = t entonces
    escribir('El numero deseado esta presente y ocupa el lugar',i)
  si_no
    escribir(t, 'no existe en el vector')
  fin_si
fin
```

Método 4

```
algoritmo busqueda_secuencial_4
  //...
inicio
  llamar_a llenar(A,n)
  leer(t)
  i ← 1

  mientras i <= n hacer
    si t = A[i] entonces
      escribir('Se encontró el elemento buscado en la posicion',i)
      i ← n + 1
    si_no
      i ← i+1
    fin_si
  fin_mientras
fin
```

Búsqueda secuencial con centinela

Una manera muy eficaz de realizar una búsqueda secuencial consiste en modificar los algoritmos anteriores utilizando un elemento centinela. Este elemento se agrega al vector al final del mismo. El valor del elemento centinela es el del argumento. El propósito de este elemento centinela, `A[n + 1]`, es significar que la búsqueda siempre tendrá éxito. El elemento `A[n + 1]` sirve como centinela y se le asigna el valor de `t` antes de iniciar la búsqueda. En cada paso se evita la comparación de `i` con `N` y, por consiguiente, este algoritmo será preferible a los métodos anteriores, concretamente el método 4. Si el índice alcanzase el valor `n + 1`, supondría que el argumento no pertenece al vector original y en consecuencia la búsqueda no tiene éxito.

Método 5

```
algoritmo busqueda_secuencial_5
  //declaraciones
inicio
  llenar(A,n)
  leer(t)
```

```
  i ← 1
  A[n + 1] ← t
  mientras A[i] <> t hacer
    i ← i + 1
  fin_mientras
  si i = n + 1 entonces
    escribir('No se ha encontrado elemento')
  si_no
    escribir('Se ha encontrado el elemento')
  fin_si
fin
```

Una variante del método 5 es utilizar una variable lógica (interruptor o *switch*), que represente la existencia o no del elemento buscado.

Localizar si el elemento `t` existe en una lista `A[i]`, donde `i` varía desde `1` a `n`.

En este ejemplo se trata de utilizar una variable lógica `ENCONTRADO` para indicar si existe o no el elemento de la lista.

Método 6

```
algoritmo busqueda_secuencia_6
  //declaraciones
inicio
  llenar (A,n)
  leer(t)
  i ← 1
  ENCONTRADO ← falso
  mientras (no ENCONTRADO) y (i =< n) hacer
    si A[i] = t entonces
      ENCONTRADO ← verdadero
    fin_si
    i ← i + 1
  fin_mientras
  si ENCONTRADO entonces
    escribir('El numero ocupa el lugar', i - 1)
  si_no
    escribir('El numero no esta en el vector')
  fin_si
fin
```

Nota

De todas las versiones anteriores, tal vez la más adecuada sea la incluida en el método 6. Entre otras razones, debido a que el bucle **mientras** engloba las acciones que permiten explorar el vector, bien hasta que *t* se encuentre o bien cuando se alcance el final del vector.

Método 7

```
algoritmo busqueda_secuencia_7
  //declaraciones
inicio
  llenar (A,n)
  leer(t)
```

```
  i ← 1
  ENCONTRADO ← falso
  mientras i =< n hacer
    si A[i] = t entonces
      ENCONTRADO ← verdad
      escribir ('El número ocupa el lugar'; i)
    fin_si
    i ← i + 1
  fin_mientras
  si_no (ENCONTRADO) entonces
    escribir('El numero no esta en el vector')
  fin_si
fin
```

Método 8

```
algoritmo busqueda_secuencial_8
  //declaraciones
inicio
  llenar (A,n)
  ENCONTRADO ← falso
  i ← 0
  leer(t)
  repetir
    i ← i+1
    si A[i] = t entonces
      ENCONTRADO ← verdad
    fin_si
  hasta_que ENCONTRADO o (i = n)
fin
```

Método 9

```
algoritmo busqueda_secuencial_9
  //declaraciones
inicio
  llenar (A,n)
  ENCONTRADO ← falso
  leer(t)
  desde i ← 1 hasta i ← n hacer
    si A[i] = t entonces
      ENCONTRADO ← verdad
    fin_si
  fin_desde
  si ENCONTRADO entonces
    escribir('Elemento encontrado')
  si_no
    escribir('Elemento no encontrado')
  fin_si
fin
```

Consideraciones sobre la búsqueda lineal

El método de búsqueda lineal tiene el inconveniente del consumo excesivo de tiempo en la localización del elemento buscado. Cuando el elemento buscado no se encuentra en el vector, se verifican o comprueban sus *n* elementos.

En los casos en que el elemento se encuentra en la lista, el número podrá ser el primero, el último o alguno comprendido entre ambos.

Se puede suponer que el número medio de comprobaciones o comparaciones a realizar es de (n+1)/2 (aproximadamente igual a la mitad de los elementos del vector).

La búsqueda secuencial o lineal no es el método más eficiente para vectores con un gran número de elementos. En estos casos, el método más idóneo es el de *búsqueda binaria*, que presupone una ordenación previa en los elementos del vector. Este caso suele ser muy utilizado en numerosas facetas de la vida diaria. Un ejemplo de ello es la búsqueda del número de un abonado en una guía telefónica; normalmente no se busca el nombre en orden secuencial, sino que se busca en la primera o segunda mitad de la guía; una vez en esa mitad, se vuelve a tantear a una de sus dos submitades, y así sucesivamente se repite el proceso hasta que se localiza la página correcta.

10.3.2. Búsqueda binaria

En una búsqueda secuencial se comienza con el primer elemento del vector y se busca en él hasta que se encuentra el elemento deseado o se alcanza el final del vector. Aunque este puede ser un método adecuado para pocos datos, se necesita una técnica más eficaz para conjuntos grandes de datos.

Si el número de elementos del vector es grande, el algoritmo de búsqueda lineal se ralentizaría en tiempo de un modo considerable. Por ejemplo, si tuviéramos que consultar un nombre en la guía telefónica de una gran ciudad como Madrid, con una cifra aproximada de un millón de abonados, el tiempo de búsqueda —según el nombre— se podría eternizar. Naturalmente, las personas que viven en esa gran ciudad nunca utilizarán un método de búsqueda secuencial, sino un método que se basa en la división sucesiva del espacio ocupado por el vector en sucesivas mitades, hasta encontrar el elemento buscado.

Si los datos que se buscan están clasificados en un determinado orden, el método citado anteriormente se denomina *búsqueda binaria*.

La búsqueda binaria utiliza un método de *"divide y vencerás"* para localizar el valor deseado. Con este método se examina primero el elemento central de la lista; si este es el elemento buscado, entonces la búsqueda ha terminado. En caso contrario se determina si el elemento buscado está en la primera o la segunda mitad de la lista y, a continuación, se repite este proceso, utilizando el elemento central de esa sublista. Supongamos la lista

```
1231
1473
1545
1834
1892
1898        elemento central
1983
2005
2446
2685
3200
```

Si está buscando el elemento `1983`, se examina el número central, `1898`, en la sexta posición. Ya que `1983` es mayor que `1898`, se desprecia la primera sublista y nos centramos en la segunda

```
1983
2005
2446        elemento central
2685
3200
```

El número central de esta sublista es `2446` y el elemento buscado es `1983`, menor que `2446`; eliminamos la segunda sublista y nos queda

```
1983
2005
```

Como no hay término central, elegimos el término inmediatamente anterior al término central, `1983`, que es el buscado.

Se han necesitado tres comparaciones, mientras que la búsqueda secuencial hubiese necesitado siete.

La búsqueda binaria se utiliza en vectores ordenados y se basa en la constante división del espacio de búsqueda (recorrido del vector). Como se ha comentado, se comienza comparando el elemento que se busca, no con el primer elemento, sino con el elemento central. Si el elemento buscado —*t*— es menor que el elemento central, entonces *t* deberá estar en la mitad izquierda o inferior del vector; si es mayor que el valor central, deberá estar en la mitad derecha o superior, y si es igual al valor central, se habrá encontrado el elemento buscado.

El funcionamiento de la búsqueda binaria en un vector de enteros se ilustra en la Figura 10.3 para dos búsquedas: *con éxito* (localizado el elemento) y *sin éxito* (no encontrado el elemento).

El proceso de búsqueda debe terminar normalmente conociendo si la búsqueda *ha tenido éxito* (se ha encontrado el elemento) o bien *no ha tenido éxito* (no se ha encontrado el elemento) y normalmente se deberá devolver la posición del elemento buscado dentro del vector.

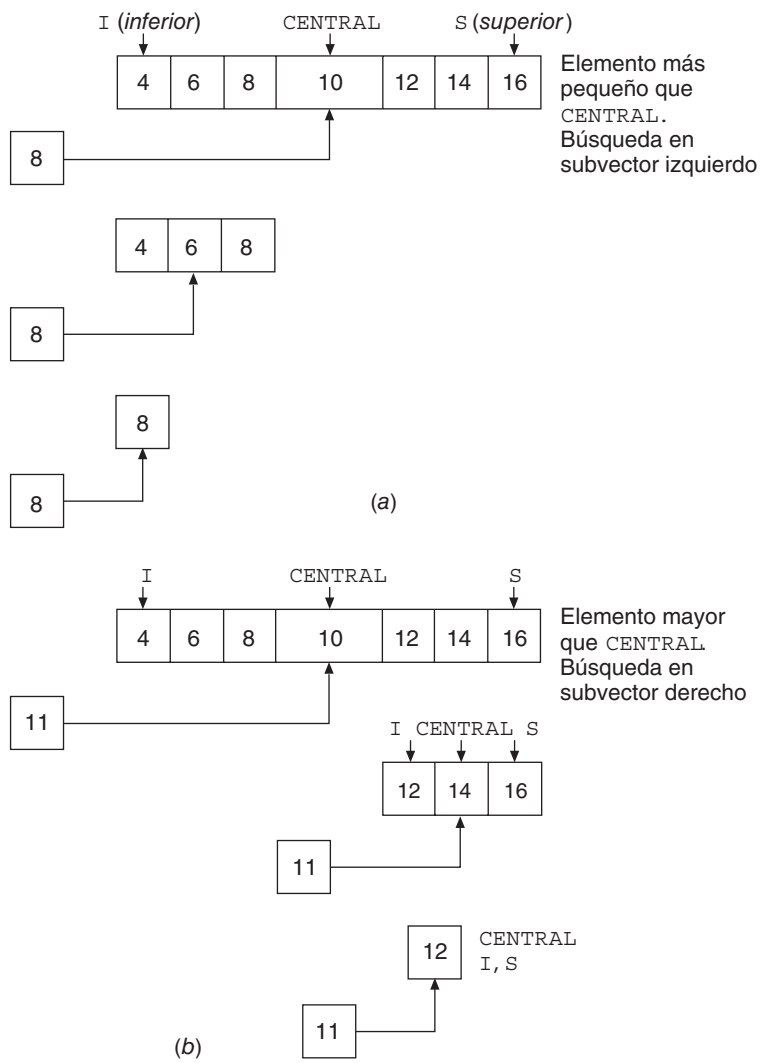


Figura 10.3. Ejemplo de búsqueda binaria: (*a*) con éxito, (*b*) sin éxito

EJEMPLO 10.6

Encontrar el algoritmo de búsqueda binaria para encontrar un elemento `K` *en una lista de elementos* $X_1, X_2, \ldots, X_n$ *previamente clasificados en orden ascendente.*

El array o vector *X* se supone ordenado en orden creciente si los datos son numéricos, o alfabéticamente si son caracteres. Las variables BAJO, CENTRAL, ALTO indican los límites inferior, central y superior del intervalo de búsqueda.

```
algoritmo busqueda_binaria
  //declaraciones
inicio
  //llenar (X,N)
  //ordenar (X,N)
  leer(K)
  //inicializar variables
  BAJO ← 1
  ALTO ← N
  CENTRAL ← ent ((BAJO + ALTO) / 2)
  mientras (BAJO =< ALTO) y (X[CENTRAL] <> K) hacer
    si K < X[CENTRAL] entonces
      ALTO ← CENTRAL - 1
    si_no
      BAJO ← CENTRAL + 1
    fin_si
    CENTRAL ← ent ((BAJO + ALTO) / 2)
  fin_mientras
  si K = X[CENTRAL] entonces
    escribir('Valor encontrado en', CENTRAL)
  si_no
    escribir('Valor no encontrado')
  fin_si
fin
```

EJEMPLO 10.7

Se dispone de un vector tipo carácter NOMBRE *clasificado en orden ascendente y de* N *elementos. Realizar el algoritmo que efectúe la búsqueda de un nombre introducido por el usuario.*

La variable N indica cuántos elementos existen en el array.
ENCONTRADO es una variable lógica que detecta si se ha localizado el nombre buscado.

```
algoritmo busqueda_nombre
{inicializar todas las variables necesarias}
{NOMBRE       array de caracteres
N             numero de nombres del array NOMBRE
ALTO          puntero al extremo superior del intervalo
BAJO          puntero al extremo inferior del intervalo
CENTRAL       puntero al punto central del intervalo
X             nombre introducido por el usuario
ENCONTRADO    bandera o centinela}
inicio
  llenar (NOMBRE, N)
  leer (X)
  BAJO ← 1
  ALTO ← N
  ENCONTRADO ← falso
  mientras (no ENCONTRADO) y (BAJO =< ALTO) hacer
    CENTRAL ← ent (BAJO+ALTO) / 2
    //verificar nombre central en este intervalo
```

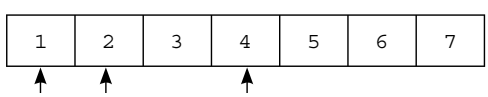
```
      si NOMBRE[CENTRAL] = X entonces
        ENCONTRADO ← verdad
      si_no
        si NOMBRE[CENTRAL] > X entonces
          ALTO ← CENTRAL - 1
        si_no
          BAJO ← CENTRAL + 1
        fin_si
      fin_si
    fin_mientras
    si ENCONTRADO entonces
      escribir('Nombre encontrado')
    si_no
      escribir('Nombre no encontrado')
    fin_si
  fin
```

Análisis de la búsqueda binaria

La búsqueda binaria es un método eficiente siempre que el vector esté ordenado. En la práctica esto suele suceder, pero no siempre. Por esta razón la búsqueda binaria exige una ordenación previa del vector.

Para poder medir la velocidad de cálculo del algoritmo de búsqueda binaria se deberán obtener el número de comparaciones que realiza el algoritmo.

Consideremos un vector de siete elementos ($n = 7$). El número 8 ($N + 1 = 8$) se debe dividir en tres mitades antes de que se alcance 1; es decir, se necesitan tres comparaciones.



El medio matemático de expresar estos números es:

$3 = log_2 (8)$

en general, para n elementos:

$K = log_2 (n + 1)$

Recuerde que $log_2 (8)$ es el exponente al que debe elevarse 2 para obtener 8. Es decir, 3, ya que $2^3 = 8$.

Si $n + 1$ es una potencia de 2, entonces $log_2 (n + 1)$ será un entero. Si $n + 1$ no es una potencia de 2, el valor del logaritmo se redondea hasta el siguiente entero. Por ejemplo, si n es 12, entonces K será 4, ya que $log_2 (13)$ (que está entre 3 y 4) se redondeará hasta 4 (2^4 es 16).

En general, en el mejor de los casos se realizará una comparación y, en el peor de los casos, se realizarán $log_2 (n + 1)$ comparaciones.

Como término medio, el número de comparaciones es

$$\frac{1 + \log_2(n + 1)}{2}$$

Esta fórmula se puede reducir para el caso de que n sea grande a

$$\frac{\log_2(n + 1)}{2}$$

Para poder efectuar una comparación entre los métodos de búsqueda lineal y búsqueda binaria, realicemos los cálculos correspondientes para diferentes valores de *n*.

`n = 100`

En la *búsqueda secuencial* se necesitarán

$$\frac{100 + 1}{2}$$ **50 comparaciones**

En la *búsqueda binaria* $\log_2 (100) = 6...$

$log_2 (100) = x$ donde $2^x = 100$ y $x = 6...$:

$27 = 128 > 100$ **7 comparaciones**

`n = 1.000.000`

En la *búsqueda secuencial:*

$$\frac{1.000.000 + 1}{2}$$ **500.000 comparaciones**

En la *búsqueda binaria* $log_2 (1.000.000) = x$

$2^x = 1.000.000$ donde $x = 20$ y $2^{20} > 1.000.000$

20 comparaciones

Como se observa en los ejemplos anteriores, el tiempo de búsqueda es muy pequeño, aproximadamente siete comparaciones para 100 elementos y veinte para 1.000.000 de elementos. *(Compruebe el lector que para 1.000 elementos se requiere un máximo de diez comparaciones.)*

La búsqueda binaria tiene, sin embargo, inconvenientes a resaltar:

El *vector debe estar ordenado* y *el almacenamiento de un vector ordenado* suele plantear problemas en las inserciones y eliminaciones de elementos. (En estos casos será necesario utilizar listas enlazadas o árboles binarios. Véanse Capítulos 12 y 13.)

La Tabla 10.2 compara la eficiencia de la búsqueda lineal y búsqueda binaria para diferentes valores de *n*. Como se observará en dicha tabla, la ventaja del método de búsqueda binaria aumenta a medida que *n* aumenta.

Tabla 10.2. Eficiencia de las búsquedas lineal y binaria

Búsqueda secuencial		**Búsqueda binaria**
Número de comparaciones		Número máximo de comparaciones
n	Elemento no localizado	Elemento no localizado
7	7	3
100	100	7
1.000	1.000	10
1.000.000	100.000	20

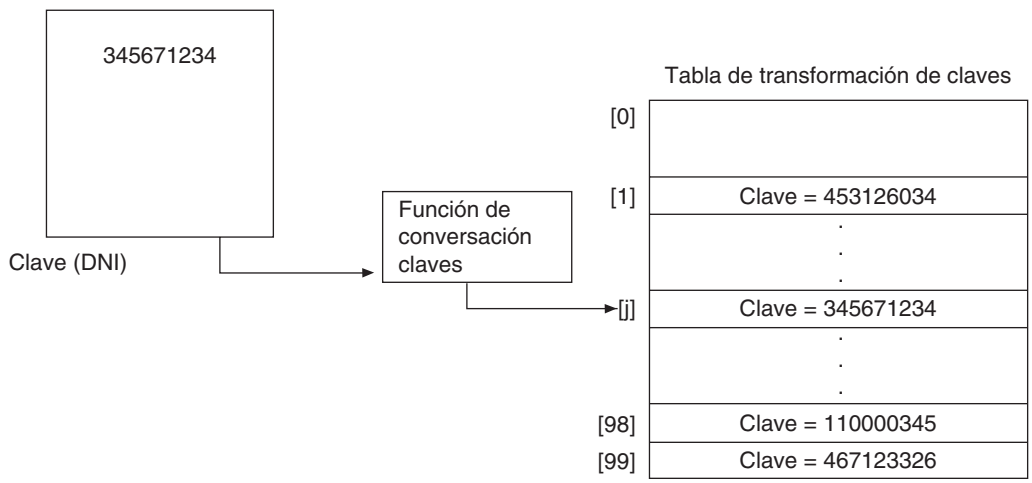
10.3.3. Búsqueda mediante transformación de claves (*hashing*)

La búsqueda binaria proporciona un medio para reducir el tiempo requerido para buscar en una lista. Este método, sin embargo, exige que los datos estén ordenados. Existen otros métodos que pueden aumentar la velocidad de búsqueda en el que los datos no necesitan estar ordenados, este método se conoce como transformación de claves (clave-dirección) o *hashing*.

El método de transformación de claves consiste en convertir la clave dada (numérica o alfanumérica) en una dirección (índice) dentro del array. La correspondencia entre las claves y la dirección en el medio de almacenamiento o en el array se establece por una función de conversión (función o *hash*).

Así, por ejemplo, en el caso de una lista de empleados (100) de una pequeña empresa. Si cada uno de los cien empleados tiene un número de identificación (clave) del 1 al 100, evidentemente puede existir una correspondencia directa entre la clave y la dirección definida en un vector o array de 100 elementos.

Supongamos ahora que el campo clave de estos registros o elementos es el número del DNI o de la Seguridad Social, que contenga nueve dígitos. Si se desea mantener en un array todo el rango posible de valores, se necesitarán 10^10 elementos en la tabla de almacenamiento, cantidad difícil de tener disponibles en memoria central, aproximadamente 1.000.000.000 de registros o elementos. Si el vector o archivo sólo tiene 100, 200 o 1.000 empleados, cómo hacer para introducirlos en memoria por el campo clave DNI. Para hacer uso de la clave DNI como un índice en la tabla de búsqueda, se necesita un medio para convertir el campo clave en una dirección o índice más pequeño. En la figura se presenta un diagrama de cómo realizar la operación de conversión de una clave grande en una tabla pequeña.



Los registros o elementos del campo clave no tienen por qué estar ordenados de acuerdo con los valores del campo clave, como estaban en la búsqueda binaria.

Por ejemplo, el registro del campo clave 345671234 estará almacenado en la tabla de transformación de claves (array) en una posición determinada; por ejemplo, 75.

La función de transformación de clave, $H(k)$ convierte la clave (k) en una dirección (d).

Imaginemos que las claves fueran nombres o frases de hasta dieciséis letras, que identifican a un conjunto de un millar de personas. Existirán 26^16 combinaciones posibles de claves que se deben transformar en 103 direcciones o índices posibles. La función H es, por consiguiente, evidentemente una función de paso o conversión de múltiples claves a direcciones. Dada una clave k, el primer paso en la operación de búsqueda es calcular su índice asociado $d \leftarrow H(k)$ y el segundo paso —evidentemente necesario— es verificar *sí* o *no* el elemento con la clave k es identificado verdaderamente por d en el array T; es decir, para verificar si la clave `T[H(K)] = K` se deben considerar dos preguntas:

- ¿Qué clase de función H se utilizará?
- ¿Cómo resolver la situación de que H no produzca la posición del elemento asociado?

La respuesta a la segunda cuestión es que se debe utilizar algún método para producir una posición alternativa, es decir, el índice d', y si ésta no es aún la posición del elemento deseado, se produce un tercer índice d'', y así sucesivamente. El caso en el que una clave distinta de la deseada está en la posición identificada se denomina *colisión*; la tarea de generación de índices alternativos se denomina tratamiento de colisiones.

Un ejemplo de colisiones puede ser:

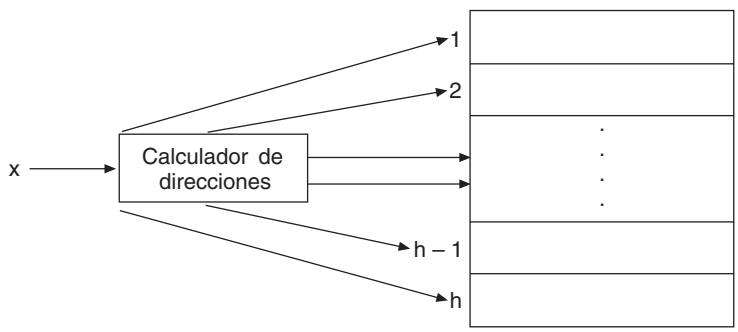
```
clave 345123124
clave 416457234     función de conversión H → dirección 200
                    función de conversión H → dirección 200
```

Dos claves distintas producen la misma dirección, es decir, *colisiones*. La elección de una buena función de conversión exige un tratamiento idóneo de colisiones, es decir, la reducción del número de colisiones.

10.3.3.1. Métodos de transformación de claves

Existen numerosos métodos de transformación de claves.

Todos ellos tienen en común la necesidad de convertir claves en direcciones. En esencia, la función de conversión equivale a una caja negra que podríamos llamar *calculador de direcciones*. Cuando se desea localizar un elemento de clave *x*, el indicador de direcciones indicará en qué posición del array estará situado el elemento.



Truncamiento

Ignora parte de la clave y se utiliza la parte restante directamente como índice (considerando campos no numéricos y sus códigos numéricos). Si las claves, por ejemplo, son enteros de ocho dígitos y la tabla de transformación tiene mil posiciones, entonces el primero, segundo y quinto dígitos desde la derecha pueden formar la función de conversión. Por ejemplo, 72588495 se convierte en 895. El truncamiento es un método muy rápido, pero falla para distribuir las claves de modo uniforme.

Plegamiento

La técnica del plegamiento consiste en la partición de la clave en diferentes partes y la combinación de las partes en un modo conveniente (a menudo utilizando suma o multiplicación) para obtener el índice.

La clave *x* se divide en varias partes, x_1, x_2, ..., x_n, donde cada parte, con la única posible excepción de la última parte, tiene el mismo número de dígitos que la dirección más alta que podría ser utilizada.

A continuación se suman todas las partes

$h(x) = x_1 + x_2 + ... + x_n$

En esta operación se desprecian los dígitos más significativos que se obtengan de arrastre o acarreo.

EJEMPLO 10.8

Un entero de ocho dígitos se puede dividir en grupos de tres, tres y dos dígitos, los grupos se suman juntos y se truncan si es necesario para que estén en el rango adecuado de índices.

Por consiguiente, si la clave es:

```
62538194
```

y el número de direcciones es `100`, la función de conversión será

```
625 + 381 + 94 = 1100
```

que se truncará a `100` y que será la dirección deseada.

EJEMPLO 10.9

Los números empleados —campo clave— de una empresa constan de cuatro dígitos y las direcciones reales son 100. Se desea calcular las direcciones correspondientes por el método de plegamiento de los empleados.

```
4205      8148   3355
```

Solución

```
h(4205) = 42 + 05 = 47
h(8148) = 81 + 48 = 129        y se convierte en 29 (129 – 100), es decir, se ignora el acarreo 1
h(3355) = 33 + 55 = 88
```

Si se desea afinar más se podría hacer la inversa de las partes pares y luego sumarlas.

Aritmética modular

Convertir la clave a un entero, dividir por el tamaño del rango del índice y tomar el resto como resultado. La función de conversión utilizada es **mod** (módulo o resto de la división entera).

```
h(x) = x mod m
```

donde *m* es el tamaño del array con índices de 0 a *m* – 1. Los valores de la función —direcciones— (el resto) irán de 0 a *m* – 1, ligeramente menor que el tamaño del array. La mejor elección de los módulos son los números primos. Por ejemplo, en un array de 1.000 elementos se puede elegir 997 o 1.009. Otros ejemplos son

```
18 mod 6    19 mod 6    20 mod 6
```

que proporcionan unos restos de 0, 1 y 2 respectivamente.

Si se desea que las direcciones vayan de 0 hasta *m*, la función de conversión debe ser

```
h(x) = x mod (m + 1)
```

EJEMPLO 10.10

Un vector T tiene cien posiciones, 0..100. Supongamos que las claves de búsqueda de los elementos de la tabla son enteros positivos (por ejemplo, número del DNI).

Una función de conversión h debe tomar un número arbitrario entero positivo x y convertirlo en un entero en el rango 0..100, esto es, h es una función tal que para un entero positivo x.

$h(x) = n$, donde n es entero en el rango 0..100

El método del módulo, tomando 101, será

```
h(x) = x mod 101
```

Si se tiene el DNI número 234661234, por ejemplo, se tendrá la posición 56:

```
234661234 mod 101 = 56
```

EJEMPLO 10.11

La clave de búsqueda es una cadena de caracteres —tal como un nombre—. Obtener las direcciones de conversión.

El método más simple es asignar a cada carácter de la cadena un valor entero (por ejemplo, A = 1, B = 2, ...) y sumar los valores de los caracteres en la cadena. Al resultado se le aplica entonces el módulo 101, por ejemplo.

Si el nombre fuese JONAS, esta clave se convertiría en el entero

```
10 + 15 + 14 + 1 + 19 = 63
```

```
63 mod 101 = 63
```

Mitad del cuadrado

Este método consiste en calcular el cuadrado de la clave *x*. La función de conversión se define como

$h(x) = c$

donde *c* se obtiene eliminando dígitos a ambos extremos de x^2. Se deben utilizar las mismas posiciones de x^2 para todas las claves.

EJEMPLO 10.12

Una empresa tiene ochenta empleados y cada uno de ellos tiene un número de identificación de cuatro dígitos y el conjunto de direcciones de memoria varía en el rango de 0 a 100. Calcular las direcciones que se obtendrán al aplicar función de conversión por la mitad del cuadrado de los números empleados:

```
4205   7148   3350
```

Solución

```
x          4205          7148          3350
x²   17 682 025    51 093 904    11 122 250
```

Si elegimos, por ejemplo, el cuarto y quinto dígito significativo, quedaría

```
h(x)      82    93    22
```

10.3.3.2. Colisiones

La función de conversión $h(x)$ no siempre proporciona valores distintos, puede suceder que para dos claves diferentes x_1 y x_2 se obtenga la misma dirección. Esta situación se denomina *colisión* y se deben encontrar métodos para su correcta resolución.

Los ejemplos vistos anteriormente de las claves DNI correspondientes al archivo de empleados, en el caso de cien posibles direcciones. Si se considera el método del módulo en el caso de las claves, y se considera el número primero 101

```
123445678   123445880
```

proporcionarían las direcciones:

```
h (123445678) = 123445678 mod 101 = 44
h (123445880) = 123445880 mod 101 = 44
```

Es decir, se tienen dos elementos en la misma posición del vector o array, [44]. En terminología de claves se dice que las claves 123445678 y 123445880 han *colisionado*.

El único medio para evitar el problema de las colisiones totalmente es tener una posición del array para cada posible número de DNI. Si, por ejemplo, los números de DNI son las claves y el DNI se representa con nueve dígitos, se necesitaría una posición del array para cada entero en el rango 000000000 a 999999999. Evidentemente, sería necesario una gran cantidad de almacenamiento. En general, el único método para evitar colisiones totalmente es que el array sea lo bastante grande para que cada posible valor de la clave de búsqueda pueda tener su propia posición. Ya que esto normalmente no es práctico ni posible, se necesitará un medio para tratar o resolver las colisiones cuando sucedan.

Resolución de colisiones

Consideremos el problema producido por una colisión. Supongamos que desea insertar un elemento con número nacional de identidad DNI 12345678, en un array `T`. Se aplica la función de conversión del módulo y se determina que el nuevo elemento se situará en la posición `T[44]`. Sin embargo, se observa que `T[44]` ya contiene un elemento con DNI 123445779.

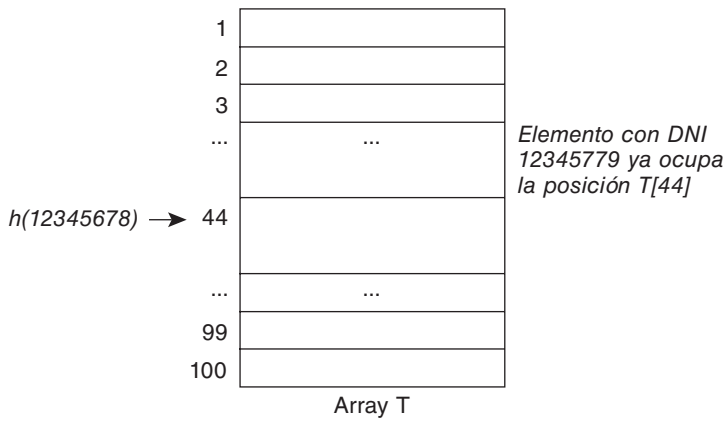


Figura 10.4. Colisión.

La pregunta que se plantea inmediatamente es ¿qué hacer con el nuevo elemento?

Un método comúnmente utilizado para resolver una colisión es cambiar la estructura del array `T` de modo que pueda alojar más de un elemento en la misma posición. Se puede, por ejemplo, modificar `T` de modo que cada posición `T[i]` sea por sí misma un array capaz de contener `N` elementos. El problema, evidentemente, será saber la magnitud de `N`. Si `N` es muy pequeño, el problema de las colisiones aparecerá cuando aparezca `N + 1` elementos.

Una solución mejor es permitir una lista enlazada o encadenada de elementos para formar a partir de cada posición del array. En este método de resolución de colisiones, conocido como *encadenamiento*, cada entrada `T[i]` es un puntero que apunta al elemento del principio de la lista de elementos (véase Capítulo 12), de modo que la función de transformación de clave lo convierte en la posición *i*.

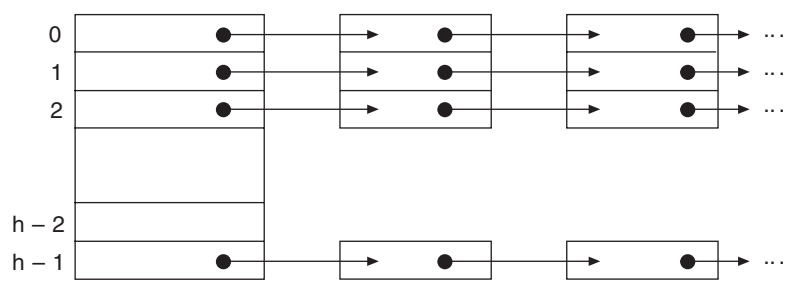


Figura 10.5. Encadenamiento.

10.4. INTERCALACIÓN

La *intercalación* es el proceso de mezclar (intercalar) dos vectores ordenados y producir un nuevo vector ordenado.

Consideremos los vectores (listas de elementos) ordenados:

```
A:  6  23  34
B:  5  22  26  27  39
```

El vector clasificado es:

```
C:  5    6    22    23    24    26    27    39
```

La acción requerida para solucionar el problema es muy fácil de visualizar. Un algoritmo sencillo puede ser:

1. Poner todos los valores del vector `A` en el vector `C`.
2. Poner todos los valores del vector `B` en el vector `C`.
3. Clasificar el vector `C`.

Es decir, todos los valores se ponen en el vector `C`, con todos los valores de `A` seguidos por todos los valores de `B`. Seguidamente, se clasifica el vector `C`. Evidentemente es una solución correcta. Sin embargo, se ignora por completo el hecho de que los vectores `A` y `B` están clasificados.

Supongamos que los vectores `A` y `B` tienen `M` y `N` elementos. El vector `C` tendrá `M + N` elementos.

El algoritmo comenzará seleccionando el más pequeño de los dos elementos `A` y `B`, situándolo en `C`. Para poder realizar las comparaciones sucesivas y la creación del nuevo vector `C`, necesitaremos dos índices para los vectores `A` y `B`. Por ejemplo, *i* y *j*. Entonces nos referiremos al elemento *i* en la lista `A` y al elemento `j` en la lista `B`. Los pasos generales del algoritmo son:

```
si elemento i de A es menor que elemento j de B entonces
  transferir elemento i de A a C
  avanzar i (incrementar en 1)
si_no
  transferir elemento j de B a C
  avanzar j
fin_si
```

Se necesita un índice `K` que represente la posición que se va rellenando en el vector `C`. El proceso gráfico se muestra en la Figura 10.6.

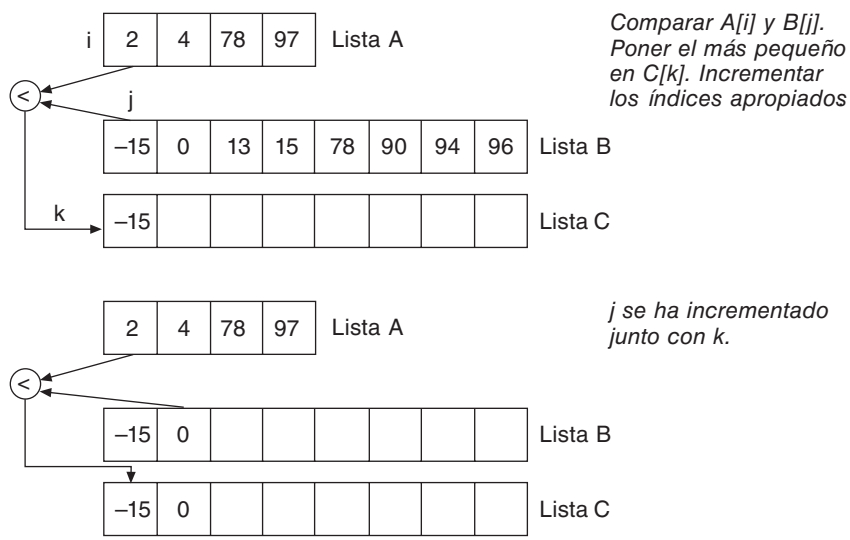


Figura 10.6. Intercalación (B[j] < A[i], de modo que C[k] se obtiene de B[j]).

El primer refinamiento del algoritmo.

```
{estado inicial de los algoritmos}
i ← 1
j ← 1
k ← 0
mientras (i <= M) y (j <= N) hacer
  //seleccionar siguiente elemento de A o B y añadir a C
  k ← k + 1
  //incrementar K}
```

```
    si A[i] < B[j] entonces
      C[k] ← A[i]
      i ← i + 1
    si_no
      C[k] ← B[j]
      j ← j + 1
    fin_si
  fin_mientras
```

Si los vectores tienen elementos diferentes, el algoritmo anterior no requiere seguir haciendo comparaciones cuando el vector más pequeño se termine de situar en C. La operación siguiente deberá copiar en C los elementos que restan del vector más grande. Así, por ejemplo, supongamos:

```
A = 6     23    24                    i = 4
B = 5     22    26    27    39        j = 3
C = 5      6    22    23    24        k = 5
```

Todos los elementos del vector A se han relacionado y situado en el vector C. El vector B contiene los elementos no seleccionados y que deben ser copiados, en orden, al final del vector C. En general, será necesario decidir cuál de los vectores A o B tienen elementos no seleccionados y a continuación ejecutar la asignación necesaria.

El algoritmo de copia de los elementos restantes es:

```
si i <= M entonces
  desde r ← i hasta M hacer
    k ← k + 1
    C[k] ← A[r]
  fin_desde
si_no
  desde r ← j hasta N hacer
    k ← k + 1
    C[k] ← B[r]
  fin_desde
fin_si
```

El algoritmo total resultante de la intercalación de dos vectores A y B ordenados en uno C es:

```
algoritmo intercalacion
inicio
  leer(A, B)   //A, B vectores de M y N elementos
  i ← 1
  j ← 1
  k ← 0

  mientras (i <= M) y (j <= N) hacer
    //seleccionar siguiente elemento de A o B y añadirlo a C
    k ← k+1
    si A[i] < B[j] entonces
      C[k] ← A[i]
      i ← i + 1
    si_no
      C[k] ← B[j]
      j ← j + 1
    fin_si
  fin_mientras
  //copiar el vector restante
  si i <= M entonces
```

```
    desde r ← i hasta M hacer
      k ← k + 1
      C[k] ← A[r]
    fin_desde
  si_no
    desde r ← j hasta N hacer
      k ← k + 1
      C[k] ← B[r]
    fin_desde
  fin_si
  escribir(C)   //vector clasificado
fin
```

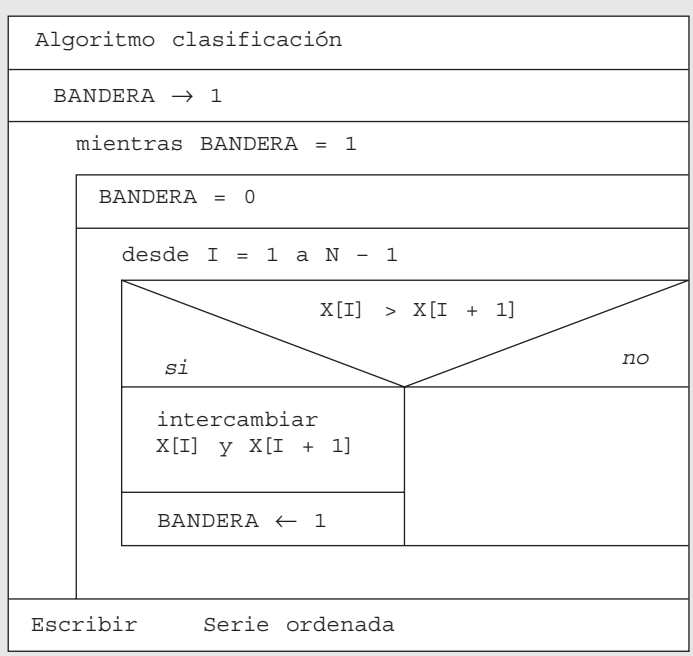
ACTIVIDADES DE PROGRAMACIÓN RESUELTAS

10.1. *Clasificar una serie de números* `X1, X2, ..., Xn` *en orden creciente por el método del intercambio o de la burbuja.*

Análisis

Se utiliza un indicador (bandera) igual a 0 si la serie está bien ordenada y a 1 en caso contrario. Como a priori la serie no está bien ordenada, se inicializa el valor de la bandera a 1 y después se repiten las siguientes acciones:

- Se fija la bandera a 0.
- A partir del primero se comparan dos elementos consecutivos de la serie; si están bien ordenados, se pasa al elemento siguiente, si no se intercambian los valores de los dos elementos y se fija el valor de la bandera a 1; si después de haber pasado revista —leído— toda la serie, la bandera permanece igual a 0, entonces la clasificación está terminada.



10.2. *Clasificar los números* `A` *y* `B`.

Método 1

```
algoritmo clasificar
inicio
  leer(A, B)
  si A < B entonces
    permutar (A , B)
  fin_si
  escribir('Mas grande', A)
  escribir('Más pequeña', B)
fin
```

Método 2

```
algoritmo clasificar
inicio
  leer(A)
  MAX ← A
  leer(B)
  MIN ← B
  si B > A entonces
    MAX ← B
    MIN ← A
  fin_si
  escribir('Maximo =', MAX)
  escribir('Mínimo =', MIN)
fin
```

10.3. *Se dispone de una lista de números enteros clasificados en orden creciente. Se desea conocer si un número dado introducido desde el terminal se encuentra en la lista. En caso afirmativo, averiguar su posición, y en caso negativo, se desea conocer su posición en la lista e insertarlo en su posición.*

Análisis

Como ya conoce el lector, existen dos métodos fundamentales de búsqueda: lineal y binaria. Resolvemos el problema con los dos métodos a fin de consolidar las ideas sobre ambos.

Búsqueda lineal

El método consiste en comparar el número dado en orden sucesivo con todos los elementos del conjunto de números, efectuando un recorrido completo del vector que representa la lista.

El proceso termina cuando se encuentra un número igual o superior al número dado.

El método de inserción o intercalación de un elemento en el vector será el descrito en el apartado 6.3.4.

La tabla de variables es la siguiente:

`N`	número de elementos de la lista: *entero*.
`J`	posición del elemento en la lista: *entero*.
`K`	contador del bucle de búsqueda: *entero*.
`X`	número dado: *entero*.
`LISTA`	conjunto de números enteros.

Búsqueda dicotómica

La condición para realizar este método —más rápido y eficaz— es que la lista debe estar clasificada en orden creciente o decreciente.

Se obtiene el número de elementos de la lista y se calcula el número central de la lista.

Si el número dado es igual al número central de la lista, la búsqueda ha terminado. En caso contrario, pueden suceder dos casos:

- El número está en la sublista inferior.
- El número está en la sublista superior.

Tras localizar la sublista donde se encuentra, se consideran variables `MIN` y `MAX` que contienen los elementos menor y mayor de cada sublista —que coincidirán con los extremos al estar ordenada la lista—, así como el término central (`CENTRAL`), de acuerdo al siguiente esquema.

Sublista inferior `L[1] L[2] ...L[CENTRAL]`
Sublista superior `L[CENTRAL + 1]...L[N]`

Los valores de las variables `INF`, `SUP` y `CENTRAL` serán:

Primera búsqueda

$$\texttt{CENTRAL} = \frac{(\texttt{SUP} - \texttt{INF})}{2} + \frac{\texttt{INF} = \texttt{N} - 1}{2} + 1 = \frac{\texttt{N} - 1}{2}$$

```
SUP = N
INF = 1
```

- *Si el número* `X` *está en la sublista inferior, entonces*

```
INF = 1
SUP = CENTRAL - 1
```

y se realiza una segunda búsqueda entre los elementos de orden `1` y `CENTRAL`.

- *Si el número X está en la sublista superior, entonces*

```
INF = CENTRAL + 1
SUP = N
```

y se realiza una segunda búsqueda entre los elementos de orden `CENTRAL + 1` y `N`.

El proceso de variables es:

`N`	número de elementos de la lista: *entero*.
`I`	contador del bucle de búsqueda: *entero*.
`SW`	interruptor o bandera para indicar si el número dado está en la lista: *lógico*.
`LISTA`	conjunto de números enteros: *entero*.
`X`	número buscado: *entero*.
`INF`	posición inicial de la lista o sublista: *entero*.
`SUP`	posición superior de la lista o sublista: *entero*.
`POSICION`	lugar del orden ocupado por el número buscado: *entero*.

Pseudocódigo

Búsqueda lineal

```
algoritmo busqueda_1
var
  entero : I, K, X, N
  array[1..50] de entero : lista
  //se supone dimensión de la lista a 50 elementos y que se trata de una
  //lista ordenada
inicio
  leer(N)
  //lectura de la lista
```

```
  desde I ← 1 hasta N hacer
    leer(LISTA[I])
  fin_desde
  Ordenar (LISTA, N)
  leer(X)
  I ← 0
  repetir
    I ← I + 1
  hasta_que (LISTA[I] >= X) o (I = 50)
  si LISTA[I] = X entonces
    escribir('se encuentra en',I)
  si_no
    escribir('El numero dado no esta en el lista')
    //insertar el elemento X en la lista
    si N < 50 entonces
      desde K ← N hasta I decremento 1 hacer
        LISTA[K + 1] ← LISTA[K]
      fin_desde
      LISTA[I] ← X
      N ← N + 1
      escribir('Insertado en',I)
    fin_si
  fin_si
  //escritura del vector LISTA
  desde I ← 1 hasta N hacer
    escribir(LISTA[I])
  fin_desde
fin
```

Búsqueda dicotómica

```
algoritmo busqueda_b
var
  entero: I, N, X, K, INF, SUP, CENTRAL, POSICION
  lógico: SW
  array [1...50] de entero: LISTA
inicio
  leer(N)
  desde I ← 1 hasta N hacer
    leer(LISTA[I])   //la lista ha de estar ordenada
  fin_desde
  Ordenar (LISTA, N)
  leer(X)
  SW ← falso
  INF ← 1
  SUP ← N
  repetir
    CENTRAL ← (SUP - INF)DIV 2 + INF
    si LISTA[CENTRAL] = X entonces
      escribir('Numero encontrado en la lista')
      POSICION ← CENTRAL
      escribir(POSICION)
      SW ← verdad
    si_no
      si X < LISTA[CENTRAL] entonces
        SUP ← CENTRAL
      si_no
        INF ← CENTRAL+1
```

```
      fin_si
      si (INF = SUP) y (LISTA[INF] = X) entonces
        escribir('El numero esta en la lista')
        POSICION ← INF
        escribir(POSICION)
        SW ← verdad
    fin_si
  fin_si
  hasta_que (INF = SUP) o SW
  si no (SW) entonces
    escribir('Numero no existe en la lista')
    si X < lista(INF) entonces
      POSICION ← INF
    si_no
      POSICION ← INF+1
    fin_si
    escribir(POSICION)
    desde K ← N hasta POSICION decremento 1 hacer
      LISTA[K + 1] ← LISTA[K]
    fin_desde
    LISTA[POSICION] ← X
    N ← N + 1
  fin_si
  //escritura de la lista
  desde I ← 1 hasta N hacer
    escribir(LISTA[I])
  fin_desde
fin
```

10.4. *Ordenar de mayor a menor un vector de N elementos (N <= 40), cada uno de los cuales es un registro con los campos día, mes y año de tipo entero.*

Utilice una función `ESMENOR(fecha1,fecha2)` que nos devuelva si una fecha es menor que otra.

```
algoritmo ordfechas
tipo registro: fechas
     inicio
         entero: dia
         entero: mes
         entero: año
     fin_registro
     array[1..40] de fechas: arr
var arr    : f
    entero : n
inicio
 pedirfechas(f,n)
 ordenarfechas(f,n)
 presentarfechas(f,n)
fin
logico función esmenor(E fechas: fecha1,fecha2)
inicio
 si (fecha1.año<fecha2.año) o
    (fecha1.año=fecha2.año) y (fecha1.mes<fecha2.mes) o
    (fecha1.año=fecha2.año) y (fecha1.mes=fecha2.mes) y
    (fecha1.día<fecha2.día) entonces
          devolver(verdad)
 si_no
          devolver(falso)
 fin_si
fin_función
```

```
procedimiento pedirfechas(S arr:f; S entero:n)
 var entero:i
     entero:dia
 inicio
  i←1
  escribir ("Deme la ",i,"ª fecha")
  escribir("Día: ")
  leer(dia)
  mientras (dia<>0) y (i<=40) hacer
      f[i].dia ← dia
      escribir("Mes:")
      leer(f[i].mes)
      escribir("Año:")
      leer(f[i].año)
      n ← i
      i ← i+1
      si i<=40 entonces
         escribir ("Deme la ",i,"ª fecha")
         escribir ("Día: ")
         leer(dia)
      fin_si
  fin_mientras
fin_procedimiento

procedimiento ordenarfechas(E/S arr:f; E entero:n)
 var entero:salto
     lógico:ordenada
     entero:j
     fechas:AUXI
inicio
  salto ← n
  mientras salto > 1 hacer
   salto ← salto div 2
   repetir
    ordenada ← verdad
    desde j ← 1 hasta n-salto hacer
      si esmenor(f[j], f[j+salto]) entonces
          AUXI ← f[j]
          f[j] ← f[j+salto]
          f[j+salto] ← AUXI
          ordenada ← falso
      fin_si
    fin_desde
   hasta ordenada
 fin_mientras
fin_procedimiento

procedimiento presentarfechas(E arr:f; E entero:n)
 var entero:i
 inicio
 desde i←1 hasta n hacer
   escribir(f[i].día,f[i].mes,f[i].año)
 fin_desde
fin_procedimiento
```

Considere otras posibilidades, usando el mismo método de ordenación, para resolver el ejercicio.

10.5. *Dada la lista de fechas ordenada en orden decreciente del ejercicio anterior, diseñar los procedimientos:*

1. Buscar, que nos informará sobre si una determinada fecha se encuentra o no en la lista
   - — si no está, indicará la posición donde correspondería insertarla,
   - — si está, nos dirá la posición donde la hemos encontrado o, si estuviera repetida, a partir de qué posición y cuántas veces son las que aparece.
2. Insertar, que nos permitirá insertar una fecha en una determinada posición. Se deberá utilizar en un algoritmo haciendo uso previo de buscar; así, cuando una fecha no se encuentre en la lista, la insertará en el lugar adecuado para que no se pierda la ordenación inicial.

```
algoritmo buscar_insertar_fechas
tipo registro: fechas
     inicio
         entero: dia
         entero: mes
         entero: año
     fin_registro
     array[1..40] de fechas: vector

var vector : f
    entero : n
    fechas : fecha
    lógico : esta
    entero : posic, cont

inicio
  pedirfechas(f,n)
  ordenarfechas(f,n)
  presentarfechas(f,n)
  escribir('Deme fecha a buscar (dd mm aa)')
  leer(fecha.dia,fecha.mes,fecha.año)
  buscar(f,n,fecha,esta,posic,cont)
  si esta entonces
    si cont > 1 entonces
      escribir('Aparece a partir de la posición: ', posic, ' ', cont, ' veces')
    si_no
      escribir('Está en la posición: ', posic )
    fin_si
  si_no
    si n=40 entonces
      escribir('No está. Array lleno')
    si_no
      insertar(f,n,fecha,posic)
      presentarfechas(f,n)
    fin_si
  fin_si
fin

logico función esmenor(E fechas: fecha1,fecha2)
 inicio
   ..................
 fin_función

logico función esigual(E fechas: fecha1,fecha2)
 inicio
  si (fecha1.año=fecha2.año) y (fecha1.mes=fecha2.mes) y (fecha1.día=fecha2.día) entonces
     devolver(verdad)
```

```
  si_no
     devolver(falso)
  fin_si
 fin_función

procedimiento pedirfechas(S vector: f; S entero n)
 var entero: i
     entero: dia
 inicio
   ...
 fin_procedimiento

procedimiento ordenarfechas(E/S vector: f; E entero: n)
 var entero : salto
     lógico : ordenada
     entero : j
     fechas : AUXIi
 inicio
  ...
 fin_procedimiento

procedimiento buscar(E vector: f; E entero:n; E fechas: fecha; S lógico:esta; S entero:
                     posic, cont)
 var entero : primero,ultimo,central,i
     lógico : encontrado
 inicio
  primero ← 1
  ultimo ← n
  esta ← falso
  mientras (primero<=ultimo) y (no esta) hacer
    central ← (primero+ultimo) div 2
    si esigual(f[central],fecha) entonces
           esta ← verdad
    si_no
           si esmenor(f[central],fecha) entonces
                  ultimo ← central-1
           si_no
                  primero ← central+1
           fin_si
    fin_si
  fin_mientras
  cont ← 0
  si esta entonces
       i ← central-1
       encontrado ← verdad
       mientras (i>=1) y (encontrado) hacer
             si esigual(f[i],f[central]) entonces
                  i ← i-1
             si_no
                  encontrado ← falso
             fin_si
       fin_mientras
       i ← i+1
       encontrado ← verdad
       posic ← i
       mientras (i<=40) y encontrado hacer
             si esigual(f[i],f[central]) entonces
                  cont ← cont+1
                  i ← i+1
```

```
              si_no
                   encontrado ← falso
              fin_si
        fin_mientras
      si_no
        posic ← primero
  fin_si
 fin_procedimiento

procedimiento insertar(E/S vector: f; E/S entero: n
                       E fechas:fecha; E entero:posic)
 var entero:i
 inicio
  desde i ← n hasta posic decremento 1 hacer
    f[i+1] ← f[i]
  fin_desde
  f[posic] ← fecha
  n ← n+1
 fin_procedimiento

procedimiento presentarfechas(E vector:f; E entero:n)
 var entero:i
 inicio
   ...
 fin_procedimiento
```

10.6. *Escriba el procedimiento de búsqueda binaria de forma recursiva*

```
algoritmo busqueda_binaria
  tipo
       array[1..10] de entero: arr
  var
       arr : a
       entero : num, posic, i
inicio
    desde i ← 1 hasta 10 hacer
      leer(a[i])
    fin_desde
    ordenar(a)
    escribir('Indique el número a buscar en el array ')
    leer(num)
    busqueda(a,posic,1,10,num)
    si posic > 0 entonces
       escribir('Existe el elemento en la posición ', posic)
    si_no
       escribir('No existe el elemento en el array.')
    fin_si
fin

  procedimiento ordenar(E/S arr: a)
    ...
    inicio
       ...
    fin_procedimiento

  procedimiento busqueda(E arr: a; S entero: posic
                         E entero: primero,ultimo,num)
    //Este procedimiento devuelve 0 si no existe el elemento
    en el array, y si existe devuelve su posición
    var
       entero: central
```

```
  inicio
          si primero > ultimo entonces
              posic ← 0
          si_no
              central ← (primero+ultimo) div 2
                si a[central] = num entonces
                   posic ← central
           si_no
              si num > a[central] entonces
                 primero ← central + 1
              si_no
                 ultimo ← central - 1
              fin_si
              busqueda(a,posic,primero,ultimo,num)
           fin_si
      fin_si
fin_procedimiento
```

10.7. *Partiendo de la siguiente lista inicial:*

80 36 98 62 26 78 22 27 2 45

tome como elemento pivote el contenido del que ocupa la posición central y realice el seguimiento de los distintos pasos que llevarían a su ordenación por el método Quick-Sort. Implemente el algoritmo correspondiente.

1	2	3	4	5	6	7	8	9	10
80	**36**	**98**	**62**	**26**	**78**	**22**	**27**	**2**	**45**
2								**80**	
	22					**36**			
		26		**98**					
		j	i						

1	2	3	4	5	6	7	8	9	10
2	**22**	**26**	**62**	**98**	**78**	**36**	**27**	**80**	**45**
	22								
j		i							

1	2	3	4	5	6	7	8	9	10
2	**22**	**26**	**62**	**98**	**78**	**36**	**27**	**80**	**45**
			27				**62**		
				36		**98**			
				j	i				

1	2	3	4	5	6	7	8	9	10
2	**22**	**26**	**27**	**36**	**78**	**98**	**62**	**80**	**45**
			27						
		j		i					

1	2	3	4	5	6	7	8	9	10
2	**22**	**26**	**27**	**36**	**78**	**98**	**62**	**80**	**45**
					45				**78**
						62	**98**		
						j	i		

1	2	3	4	5	6	7	8	9	10
2	**22**	**26**	**27**	**36**	**45**	**62**	**98**	**80**	**78**
					45				
				j		i			

1	2	3	4	5	6	7	8	9	10
2	**22**	**26**	**27**	**36**	**45**	**62**	**98**	**80**	**78**
							78		**98**
								80	
							j		i

1	2	3	4	5	6	7	8	9	10
2	**22**	**26**	**27**	**36**	**45**	**62**	**78**	**80**	**98**

```
algoritmo quicksort
 tipo
  array[1..10] de entero: arr
 var
   arr    : a
   entero : k

inicio
     desde k ← 1 hasta 10 hacer
         leer (a[k])
     fin_desde
     rápido (a,10)
     desde k ← 1 hasta 10 hacer
         escribir (a[k])
     fin_desde
fin

  procedimiento intercambiar (E/S entero: m,n)
    var
      entero: AUXI
    inicio
      AUXI ← m
      m ← n
      n ← AUXI
  fin_procedimiento

  procedimiento partir (E/S arr: a E entero: primero, ultimo)
  var
    entero: i,j,central

  inicio
    i ← primero
    j ← ultimo
    // encontrar elemento pivote, central, y almacenar su contenido
    central ← a[ (primero+ultimo) div 2 ]
    repetir
       mientras a[i] < central hacer
          i ← i+1
       fin_mientras
       mientras a[j] > central hacer
          j ← j-1
       fin_mientras
       si i <= j entonces
          intercambiar( a[i],a[j] )
          i ← i+1
          j ← j-1
       fin_si
    hasta_que i > j
    si primero < j entonces
       partir (a,primero,j)
    fin_si
    si i < ultimo entonces
       partir (a,i,ultimo)
    fin_si
  fin_procedimiento

  procedimiento rapido (E/S arr: a; E entero: n)
    inicio
       partir (a,1,n)
    fin_procedimiento
```

CONCEPTOS CLAVE

- Búsqueda.
- Eficiencia de los métodos de ordenación.
- Intercalación.
- Ordenación.
- Tipos de búsqueda.

RESUMEN

La ordenación de datos es una de las aplicaciones más importantes de las computadoras. Dado que es frecuente que un programa trabaje con grandes cantidades de datos almacenados en arrays, resulta imprescindible conocer diversos métodos de ordenación de arrays y cómo, además, puede ser necesario determinar si un array contiene un valor que coincide con un cierto valor clave también resulta básico conocer los algoritmos de búsqueda.

1. La *ordenación* o *clasificación* es el proceso de organizar datos en algún orden o secuencia específica, tal como creciente o decreciente para datos numéricos o alfabéticamente para datos de caracteres. La ordenación de arrays (arreglos) se denomina ordenación interna, ya que se efectúa con todos los datos en la memoria interna de computadora.
2. Es posible ordenar arrays por diversas técnicas, como burbuja, selección, inserción, Shell o QuickSort y, cuando el número de elementos a ordenar es pequeño, todos estos métodos son aceptables.
3. Para ordenar arrays con un gran número de elementos debe tenerse en cuenta la diferente eficiencia en cuanto al tiempo de ejecución entre los métodos comentados. Entre los citados, QuickSort y Shell son los más avanzados.
4. El método de búsqueda lineal de un determinado valor clave en un array, que compara cada elemento con la clave buscada, puede ser útil en arrays pequeños o no ordenados.
5. El método de búsqueda binaria es mucho más eficiente pero requiere arrays ordenados.
6. Puesto que los arrays permiten el acceso directo a un determinado elemento o posición, la información en un array no tiene por qué ser colocada en forma secuencial. Es, por tanto, posible usar una función *hash* que transforme el valor clave en un número válido para ser utilizado como subíndice en el array y almacenar la información en la posición especificada por dicho subíndice.
7. Una función de conversión *hash* no siempre proporciona valores distintos, y puede suceder que para dos claves diferentes devuelva la misma dirección. Esta situación se denomina *colisión* y se deben encontrar métodos para su correcta resolución.
8. Entre los métodos para resolver las colisiones destacan:

   a) Reservar una zona especial en el array para colocar las colisiones.
   b) Buscar la primera posición libre que siga a aquélla donde se debiera haber colocado la información y en la que no se pudo situar por encontrarse ya ocupada debido a la colisión.
   c) Utilizar encadenamiento.

9. Si la información se coloca en un array aplicando una función *hash* a determinado campo clave y estableciendo un método de resolución de colisiones, la consulta por dicho campo clave también se efectuará de forma análoga.
10. Cuando se tienen dos vectores ordenados y se necesita obtener otro también ordenado, el proceso de intercalación o mezcla debe producirnos el resultado deseado, sin que sea necesario aplicar a continuación ningún método de ordenación.

EJERCICIOS

10.1. Realizar el diagrama de flujo y el pseudocódigo que permuta tres enteros: $n1$, $n2$ y $n3$ en orden creciente.

10.2. Escribir un algoritmo que lea diez nombres y los ponga en orden alfabético utilizando el método de selección. Utilice los siguientes datos para comprobación: Sánchez, Waterloo, McDonald, Bartolomé, Jorba, Clara, David, Robinson, Francisco, Westfalia.

10.3. Clasificar el array (vector):

```
42   57   14   40   96   19   08   68
```

por los métodos: 1) selección, 2) burbuja. Cada vez que se reorganice el vector, se debe mostrar el nuevo vector reformado.

10.4. Supongamos que se tiene una secuencia de *n* números que deben ser clasificados:

1. Utilizando el método de selección, cuántas comparaciones y cuántos intercambios se requieren para clasificar la secuencia si:
   - Ya está clasificado.
   - Está en orden inverso.
2. Repetir el paso *i* para el método de selección.

10.5. Escribir un algoritmo de búsqueda lineal para un vector ordenado.

10.6. Un algoritmo ha sido diseñado para leer una lista de no más de 1.000 enteros positivos, cada uno menos de 100, y ejecutar algunas operaciones. El cero es la marca final de la lista. El programador debe obtener en el algoritmo.

1. Visualizar los números de la lista en orden creciente.
2. Calcular e imprimir la mediana (valor central).
3. Determinar el número que ocurre más frecuentemente.
4. Imprimir una lista que contenga:
   - Números menores de 30.
   - Números mayores de 70.
   - Números que no pertenezcan a los dos grupos anteriores.
5. Encontrar e imprimir el entero más grande de la lista junto con su posición en la lista antes de que los números hayan sido ordenados.

10.7. Diseñar diferentes algoritmos para insertar un nuevo valor en una lista (vector). La lista debe estar ordenada en orden ascendente antes y después de la inserción.