Capítulo 8

Algoritmos de ordenación y búsqueda

Objetivos

Una vez que se haya leído y estudiado este capítulo usted podrá:

- Conocer los algoritmos basados en el intercambio de elementos.
- Conocer el algoritmo de ordenación por inserción.
- Conocer el algoritmo de selección.
- Distinguir entre los algoritmos de ordenación basados en el intercambio y en la inserción.
- Saber la eficiencia de los métodos básicos de ordenación.
- Conocer los métodos más eficientes de ordenación.
- Diferenciar entre búsqueda secuencial y búsqueda binaria.

Contenido

Conceptos clave

- Búsqueda binaria.
- Búsqueda secuencial.
- Complejidad cuadrática.
- Complejidad logarítmica.
- Ordenación alfabética.
- Ordenación por burbuja.
- Ordenación numérica.
- Ordenación por intercambio.
- Ordenación por inserción.
- Ordenación por selección.
- Ordenación rápida.
- Residuos.

INTRODUCCIÓN

Muchas actividades humanas requieren que a diferentes colecciones de elementos utilizados se pongan en un orden específico. Las oficinas de correo y las empresas de mensajería ordenan el correo y los paquetes por códigos postales con el objeto de conseguir una entrega eficiente; las facturas telefónicas se ordenan por la fecha de las llamadas; los anuarios o listines telefónicos se ordenan por orden alfabético de apellidos con el fin último de encontrar fácilmente el número de teléfono deseado. Los estudiantes de una clase en la universidad se ordenan por sus apellidos o por los números de expediente. Por esta circunstancia una de las tareas que realizan más frecuentemente las computadoras en el procesamiento de datos es la *ordenación*.

El estudio de diferentes métodos de ordenación es una tarea intrínsecamente interesante desde un punto de vista teórico y, naturalmente, práctico. El capítulo estudia los algoritmos y técnicas de ordenación más usuales y su implementación en C++. Además, se analizan los diferentes métodos de ordenación con el objeto de conseguir la máxima eficiencia en su uso real.

8.1. ORDENACIÓN

La **ordenación** o **clasificación** de datos (*sort* en inglés) es una operación consistente en disponer un conjunto —estructura— de datos en algún determinado orden con respecto a uno de los *campos* de los elementos del conjunto. Por ejemplo, cada elemento del conjunto de datos de una guía telefónica tiene un campo nombre, un campo dirección y un campo número de teléfono; la guía telefónica está dispuesta en orden alfabético de nombres. Los elementos numéricos se pueden ordenar en orden creciente o decreciente de acuerdo al valor numérico del elemento. En terminología de ordenación, el elemento por el cual está ordenado un conjunto de datos (o se está buscando) se denomina *clave*.

Una colección de datos *(estructura)* puede ser almacenada en memoria central o en archivos de datos externos guardados en unidades de almacenamiento magnético (discos, cintas, CD-ROM, DVD, etc.). Cuando los datos se guardan en memoria principal —un *array,* una *lista enlazada* o un *árbol*— se denomina *ordenación interna;* estos datos se almacenan exclusivamente para tratamientos internos que se utilizan para gestión masiva de datos y se guardan en arrays de una o varias dimensiones. Si los datos están almacenados en un archivo, el proceso de ordenación se llama *ordenación externa*. Este capítulo estudia los métodos de *ordenación interna.*

Una *lista está ordenada por la clave k* si la lista está en orden ascendente o descendente con respecto a esta clave. La lista está en *orden ascendente* si:

```
i < j       implica que       k[i] <= k[j]
```

y está en *orden descendente* si:

```
i > j       implica que       k[i] <= k[j]
```

para todos los elementos de la lista. Por ejemplo, para una guía telefónica, la lista está clasificada en orden ascendente por el campo clave *k*, siendo *k* el nombre del abonado (apellidos, nombre).

```
4   5   14  21  32  45                          orden ascendente
75  70  35  16  14  12                          orden descendente

Zacarias Rodriguez Martinez Lopez Garcia        orden descendente
```

Los métodos (algoritmos) de ordenación son numerosos, por ello se debe prestar especial atención en su elección. ¿Cómo se sabe cuál es el mejor algoritmo? La *eficiencia* es el factor que mide la calidad y rendimiento de un algoritmo. En el caso de la operación de ordenación, dos criterios se suelen seguir a la hora de decidir qué algoritmo —de entre los que resuelven la ordenación— es el más eficiente: 1) *tiempo menor de ejecución en computadora;* 2) *menor número de instrucciones.* Sin embargo, no siempre es fácil efectuar estas medidas: puede no disponerse de instrucciones para medida de tiempo, y las instrucciones pueden variar, dependiendo del lenguaje y del propio estilo del programador. Por esta razón, el mejor criterio para medir la eficiencia de un algoritmo es aislar una operación específica clave en la ordenación y contar el número de veces que se realiza. Así, en el caso de los algoritmos de ordenación, se utilizará como medida de su eficiencia el número de comparaciones entre elementos efectuados. El algoritmo de *ordenación A* será más eficiente que el *B,* si requiere menor número de comparaciones. En la ordenación de los elementos de un array, el número de comparaciones será *función* del número de elementos *(n)* del array. Por consiguiente, se puede expresar el número de comparaciones en términos de *n*.

Todos los métodos de este capítulo, normalmente —para comodidad del lector— se *ordena de modo ascendente* sobre listas (arrays unidimensionales). Se suelen dividir en dos grandes grupos:

- *Directos* burbuja, selección, inserción.
- *Indirectos* (avanzados) shell, ordenación rápida, ordenación por mezcla, radixsort.

En el caso de listas pequeñas, los métodos directos se muestran eficientes, sobre todo porque los algoritmos son sencillos; su uso es muy frecuente. Sin embargo, en listas grandes estos métodos se muestran ineficaces y es preciso recurrir a los métodos avanzados.

> Existen dos técnicas de ordenación fundamentales en gestión de datos: *ordenación de listas* y *ordenación de archivos*. Los métodos de ordenación se conocen como *internos* o *externos* según que los elementos a ordenar estén en la memor ia principal o en la memoria externa.

8.2. ALGORITMOS DE ORDENACIÓN BÁSICOS

Existen diferentes algoritmos de ordenación elementales o básicos cuyos detalles de implementación se pueden encontrar en diferentes libros de algoritmos. La enciclopedia de referencia es [Knuth 1973][1] y sobre todo la 2.ª edición publicada en el año 1998 [Knuth 1998][2]. Los algoritmos presentan diferencias entre ellos que los convierten en más o menos eficientes y prácticos según sea la rapidez y eficiencia demostrada por cada uno de ellos. Los algoritmos básicos de ordenación más simples y clásicos son:

- Ordenación por selección.
- Ordenación por inserción.
- Ordenación por burbuja.

[1] [Knuth 1973] Donald E. Knuth. *The Art of Computer Programming*. Volume 3: *Sorting and Searching*. Addison-Wesley, 1973.

[2] [Knuth 1998] Donald E. Knuth. *The Art of Computer Programming*. Volume 3: *Sorting and Searching*. Second Edition. Addison-Wesley, 1998.

Los métodos más recomendados son el de *selección* y el de *inserción*, aunque se estudiará el método de *burbuja*, por aquello de ser el más sencillo aunque a la par también es el más *ineficiente*; por esta causa no se recomienda su uso, pero sí conocer su técnica.

Con el objeto de facilitar el aprendizaje del lector y aunque no sea un método utilizado por su poca eficiencia, se describe en primer lugar el método de *ordenación por intercambio*, debido a la sencillez de su técnica y con el objetivo de que el lector no introducido en los algoritmos de ordenación pueda comprender su funcionamiento y luego asimile más eficazmente los tres algoritmos básicos ya citados y los avanzados que se estudian más adelante.

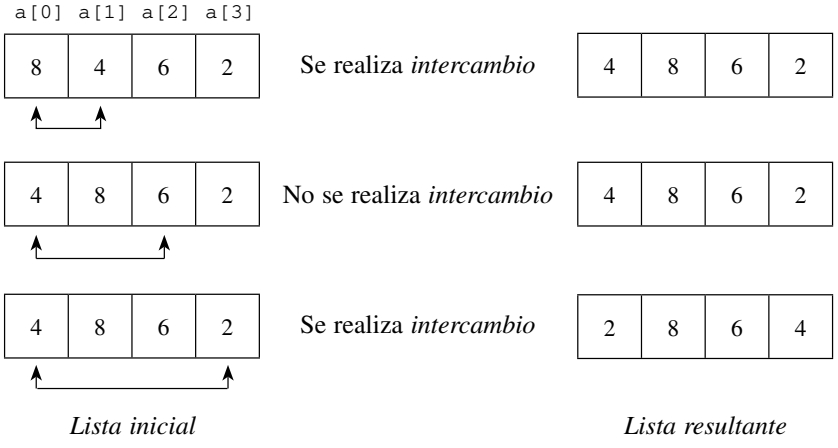
8.3. ORDENACIÓN POR INTERCAMBIO

El algoritmo se basa en la lectura sucesiva de la lista a ordenar, comparando el elemento inferior de la lista con los restantes y efectuando intercambio de posiciones cuando el orden resultante de la comparación no sea el correcto.

Los pasos que sigue el algoritmo se muestran al aplicarlo a la lista original 8, 4, 6, 2 que ha de convertirse en la lista ordenada 2, 4, 6, 8. El algoritmo efectua *n – 1* pasadas (3 en el ejemplo), siendo *n* el número de elementos.

Pasada 1

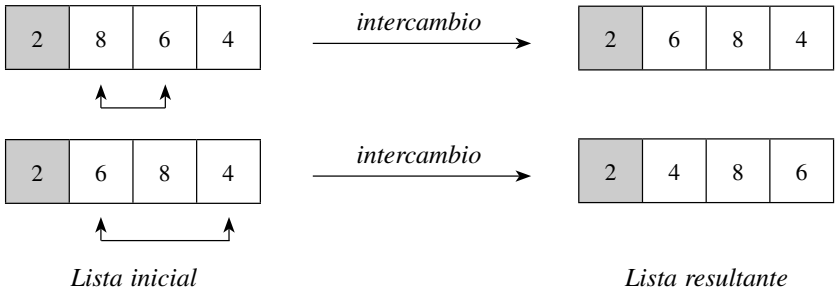
El elemento de índice `0 (a[0])` se compara con cada elemento posterior de la lista, de índices `1, 2` y `3`. Cada comparación comprueba si el elemento siguiente es más pequeño que el elemento de índice `0`, en cuyo caso se intercambian. Después de terminar todas las comparaciones, el elemento más pequeño se sitúa en el índice o.



Pasada 2

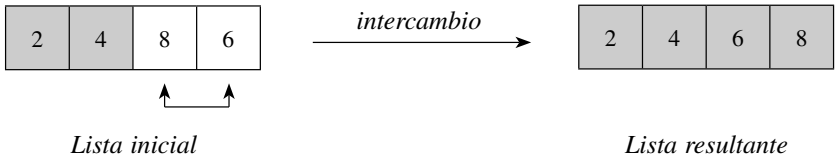
El elemento más pequeño ya está en la posición de índice `0`, ahora se considera la sublista restante 8, 6, 4. El algoritmo continúa comparando el elemento de índice `1` con los elementos posteriores de índices `2` y `3`. Por cada comparación, si el elemento mayor está en el índice `1`

se intercambian los elementos. Después de hacer todas las comparaciones, el segundo elemento más pequeño de la lista se almacena en el índice `1`.



Pasada 3

Ahora la sublista a considerar es 8, 6 ya que 2, 4 están ordenads. Una comparación única se produce entre los dos elementos de la sublista.



8.3.1. Codificación del algoritmo de ordenación por intercambio

El método `ordIntercambio()` implementa el algoritmo descrito, para ello utiliza dos bucles anidados. Separa una lista de tamaño *n*, el rango del bucle externo va de `0` a `n - 2`. Por cada índice *i,* se comparan los elementos posteriores de índices *j = i + 1, i + 2 ,..., n – 1*. El intercambio *(swap)* de dos elementos `a[i], a[j]` se realiza en esta función:

```
void intercambiar(int& x, int& y)
{
  int aux = x;
  x = y;
  y = aux;
}
```

Se supone que se ordena un *array* de n enteros:

```
void ordIntercambio (int a[], int n)
{
  int i, j;

  for (i = 0; i < n - 1; i++)
    // sitúa mínimo de a[i+1]...a[n-1] en a[i]
    for (j = i + 1; j < n; j++)
```

```
        if (a[i] > a[j])
        {
          intercambiar(a[i], a[j]);
        }
}
```

8.3.2. Complejidad del algoritmo de ordenación por intercambio

El algoritmo consta de dos bucles anidados, está *dominado* por los dos bucles. De ahí, que el análisis del algoritmo en relación a la complejidad sea inmediato, siendo `n` el número de elementos, el primer bucle hace `n-1` pasadas y el segundo `n-i-1` comparaciones en cada pasada (`i` es el índice del bucle externo, `i = 0 .. n-2`). El número total de comparaciones se obtiene desarrollando la sucesión matemática formada para los distintos valores de `i`:

```
n-1, n-2, n-3, ... , 1
```

El número de comparaciones se obtiene sumando los términos de la sucesión: $\frac{(n-1)\cdot n}{2}$

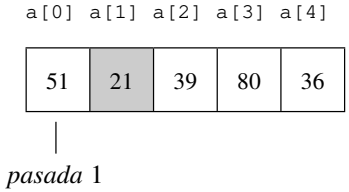
y un número similar de intercambios *en el peor de los casos*. Entonces, el número de comparaciones y de intercambios *en el peor de los casos* es $(n-1)\cdot n/2 = (n^2-n)/2$. El término dominante es n^2, por tanto la complejidad es $0(n^2)$.

8.4. ORDENACIÓN POR SELECCIÓN

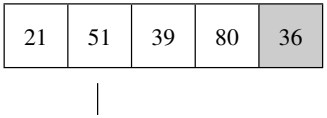
Considérese el algoritmo para ordenar un array `a[]` en orden ascendente; es decir, si el array tiene *n* elementos, se trata de ordenar los valores del array de modo que `a[0]` sea el valor más pequeño, el valor almacenado en `a[1]` el siguiente más pequeño, y así hasta `a[n-1]` que ha de contener el elemento mayor. El algoritmo de selección realiza *pasadas* que intercambian el elemento más pequeño sucesivamente, con el elemento del array que ocupa la posición igual al orden de *pasada* (hay que considerar el índice `0`).

La *pasada* inicial busca el elemento más pequeño de la lista y se intercambia con `a[0]`, primer elemento de la lista. Después de terminar esta primera pasada, el frente de la lista está ordenado y el resto de la lista `a[1], a[2] ... a[n-1]` permanece desordenada. La siguiente *pasada* busca en esta lista desordenada y *selecciona* el elemento más pequeño, que se almacena en la posición `a[1]`. De este modo los elementos `a[0]` y `a[1]` están ordenados y la sublista `a[2], a[3]...a[n-1]` desordenada. El proceso continúa hasta realizar `n-1` *pasadas*, en ese momento la lista desordenada se reduce a un elemento (el mayor de la lista) y el *array* completo ha quedado ordenado.

El siguiente ejemplo práctico ayudará a la comprensión del algoritmo:

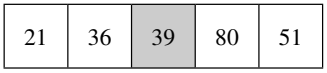


Pasada 1*: Seleccionar* `21`
Intercambiar `21` y `a[0]`



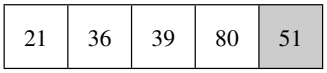
pasada 2

Pasada 2: Seleccionar `36`
Intercambiar `36` y `a[1]`



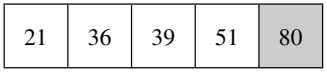
pasada 3

Pasada 3: Seleccionar `39`
Intercambiar `39` y `a[2]`



pasada 4

Pasada 4: Seleccionar `51`
Intercambiar `51` y `a[3]`



Array ordenado

8.4.1. Codificación del algoritmo de *selección*

La función `ordSeleccion()` ordena un array de números reales de `n` elementos. El proceso de selección explora, en la pasada `i`, la sublista `a[i]` a `a[n-1]` y fija el índice del elemento más pequeño. Después de terminar la exploración, los elementos `a[i]` y `a[indiceMenor]` se intercambian.

```
  /*
      ordenar un array de n elementos de tipo double
      utilizando el algoritmo de ordenación por selección
  */

void ordSeleccion (double a[], int n)
{
  int indiceMenor, i, j
         // ordenar a[0]..a[n-2] y a[n-1] en cada pasada
  for (i = 0; i < n - 1; i++)
  {
     // comienzo de la exploración en índice i
     indiceMenor = i;
                     // j explora la sublista a[i+1]..a[n-1]
     for (j = i + 1; j < n; j++)
       if (a[j] < a[indiceMenor])
          indiceMenor = j;
     // sitúa el elemento mas pequeño en a[i]
     if (i != indiceMenor)
```

```
        intercambiar(a[i], a[indiceMenor]);
    }
  }

  void intercambiar(double& x, double& y)
  {
    double aux = x;
    x = y;
    y = aux;
  }
```

8.4.2. Complejidad del algoritmo de *selección*

El análisis del algoritmo, con el fin de determinar la función *tiempo de ejecución* `t(n)`, es sencillo y claro, ya que requiere un número fijo de comparaciones que sólo dependen del tamaño del *array* y no de la distribución inicial de los datos. El término *dominante* del algoritmo es el bucle externo que anida a un bucle interno. Por ello, el número de comparaciones que realiza el algoritmo es el número decreciente de iteraciones del bucle interno: `n-1, n-2, ... 2, 1` (`n` es el número de elementos). La suma de los términos de la sucesión se ha obtenido en el apartado anterior, 8.3.2, y se ha comprobado que depende de n^2. Como conclusión, la complejidad del algoritmo de selección es $0(n^2)$.

8.5. ORDENACIÓN POR INSERCIÓN

Este método de ordenación es similar al proceso típico de ordenar tarjetas de nombres (cartas de una baraja) por orden alfabético, que consiste en insertar un nombre en su posición correcta dentro de una lista que ya está ordenada. Así el proceso en el caso de la lista de enteros `a[] = 50, 20, 40, 80, 30`

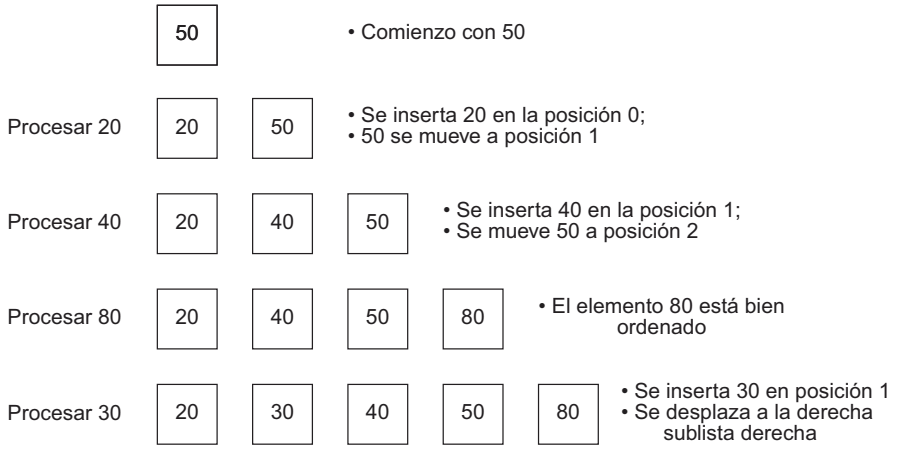


Figura 8.1. Método de ordenación por inserción

8.5.1. Algoritmo de ordenación por inserción

El algoritmo correspondiente a la ordenación por inserción contempla los siguientes pasos:

1. El primer elemento `a[0]` se considera ordenado; es decir, la lista inicial consta de un elemento.
2. Se inserta `a[1]` en la posición correcta; delante o detrás de `a[0]`, dependiendo de que sea menor o mayor.
3. Por cada iteración `i` (`desde i = 1 hasta n - 1`) se explora la sublista `a[i-1] ... a[0]` buscando la posición correcta de inserción de `a[i]`; a la vez se mueve *hacia abajo* (a la derecha en la sublista) una posición todos los elementos mayores que el elemento a insertar `a[i]`, para dejar vacía esa posición.
4. Insertar el elemento `a[i]` en la posición correcta.

8.5.2. Codificación del algoritmo de ordenación por *inserción*

La codificación del algoritmo se realiza en la función `ordInsercion()`. Los elementos del *array* son de tipo entero, en realidad puede ser cualquier tipo básico y ordinal.

```
void ordInsercion (int a[], int n)
{
  int i, j, aux;

  for (i = 1; i < n; i++)
  {
    /* indice j es para explorar la sublista a[i-1]..a[0] buscando la po-
       sicion correcta del elemento destino */
    j = i;
    aux = a[i];
      // se localiza el punto de inserción explorando hacia abajo
    while (j > 0 && aux < a[j-1])
    {
      // desplazar elementos hacia arriba para hacer espacio
      a[j] = a[j-1];
      j--;
    }
    a[j] = aux;
  }
}
```

8.5.3. Complejidad del algoritmo de *inserción*

A la hora de analizar este algoritmo se observa que el número de instrucciones que realiza depende del bucle externo que anida al bucle condicional `while`. Siendo `n` el número de elementos, el bucle externo realiza `n - 1` *pasadas*, por cada una de ellas y *en el peor de los casos* (`aux` siempre menor que `a[j-1]`), el bucle interno `while` itera un número creciente de veces que da lugar a la sucesión: `1, 2, 3, ... n-1` (para `i == n-1`). La suma de los términos de la sucesión se ha obtenido en el Apartado 8.3.2, y se ha comprobado que el término dominante es n^2. Como conclusión, la complejidad del algoritmo de inserción es $0(n^2)$.

8.6. ORDENACIÓN POR BURBUJA

El método de *ordenación por burbuja* es el más conocido y popular entre estudiantes y aprendices de programación, por su facilidad de comprender y programar; por el contrario, es el menos eficiente y por ello, normalmente, se aprende su técnica pero no suele utilizarse.

La técnica utilizada se denomina ***ordenación por burbuja*** u ***ordenación por hundimiento*** debido a que los valores más pequeños "*burbujean*" gradualmente (suben) hacia la cima o parte superior del *array* de modo similar a como suben las burbujas en el agua, mientras que los valores mayores se hunden en la parte inferior del *array*.

8.6.1. Algoritmo de la burbuja

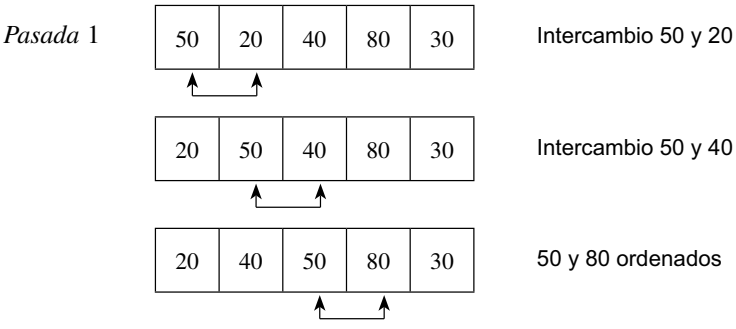
Para un *array* con *n* elementos, la ordenación por burbuja requiere hasta *n – 1* pasadas. Por cada pasada se comparan elementos adyacentes y se intercambian sus valores cuando el primer elemento es mayor que el segundo elemento. Al final de cada pasada, el elemento mayor ha "*burbujeado*" hasta la cima de la sublista actual. Por ejemplo, después que la pasada `1` está completa, la cola de la lista *a[n – 1]* está ordenada y el frente de la lista permanece desordenado. Las etapas del algoritmo son:

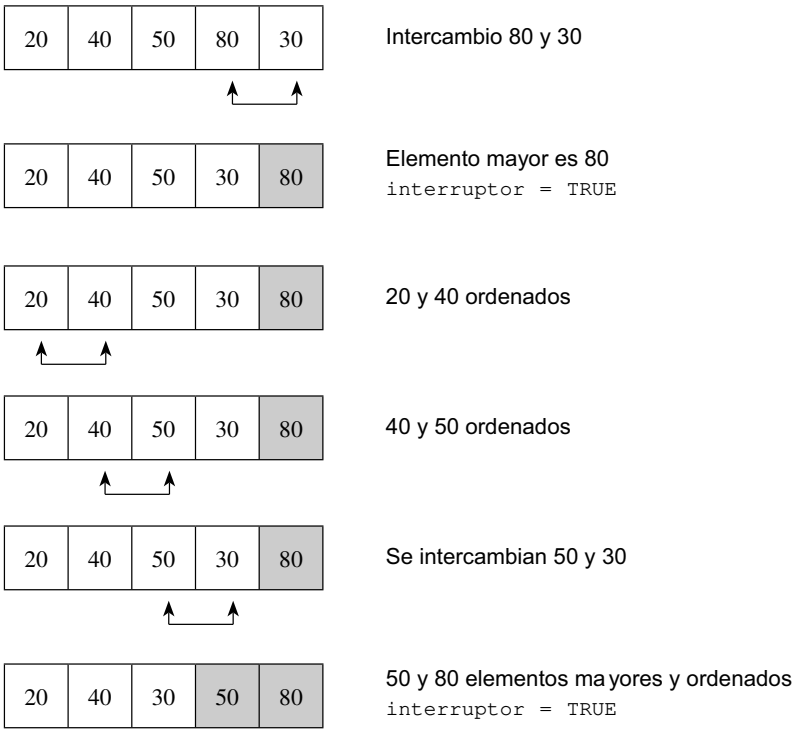
- En la pasada `1` se comparan elementos adyacentes.

  ```
  (a[0],a[1]),(a[1],a[2]),(a[2],a[3]),...(a[n-2],a[n-1])
  ```

  Se realizan *n* – 1 comparaciones, por cada pareja `(a[i],a[i+1])`, se intercambian los valores si `a[i+1] < a[i]`.
  Al final de la pasada, el elemento mayor de la lista está situado en `a[n-1]`.
- En la pasada `2` se realizan las mismas comparaciones e intercambios, terminando con el elemento de segundo mayor valor en `a[n-2]` .
- El proceso termina con la pasada *n – 1*, en la que el elemento más pequeño se almacena en `a[0]`.

El algoritmo tiene una mejora inmediata, el proceso de ordenación puede terminar en la pasada *n – 1*, o bien antes, si en un una pasada no se produce intercambio alguno entre elementos del *array* es porque ya está ordenado, entonces no es necesario más pasadas. A continuación, se ilustra el funcionamiento del algoritmo realizando las dos primeras pasadas en un array de 5 elementos; se introduce la variable `interruptor` para detectar si se ha producido intercambio en la pasada.





El algoritmo terminará cuando se termine la última pasada *(n – 1)*, o bien cuando el valor del `interruptor` sea *falso*, es decir no se haya hecho ningún intercambio.

8.6.2. Codificación del algoritmo de la burbuja

El algoritmo de ordenación de burbuja *mejorado* contempla dos bucles anidados: el *bucle externo* controla la cantidad de pasadas, el *bucle interno* controla cada pasada individualmente y cuando se produce un intercambio, cambia el valor de `interruptor` a *verdadero* `(true)`.

```
void ordBurbuja (long a[], int n)
{
  bool interruptor = true;
  int pasada, j;
    // bucle externo controla la cantidad de pasadas
  for (pasada = 0; pasada < n - 1 && interruptor; pasada++)
  {
    interruptor = false;
    for (j = 0; j < n - pasada - 1; j++)
      if (a[j] > a[j + 1])
      {
          // elementos desordenados, se intercambian
```

```
            interruptor = true;
            intercambiar(a[j], a[j + 1]);
          }
      }
  }
```

8.6.3. Análisis del algoritmo de la burbuja

¿Cuál es la eficiencia del algoritmo de ordenación de la burbuja?

La ordenación de burbuja hace una sola pasada en el caso de una lista que ya está ordenada en orden ascendente y, por tanto, su complejidad es *0(n)*. En el *caso peor* se requieren *(n – i – 1)* comparaciones y *(n – i – 1)* intercambios. La ordenación completa requiere $\frac{n(n-1)}{2}$ comparaciones y un número similar de intercambios. La complejidad para el caso peor es $0(n^2)$ comparaciones y $0(n^2)$ intercambios.

De cualquier forma, el análisis del caso general es complicado dado que alguna de las pasadas pueden no realizarse. Se podría señalar que el número medio de pasadas *k* es *0(n)* y el número total de comparaciones es $0(n^2)$. En el mejor de los casos, la ordenación por burbuja puede terminar en menos de *n – 1* pasadas pero requiere, normalmente, muchos más intercambios que la ordenación por selección y su prestación media es mucho más lenta, sobre todo cuando los arrays a ordenar son grandes.

Consejo de programación

Los algoritmos de ordenación interna: intercambio, selección, inserción y burbuja son fáciles de entender y de codificar, sin embargo poco eficientes y no recomendables para ordenar listas de muchos elementos. La complejidad de todos ellos es cuadrática, $0(n^2)$.

8.7. ORDENACIÓN SHELL

La ordenación Shell debe el nombre a su inventor, *D. L. Shell*. Se suele denominar también *ordenación por inserción con incrementos decrecientes*. Se considera que es una mejora del método de inserción directa.

En el algoritmo de inserción, cada elemento se compara con los elementos contiguos de su izquierda, uno tras otro. Si el elemento a insertar es el más pequeño hay que realizar muchas comparaciones antes de colocarlo en su lugar definitivo. El algoritmo de Shell modifica los saltos contiguos por saltos de mayor tamaño y con ello consigue que la ordenación sea más rápida. Generalmente, se toma como salto inicial $n/2$ (siendo n el número de elementos), luego en cada iteración se reduce el salto a la mitad, hasta que el salto es de tamaño 1. El Ejemplo 8.1 muestra paso a paso el método de Shell.

EJEMPLO 8.1. Aplicar el método Shell para ordenar en orden creciente la lista: 6 1 5 2 3 4 0

El número de elementos que tiene la lista es 6, por lo que el salto inicial es 6/2 = 3. La siguiente tabla muestra el número de recorridos realizados en la lista con los saltos correspondiente.

Recorrido	*Salto*	*Intercambios*	*Lista*
1	3	(6,2),(5,4),(6,0)	2 1 4 0 3 5 6
2	3	(2, 0)	0 1 4 2 3 5 6
3	3	ninguno	0 1 4 2 3 5 6
salto 3/2 = 1			
4	1	(4,2),(4,3)	0 1 2 3 4 5 6
5	1	ninguno	0 1 2 3 4 5 6

8.7.1. Algoritmo de ordenación Shell

Los pasos a seguir por el algoritmo para una lista de `n` elementos:

1. Se divide la lista original en `n/2` grupos de dos, considerando un incremento o salto entre los elementos de `n/2`.
2. Se clasifica cada grupo por separado, comparando las parejas de elementos y si no están ordenados se intercambian.
3. Se divide ahora la lista en la mitad de grupos (`n/4`), con un salto entre los elementos también mitad (`n/4`), y nuevamente se clasifica cada grupo por separado.
4. Así sucesivamente, se sigue dividiendo la lista en la mitad de grupos que en el recorrido anterior con un salto decreciente en la mitad que el salto anterior, y luego clasificando cada grupo por separado.
5. El algoritmo termina cuando el tamaño del salto es 1.

Por consiguiente, los recorridos por la lista están condicionados por el bucle,

```
salto ← n / 2
mientras (salto > 0) hacer
```

Para dividir la lista en grupos y clasificar cada grupo se anida este código:

```
desde i ← (salto + 1) hasta n hacer
  j ← i – salto
  mientras (j > 0) hacer
    k ← j + salto
    si (a[j] <= a[k]) entonces
      j ← 0
    sino
      Intercambio (a[j], a[k])
      j ← j – salto
    fin_si
  fin_mientras
fin_desde
```

Donde se observa que se comparan pares de elementos de índice `j` y `k`, separados por un *salto* de `salto` . Así, si `n = 8` el primer valor de `salto = 4`, y los índices `i = 5`, `j = 1`, `k = 6` . Los siguiente valores que toman son `i = 6`, `j = 2`, `k = 7`, y así hasta recorrer la lista.

8.7.2. Codificación del algoritmo de ordenación Shell

Al codificar el algoritmo se considera que el rango de elementos es `0 ... n-1` y, por consiguiente, se ha de desplazar una posición a la *izquierda* las variables índice respecto a lo expuesto en el algoritmo.

```
void ordenacionShell(double a[], int n)
{
  int salto, i, j, k;
  salto = n / 2;
  while (salto > 0)
  {
    for (i = salto; i < n; i++)
    {
      j = i - salto;
      while (j >= 0)
      {
        k = j + salto;
        if (a[j] <= a[k])
          j = -1;          // par de elementos ordenado
        else
        {
          intercambiar(a[j], a[j+1]);
          j -= salto;
        }
      }
    }
    salto = salto / 2;
  }
}
```

8.7.3. Análisis del algoritmo de ordenación Shell

A pesar de que el algoritmo tiene tres bucles anidados `(while-for-while)`, es más eficiente que el algoritmo de inserción y que cualquiera de los algoritmos simples analizados en los apartados anteriores. El análisis del tiempo de ejecución del algoritmo *Shell* no es sencillo. Su inventor, *Shell*, recomienda que el salto inicial sea `n/2`, y continuar dividiendo el salto por la mitad hasta conseguir un salto `1`. Con esta elección se puede probar que el tiempo de ejecución es $O(n^2)$ *en el peor de los casos*, y el tiempo medio de ejecución es $O(n^{3/2})$.

Posteriormente, se han encontrado secuencias de saltos que mejoran el rendimiento del algoritmo. Así, dividiendo el salto por 2.2 en lugar de la mitad se consigue un tiempo medio de ejecución de complejidad menor de $O(n^{5/4})$.

Nota de programación

La codificación del algoritmo Shell con el salto igual al salto anterior dividido por `2.2`, puede hacer el salto igual a `0`. Si esto ocurre, se ha de codificar que el salto sea igual a `1`, en caso contrario no funcionaría el algoritmo.

```
salto = (int) salto / 2.2;
salto = (salto == 0) ? 1 : salto;
```

8.8. ORDENACIÓN RÁPIDA (*QUICKSORT*)

El algoritmo conocido como *quicksort* (ordenación rápida) recibe el nombre de su autor, *Tony Hoare*. El fundamento del algoritmo es simple, se basa en la división de la lista en particiones a ordenar, en definitiva aplica la técnica `"divide y vencerás"`. El método es, posiblemente, el más pequeño de código, más rápido de media, más elegante y más interesante y eficiente de los algoritmos conocidos de ordenación.

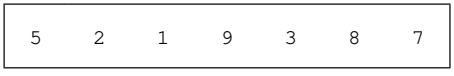
El algoritmo divide los *n* elementos de la lista a ordenar en dos partes o particiones separadas por un elemento: una partición *izquierda*, un elemento *central* denominado *pivote*, y una partición *derecha*. La partición se hace de tal forma que todos los elementos de la primera sublista (partición izquierda) son menores que todos los elementos de la segunda sublista (partición derecha). Las dos sublistas se ordenan entonces independientemente.

Para dividir la lista en particiones (sublistas) se elige uno de los elementos de la lista como *pivote* o *elemento de partición*. Si los elementos de la lista están en orden aleatorio, se puede elegir cualquier elemento como *pivote*, por ejemplo, el primer elemento de la lista. Si la lista tiene algún orden parcial, que se conoce, se puede tomar otra decisión para el *pivote*. Idealmente, el *pivote* se debe elegir de modo que se divida la lista por la mitad, de acuerdo al tamaño relativo de las claves. Por ejemplo, si se tiene una lista de enteros de 1 a 10, 5 o 6 serían *pivotes* ideales, mientras que 1 o 10 serían elecciones "pobres" de *pivotes*.

Una vez que el *pivote* ha sido elegido, se utiliza para ordenar el resto de la lista en dos sublistas: una tiene todas las claves menores que el *pivote* y la otra todas las claves mayores o iguales que el *pivote*. Estas dos listas parciales se ordenan recursivamente utilizando el mismo algoritmo; es decir, se llama sucesivamente al propio algoritmo *quicksort*. La lista final ordenada se consigue concatenando la primera sublista, el *pivote* y la segunda sublista, en ese orden, en una única lista. La primera etapa de *quicksort* es la división o *"particionado"* recursivo de la lista hasta que todas las sublistas consten de sólo un elemento.

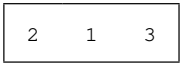
EJEMPLO 8.2. Se ordena una lista de números enteros aplicando el algor itmo *quicksort*, se elige como pivote el primer elemento de la lista.

1. *Lista original*

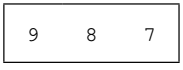


pivote elegido `5`

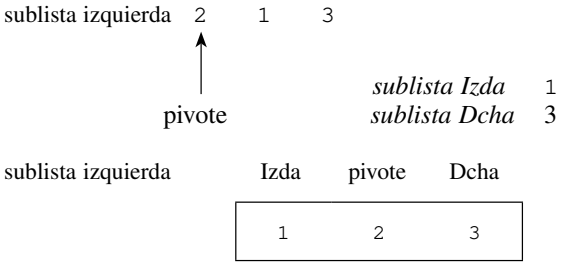
sublista *izquierda* (elementos menores que `5`)



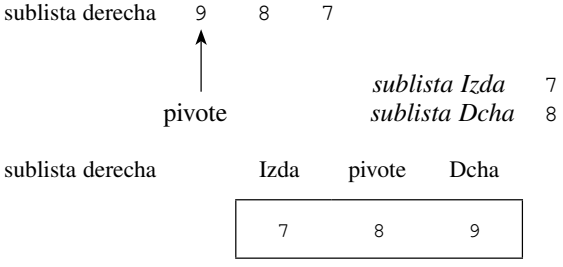
sublista *derecha* (elementos mayores o iguales a `5`)



2. El algoritmo se aplica a la sublista izquierda



3. El algoritmo se aplica a la sublista derecha



4. Lista ordenada final

Sublista izquierda *pivote* *Sublista derecha*

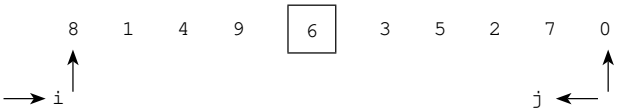
1 2 3 5 7 8 9

EJEMPLO 8.3. Se aplica el algoritmo *quicksort* para dividir una lista de números enteros en dos sublistas. El pivote es el elemento central de la lista.

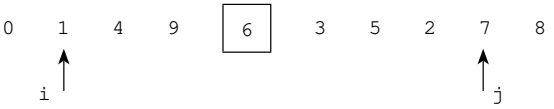
Lista original: 8 1 4 9 6 3 5 2 7 0
pivote (elemento central) 6

Una vez elegido el *pivote*, la segunda etapa requiere mover todos los elementos menores al pivote a la parte izquierda del *array* y los elementos mayores a la parte derecha. Para ello se recorre la lista de izquierda a derecha utilizando un índice `i`, que se inicializa a la posición más baja (`inferior`), buscando un elemento mayor al pivote. También se recorre el array de derecha a izquierda buscando un elemento menor. Para esto se utilizará el índice `j`, inicializado a la posición más alta (`superior`).

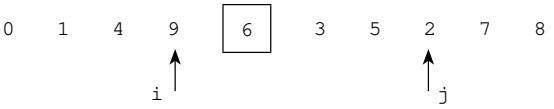
El índice `i` se detiene en el elemento `8` (mayor que el *pivote*) y el índice `j` se detiene en el elemento `0` (menor que el *pivote*).



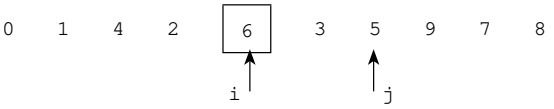
Ahora, se intercambian `a[i]` y `a[j]` para que estos dos elementos se sitúen correctamente en cada sublista; y se incrementa el índice `i`, y se decrementa `j` para seguir los intercambios.



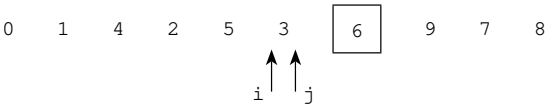
A medida que el algoritmo continúa, `i` se detiene en el elemento mayor, `9`, y `j` se detiene en el elemento menor, `2`,



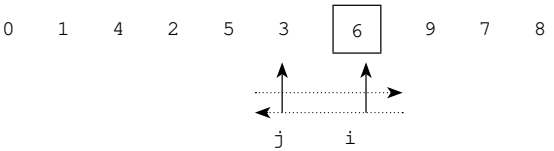
Se intercambian los elementos mientras que `i` y `j` no se crucen. En caso contrario se detiene este bucle. En el caso anterior se intercambian `9` y `2`.



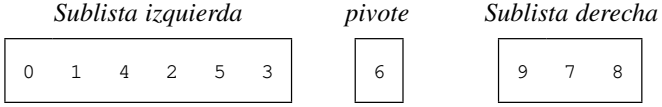
Continúa la exploración y ahora el contador `i` se detiene en el elemento `6` (que es el *pivote*) y el índice `j` se detiene en el elemento menor `5`



Los índices tienen actualmente los valores `i = 5`, `j = 5`. Continúa la exploración hasta que `i > j`, acaba con `i = 6`, `j = 5`

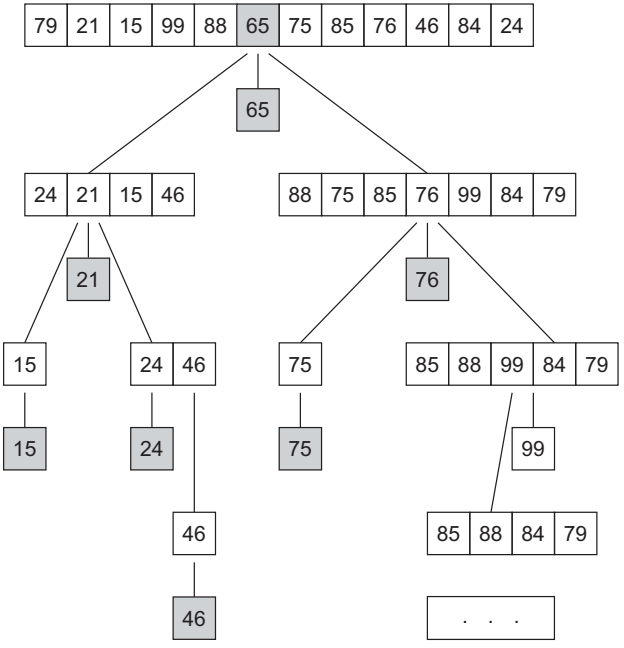


En esta posición los índices `i` y `j` han cruzado posiciones en el *array*, se detiene la búsqueda y no se realiza ningún intercambio ya que el elemento al que accede `j` está ya correctamente situado. Las dos sublistas ya han sido creadas, la lista original se ha dividido en dos particiones:



8.8.1. Algoritmo *Quicksort*

El primer problema a resolver en el diseño del algoritmo de *quicksort* es seleccionar el *pivote*. Aunque la posición del *pivote*, en principio puede ser cualquiera, una de las decisiones más ponderadas es aquella que considera el *pivote* como el elemento central o próximo al central de la lista. Una vez que se ha seleccionado el *pivote*, se ha de buscar el sistema para situar en la sublista izquierda todos los elementos menores que el *pivote* y en la sublista derecha todos los elementos mayores. La Figura 8.2 muestra las operaciones del algoritmo para ordenar la lista `a[]` de `n` elementos enteros.



Izquierda : 24, 21, 15, 46
Pivote : 65
Derecha : 88, 75, 85, 76, 99, 84, 79

Figura 8.2. Ordenación rápida eligiendo como pivote el elemento central.

Los pasos que sigue el algoritmo *quicksort* son:

Seleccionar el elemento central de `a[]` como *pivote*.
Dividir los elementos restantes en particiones *izquierda* y *derecha*, de modo que ningún elemento de la izquierda tenga una clave mayor que el *pivote* y que ningún elemento a la derecha tenga una clave más pequeña que la del *pivote*.
Ordenar la partición izquierda utilizando *quicksort* recursivamente.
Ordenar la partición derecha utilizando *quicksort* recursivamente.
La solución es partición *izquierda* seguida por el *pivote* y la partición *derecha*.

8.8.2. Codificación del algoritmo *QuickSort*

La implementación, al igual que el algoritmo, es recursiva; la función `quicksort()` tiene como argumentos el array `a[]` y los índices que le delimitan: `primero` y `ultimo`.

```
void quicksort(double a[], int primero, int ultimo)
{

  int i, j, central;
  double pivote;

  central = (primero + ultimo) / 2;
  pivote = a[central];
  i = primero;
  j = ultimo;

  do {
     while (a[i] < pivote) i++;
     while (a[j] > pivote) j--;

     if (i <= j)
     {
        intercambiar(a[i], a[j]);
        i++;
        j--;
     }
  }while (i <= j);

   if (primero < j)
     quicksort(a, primero, j); // mismo proceso con sublista izqda
   if (i < ultimo)
     quicksort(a, i, ultimo); // mismo proceso con sublista drcha
}
```

8.8.3. Análisis del algoritmo *Quicksort*

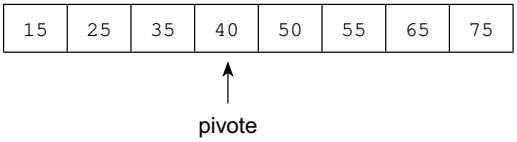
El análisis general de la eficiencia del *quicksort* es difícil. La mejor forma de determinar la complejidad del algoritmo es considerar el número de comparaciones realizadas teniendo en cuenta circunstancias ideales. Supongamos que *n* (número de elementos) es una potencia de *2*,

$n = 2^k$ ($k = log_2 n$). Además, supongamos que el *pivote* es el elemento central de cada lista, de modo que *quicksort* divide la sublista en dos sublistas aproximadamente iguales.

En el primer recorrido se realizan $n - 1$ comparaciones. Este recorrido crea dos sublistas, aproximadamente de tamaño $n/2$. En la siguiente etapa, el proceso de cada sublista requiere aproximadamente $n/2$ comparaciones. Las comparaciones totales de esta fase son $2*(n/2) = n$. A continuación, se procesan cuatro sublistas que requieren un total de $4*(n/4)$ comparaciones, etcétera. Eventualmente, el proceso de división termina después de k pasadas cuando la sublista resultante tenga tamaño 1. El número total de comparaciones es aproximadamente:

$$n + 2*(n/2) + 4*(n/4) + ... + n*(n/n) = n + n + ... + n = n * k = n * log_2 n$$

El caso ideal que se ha examinado se realiza realmente cuando la lista está ordenada en orden ascendente. En este caso el *pivote* es siempre el centro de cada sublista y el algoritmo tiene la complejidad $0(n log n)$.



El escenario del caso peor de *quicksort* ocurre cuando el *pivote* cae consistentemente en una sublista de un elemento y deja el resto de los elementos en la segunda sublista. Esto sucede cuando el pivote es siempre el elemento más pequeño de su sublista. En el recorrido inicial, hay n comparaciones y la sublista grande contiene $n - 1$ elementos. En el siguiente recorrido, la sublista mayor requiere $n - 1$ comparaciones y produce una sublista de $n - 2$ elementos, etc. El número total de comparaciones es:

$$n + n - 1 + n - 2 + ... + 2 = (n - 1) * (n + 2)/2$$

Entonces, la complejidad en el caso peor es $0(n^2)$. En general, el algoritmo de ordenación *quicksort* tiene como complejidad media $0(n log n)$ siendo posiblemente el algoritmo más rápido. La Tabla 8.1 muestra las complejidades de los algoritmos empleados en los métodos explicados en el libro.

Tabla 8.1. Comparación complejidad métodos de ordenación.

Método	Complejidad
Burbuja	n^2
Inserción	n^2
Selección	n^2
Montículo	$n log n$
Mergersort	$n log n$
Shell	$n^{3/2}$
Quicksort	$n log n$

En conclusión, se suele recomendar que para listas pequeñas, los métodos más eficientes son: inserción y selección, y para listas grandes: *quicksort, Shell, mergesort* (Apartado 7.5), y *montículo* (se desarrolla en Capítulo 13; *Colas de Prioridades y Montículos*).

8.9. ORDENACIÓN CON *URNAS*: *BINSORT* Y *RADIXSORT*

Estos métodos de ordenación utilizan *urnas* en el proceso de ordenación. En cada recorrido se deposita en una $urna_i$ aquellos elementos del array cuya clave tienen cierta correspondencia con el índice i.

8.9.1. *Binsort* (ordenación por urnas)

Este método, también denominado *clasificación por urnas,* se propone conseguir funciones tiempo de ejecución de complejidad menor que *O(n log n)* para ordenar una lista de `n` elementos, siempre que se conozca alguna relación del campo *clave* de los elementos respecto de las urnas.

Supóngase este caso ideal: se desea ordenar un array `v[]` respecto un campo clave de tipo entero, además los valores de las claves se encuentran en el rango de `1` a `n`, sin claves duplicadas y siendo `n` el número de elementos. En estas circunstancias ideales es posible ubicar los registros ordenados en un array auxiliar `t[]` mediante este único bucle:

```
desde i ← 1 hasta n hacer
  t[v[i].clave] ← v[i];
fin_desde
```

Sencillamente, determina la posición que le corresponde al registro según el valor del campo clave. El bucle lleva un tiempo de ejecución de complejidad lineal `O(n)` .

Esta ordenación tan sencilla e intuitiva es un caso particular del método binsort. El método utiliza urnas, de tal forma que una urna contiene todos los registros con una misma clave.

El proceso consiste en examinar cada elemento, `r`, del array y situarle en la urna `i`, siendo `i` el valor del campo clave de `r`. Es previsible que sea necesario guardar más de un elemento en una misma urna por tener claves repetidas. Entonces, las urnas hay que concatenarlas, en el orden de menor índice de urna a mayor, para que el array quede ordenado (ascendentemente) respecto al campo clave.

La Figura 8.3 muestra un vector de `m` urnas. Cada urna están representada mediante una lista enlazada.

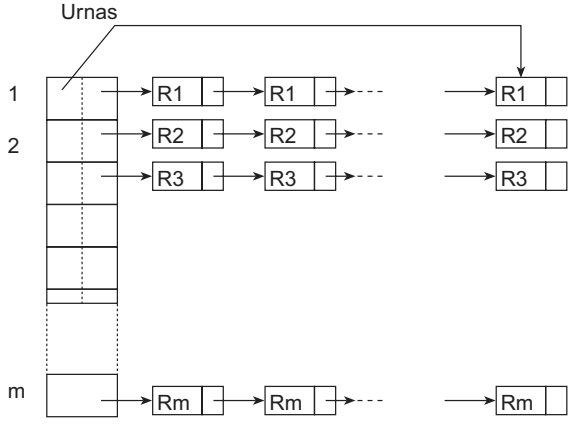


Figura 8.3. Estructura formada por m urnas.

Algoritmo *Binsort*

El algoritmo se aplica sobre elementos que se ordenan respecto una clave cuyo rango de valores es relativamente pequeño respecto al número de elementos. Si la clave es de tipo entero, en el rango `1 .. m`, son necesarias `m` urnas agrupadas en un vector. Las urnas son listas enlazadas, cada nodo de la lista contiene un elemento cuya clave se corresponde con el índice de la urna en la que se encuentra. Para una mejor comprensión del algoritmo, la clave será igual al índice de la urna. Así, en la urna `1` se sitúan los elementos cuya clave es `1`, en la urna `2` los elementos de clave `2`, y así sucesivamente en la urna *i* se sitúan los registros cuya clave sea *i*.

La primera acción del algoritmo consiste en distribuir los elementos en las diversas urnas. A continuación, se concatena las listas enlazadas para formar un única lista, que contendrá los elementos en orden creciente; por último, se recorre la lista asignando cada nodo al array. La Figura 8.4 se muestra cómo realizar la concatenación.

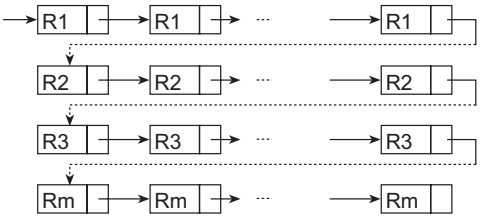


Figura 8.4. Concatenación de urnas representadas por listas enlazadas.

El algoritmo expresado en seudocódigo para un vector de *n* elementos:

```
OrdenacionBinsort(vector, n)

inicio
  CrearUrnas(Urnas);
    {Distribución de elementos en sus correspondientes urnas}
  desde j ← 1 hasta n hacer
    AñadirEnUrna(Urnas[vector[j].clave], vector[j]);
  fin_desde

    {Concatena las listas que representan a las urnas
      desde Urna_i hasta Urna_m}

  i ← 1;                          {búsqueda de primera urna no vacía}
  mientras EsVacia(Urnas[i]) hacer
    i ← i+1
  fin_mientras

  desde j ← i+l a m hacer
    EnlazarUrna(Urnas[i], Urnas[j]);
  fin_desde

  {recorre las lista(urnas) resultado de la concatenación}
  j ← 1;
  dir = <frente Urnas[i]>;
```

```
  mientras dir <> nulo hacer
    vector[j] = < elemento apuntado por dir>;
    j ← j+i;
    dir ← Sgte(dir)
  fin_mientras

fin
```

8.9.2. RadixSort *(ordenación por residuos)*

Este método de ordenación es un caso particular del algoritmo de clasificación por urnas. La manera de ordenar, manualmente, un conjunto de fichas nos da una idea intuitiva de este método de ordenación: se forman *montones* de fichas, cada uno caracterizado por tener sus componentes un mismo dígito (letra, si es ordenación alfabética) en la misma posición. Inicialmente se forman los *montones* por las unidades (dígito de menor peso); estos montones se recogen y agrupan en orden ascendente, desde el montón del dígito 0 al montón del dígito 9. Entonces, las fichas están ordenadas respecto a las unidades, a continuación, se vuelve a distribuir las fichas en montones, según el dígito de las decenas. El proceso de distribuir las fichas por montones y posterior acumulación en orden se repite tantas veces como número de dígitos tiene la ficha de mayor valor.

Suponer que las fichas están identificadas por un campo entero de tres dígitos, los pasos del algoritmo *RadixSort* para los siguientes valores:

```
345, 721, 425, 572, 836, 467, 672,194, 365, 236, 891, 746, 431, 834, 247,
529, 216, 389
```

Atendiendo al dígito de menor peso (unidades) los *montones*:

				216		
431			*365*	*746*		
891	*672*	*834*	*425*	*236*	*247*	*389*
721	*572*	*194*	*345*	*836*	*467*	*529*
1	*2*	*4*	*5*	*6*	*7*	*9*

Una vez agrupados los montones en orden ascendente la lista es la siguiente:

```
721, 891, 431, 572, 672, 194, 834, 345, 425, 365, 836, 236, 746, 216, 467,
247, 529, 389
```

Esta lista ya está ordenada respecto al dígito de menor peso, respecto a las unidades. Pues bien, ahora se vuelven a distribuir en *montones* respecto al segundo dígito (decenas):

		236					
	529	*836*	*247*				
	425	*834*	*746*	*467*	*672*		*194*
216	*721*	*431*	*345*	*365*	*572*	*389*	*891*
1	*2*	*3*	*4*	*6*	*7*	*8*	*9*

Una vez agrupados los *montones* en orden ascendente la lista es la siguiente:

```
216, 721, 425, 529, 431, 834, 836, 236, 345, 746, 247, 365, 467, 572, 672,
389, 891, 194
```

La lista fichas ya está ordenada respecto a los dos últimos dígitos, es decir, respecto a las decenas. Por último, se vuelven a distribuir en *montones* respecto al tercer dígito:

	247	389	467				891
	236	365	431	572		746	836
194	216	345	425	529	672	721	834
1	2	3	4	5	6	7	8

Se agrupan los *montones* en orden ascendente y la lista ya está ordenada:

```
194, 216, 236, 247, 345, 365, 389, 425, 431, 467, 529, 572, 672, 721, 746,
834, 836, 891
```

Algoritmo de ordenación RadixSort

La idea clave de la ordenación RadixSort es clasificar por urnas tantas veces como máximo número de dígitos (o de letras) tengan los elementos de la lista. En cada paso se realiza una distribución en las urnas y una unión de éstas en orden ascendente, desde la urna *0* a la urna 9.

Al igual que en el método de *BinSort*, las urnas se representan mediante un array de listas enlazadas. Se ha de disponer de tantas urnas como dígitos, `10`, numeradas de `0` a 9. Si la clave respecto a la que se ordena es alfabética, habrá tantas urnas como letras distintas, desde la urna que representa a la letra *a* hasta la *z*.

El algoritmo que se escribe, en primer lugar determina el número máximo de dígitos que puede tener una clave. Un bucle externo, de tantas iteraciones como el máximo de dígitos, realiza las acciones de distribuir por urnas los elementos y concatenar.

```
OrdenacionRadixsort(vector, n)

inicio
  { cálculo el número máximo de dígitos: ndig }
  ndig ← 0;
  temp ← maximaClave;
  mientras (temp > 0) hacer
    ndig ← ndig+1
    tem ← temp / 10;
  fin_mientras

  peso ← 1 { permite obtener los dígitos de menor a mayor peso}
  desde i ← 1 hasta ndig hacer
    CrearUrnas(Urnas);
    desde j ← 1 hasta n hacer
      d ← (vector[j] / peso) modulo 10;
      AñadirEnUma(Urnas[d], vector[j]);
    fin_desde

    { búsqueda de primera urna no vacía: j }
        j ← 0;
        mientras frente (Urnas i, j <> nulo) hacer
        j ← j + 1;
    desde r ← j+1 hasta M hace { M: número de urnas }
      EnlazarUma(Urnas[r], Urnas[j]);
    fin_desde
```

```
      {Se recorre la lista resultado de la concatenación}
    r ← 1;
    dir ← frente(Urnas[j]);
    mientras dir <> nulo hacer
      vecto[r] ← dir.elemento;
      r ← r+1;
      dir ← siguiente(dir)
    fin_mientras
    peso ← peso * 10;
  fin_desde
fin_ordenacion
```

8.10. BÚSQUEDA EN LISTAS: BÚSQUEDA SECUENCIAL Y BINARIA

Con mucha frecuencia los programadores trabajan con grandes cantidades de datos almacenados en arrays y registros y, por ello, será necesario determinar si un array contiene un valor que coincida con un *valor clave*. El proceso de encontrar un elemento específico de un array se denomina *búsqueda*. En esta sección se examinarán dos técnicas de búsqueda: *búsqueda lineal* o *secuencial*, la más sencilla, y *búsqueda binaria* o *dicotómica*, la más eficiente.

8.10.1. Búsqueda secuencial

La búsqueda secuencial busca un elemento de una lista utilizando un valor destino llamado ***clave***. En una búsqueda secuencial (a veces llamada ***búsqueda lineal***), los elementos de una lista se exploran (se examinan) en secuencia, uno después de otro.

El algoritmo de búsqueda secuencial compara cada elemento del array con la *clave* de búsqueda. Dado que el array no está en un orden prefijado, es probable que el elemento a buscar pueda ser el primer elemento, el último elemento o cualquier otro. De promedio, al menos el programa tendrá que comparar la clave de búsqueda con la mitad de los elementos del array. El método de búsqueda lineal funcionará bien con arrays pequeños o no ordenados.

8.10.2. Búsqueda binaria

La búsqueda secuencial se aplica a cualquier lista. Si la lista está ordenada, la *búsqueda binaria* proporciona una técnica de búsqueda mejorada. Localizar una palabra en un diccionario es un ejemplo típico de búsqueda binaria. Dada la palabra, se abre el libro cerca del principio, del centro o del final dependiendo de la primera letra de la palabra que busca. Se puede tener suerte y acertar con la página correcta; pero, normalmente, no será así y se mueve el lector a la página anterior o posterior del libro. Por ejemplo, si la palabra comienza con "J" y se está en la "L" se mueve uno hacia atrás. El proceso continúa hasta que se encuentra la página buscada o hasta que se descubre que la palabra no está en la lista.

Una idea similar se aplica en la búsqueda en una lista ordenada. Se sitúa el índice de búsqueda en el centro de la lista y se comprueba si nuestra clave coincide con el valor del elemento central. Si no se encuentra el valor de la clave, se sitúa en la mitad inferior o superior del elemento central de la lista. En general, si los datos de la lista están ordenados se puede utilizar esa información para acortar el tiempo de búsqueda.

EJEMPLO 8.4. Se desea buscar el elemento 225 en el conjunto de datos siguiente:

```
a[0]  a[1]  a[2]  a[3]  a[4]  a[5]  a[6]  a[7]
 13    44    75   100   120   275   325   510
```

El punto central de la lista es el elemento `a[3] (100)`. El valor que se busca es `225` que es mayor que `100`; por consiguiente, la búsqueda continúa en la mitad superior del conjunto de datos de la lista, es decir, en la sublista:

```
a[4]  a[5]  a[6]  a[7]
120   275   325   510
```

Ahora el elemento mitad de esta sublista es `a[5] (275)`. El valor buscado, `225`, es menor que `275` y, por consiguiente, la búsqueda continúa en la mitad inferior del conjunto de datos de la lista actual; es decir en la sublista unitaria: `a[4](120)`.

El elemento mitad de esta sublista es el propio elemento `a[4](120)` y al ser `225` mayor que `120` la búsqueda continuar en una sublista vacía. Se concluye indicando que no se ha encontrado la clave en la lista.

8.10.3. Algoritmo y codificación de la búsqueda binaria

Suponiendo que la lista está en un array delimitado por índices `bajo` y `alto`, los pasos a seguir:

1. Calcular el índice del punto central del array:

   ```
   central = (bajo + alto)/2
   ```
   (división entera)

2. Comparar el valor de este elemento central con la clave:

   ```
   clave = a[central]      a[central] > clave      a[central] < clave
   ├──────┼──────┤         ├──────┼──────┤         ├──────┼──────┤
   bajo central  alto      bajo central alto       bajo central alto
   ```

   Clave encontrada *Búsqueda lista inferior* *Búsqueda lista superior*

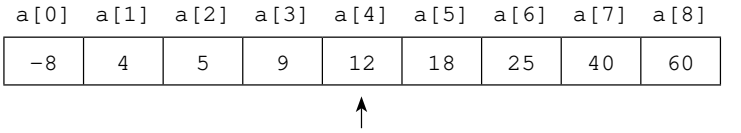
   Si `a[central] < clave`, la nueva sublista de búsqueda queda delimitada por:
   `bajo = central+1 .. alto`
   Si `a[central] > clave`, la nueva sublista de búsqueda queda delimitada por:
   `bajo .. alto central-1`

   ```
          clave                          clave
   [──────────────────]          [──────────────────]
   bajo        central-1=alto    bajo=central+1     alto
   ```

El algoritmo termina bien porque se ha encontrado la clave o porque el valor de `bajo` excede a `alto` y el algoritmo devuelve el indicador de fallo de `-1`.

EJEMPLO 8.5. Sea el array de enteros a `(-8, 4, 5, 9, 12, 18, 25, 40, 60)`, buscar la clave 40.

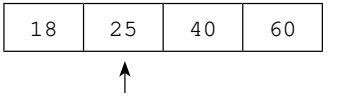
1.



$$central = \frac{bajo + alto}{2} = \frac{0 + 8}{2} = 4$$

`clave (40) > a[4] (12)`

2. Buscar en sublista derecha



$$central = \frac{bajo + alto}{2} = \frac{5 + 8}{2} = 6 \qquad \text{(división entera)}$$

`clave (40) > a[6] (25)`

3. Buscar en sublista derecha

$$central = \frac{bajo + alto}{2} = \frac{7 + 8}{2} = 7$$

`clave (40) = a[7](40)` *búsqueda con éxito*

El algoritmo ha requerido `3` comparaciones frente a `8` comparaciones `(n-1)` que se hubieran realizado con la búsqueda secuencial.

Codificación

```
int busquedaBin(int a[], int n, int clave)
{
  int central, bajo, alto;
  int valorCentral;
  bajo = 0;
  alto = n - 1;
  while (bajo <= alto)
```

```
  {
    central = (bajo + alto)/2;       // índice de elemento central
    valorCentral = a[central];       // valor del índice central
    if (clave == valorCentral)
      return central;                // encontrado, devuelve posición
    else if (clave < valorCentral)
      alto = central - 1;            // ir a sublista inferior
    else
      bajo = central + 1;            // ir a sublista superior
  }
  return -1;                         //elemento no encontrado
}
```

8.10.4. Análisis de los algoritmos de búsqueda

Al igual que sucede con las operaciones de ordenación, cuando se realizan operaciones de búsqueda es preciso considerar la eficiencia (complejidad) de los algoritmos empleados en la búsqueda. El grado de eficiencia en una búsqueda será vital cuando se trata de localizar un dato en una lista o tabla en memoria de muchos elementos.

Complejidad de la búsqueda secuencial

La complejidad diferencia entre el comportamiento en el *caso peor y mejor*. El *mejor caso* se encuentra cuando aparece una coincidencia en el primer elemento de la lista y en ese caso el tiempo de ejecución es *O(1)*. El caso peor se produce cuando el elemento no está en la lista o se encuentra al final de la lista. Esto requiere buscar en todos los *n* términos, lo que implica una complejidad de *O(n)*.

Análisis de la búsqueda binaria

El *caso mejor* se presenta cuando una coincidencia se encuentra en el punto central de la lista. En este caso la complejidad es *O(1)* dado que sólo se realiza una prueba de comparación de igualdad. La complejidad del *caso peor* es *O(log2n)* que se produce cuando el elemento no está en la lista o el elemento se encuentra en la última comparación. Se puede deducir intuitivamente esta complejidad. El caso peor se produce cuando se debe continuar la búsqueda y llegar a una sublista de longitud de `1`. Cada iteración que falla debe continuar disminuyendo la longitud de la sublista por un factor de 2. Si la división de sublistas requiere *m* iteraciones, la sucesión de tamaños de las sublistas hasta una sublista de longitud 1:

$$n \quad n/2 \quad n/2^2 \quad n/2^3 \quad n/2^4 \ldots n/2^m$$

siendo $n/2^m = 1$. Tomando logaritmos en base 2 en la expresión anterior:

$$n = 2^m$$
$$m = \log_2 n$$

Por esa razón, la complejidad del caso peor es $O(\log_2 n)$. Cada iteración requiere una operación de comparación:

$$\text{Total comparaciones} \approx 1 + \log_2 n$$

Comparación de la búsqueda binaria y secuencial

La comparación en tiempo entre los algoritmos de búsqueda secuencial y binaria se va haciendo espectacular a medida que crece el tamaño de la lista de elementos. Tengamos presente que en el caso de la búsqueda secuencial en el peor de los casos coincidirá el número de elementos examinados con el número de elementos de la lista tal como representa su complejidad *O(n)*.

Sin embargo, en el caso de la búsqueda binaria, tengamos presente, por ejemplo, que $2^{10} = 1024$, lo cual implica el examen de 11 posibles elementos; si se aumenta el número de elementos de una lista a 2048 y teniendo presente que $2^{11} = 2048$ implicará que el número máximo de elementos examinados en la búsqueda binaria es 12. Si se sigue este planteamiento, se puede encontrar el número *m* más pequeño para una lista de 1000000, tal que:

$$2^n \geq 1.000.000$$

Es decir, $2^{19} = 524.288$, $220 = 1.048.576$ y, por tanto, el número de elementos examinados (en el peor de los casos) es 21.

Tabla 8.2. Comparación de las búsquedas binaria y secuencial.

Números de elementos examinados		
Tamaño de la lista	**Búsqueda binaria**	**Búsqueda secuencial**
1	1	1
10	4	10
1.000	11	1.000
5.000	14	5.000
100.000	18	100.000
1.000.000	21	1.000.000

Consejo de programación

La búsqueda secuencial se aplica para localizar una clave en un array no ordenado. Para aplicar el algoritmo de búsqueda binaria la lista, o array, debe de estar ordenado.

RESUMEN

Una de las aplicaciones más frecuentes en programación es la ordenación. Los datos se pueden ordenar en orden ascendente o en orden descendente. Existen dos técnicas de ordenación fundamentales en gestión de datos: *ordenación de listas* y *ordenación de archivos*. También se denominan *ordenación interna* y *ordenación externa* respectivamente.

Los algoritmos de ordenación exploran los datos de las listas o arrays para realizar comparaciones y, si es necesario, cambios de posición. Cada recorrido de los datos durante el proceso de ordenación se conoce como *pasada* o *iteración*.

Los algoritmos de ordenación básicos son:

- Selección.
- Inserción.
- Burbuja.

Los algoritmos de ordenación más avanzados son:

- *Shell.*
- *Heapsort (por montículos).*
- *Mergesort.*
- *Radixsort.*
- *Binsort*
- *Quicksort.*

La eficiencia de los algoritmos de burbuja, inserción y selección es $0(n^2)$. La eficiencia de los algoritmos heapsort, radixsort, mergesort y quicksort es $0(n \text{ Log } n)$.

La búsqueda es el proceso de encontrar la posición de un elemento destino dentro de una lista. Existen dos métodos básicos de búsqueda en arrays: **búsqueda secuencial** y **binaria**.

La **búsqueda secuencial** se utiliza normalmente cuando el array no está ordenado. Comienza en el principio del array y busca hasta que se encuentra el dato buscado y se llega al final de la lista.

Si un array está ordenado, se puede utilizar un algoritmo más eficiente denominado **búsqueda binaria**.

La eficiencia de una búsqueda secuencial es $0(n)$. La eficiencia de una búsqueda binaria es $0(\log n)$.

EJERCICIOS

8.1. ¿Cuál es la diferencia entre ordenación por intercambio y ordenación por el método de la burbuja?

8.2. Se desea eliminar todos los números duplicados de una lista o vector (array). Por ejemplo, si el array toma los valores:

```
4    7    11    4    9    5    11    7    3    5
```

ha de cambiarse a

```
4    7    11    9    5    3
```

Escribir una función que elimine los elementos duplicados de un array.

8.3. Escribir una función que elimine los elementos duplicados de un vector ordenado. ¿Cuál es la eficiencia de esta función? Compare la eficiencia con la que tiene la función del Ejercicio 8.2.

8.4. Un vector contiene los elementos mostrados a continuación. Los primeros dos elementos se han ordenado utilizando un algoritmo de inserción. ¿Cuál será el valor de los elementos del vector después de tres pasadas más del algoritmo?

3	13	8	25	45	23	98	58

8.5. Dada la siguiente lista

47	3	21	32	56	92

Después de dos pasadas de un algoritmo de ordenación, el array se ha quedado dispuesto así

3	21	47	32	56	92

¿Qué algoritmo de ordenación se está utilizando (selección, burbuja o inserción)? Justifique la respuesta.

8.6. Un array contiene los elementos indicados más abajo. Utilizando el algoritmo de ordenación *Shell* encuentre las pasadas y los intercambios que se realizan para su ordenación.

8	43	17	6	40	16	18	97	11	7

8.7. Partiendo del mismo array que en el Ejercicio 8.6, encuentre las particiones e intercambios que realiza el algoritmo de ordenación quicksort para su ordenación.

8.8. Un array de registros se quiere ordenar según el campo clave *fecha de nacimiento*. Dicho campo consta de tres subcampos: dia, mes y año de 2, 2 y 4 dígitos respectivamente. Adaptar el método de ordenación radixsort a esta ordenación.

8.9. Un array contiene los elementos indicados más abajo. Utilizando el algoritmo de búsqueda binaria, trazar las etapas necesarias para encontrar el número 88.

8	13	17	26	44	56	88	97

Igual búsqueda pero para el número 20.

8.10. Escribir la función de ordenación correspondiente al método radixsort para poner en orden alfabético una lista de *n* nombres.

8.11. Escribir una función de búsqueda binaria aplicado a un array ordenado descendentemente.

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

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

PROBLEMAS

8.1. Un método de ordenación muy simple, pero no muy eficiente, de elementos x_1, x_2, x_3, ... x_n en orden ascendente es el siguiente:

Paso 1: Localizar el elemento más pequeño de la lista x_1 a x_n; intercambiarlo con x_1.
Paso 2: Localizar el elemento más pequeño de la lista x_2 a x_n, intercambiarlo con x_2.
Paso 3: Localizar el elemento más pequeño de la lista x_3 a x_n, intercambiarlo con x_n.

En el último paso, los dos últimos elementos se comparan e intercambian, si es necesario, y la ordenación se termina. Escribir un programa para ordenar una lista de elementos, siguiendo este método.

8.2. Dado un vector *x* de *n* elementos reales, donde *n* es impar, diseñar una función que calcule y devuelva la mediana de ese vector. La mediana es el valor tal que la mitad de los números son mayores que el valor y la otra mitad son menores. Escribir un programa que compruebe la función.

8.3. Se trata de resolver el siguiente problema escolar. Dadas las notas de los alumnos de un colegio en el primer curso de bachillerato, en las diferentes asignaturas (5, por comodidad), se trata de calcular la media de cada alumno, la media de cada asignatura, la media total de la clase y ordenar los alumnos por orden decreciente de notas medias individuales. *Nota:* utilizar como algoritmo de ordenación el método Shell.

8.4. Escribir un programa de consulta de teléfonos. Leer un conjunto de datos de 1.000 nombres y números de teléfono de un archivo que contiene los números en orden aleatorio. Las consultas han de poder realizarse por nombre y por número de teléfono.

8.5. Realizar un programa que compare el tiempo de cálculo de las búsquedas secuencial y binaria. Una lista A se rellena con 2.000 enteros aleatorios en el rango 0 .. 1.999 y, a continuación, se ordena. Una segunda lista B se rellena con 500 enteros aleatorios en el mismo rango. Los elementos de B se utilizan como claves de los algoritmos de búsqueda.

8.6. Se dispone de dos vectores, *Maestro* y *Esclavo*, del mismo tipo y número de elementos. Se deben imprimir en dos columnas adyacentes. Se ordena el vector *Maestro*, pero siempre que un elemento de *Maestro* se mueva, el elemento correspondiente de *Esclavo* debe moverse también; es decir, cualquier cosa que se haga a *Maestro[i]* debe hacerse a *Esclavo[i]*. Después

de realizar la ordenación se imprimen de nuevo los vectores. Escribir un programa que realice esta tarea. *Nota:* utilizar como algoritmo de ordenación el método Quicksort.

8.7. Cada línea de un archivo de datos contiene información sobre una compañía de informática. La línea contiene el nombre del empleado, las ventas efectuadas por el mismo y el número de años de antigüedad del empleado en la compañía. Escribir un programa que lea la información del archivo de datos y se ordene por ventas de mayor a menor. Visualizar la información ordenada.

8.8. Se desea realizar un programa que realice las siguientes tareas

a) Generar, aleatoriamente, una lista de 999 de números reales en el rango de 0 a 2.000.
b) Ordenar en modo creciente por el método de la burbuja.
c) Ordenar en modo creciente por el método *Shell*.
d) Ordenar en modo creciente por el método *Radixsort*.
e) Buscar si existe el número *x* (leído del teclado) en la lista. Búsqueda debe ser binaria.

Ampliar el programa anterior de modo que pueda obtener y visualizar en el programa principal los siguientes tiempos:

*t*1. Tiempo empleado en ordenar la lista por cada uno de los métodos.
*t*2. Tiempo que se emplearía en ordenar la lista ya ordenada.
*t*3. Tiempo empleado en ordenar la lista ordenada en orden inverso.

8.9. Construir un método que permita ordenar por fechas y de mayor a menor un vector de *n* elementos que contiene datos de contratos ($n <= 50$). Cada elemento del vector debe ser un objeto con los campos día, mes, año y número de contrato. Pueden existir diversos contratos con la misma fecha, pero no números de contrato repetidos. *Nota*: El método a utilizar para ordenar será *Radixsort*.

8.10. Escribir un programa que genere un vector de 10.000 números aleatorios de 1 a 500. Realice la ordenación del vector por dos métodos:

- Binsort
- Radixsort.

Escriba el tiempo empleado en la ordenación de cada método.

8.11. Se leen dos listas de números enteros, A y B de 100 y 60 elementos, respectivamente. Se desea resolver las siguientes tareas:

a) Ordenar aplicando el método de Quicksort cada una de las listas A y B.
b) Crear una lista C por intercalación o mezcla de las listas A y B.
c) Visualizar la lista C ordenada.