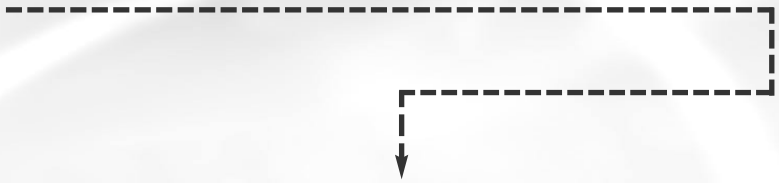
CAPÍTULO 10

ALGORITMOS DE ORDENACIÓN Y BÚSQUEDA



CONTENIDO

INTRODUCCIÓN

Muchas actividades humanas requieren que 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; 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. De igual modo se estudiará el análisis de los diferentes métodos de ordenación con el objeto de conseguir la máxima eficiencia en su uso real.

En el capítulo se analizarán los métodos básicos y los más avanzados empleados en programas profesionales.

CONCEPTOS CLAVE

- Búsqueda en listas
- Complejidad logarítmica
- Ordenación numérica
- Ordenación alfabética
- Complejidad cuadrática
- Ordenación por intercambio
- Ordenación por inserción
- Ordenación por selección
- Ordenación por burbuja
- Ordenación rápida
- Residuos

10.1. 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 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 un *archivo*, un *array* (*vector o tabla*), *un array de registros, una lista enlazada* o un *árbol*. Cuando los datos están almacenados en un *array*, una lista enlazada o un árbol, se denomina *ordenación interna*. Si los datos están almacenados en un archivo, el proceso de ordenación se llama *ordenación externa*.

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

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

y se dice que 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*, donde *k*[*i*] es 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, aunque no sea éste el caso del lenguaje C , 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. Así, en el caso de ordenar los elementos de un vector, el número de comparaciones será *función* del número de elementos (*n*) del vector (*array*). Por consiguiente, se puede expresar el número de comparaciones en términos de *n* (por ejemplo, *n+4*), o bien, n^2 en lugar de números enteros (por ejemplo, 325).

En *todos los métodos de este capítulo*, normalmente –para comodidad del lector– *se utiliza el orden ascendente* sobre vectores o listas (*array*s unidimensionales).

Los métodos de ordenación 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.

A recordar

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 memoria principal o en la memoria externa.

10.2. 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*. La técnica consiste en hacer varias pasadas a través del *array*. En cada pasada, se comparan parejas sucesivas de elementos. Si una pareja está en orden creciente (o los valores son idénticos), se dejan los valores como están. Si una pareja está en orden decreciente, sus valores se intercambian en el *array*.

10.2.1. Algoritmo de la burbuja

En el caso de un *array* (lista) 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 0 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 0 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]`.

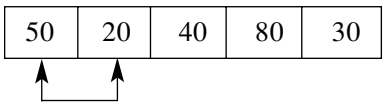
Al final de la pasada, el elemento mayor de la lista está situado en `A[n-1]`.

- En la pasada 1 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]` .

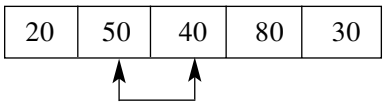
El algoritmo tiene una mejora inmediata, el proceso de ordenación puede terminar en la pasada *n-1*, o bien antes. Si en un una pasada no se produce intercambio alguno entre elementos del *array* es porque ya está ordenado, entonces no es necesario mas pasadas.

El ejemplo siguiente ilustra el funcionamiento del algoritmo de la burbuja con un *array* de 5 elementos (`A = 50, 20, 40, 80, 30`) donde se introduce una variable `interruptor` para detectar si se ha producido intercambio en la pasada.

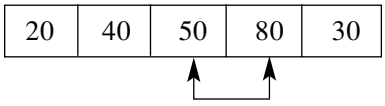
Pasada 0



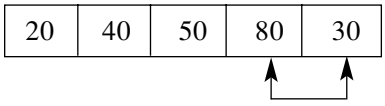
Intercambio 50 y 20



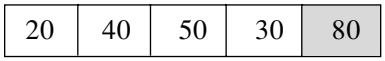
Intercambio 50 y 40



50 y 80 ordenados

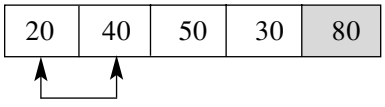


Intercambio 80 y 30

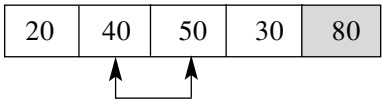


- Elemento mayor es 80
- `Interruptor = TRUE`

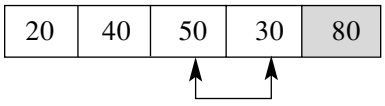
Pasada 1



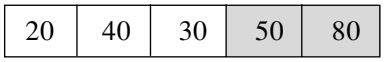
20 y 40 ordenados



40 y 50 ordenados



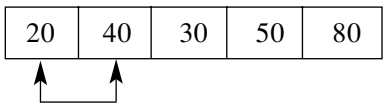
Se intercambian 50 y 30



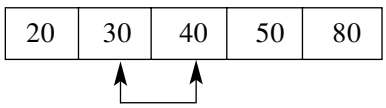
- 50 y 80 elementos mayores ordenados
- `Interruptor = TRUE`

En la pasada 2, sólo se hacen dos comparaciones y se produce un intercambio.

Pasada 2

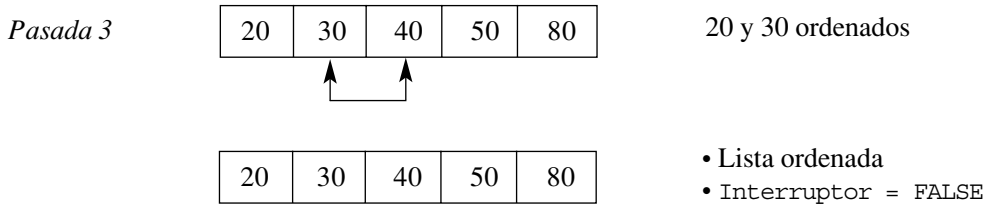


20 y 40 ordenados



- Se intercambian 40 y 30
- `Interruptor = TRUE`

En la pasada 3, se hace una única comparación de 20 y 30, y no se produce intercambio.



En consecuencia, el algoritmo de ordenación de burbuja mejorado contempla dos bucles anidados : el *bucle externo* controla la cantidad de pasadas (al principio de la primera pasada todavía no se ha producido ningún intercambio por tanto la variable `interruptor` se pone a valor falso (0) ; el *bucle interno* controla cada pasada individualmente y cuando se produce un intercambio, cambia el valor de `interruptor` a verdadero (1).

El algoritmo terminará, bien cuando se finalice la última pasada (*n-1*) o bien cuando el valor del `interruptor` sea falso (0), es decir no se haya hecho ningún intercambio. La condición para realizar una nueva pasada se define en la expresión lógica:

```
(pasada < n-1) && interruptor
```

10.2.2. Codificación del algoritmo de la burbuja

La función `ordBurbuja()` implementa el algoritmo de ordenación de la burbuja. Tiene dos argumentos, el *array* que se va a ordenar crecientemente, y el número de elementos `n`. En la codificación se supone que los elementos son de tipo entero largo.

```
void ordBurbuja (long a[], int n)
{
  int interruptor = 1;
  int pasada, j;

  for (pasada = 0; pasada < n-1 && interruptor; pasada++)
  {
              /* bucle externo controla la cantidad de pasadas */
    interruptor = 0;
    for (j = 0; j < n-pasada-1; j++)
      if (a[j] > a[j+1])
      {
              /* elementos desordenados, es necesario intercambio */
        long aux;
        interruptor = 1;
        aux = a[j];
        a[j] = a[j+1];
        a[j+1] = aux;
      }
  }
}
```

Una modificación del algoritmo anterior puede ser utilizar, en lugar de una variable bandera `interruptor`, una variable `indiceIntercambio` que se inicie a 0 (cero) al principio de cada pasada y se esta-

blezca al índice del último intercambio, de modo que cuando al terminar la pasada el valor de `indiceIntercambio` siga siendo 0 implicará que no se ha producido ningún intercambio (o bien que el intercambio ha sido con el primer elemento) y, por consiguiente, la lista estará ordenada. En caso de no ser 0, el valor de `indiceIntercambio` representa el índice del vector a partir del cual los elementos están ordenados. La codificación en C de esta alternativa sería:

```
/*
  Ordenación por burbuja : array  de n elementos
  Se realizan una serie de pasadas mientras indiceIntercambio > 0
*/
void ordBurbuja2 (long a[], int n)
{
  int i, j;
  int indiceIntercambio;

       /* i es el índice del último elemento de la sublista */
  i = n-1;

       /* el proceso continúa hasta que no haya intercambios */
  while (i > 0)
  {
    indiceIntercambio = 0;
                    /* explorar la sublista a[0] a a[i] */
    for (j = 0; j < i; j++)
      /* intercambiar pareja y actualizar IndiceIntercambio */
      if (a[j+1] < a[j])
      {
        long aux = a[j];
        a[j] = a[j+1];
        a[j+1] = aux;
        indiceIntercambio = j;
      }
             /* i se pone al valor del índice del último intercambio */
    i = indiceIntercambio;
  }
}
```

10.2.3. Análisis del algoritmo de la burbuja

¿Cuál es la eficiencia del algoritmo de ordenación de la burbuja ? Dependerá de la versión utilizada. En la versión más simple se hacen *n-1* pasadas y *n-1* comparaciones en cada pasada. Por consiguiente, el número de comparaciones es $(n-1) * (n-1) = n^2 - 2n + 1$, es decir la complejidad es $0(n^2)$.

Si se tienen en cuenta las versiones mejoradas haciendo uso de las variables `interruptor` o `indiceIntercambio`, entonces se tendrá una eficiencia diferente para cada algoritmo. En el mejor de los casos, 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, entonces, que el número medio de pasadas *k* sea *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 *array*s a ordenar son grandes.

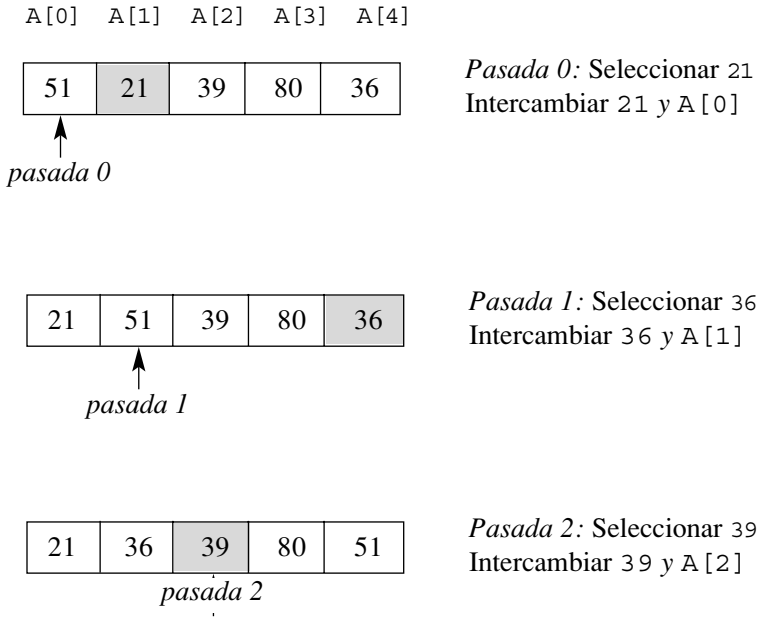
10.3. ORDENACIÓN POR SELECCIÓN

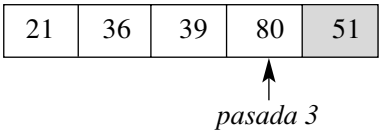
Considérese el algoritmo para ordenar un *array* `A` de enteros en orden ascendente, esto es del número más pequeño al mayor. Si el *array* `A` tiene *n* elementos, se trata de ordenar los valores del *array* de modo que el dato contenido en `A[0]` sea el valor más pequeño, el valor almacenado en `A[1]` el siguiente más pequeño, y así hasta `A[n-1]` que ha de contener el elemento de mayor valor. El algoritmo de selección se apoya en sucesivas pasadas que intercambian el elemento más pequeño sucesivamente con el primer elemento de la lista, `A[0]` en la primera pasada. En síntesis, se busca el elemento más pequeño de la lista y se intercambia con `A[0]`, primer elemento de la lista.

```
A[0] A[1] A[2]....A[n-1]
```

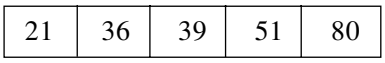
Después de terminar esta primera pasada, el frente de la lista está ordenado y el resto de la lista `A[1], A[2]...A[n-1]` permanece desordenada. La siguiente pasada busca en esta lista desordenada y *selecciona* el elemento más pequeño, que se almacena entonces en la posición `A[1]`. De este modo los elementos `A[0]` y `A[1]` están ordenados y la sublista `A[2], A[3]...A[n-1]` desordenada; entonces, se selecciona el elemento más pequeño y se intercambia con `A[2]`. El proceso continúa *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.

Un ejemplo práctico ayudará a la comprensión del algoritmo. Consideremos un *array* `A` con 5 valores enteros `51, 21, 39, 80, 36`:





Pasada 3: Seleccionar `51`
Intercambiar `51` *y* `A[3]`



Lista ordenada

10.3.1. Algoritmo de selección

Los pasos del algoritmo son :

1. Seleccionar el elemento más pequeño de la lista `A`. Intercambiarlo con el primer elemento `A[0]`. Ahora la entrada más pequeña está en la primera posición del vector.
2. Considerar las posiciones de la lista `A[1], A[2], A[3]...` seleccionar el elemento más pequeño e intercambiarlo con `A[1]`. Ahora las dos primeras entradas de `A` están en orden.
3. Continuar este proceso encontrando o seleccionando el elemento más pequeño de los restantes elementos de la lista, intercambiándolos adecuadamente.

10.3.2. Codificación del algoritmo de selección

La función `ordSeleccion()` ordena una lista o vector de números reales de `n` elementos. En la pasada `i`, el proceso de selección explora 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]` intercambian las posiciones.

```
/*
   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;
              /* situa el elemento mas pequeño en a[i] */
    if (i != indiceMenor)
    {
```

```
            double aux = a[i];
            a[i] = a[indiceMenor];
            a[indiceMenor] = aux ;
        }
    }
}
```

El análisis del algoritmo de selección es sencillo y claro, ya que requiere un número fijo de comparaciones que sólo dependen del tamaño de la lista o *array* y no de la distribución inicial de los datos.

10.4. ORDENACIÓN POR INSERCIÓN

El método de ordenación por inserción 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 o archivo que ya está ordenado. Así el proceso en el caso de la lista de enteros `A = 50, 20, 40, 80, 30.`

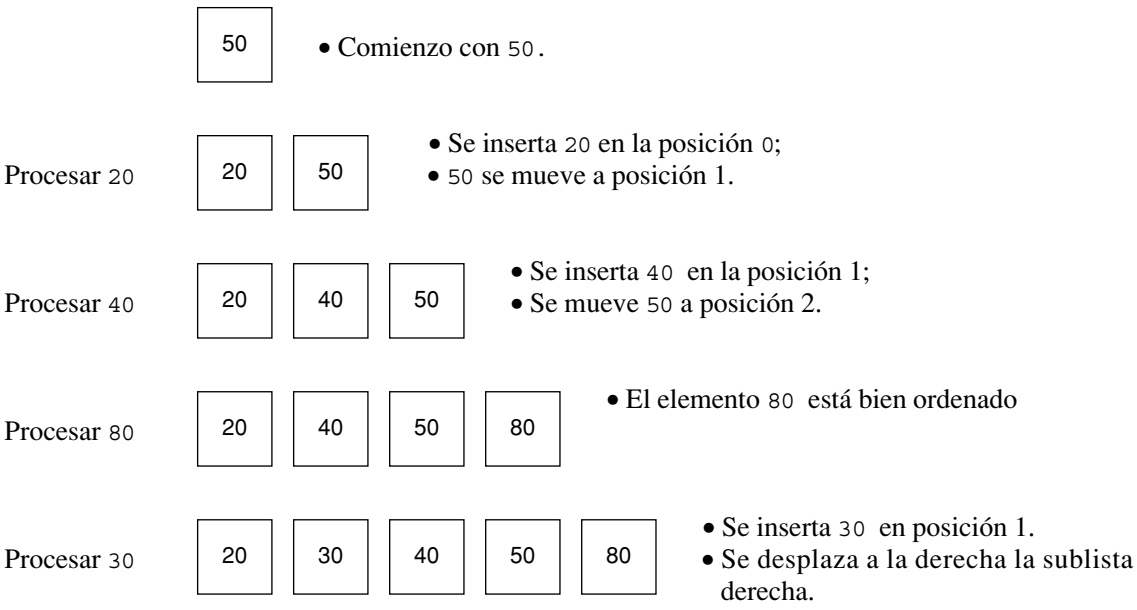


Figura 10.1 Método de ordenación por inserción.

10.4.1. 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 bucle o iteración i (`desde i = 1 hasta n-1`) se explora la sublista `A[i-1] . . A[0]` buscando la posición correcta de inserción; 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 en la posición correcta.

10.4.2. Codificación del algoritmo de inserción

El método `ordInsercion()` tiene dos argumentos, el *array* `a[]` que se va a ordenar crecientemente y el número de elementos `n`. En la codificación se supone que los elementos son de tipo entero.

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

  for (i = 1; i < n; i++)
  {
    /* indice j explora la sublista a[i-1]..a[0] buscando la
      posicion correcta del elemento destino, lo asigna a a[j] */
    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;
  }
}
```

La complejidad del algoritmo es cuadrática, $0(n^2)$, debido a que todo el proceso se controla con dos bucles anidados que en el peor de los casos realizan `n-1` iteraciones .

10.5. 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 el método Shell es una mejora de los métodos de inserción directa.

En el álgoritmo de inserción, cada elemento se compara con los elementos contiguos de su izquierda, uno tras otro. Si el elemento a insertar es el mas pequeño hay que realizar muchas comparaciones antes de colocarlo en su lugar definitivo.

El algoritmo de Shell modifica los saltos contiguos resultantes de las comparaciones por saltos de mayor tamaño y con ello se consigue que la ordenación sea más rápida. Generalmente se toma como salto inicial $n/2$ (siendo n el número de elementos), luego se reduce el salto a la mitad en cada repetición hasta que el salto es de tamaño 1. El ejemplo 10.1. ordena una lista de elementos siguiendo paso a paso el método de Shell.

Ejemplo 10.1.

Obtener las secuencias parciales del vector al 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 7, por lo que el salto inicial es *7/2 = 3*. La siguiente tabla muestra el número de recorridos realizados en la lista con los saltos correspondientes.

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

10.5.1. 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 incremento o 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 incremento o salto decreciente en la mitad que el salto anterior y después clasificando cada grupo por separado.
5. El algoritmo termina cuando se alcanza el tamaño de salto 1.

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

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

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

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

Se puede observar que se comparan pares de elementos indexados por `j` y `k`, `(a[j],a[k])`, separados por un *salto,* `intervalo` . Así, si `n = 8` el primer valor de `intervalo = 4`, y los índices `i = 5`, `j = 1`, `k = 6` . Los siguiente valores que toman `i = 6`, `j = 2`, `k = 7` y así hasta recorrer la lista.

Para realizar un nuevo recorrido de la lista con la mitad de grupos, el intervalo se hace la mitad.

intervalo ← intervalo / 2

Y así sucesivamente se repiten los recorridos por la lista, con el bucle: *mientras intervalo > 0.*

10.5.2. Codificación del método Shell

Al codificar en C este método de ordenación es necesario tener en cuenta que el operador / realiza una división entera si los operandos son enteros y esto es importante al calcular el ancho del saltos entre pares de elementos: `intervalo = n/2.`

En cuanto a los índices, C toma como base el índice 0, como consecuencia hay que desplazar una posición a la izquierda las variables índice respecto a lo expuesto en el algoritmo.

```
void ordenacionShell(double a[], int n)
{
  int intervalo, i, j, k;

  intervalo = n / 2;
  while (intervalo > 0)
  {
    for (i = intervalo; i < n; i++)
    {
     j = i - intervalo;
     while (j >= 0)
     {
       k = j + intervalo;
       if (a[j] <= a[k])
         j = -1;           /* así termina el bucle, par ordenado */
       else
       {
         double temp;
         temp = a[j];
         a[j] = a[k];
         a[k] = temp;
         j -= intervalo;
       }
      }
     }
     intervalo = intervalo / 2;
   }
}
```

10.6. ORDENACIÓN RAPIDA (*QuickSort*)

El algoritmo conocido como *quicksort* (ordenación rápida) fue creado por Tony Hoare. La idea del algoritmo es simple, se basa en la división en particiones de la lista a ordenar, por lo que se puede considerar

que aplica la técnica *"divide y vence"*. El método es, posiblemente, el más pequeño en código, más rápido, más elegante y más interesante y eficiente de los algoritmos conocidos de ordenación.

El método se basa en dividir 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* o elemento de partición, y una partición *derecha*. La partición o división 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 y se utiliza como *pivote* o *elemento de partición*. Si se elige una lista cualquiera con los elementos en orden aleatorio, se puede elegir cualquier elemento de la lista 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 divida la lista exactamente 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 en la que todos los elementos (claves) son mayores o iguales que el pivote (o al revés). 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 lista, 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 constan de sólo un elemento.

A recordar

El algoritmo de ordenación *quicksort* sigue la estrategia típica de los algoritmos recursivos *"divide y vence"*. El problema de tamaño n (ordenar una lista de n elementos) se divide en dos subproblema o tareas, cada una de las cuales se vuelven a dividir en dos tareas, cada vez de menor tamaño. Como cada tarea realiza las mismas acciones, el algoritmo se expresa recursivamente. El *caso base* (condición de parada) es conseguir una tarea (sublista) de tamaño 1.

Ejemplo 10.2.

Se ordena una lista de números enteros aplicando el algoritmo quicksort, como pivote se elige el primer elemento de la lista.

1. *Lista original* | 5 2 1 8 3 9 7 |

pivote *elegido* | 5 |

↑

sublista izquierda1 (elementos menores que 5) | 2 1 3 |

sublista derecha1 (elementos mayores o iguales a 5) | 8 9 7 |

2. El algoritmo se aplica a la sublista izquierda:

Sublista Izda 1 `2 1 3`

↑

pivote

sublista Izda 1
sublista Dcha 3

Izda	pivote	Dcha
1	2	3

3. El algoritmo se aplica a la sublista derecha:

Sublista dcha 1 `8 9 7`

↑

pivote

sublista Izda 7
sublista Dcha 9

Izda	pivote	Dcha
7	8	9

4. *Lista ordenada final:*

Sublista izquierda	*pivote*	*Sublista derecha*
1 2 3	5	7 8 9

A recordar

El algoritmo *quicksort* requiere una estrategia de partición y la selección idónea del pivote. Las etapas fundamentales del algoritmo dependen del pivote elegido aunque la estrategia de partición suele ser similar

10.6.1. Algoritmo *quicksort*

La primera etapa en el algoritmo de partición es obtener el elemento pivote; una vez que se ha seleccionado se ha de buscar la forma de situar en la sublista izquierda todos los elementos menores que el pivote y en la sublista derecha todos los elementos mayores que el pivote. Supongamos que todos los elementos de la lista son distintos, aunque será preciso tener en cuenta los casos en que existan elementos idénticos. En el ejemplo 10.3. se elige como pivote el elemento central de la lista actual.

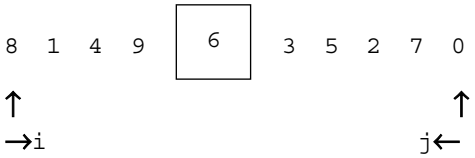
Ejemplo 10.3.

Se ordena una lista de números enteros aplicando el algoritmo quicksort. Como pivote se elige el elemento central de la lista.

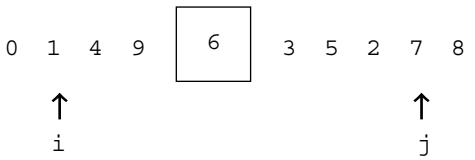
Lista original:	8 1 4 9 6 3 5 2 7 0
pivote (elemento central)	6

La etapa 2 requiere mover todos los elementos menores que el 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 la lista de derecha a izquierda buscando un elemento menor. Para hacer esto se utilizará un índice `j` inicializado a la posición más alta (`superior`).

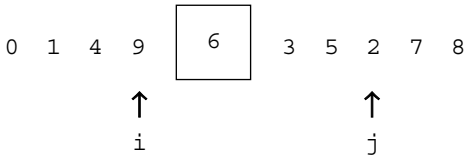
El índice `i` se detiene en el elemento 8 (mayor que el pivote) y el índice `j` se detiene en el elemento 0 (menor que el pivote)



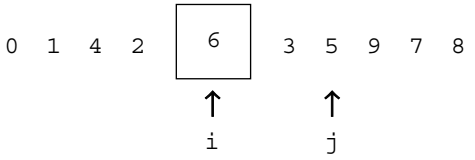
Ahora se intercambian 8 y 0 para que estos dos elementos se sitúen correctamente en cada sublista; y se incrementa el índice `i`, y se decrementa `j` para seguir los intercambios.



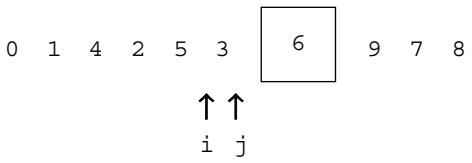
A medida que el algoritmo continúa, `i` se detiene en el elemento mayor, 9, y `j` se detiene en el elemento menor, 2.



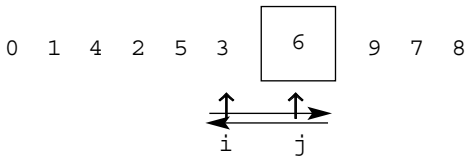
Se intercambian los elementos mientras que `i` y `j` no se *crucen*. En el momento en que se cruzan los índices se detiene este bucle. En el caso anterior se intercambian 9 y 2.



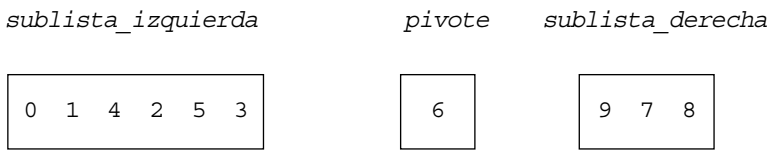
Continúa la exploración y ahora el contador `i` se detiene en el elemento 6 (que es el pivote) y el índice `j` se detiene en el elemento menor 5. Se intercambian 6 y 5, se incrementa `i` y se decrementa `j`.



Los índices tienen actualmente los valores `i = 5`, `j = 5`. Continua 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*. En este caso 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:



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. La Figura 10.2 muestra las operaciones del algoritmo para ordenar la lista `a[]` de `n` elementos enteros.

Pasos que sigue el algoritmo *quicksort*:

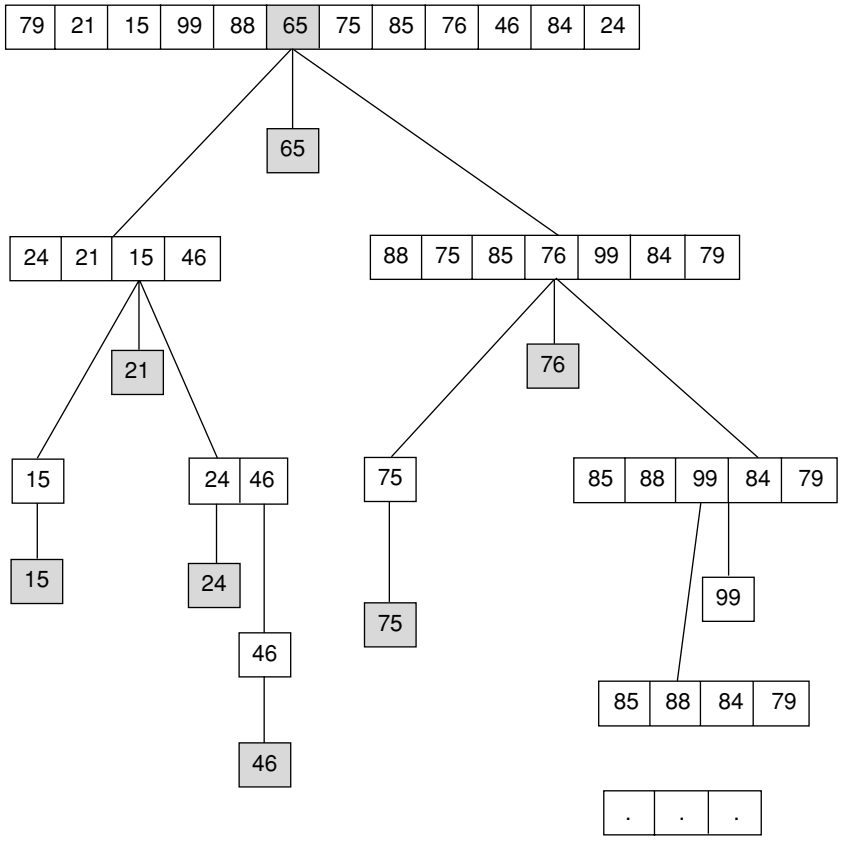
Seleccionar el elemento central de a[0:n-1] como pivote

Dividir los elementos restantes en particiones *izquierda* y *derecha*, de modo que ningún elemento de la izquierda tenga una clave (valor) 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 a continuación partición *derecha*.



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

Figura 10.2 Ordenación rápida eligiendo como pivote el elemento central.

10.6.2. Codificación del algoritmo *quicksort*

La función `quicksort()` refleja el algoritmo citado anteriormente; a esta función se la llama pasando como argumento el *array* `a[]` y los índices que la delimitan `0` y `n-1` (índice inferior y superior). La llamada a la función:

```
quicksort(a, 0, n-1);
```

Y la codificación recursiva de la función:

```
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)
      {
         double tmp;
         tmp = a[i];
         a[i] = a[j];
         a[j] = tmp;         /* intercambia a[i] con 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 */
}
```

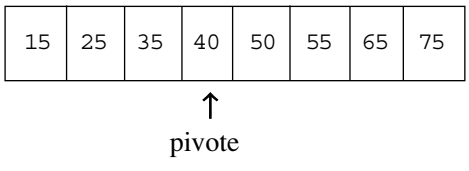
10.6.3. Análisis del algoritmo *quicksort*

El análisis general de la eficiencia de *quicksort* es difícil. La mejor forma de ilustrar y calcular la complejidad del algoritmo es considerar el número de comparaciones realizadas teniendo en cuenta circunstancias ideales. Supongamos que n (número de elementos de la lista) 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 la primera exploración o recorrido se hacen $n-1$ comparaciones. El resultado de la etapa crea dos sublistas aproximadamente de tamaño $n/2$. En la siguiente fase, el proceso de cada sublista requiere aproximadamente $n/2$ comparaciones. Las comparaciones totales de esta fase son $2(n/2) = n$. La siguiente fase procesa cuatro sublistas que requieren un total de $4(n/4)$ comparaciones, etc. Eventualmente, el proceso de división termina después de k pasadas cuando la sublista resultante tenga tamaño 1. El número total de comparaciones es aproximadamente :

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

Para una lista normal la complejidad de *quicksort* es $0(n log_2 n)$.El caso ideal que se ha examinado se realiza realmente cuando la lista (el *array*) está ordenado en orden ascendente. En este caso el pivote es precisamente el centro de cada sublista.



Si el *array* está en orden ascendente, el primer recorrido encuentra el pivote en el centro de la lista e intercambia cada elemento en las sublistas inferiores y superiores. La lista resultante está casi ordenada y el algoritmo tiene la complejidad $0(nlog_2n)$.

El escenario del caso peor de *quicksort* ocurre cuando el pivote cae 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:

$$n + n\text{-}1 + n\text{-}2 + ... + 2 = (n\text{-}1)(n+2)/2$$

Entonces la complejidad es $0(n^2)$. En general el algoritmo de ordenación tiene como complejidad media $0(nlog_2n)$ siendo posiblemente el algoritmo más rápido. La Tabla 10.1 muestra las complejidades de los algoritmos empleados en los métodos explicados en el libro.

Tabla 10.1. Comparación complejidad métodos de ordenación

Método	Complejidad
Burbuja	n^2
Inserción	n^2
Selección	n^2
Montículo	$nlog_2n$
Fusión	$nlog_2n$
Shell	$nlog_2n$
Quicksort	$nlog_2n$

Consejo de programación

Se recomienda utilizar los algoritmos de inserción o selección para ordenar listas pequeñas. Para listas grandes utilizar el algoritmo *quicksort*.

10.7. BÚSQUEDA EN LISTAS: BÚSQUEDA SECUENCIAL Y BINARIA

Con mucha frecuencia los programadores trabajan con grandes cantidades de datos almacenados en *array*s y registros, y por ello será necesario determinar si un *array* contiene un valor que coincida con un cierto *valor clave*. El proceso de encontrar un elemento específico de un *array* se denomina *búsqueda*. Las técnicas de búsqueda más utilizadas son : *búsqueda lineal* o *secuencial*, la técnica más sencilla y *búsqueda binaria* o *dicotómica*, la técnica más eficiente. El algoritmo de búsqueda secuencial se puede consultar en el apartado 9.6; en esta sección se desarrolla la técnica de búsqueda binaria de forma iterativa. Debido a que la búsqueda binaria es un ejemplo claro de algoritmo *divide y vencerás*, su implementación recursiva aparece en el capítulo 8, apartado 5.

10.7.1. Búsqueda binaria

La búsqueda secuencial se aplica a cualquier lista. Si la lista está ordenada, la *búsqueda binaria* proporciona una técnica de búsqueda mejorada. Una búsqueda binaria típica es la búsqueda de una palabra en un diccionario. Dada la palabra, se abre el libro cerca del principio, del centro o del final dependiendo de la primera letra del primer apellido o 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 la lectura 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 uno 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 10.4.

Se desea buscar a ver si el elemento 225 se encuentra 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 `a[5]` (`275`). El valor buscado, `225`, es menor que `275` y la búsqueda continua en la mitad inferior del conjunto de datos de la lista actual; es decir en la sublista de un único elemento:

```
a[4]
120
```

El elemento mitad de esta sublista es el propio elemento `a[4]` (`120`). Al ser `225` mayor que `120` la búsqueda debe continuar en una sublista vacía. En este ejemplo se termina indicando que no se ha encontrado la clave en la lista.

10.7.2. Algoritmo y codificación de la búsqueda binaria

Suponiendo que la lista está almacenada como un *array*, donde los índices de la lista son `bajo = 0` y `alto = n-1` donde *n* es el número de elementos del *array*, 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

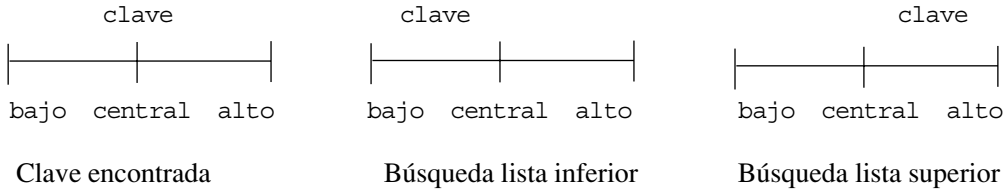


Figura 10.5 Búsqueda binaria de un elemento.

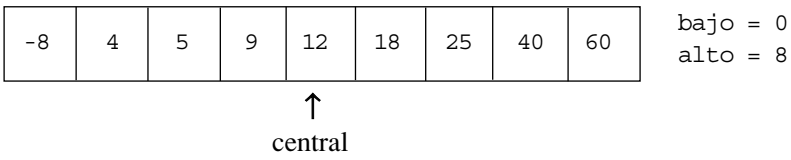
- Si `a[central] < clave`, la nueva sublista de búsqueda está en el rango `bajo = central+1 ..alto`
- Si `clave < a[central]`, la nueva sublista de búsqueda está en el rango `bajo..central-1`

El algoritmo termina o bien porque se ha encontrado la clave o porque el valor de `bajo` excede a `alto` y el algoritmo devuelve el indicador de fallo, -1 (búsqueda no encontrada).

Ejemplo 10.5.

Sea el array de enteros `A (-8, 4, 5, 9, 12, 18, 25, 40, 60)`, *buscar la clave*, *clave* = 40.

1. `a[0] a[1] a[2] a[3] a[4] a[5] a[6] a[7] a[8]`

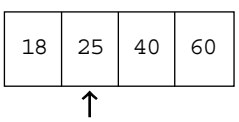


`bajo = 0`
`alto = 8`

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

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

2. Buscar en sublista derecha

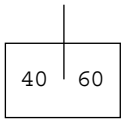


`bajo = 5`
`alto = 8`

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

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

3. Buscar en sublista derecha.




```
bajo = 7
alto = 8
```

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

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

El algoritmo ha requerido 3 comparaciones frente a 8 comparaciones `(n-1, 9-1 = 8)` que se hubieran realizado con la búsqueda secuencial.

Codificación del algoritmo de búsqueda binaria:

```
/*
  búsqueda binaria.
  devuelve el índice del elemento buscado, o bien -1 en caso de fallo
*/
int busquedaBin(int lista[], 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 = lista[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 */
}
```

10.7.3. 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 trate de localizar una información en una lista o tabla en memoria, o bien en un archivo de datos.

Complejidad de la búsqueda secuencial

La complejidad de la búsqueda secuencial (consultar apartado 9.6) distingue entre el comportamiento en el caso peor y mejor. El *mejor caso* se da 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. Esto requiere buscar en todos los *n* elementos, lo que implica una complejidad lineal, dependiente de *n*, *O(n)*.

El caso medio requiere un poco de razonamiento probabilista. Para el caso de una lista aleatoria es probable que una coincidencia ocurra en cualquier posición. Después de la ejecución de un número grande de búsquedas, la posición media para una coincidencia es el elemento central *n/2*. El elemento central ocurre después de *n/2* comparaciones, que define el coste esperado de la búsqueda. Por esta razón, se dice que la prestación media de la búsqueda secuencial es *O(n)*.

Análisis de la búsqueda binaria

El *caso mejor* de la busqueda binaria, 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 $O(log_2 n)$ 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. El tamaño de las sublistas es:

$n\ n/2\ n/4\ n/8 \ldots\ 1$

La división de sublistas requiere *m* iteraciones, en cada iteración el tamaño de la sublista se reduce a la mitad. La sucesión de tamaños de las sublistas hasta una sublista de longitud 1:

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

siendo $n/2^m = 1$

Tomando logaritmos en base 2 en la expresión anterior quedará:

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

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

$Total\ comparaciones \approx 1 + log_2 n$

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 bús-

queda 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, el número máximo de comparaciones es $1 + log_2 n$. Si por ejemplo, la lista tiene $2^{10} = 1024$ elementos, el total de comparaciones será $1 + log_2 2^{10}$; es decir, 11. Para una lista de 2048 elementos y teniendo presente que $2^{11} = 2048$ implicará que el número máximo de elementos examinados (comparaciones) 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:

$2^m \geq 1.000.000$

Es decir $2^{19} = 524.288$, $2^{20} = 1.048.576$ y por tanto el número de elementos examinados (en el peor de los casos) es 21.

Tabla 10.2. 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

La Tabla 10.2 muestra la comparación de los métodos de búsqueda secuencial y búsqueda binaria. En la misma tabla se puede apreciar una comparación del número de elementos que se deben examinar utilizando búsqueda secuencial y binaria. Esta tabla muestra la eficiencia de la búsqueda binaria comparada con la búsqueda secuencial y cuyos resultados de tiempo vienen dados por las funciones de complejidad $O(log_2 n)$ y *O(n)* de las búsquedas binaria y secuencial respectivamente.

A recordar

La búsqueda secuencial se aplica para localizar una clave en un vector no ordenado. Para aplicar el algoritmo de búsqueda binaria la lista, o vector, donde se busca debe de estar ordenado. La complejidad de la búsqueda binaria es logarítmica, *O(Log n)*, mas eficiente que la búsqueda secuencial que tiene complejidad lineal, *O(n)*.

10.8. RESUMEN

- Una de las aplicaciones más frecuentes en programación es la ordenación.
- Existen dos técnicas de ordenación fundamentales en gestión de datos: *ordenación de listas* y *ordenación de archivos*.
- Los datos se pueden ordenar en orden ascendente o en orden descendente.
- 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 mas avanzados son:
  - *Shell*.
  - *Quicksort*.
  - *Heapsort* (por montículos).
  - *Mergesort*.
- La eficiencia de los algoritmos de burbuja, inserción y selección es $0(n^2)$.
- La eficiencia del algoritmo *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 *array*s: **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 o 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 $0(\log n)$.

10.9. EJERCICIOS

10.1. ¿Cuál es la diferencia entre ordenación por selección y ordenación por inserción?

10.2. 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*.

10.3. 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 10.2

10.4. 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

10.5. Dada la siguiente lista:

47	3	21	32	56	92

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

3	21	47	32	56	92

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

10.6. 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

10.7. Partiendo del mismo *array* que en el ejercicio 10.6, encuentre las particiones e intercambios que realiza el algoritmo de ordenación *quicksort* para su ordenación.

10.8. 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 la burbuja a esta ordenación.

10.9. Un *array* contiene los elementos indicados a continuación. 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.

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

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

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

1. Si se 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. Realizar los mismos cálculos si se utiliza el algoritmo *quicksort*.

10.10. PROBLEMAS

10.1. 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.

10.2. 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.

10.3. 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*.

10.4. Escribir un programa de consulta de teléfonos. Leer un conjunto de datos de 1000 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.

10.5. Realizar un programa que compare el tiempo de cálculo de las búsquedas secuencial y binaria. Una lista A se rellena con 2000 enteros aleatorios en el rango 0 .. 1999 y a continuación se ordena. Una segunda lista B se rellena con 500 enteros aleato-

rios en el mismo rango. Los elementos de B se utilizan como claves de los algoritmos de búsqueda.

10.6. 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 acción hecha con *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*.

10.7. 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 a continuación se visualiza. La información debe ser ordenada por ventas de mayor a menor y visualizada de nuevo.

10.8. Se desea realizar un programa que realice las siguientes tareas:

a) Generar, aleatoriamente, una lista de 999 números reales en el rango de 0 a 2000.0
b) Ordenar en modo creciente por el método de la burbuja.
c) Ordenar en modo creciente por el método Shell.
e) Buscar si existe el número x (leído del teclado) en la lista. Aplicar la búsqueda binaria.

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

t1. Tiempo empleado en ordenar la lista por cada uno de los métodos.
t2. Tiempo que se emplearía en *ordenar* la lista ya ordenada.
t3. Tiempo empleado en ordenar la lista ordenada en orden inverso.

10.9. 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 una estructura 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 par ordenar será el de *quicksort*.

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

a) Ordenar, aplicando el método de inserción, 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.