

# Programación en Java 2

Algoritmos, Estructuras de Datos  
y Programación Orientada a Objetos



Luis JOYANES AGUILAR

Ignacio ZAHONERO MARTÍNEZ

**Mc  
Graw  
Hill**



### 13.1. 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]<sup>1</sup> y sobre todo la segunda edición publicada en el año 1998 [Knuth 1998]<sup>2</sup>. 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.

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

Los datos se pueden almacenar en memoria central o en archivos de datos externos guardados en unidades de almacenamiento magnético (discos, cintas, disquetes, CD-ROM, DVD, etc.) Cuando los datos se guardan en listas y en pequeñas cantidades, se suelen almacenar de modo temporal en arrays y registros; 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. Los datos, sin embargo, se almacenan de modo permanente en archivos y bases de datos que se guardan en discos y cintas magnéticas.

Así pues, 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.

Las técnicas que se analizarán a continuación considerarán, esencialmente, la ordenación de elementos de una lista (*array*) en orden ascendente. En cada caso se desarrollará la eficiencia computacional del algoritmo.

Con el objeto de facilitar el aprendizaje del lector y aunque no sea un método utilizado por su poca eficiencia se describirá en primer lugar el método de ordenación por intercambio con un programa completo que manipula al correspondiente método `ordIntercambio`, por 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 pueda asimilar más eficazmente los tres algoritmos básicos ya citados y los avanzados que se estudiarán más adelante.

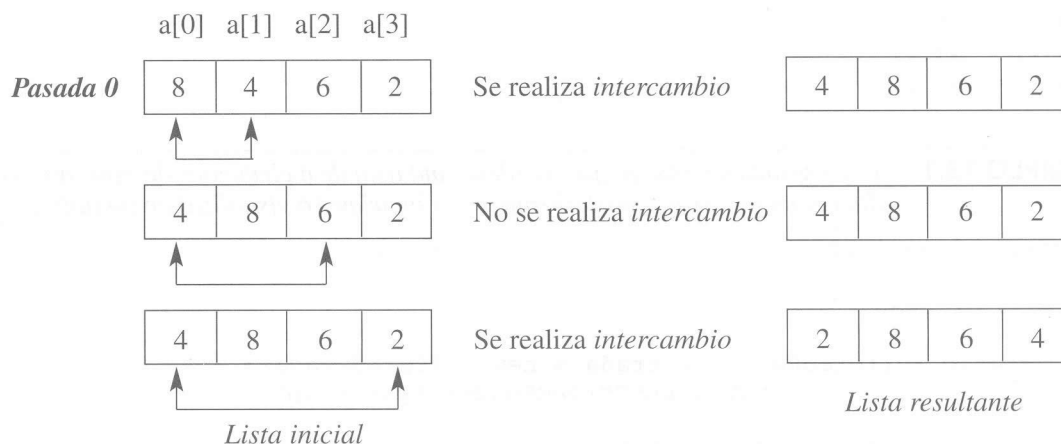
### 13.2. ORDENACIÓN POR INTERCAMBIO

El algoritmo de ordenación tal vez más sencillo sea el denominado de *intercambio*, que ordena los elementos de una lista en orden ascendente. 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.

El algoritmo se ilustra con la lista original 8, 4, 6, 2 que ha de convertirse en la lista ordenada 2, 4, 6, 8. El algoritmo realiza  $n-1$  pasadas (tres en el ejemplo), siendo  $n$  el número de elementos, realizando las siguientes operaciones:

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

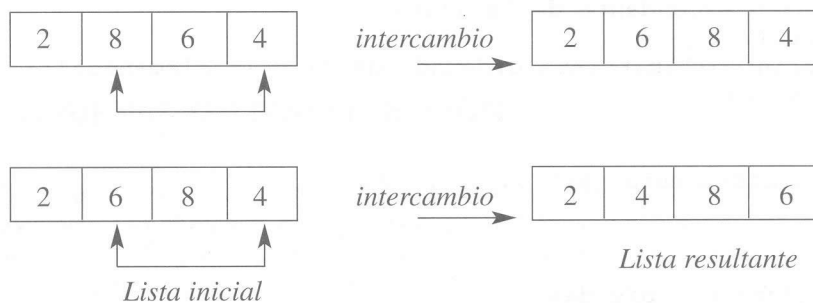
<sup>2</sup> [Knuth 1998] Donald E. Knuth. *The Art of Computer Programming*. Volume 3: *Sorting and Searching*. Second Edition. Addison-Wesley, 1998.



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

### Pasada 1

El elemento más pequeño ya está localizado en el índice 0, y 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 2

La sublista a considerar ahora es 8, 6, ya que 2, 4 está ordenada. Una comparación única se produce entre los dos elementos de la sublista



El método `ordIntercambio` utiliza dos bucles anidados. Suponiendo que la lista es de tamaño  $n$ , el rango del bucle externo irá desde el índice 0 hasta  $n-2$ . Por cada índice  $i$ , se comparan los elementos posteriores de índices  $j = i+1, i+2, \dots, n-1$ . El intercambio (*swap*) de los elementos  $a[i], a[j]$  se realiza en un bloque que utiliza el algoritmo siguiente:

```

aux = a[i];
a[i] = a[j];
a[j] = aux ;

```

**EJEMPLO 13.1.** *El programa OrdSwap.java ordena una lista de n elementos de tipo entero introducida en un array y posteriormente la imprime (o visualiza) en pantalla.*

```

import java.io.*;

class OrdSwap
{
    static BufferedReader entrada = new BufferedReader(
        new InputStreamReader(System.in));

    public static void main(String[] ar) throws IOException
    {
        int n;
        int []v;

        System.out.print("Introduzca el número de elementos: ");
        System.out.flush();
        n = Integer.parseInt(entrada.readLine());
        v = new int[n];
        entradaLista(v,n);
        // muestra lista original
        System.out.println("\n\tLista original de "+ n +" elementos");
        imprimirLista(v,n);
        // ordenación ascendente de la lista
        ordIntercambio(v,n);
        System.out.println("\n\tLista ordenada de "+ n +" elementos");
        imprimirLista(v,n);
    }

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

        // se realizan n-1 pasadas
        // a[0], ... , a[n-2]
        for (i = 0 ; i <= n-2 ; i++)
            // se coloca el mínimo de a[i+1]...a[n-1] en a[i]
            for (j = i+1 ; j <= n-1 ; j++)
                if (a[i] > a[j])
                {
                    int aux;
                    aux = a[i];
                    a[i] = a[j];
                    a[j] = aux ;
                }
    }

    static void imprimirLista (int a[], int n)
    {
        for (int i = 0 ; i < n ; i++)
        {

```



```

        String c;
        c = (i%10==0)?"\n":"" ";
        System.out.print(c + a[i]);
    }
}

static void entradaLista (int a[], int n) throws IOException
{
    System.out.println("\n Entrada de los elementos");
    for (int i = 0 ; i < n ; i++)
    {
        System.out.print("a[" + i + "] = ");
        a[i] = Integer.parseInt(entrada.readLine());
    }
}
}

```

La ejecución del programa OrSwap con los datos de entrada:

Lista original de 20 elementos  
 30 35 38 58 14 15 50 27 10 20  
 12 85 49 65 85 60 25 90 5 16

Lista ordenada  
 5 10 12 14 15 16 20 25 27 30  
 35 38 49 50 58 60 65 85 86 90

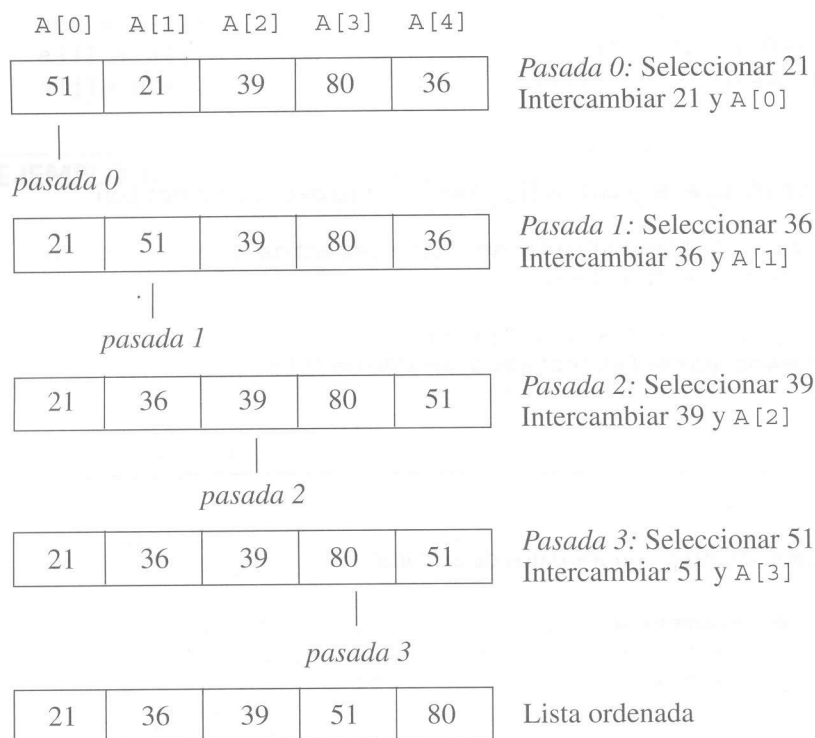
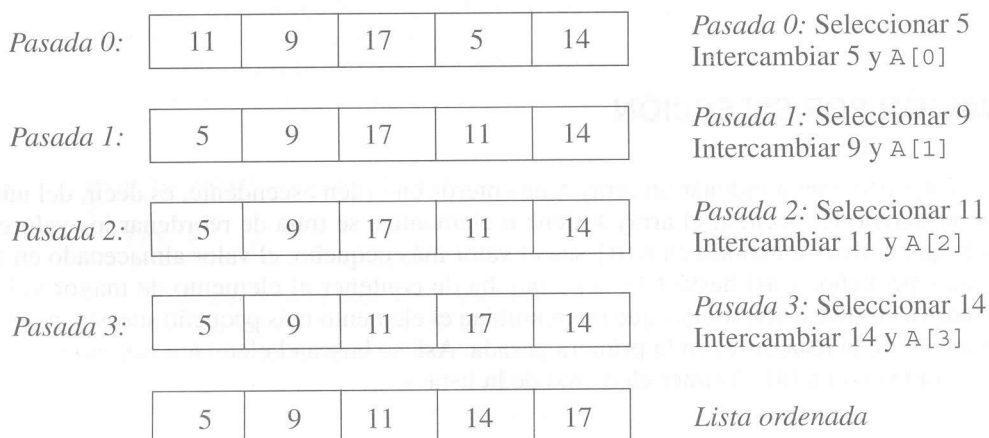
### 13.3. ORDENACIÓN POR SELECCIÓN

Considérese el algoritmo para ordenar un array  $A$  de enteros en orden ascendente, es decir, del número más pequeño al mayor. Es decir, si el array  $A$  tiene  $n$  elementos, se trata de reordenar 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 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. Así, 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 la última 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 cinco valores enteros 51, 21, 39, 80, 36:

**EJEMPLO 13.2.** Ordenar por selección la lista de enteros 11, 9, 17, 5, 14.**13.3.1. 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.

El método `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  $A[\text{indiceMenor}]$  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;
        // Cuando se termina, situar el elemento mas pequeño en A[i]
        if (i != indiceMenor)
        {
            double aux = a[i];
            a[i] = a[indiceMenor];
            a[indiceMenor] = aux;
        }
    }
}
```

### 13.3.2. Análisis del algoritmo de ordenación por selección

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 vector (array) y no de la distribución inicial de los datos. En la primera pasada se hacen  $n-1$  comparaciones, en la segunda pasada  $n-2$  y así sucesivamente. Matemáticamente se puede decir que en la pasada  $i$ , el número de comparaciones con la sublista  $A[i+1]$  a  $A[n-1]$  es

$$(n-1) - (i+1) + 1 = n - i - 1$$

de modo que el número total de comparaciones es

$$\begin{aligned} \sum_{i=0}^{n-2} (n-1-i) &= (n-1)^2 - \sum_{i=0}^{n-2} i \\ &= (n-1)^2 - (n-1)(n-2)/2 \\ &= \frac{1}{2} n (n-1) \end{aligned}$$

o de otra manera

$$(n-1) + (n-2) + (n-3) + \dots + 1 = \frac{n * (n-1)}{2}$$

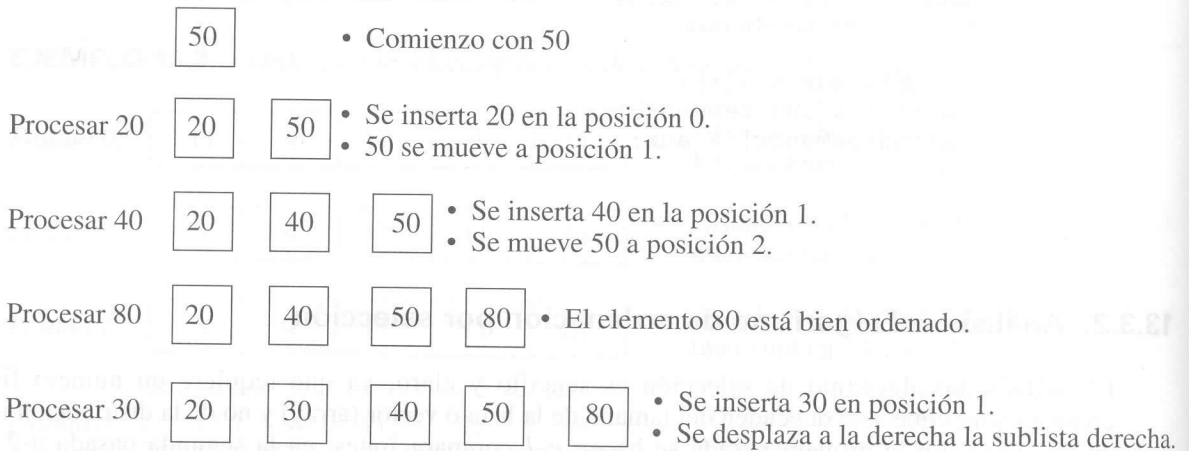
Es decir, para ordenar un vector de  $n$  elementos, el número de comparaciones se calcula sumando los  $n-1$  primeros enteros

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

La complejidad del algoritmo se mide por el número de comparaciones y es *cuadrática*, es decir,  $O(n^2)$ ; el número de intercambios es  $O(n)$ . No hay caso mejor ni peor, dado que el algoritmo realiza un número fijo de pasadas y explora o rastrea un número especificado de elementos en cada pasada.

### 13.4. 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 13.1.** Método 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  $A[i-1] \dots A[0]$  buscando la posición correcta de inserción; a la vez se mueven 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 la posición correcta.



### 13.4.1. Codificación en Java del algoritmo de ordenación por 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
            // para inserción
            a[j] = a[j-1];
            j--;
        }
        a[j] = aux;
    }
}
```

### 13.4.2. Análisis del algoritmo de ordenación por inserción

La ordenación por inserción requiere un número fijo de pasadas. La pasada  $n-1$  inserta en la sublista  $A[n-1]$  a  $A[0]$ . Para una pasada general  $i$ , la inserción ocurre en la sublista  $A[i-1]$  a  $A[0]$  y requiere la media de  $i/2$  comparaciones. El número total de comparaciones es

$$\frac{1}{2} + \frac{2}{2} + \frac{3}{2} + \dots + \frac{(n-2)}{2} + \frac{(n-1)}{2} = \frac{n(n-1)}{4}$$

Al contrario que otros métodos, la ordenación por inserción no utiliza intercambios. La complejidad del algoritmo es  $O(n^2)$ , que mide el número de comparaciones. El mejor caso sucede cuando la lista original está ya ordenada; en la pasada  $i$ , la inserción ocurre en la posición  $i$  y el número total de comparaciones es  $n-1$  con complejidad  $O(n)$ . El caso peor se produce cuando la lista está ordenada en orden descendente (inverso) y el número total de comparaciones es

$$(n-1) + (n-2) + \dots + 3 + 2 + 1 = \frac{(n-1)n}{2}$$

Es decir, la complejidad sigue siendo  $O(n^2)$ . Este método se mejora utilizando una *búsqueda binaria* con el propósito de encontrar la posición correcta para  $A[i]$  en la lista ordenada  $A[0] \dots A[i-1]$ . Esta operación reduce la cantidad total de comparaciones de  $O(n^2)$  a  $O(n \log n)$ . Sin embargo, incluso si la posición correcta de inserción,  $j$ , se encuentra en  $O(n \log n)$  pasos, cada uno de los elementos  $A[j] \dots A[i-1]$  debe moverse una posición y esta operación requiere  $O(n^2)$  sustituciones. Es decir,

desafortunadamente, aunque la técnica de búsqueda binaria mejora algo el algoritmo, no compensa la mejora en tiempo de ejecución la complejidad que se introduce en el algoritmo.

## 13.5. 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.

### 13.5.1. 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

$(A[0], A[1]), (A[1], A[2]), (A[2], A[3]), \dots, (A[n-2], A[n-1])$

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  $A[0]$ .

Un ejemplo ilustrará la técnica. Sea la lista:

$A[] \quad 25 \quad 60 \quad 45 \quad 35 \quad 12 \quad 92 \quad 85 \quad 30$

En la primera pasada se hacen las siguientes comparaciones:

$A[0]$	con	$A[1]$	(25 con 60)	no hay intercambio
$A[1]$	con	$A[2]$	(60 con 45)	intercambio
$A[2]$	con	$A[3]$	(60 con 35)	intercambio
$A[3]$	con	$A[4]$	(60 con 12)	intercambio
$A[4]$	con	$A[5]$	(60 con 92)	no hay intercambio
$A[5]$	con	$A[6]$	(92 con 85)	intercambio
$A[6]$	con	$A[7]$	(92 con 30)	intercambio



Después de la primera pasada la lista está en el orden

25 45 35 12 60 85 30 92

Realizando las mismas comparaciones después de la segunda pasada la lista está en el orden

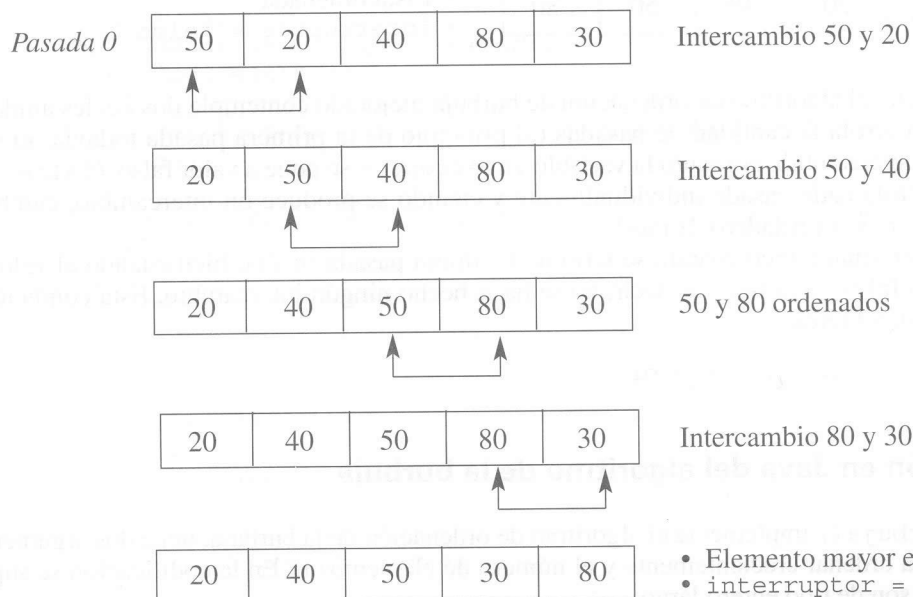
25 35 12 45 60 30 85 92

El conjunto completo de pasadas produciría:

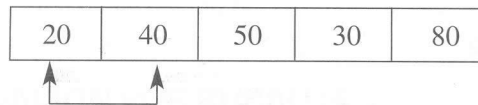
<i>Pasada 0</i> (lista original)	25	60	45	35	12	92	85	30
<i>Pasada 1</i>	25	45	35	12	60	85	30	92
<i>Pasada 2</i>	25	35	12	45	60	30	85	92
<i>Pasada 3</i>	25	12	35	45	30	60	85	92
<i>Pasada 4</i>	12	25	35	30	45	60	85	92
<i>Pasada 5</i>	12	25	30	35	45	60	85	92
<i>Pasada 6</i>	12	25	30	35	45	60	85	92
<i>Pasada 7</i>	12	25	30	35	45	60	85	92

Obsérvese que la lista se ha quedado ordenada después de cinco *pasadas* (o *iteraciones*), por lo que las dos últimas pasadas son innecesarias y han consumido tiempo de ejecución. Es decir, el método de la burbuja admite alguna mejora en forma de eliminación de pasadas innecesarias. Con el objeto de eliminar esas pasadas innecesarias será preciso detectar cuándo la lista está ya ordenada. Esta situación es fácil de detectar, ya que se producirá cuando no se haya producido ningún intercambio; este hecho se puede verificar mediante una *variable bandera* de tipo interruptor (verdadero/falso) que cambia de valor (estado) cuando se produce un intercambio.

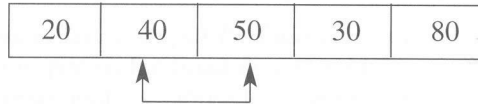
El ejemplo siguiente ilustra el funcionamiento del algoritmo de la burbuja con un array de cinco elementos ( $A = 50, 20, 40, 80, 30$ ), donde se introduce una variable interruptor para detectar si se ha producido intercambio en la pasada.



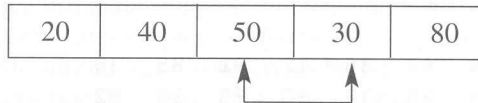
En la pasada 1



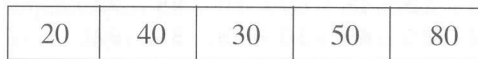
20 y 40 ordenados



40 y 50 ordenados

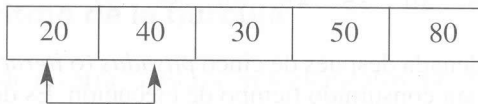


Se intercambian 50 y 30

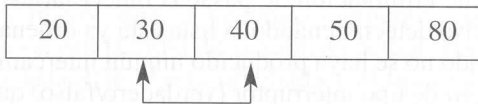


- 50 y 80 elementos mayores y ordenados
- interruptor = true

En la pasada 2 sólo se hacen dos comparaciones

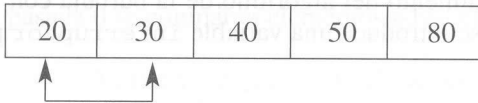


20 y 40 ordenados



- 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



20 y 30 ordenados



- Lista ordenada
- interruptor = false

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 (*false*)); el *bucle interno* controla cada pasada individualmente y cuando se produce un intercambio, cambia el valor de *interruptor* a verdadero (*true*).

El algoritmo terminará, bien cuando se termine la última pasada ( $n-1$ ) o bien cuando el valor del *interruptor* sea falso (*false*), es decir, no se haya hecho ningún intercambio. Esta condición se define en la expresión lógica

$(\text{pasada} < n-1) \ \&\& \ \text{interruptor}$

### 13.5.2. Codificación en Java del algoritmo de la burbuja

El método `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)
{
    boolean interruptor = true;

    for (int pasada = 0; pasada < n-1 && interruptor; pasada++)
    {
        // bucle externo controla la cantidad de pasadas
        interruptor = false; // no se han hecho intercambio todavía
        for (int j = 0; j < n-pasada-1; j++)
            // bucle interno controla cada pasada
            if (a[j] > a[j+1])
            {
                // elementos desordenados,
                // es necesario intercambio
                long aux;
                interruptor = true;
                aux = a[j];
                a[j] = a[j+1];
                a[j+1] = aux;
            }
    }
}

```

Una modificación al algoritmo anterior podría ser utilizar, en lugar de una variable interruptor con valor lógico, una variable indiceIntercambio de tipo contador que se inicie a 0 (cero) al principio de cada pasada y se incremente en 1 cada vez que se produce un intercambio, de modo que, cuando al terminar la pasada el valor de indiceIntercambio siga siendo 0, implicará que no se ha producido ningún intercambio y, por consiguiente, la lista estará ordenada. La codificación en Java de esta alternativa:

```

// 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;
    // Índice de ultimo intercambio
    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
        // continúa ordenación de la sublista a[0] a a[i]
        i = indiceIntercambio
    }
}

```

### 13.5.3. 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  $O(n^2)$ .

Si se tienen en cuenta las versiones mejoradas haciendo uso de las variables interruptor o `indiceIntercambio`, entonces se tendrá una eficiencia diferente a 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  $O(n)$ . 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  $O(n^2)$  comparaciones y  $O(n^2)$  intercambios.

En cualquier forma, el análisis del caso general es complicado, dado que algunas de las pasadas pueden no realizarse. Se podría señalar, entonces, que el número medio de pasadas  $k$  sea  $O(n)$  y el número total de comparaciones es  $O(n^2)$ . 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.

## 13.6. 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 cierto *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 técnica más sencilla, y *búsqueda binaria* o *dicotómica*, la técnica más eficiente.

### 13.6.1. 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 o vector se exploran (se examinan) en secuencia, uno después de otro. La búsqueda secuencial es necesaria, por ejemplo, si se desea encontrar la persona cuyo número de teléfono es 958-220000 en un directorio o listado telefónico de su ciudad. Los directorios de teléfonos están organizados alfabéticamente por el nombre del abonado en lugar de por números de teléfono, de modo que deben explorarse todos los números, uno después de otro, esperando encontrar el número 958-220000. Naturalmente, si se hace esta tarea de modo manual se perderá con facilidad el número, no así mediante la computadora, que realizará fácilmente tareas repetitivas.

Naturalmente, existe otro método de búsqueda cuando se busca el número de teléfono del abonado Sr. Martínez. Dado que el par *nombre de abonado/número de teléfono* está almacenado alfabéticamente por nombre, es posible buscar un nombre de modo eficiente; este método es el conocido como *búsqueda binaria*, que aprovecha la ventaja de que los datos almacenados están ordenados. Este es el sistema

usual de búsqueda de un teléfono en un listín telefónico: búsqueda mediante aproximaciones sucesivas de la página donde aparece el nombre deseado.

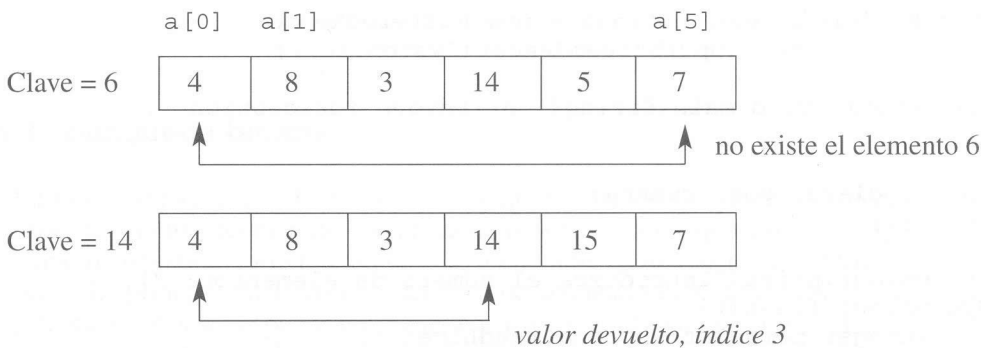
El algoritmo de búsqueda secuencial compara cada elemento del array con la *clave* de búsqueda. Dado que el array no está en un orden prefijado, es probable que el elemento a buscar pueda ser el primer elemento, el último elemento o cualquier otro. De promedio, al menos el programa tendrá que comparar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

### 13.6.2. Algoritmo de búsqueda secuencial

El algoritmo comienza en el primer elemento `lista[0]`, o bien en una posición predeterminada (*inicio*), y recorre los restantes elementos de la lista, comparando cada elemento con la clave. La exploración continúa hasta que se encuentra la clave o se termina la lista. Si la clave se encuentra devuelve el índice del elemento encontrado en la lista; en caso contrario, el valor -1.

#### EJEMPLO 13.3

Buscar si el elemento 6 o el 14 están en la lista A (4 8 3 14 5 7)



La codificación del algoritmo en Java:

```
int busquedaLineal (int []lista, int n, int clave)
{
    for (int i = 0 ; i < n; i++)
        if (lista[i] == clave)
            return i;

    return -1;
}
```

Si por el contrario se deseara comenzar la lista en un elemento distinto del cero, se declara un parámetro *inicio* que corresponde al elemento cuyo índice es *inicio*; es decir, `lista[inicio]`. El algoritmo correspondiente sería:

```
int busquedaLineal(int [] lista, int inicio, int n, int clave)
{
    for (int i = inicio; i < n; i++)
        if (lista[i] == clave)
```

```

        return i ;
    }
    return -1;
}

```

---

**PROBLEMA 13.1.** *El programa comprueba la búsqueda secuencial contando el número de ocurrencias de una clave en una lista. El programa principal introduce n enteros en un array y a continuación solicita una clave.*

El programa realiza repetidas llamadas a `busquedaLineal()` utilizando un índice inicial diferente. Inicialmente se comienza en el índice 0, el principio del array. Después de cada llamada a `busquedaLineal()`, la cuenta del número de ocurrencias se incrementa si se encuentra la clave; en caso contrario la búsqueda termina y se saca la cuenta. Si se encuentra la clave, el valor devuelto identifica su posición en la lista. La siguiente llamada a `busquedaLineal()` se hace con el valor inicial del índice de búsqueda en el elemento inmediatamente a la derecha; este artificio evitará repetir la búsqueda desde el principio y, por consiguiente, hacer más eficiente el algoritmo.

```

import java.io.*;

class BusSecuen
{
    static BufferedReader entrada = new BufferedReader(
        new InputStreamReader(System.in));

    public static void main(String[] ar) throws IOException
    {
        int n, clave, pos, cuenta;
        int []v;

        System.out.print("Introduzca el número de elementos: ");
        System.out.flush();
        n = Integer.parseInt(entrada.readLine());
        v = new int[n];
        entradaLista(v,n);

        System.out.print("Clave a buscar: ");
        System.out.flush();
        clave = Integer.parseInt(entrada.readLine());
        // inicio búsqueda en el primer elemento del array
        pos = cuenta = 0;
        while ((pos = busquedaLineal(v, pos, n, clave)) != -1)
        {
            cuenta ++;
            pos++; // avanzar al siguiente índice
        }

        System.out.println(clave + " se repite " + cuenta +
            (cuenta != 1 ? " veces" : " vez") + " en la lista");
    }
    static
    int busquedaLineal(int []lista, int inicio, int n, int clave)
    {
        for (int i = inicio; i < n; i++)

```



```

        if (lista[i] == clave)
            return i ;
        return -1;
    }
    static void entradaLista(int []v, int n) throws IOException
    {
        System.out.println("\n Entrada de " + n + " enteros.");
        for (int i = 0 ; i < n ; i++)
        {
            System.out.print("v[" + i + "] = ");
            v[i] = Integer.parseInt(entrada.readLine());
        }
    }
}

```

Al ejecutarse el programa, si suponemos que  $n = 10$  y que la lista de enteros:

4 3 7 1 2 6 3 8 3 4

La salida por pantalla:

Introduzca clave a buscar : 3  
3 se repite 2 veces en la lista

### 13.6.3. 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 un número en un directorio telefónico o de un nombre en un diccionario. Dado el nombre, 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 el nombre de la persona 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 el nombre 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 13.4.** *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 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 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. Se concluye con un no encontrado clave en la lista, devolviendo -1.

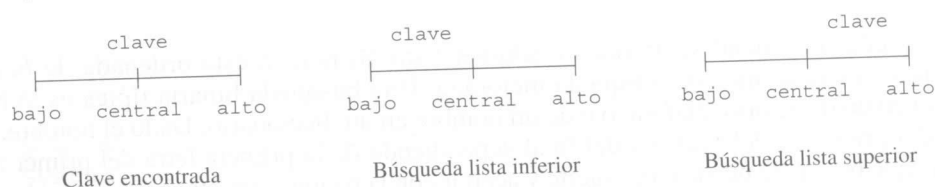
### 13.6.4. Algoritmo de búsqueda binaria

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.

1. Calcular el índice del punto central del array

$$\text{central} = (\text{bajo} + \text{alto}) / 2 \quad (\text{división entera})$$

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



**Figura 13.2.** Búsqueda binaria de un elemento.

- Si  $a[\text{central}] < \text{clave}$ , la nueva sublista de búsqueda tiene por valores extremos de su rango bajo = central+1 ..alto.
- Si  $\text{clave} < a[\text{central}]$ , la nueva sublista de búsqueda tiene por valores extremos de su rango bajo..central-1.



El algoritmo se 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 (búsqueda no encontrada).

**EJEMPLO 13.5.** 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]

-8	4	5	9	12	18	25	40	60
----	---	---	---	----	----	----	----	----

bajo = 0  
alto = 8

↑  
central

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

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

2. Buscar en sublista derecha

18	25	40	60
----	----	----	----

bajo = 5  
alto = 8

↑

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

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

3. Buscar en sublista derecha

40	60
----	----

bajo = 7  
alto = 8

↑

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

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

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

La codificación del algoritmo de búsqueda binaria:

```
// búsqueda binaria
// devuelve el índice del elemento buscado o -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 valor;
                                   // 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
}

```

## 13.7. 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 una información en una lista o tabla en memoria, o bien en un archivo de datos.

### 13.7.1. Complejidad de la búsqueda secuencial

La complejidad de la búsqueda secuencial 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  $O(n)$ .

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  $O(n)$ .

### 13.7.2. 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  $O(\log_2 n)$ , 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 con listas de una 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:

$n \quad n/2 \quad n/4 \quad n/8 \quad \dots \quad 1$

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:

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

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:

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

### 13.7.3. 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,  $2^{19} = 524.288$ ,  $2^{20} = 1.048.576$  y por tanto el número de elementos examinados (en el peor de los casos) es 21.

**Tabla 13.1.** Comparación de las búsquedas binaria y secuencial

Tamaño de la lista	Números de elementos examinados	
	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 13.1 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.

## RESUMEN

- Una de las aplicaciones más frecuentes en programación es la ordenación.
- 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.
- Existen dos técnicas de ordenación fundamentales en gestión de datos: *ordenación de listas* y *ordenación de archivos*.
- La eficiencia de los algoritmos de burbuja, inserción y selección es  $O(n^2)$ .
- 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 lista secuencial es  $O(n)$ .
- La eficiencia de una búsqueda binaria es  $O(\log_2 n)$ .

## EJERCICIOS

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

**13.2.** 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 un método que elimine los elementos duplicados de un array.

**13.3.** 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

**13.4.** 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.

**13.5.** Un array contiene los elementos siguientes:

3 13 7 26 44 23 98 57

¿Cuántas pasadas serán necesarias para ordenar en orden ascendente por el método de burbuja, selección e inserción?

**13.6.** 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.

- 13.7.** Escribir un método que tenga un primer argumento que sea un array de 10 nombres y los ponga en orden alfabético utilizando el método de selección.

- 13.8.** Clasificar el vector

42 57 14 40 96 19 08 68

por los métodos: a) burbuja; b) selección; c) inserción. Cada vez que se reorganiza el vector se debe mostrar el nuevo vector.

- 13.9.** Escribir un método de búsqueda binaria aplicado a un array ordenado descendientemente.

- 13.10.** Supongamos que se tiene una secuencia de  $n$  números que deben ser clasificados:

1. Utilizar el método de selección, ¿cuántas comparaciones y cuántos intercambios se requieren para clasificar la secuencia si:

- Ya está clasificado.
- Está en orden inverso?

2. Repetir el paso 1 para el método de la burbuja.

## PROBLEMAS

- 13.1.** Modificar el algoritmo de ordenación por selección que ordene un vector de enteros en orden descendente.

- 13.2.** Escribir un programa en el que se dé entrada a un vector de empleados y se ordene por el salario del empleado aplicando el algoritmo de ordenación por selección.

- 13.3.** 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.

- 13.4.** 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.

- 13.5.** Realizar un programa que permita la introducción de 12 números enteros en una lista y

mediante un menú seleccione y ejecute uno de los siguientes métodos de ordenación: selección, intercambio, inserción y burbuja. El programa debe presentar en pantalla las listas ordenadas en orden creciente o decreciente a opción del usuario.

- 13.6.** Construir un método que permita ordenar por fechas y de mayor a menor un vector de  $n$  elementos que contiene datos de contratos ( $n \leq 50$ ). Cada elemento del vector debe ser un objeto con los campos día, mes, año y número de contrato.

*Nota.* Pueden existir diversos contratos con la misma fecha, pero no números de contrato repetidos.

- 13.7.** Un programa ha sido diseñado para leer una lista de no más de 1.000 enteros positivos, cada uno menor de 100, y ejecutar algunas operaciones. El cero es la marca final de la lista. El programador debe conseguir:

1. Visualizar los números de la lista en orden creciente.
2. Calcular e imprimir la mediana (valor central).