Capítulo 9

ARRAYS. ALGORITMOS DE ORDENACIÓN Y BÚSQUEDA

1. INTRODUCCIÓN

El propósito fundamental de mantener estructuras de datos ordenadas es, sin duda, facilitar la búsqueda de elementos. Es incuestionable que un elemento en particular, perteneciente a una estructura determinada, es más fácil de encontrar si dicha estructura está ordenada.

Ordenar un conjunto de datos es una tarea que se realiza frecuentemente y que resulta muy útil. Por este motivo y dado que, generalmente, se trabaja con una gran cantidad de datos almacenados en la misma estructura, es objetivo primordial encontrar el mejor algoritmo de ordenación. Un buen algoritmo de ordenación influye en gran medida en la eficiencia del programa que lo use, por lo que la ordenación es una de las áreas en las que a veces se anima al programador a sacrificar claridad en los algoritmos en favor de rapidez de ejecución de los mismos.

En este capítulo se estudiará por un lado algunos algoritmos de ordenación, por otro se analizarán los algoritmos de búsqueda y, por último, los de inserción de elementos en arrays ordenados y desordenados.

2. CLASIFICACIÓN: CONCEPTO

Clasificar significa establecer un orden entre los elementos que forman parte de una estructura de datos en particular. La importancia de la ordenación radica en la minimización de los tiempos de búsqueda de los elementos de una estructura si todos están ordenados.

La ordenación se organiza en función de una propiedad o criterio determinados. Debe tenerse en cuenta que pueden variar los criterios para clasificar e, incluso, el

conjunto de elementos sobre el que aplicar una clasificación pero la regla que se aplica para ordenar, no cambia. Quiere esto decir que la misma regla sirve para ordenar conjuntos diferentes, variando o no los criterios de ordenación que se apliquen.

2.1. Tipos de clasificación

SEGÚN EL ORDEN

Los elementos de una estructura pueden estar clasificados según el orden:

- De forma **ascendente**, si los elementos que forman la estructura de datos están organizados de menor a mayor.
- De forma **descendente**, cuando los elementos se encuentran clasificados de mayor a menor.

En ambos casos los valores repetidos, si existen, quedan en posiciones contiguas.

SEGÚN LA UBICACIÓN DE LOS DATOS

Según el lugar donde estén almacenados los elementos de una estructura, la ordenación será:

- **Interna**. Cuando los datos a ordenar están en la memoria principal y las estructuras que soportan estos datos son arrays. Los métodos que se usan se conocen como *métodos de ordenación de arrays*.
- **Externa**. Si los datos a ordenar están en la memoria secundaria, discos, cintas, etc. y las estructuras donde están almacenados estos datos son ficheros secuenciales. Estos métodos se conocen como *métodos de ordenación de ficheros secuenciales*.
- **Híbrida**. Los datos se almacenan en la memoria secundaria pero se ordenan en la memoria principal. Este es el caso de los datos almacenados en ficheros secuenciales que se vuelcan en memoria sobre una estructura de array para ordenarlos y, acabada la ordenación, se vuelven de nuevo al fichero por volcado del array en dicho fichero.

En este capítulo se estudiará solamente la ordenación interna. Las otras dos formas de ordenar datos se detallarán y usarán cuando se estudien las estructuras de datos ficheros.

3. CLASIFICACIÓN INTERNA: MÉTODOS SENCILLOS

Imagine que se dispone de una colección de 30 números escritos en un folio y se pidiera escribirlos en orden ascendente. Probablemente, buscaría el elemento más pequeño del conjunto, lo escribiría en otro folio y lo tacharía del original, y repetiría el

proceso mientras existieran números en el folio original. Implementar en Java un método tal como se ha descrito, supone el uso de un array auxiliar, donde escribir los números ordenados, lo que duplicaría la memoria utilizada. Además, hay que considerar que no es fácil "tachar" los elementos del array original ya escritos en el segundo. Sin embargo, con alguna modificación del algoritmo descrito podría utilizarse el mismo array para ordenar sus componentes.

Todos los métodos de ordenación que se van a estudiar utilizan el mismo array para realizar la ordenación. Se realizará un recorrido del array de tal manera que, aplicado cualquier método de clasificación, se asegure que, una vez acabado el algoritmo y según la propiedad establecida para su ordenación, los datos quedan ordenados sobre la misma estructura. Tenga en cuenta que utilizar otras estructuras auxiliares para realizar la ordenación supondría un abuso en el uso de recursos de la máquina, por lo que respecta al uso de memoria, y, por tanto, una ralentización de los algoritmos, por tener que acceder a estructuras distintas. Por otro lado, utilizando estructuras auxiliares, si se desea que los datos queden ordenados sobre la estructura original, habría que volcar la estructura auxiliar ordenada sobre la primera, lo que añadiría más tiempo al proceso de ordenación.

Los métodos más conocidos son: el de *intercambio directo* o *burbuja*, el método de *Selección directa*, conocido también como método de *búsqueda del menor* si se realiza la ordenación de forma ascendente o *búsqueda del mayor* si se realiza de forma descendente y por último el de *Inserción directa*.

Los algoritmos de todos estos métodos dividen el array en una parte ordenada y otra por ordenar, recorriéndolo cuantas veces sean necesarias, según el método aplicado, para garantizar el orden.

En todos los algoritmos que se detallan a continuación se usa un array de enteros y se ordena de forma ascendente.

3.1. Método de Intercambio directo o método de la Burbuja

La característica que distingue particularmente a la ordenación por intercambio, es que cada iteración pone el elemento más pequeño no ordenado en su lugar correcto. También podría implementarse colocando el elemento mayor en su lugar correcto, conociéndose el algoritmo como de la *plomada*, es decir "hundiendo" el elemento de más peso. En ambos casos, considerando la ordenación de forma ascendente.

Explicación del algoritmo: Se trata de recorrer el array repetidas veces, de forma que cada iteración ponga el elemento menor no ordenado en su sitio, lo cual provocará cambios en la posición de otros elementos del array. Siendo N el tamaño del array, se comienza con el elemento de la posición *N-ésima* y se van comparando sucesivos pares de elementos, intercambiándolos si el elemento de abajo del par (el situado en la casilla de índice más alto) es más pequeño que el elemento que le precede (situado en la casilla de índice más bajo). De esta forma, el elemento menor va "*burbujeando*" hacia arriba hasta el tope del array. Así, en la primera iteración

subirá a la 1ª posición el elemento menor. En la segunda iteración, usando la misma técnica, el elemento menor de la parte no ordenada del array subirá a la 2ª posición y así sucesivamente con los elementos restantes del array.

INTERFAZ

Propósito: ordenación ascendente de un array unidimensional de tamaño *tam*.

Entradas: un array. En principio también se necesita el tamaño, pero al implementar en Java lo conocemos, por estar implícito al crearlo.

Precondiciones: el array no debe estar vacío.

Salida: el mismo array modificado.

Postcondiciones: array [0], ..., array[tam-1] está ordenado según el criterio de ordenación establecido.

Como puede observarse en la descripción de la interfaz, el array es un elemento de entrada y salida, por lo que, en principio, debe pasarse por referencia, cuando se implemente en un lenguaje concreto habrá que conocer la forma de pasarlo en dicho lenguaje. Por ejemplo, en Java se pasan como referencias y además no habrá que pasar el tamaño, puesto que es un atributo de la clase Array.

PSEUDOCÓDIGO

```
burbuja (entero array [], entero tam)
Inicio
   entero i, j, aux
   para(i=0 , mientras i<tam-1, incremento 1)
      /*hace burbujear al menor*/
      para (j=tam-1, mientras j>i,decremento 1)
         si (array[j] < array[j-1])
            //intercambio de elementos
            aux = array[j];
            array[j] = array[j-1];
            array[j-1] = aux;
         Finsi
      Finpara
   Finpara
Fin-burbuja
```

En todos los métodos de ordenación se utilizan dos índices. El índice *i* separa la parte ordenada de la no ordenada, señalando siempre el primer elemento de la parte no ordenada, mientras que el índice *j* se mueve por la parte no ordenada del array buscando un elemento para, según las especificaciones, ponerlo en su lugar. Los elementos iguales quedarán en posiciones consecutivas.

EN CÓDIGO JAVA

```
void  burbuja (int[] array)
{
  int i, j;
  for (i=0 ;  i< array.length -1 ;  i++)
      for (j = array.length -1 ;  j > i ; j--)
          if (array[j] < array[j-1])
          //intercambio de elementos
          aux = array[j];
          array[j] = array[j-1];
          array[j-1] = aux;

}
```

Nota: En el ejemplo **se muestran las iteraciones de i** y se separa la parte ordenada de la no ordenada con el sombreado gris. Se aconseja seguir la traza para observar las iteraciones de j.

Ejemplo: Suponiendo un array de enteros con los datos que aparecen en el gráfico siguiente, se aplicará el algoritmo descrito anteriormente sobre el array que aparece como original en la figura.

	ARRAY		ARRAY		ARRAY		ARRAY		ARRAY
[0]	35	[0]	5	[0]	5	[0]	5	[0]	5
[1]	23	[1]	35	[1]	11	[1]	11	[1]	11
[2]	11	[2]	23	[2]	35	[2]	13	[2]	13
[3]	5	[3]	11	[3]	23	[3]	35	[3]	23
[4]	13	[4]	13	[4]	13	[4]	23	[4]	35
	original		Primera Iteración		Segunda Iteración		Tercera iteración		Cuarta iteración

3.2. Método de Selección directa

Explicación del algoritmo: El método divide el array en una parte ordenada y otra por ordenar. En principio, la parte no ordenada es todo el array. Se recorre el array repetidas veces, de forma que, en cada iteración, se busca, en la parte no ordenada, el elemento más pequeño para colocarlo en su lugar.

El método se conoce también como método de *búsqueda del menor* si se realiza la ordenación de forma ascendente o *búsqueda del mayor* si se realiza de forma descendente.

Se detalla a continuación el pseudocódigo del algoritmo con interfaz es idéntica al método anterior.

PSEUDOCÓDIGO

```
SELECCIÓN (entero ARRAY [], entero tam)
 Inicio
  Entero I, J, MÍNIMO
  Para (I= 0, mientras I<tam, incremento 1)
      MINIMO = I //bucle para encontrar el elemento más pequeño
      Para (J=I+1, mientras J<tam, incrementa en 1)
         Si(ARRAY[J] < ARRAY[MINIMO])
           MINIMO = J
         Finsi
      Finpara
      //intercambio para poner en su lugar el más pequeño
          aux = array[I];
          array[I] = array[MINIMO];
          array[MINIMO] = aux;

  Finpara
 Fin-selección
```

Ejemplo: Con el mismo array del ejemplo anterior vamos a seguir la traza del algoritmo descrito anteriormente. Observe que, en cada iteración, el elemento más pequeño de la parte no ordenada se coloca tras el último elemento de la parte ordenada.

	ARRAY		ARRAY		ARRAY		ARRAY		ARRAY
[0]	35	[0]	5	[0]	5	[0]	5	[0]	5
[1]	23	[1]	23	[1]	11	[1]	11	[1]	11
[2]	11	[2]	11	[2]	23	[2]	13	[2]	13
[3]	5	[3]	35	[3]	35	[3]	35	[3]	23
[4]	13	[4]	13	[4]	13	[4]	23	[4]	35
	original		Primera iteración		Segunda iteración		Tercera iteración		Cuarta iteración

3.3. Método de Inserción directa

Explicación del algoritmo: El método divide el array en dos partes, una ordenada y otra por ordenar, tomando el primer elemento de la parte no ordenada y haciéndolo avanzar hasta colocarlo en su lugar correspondiente dentro de la parte ordenada. Es decir, comienza considerando el primer elemento ordenado, continúa ordenando el segundo elemento respecto del primero, es decir, o se coloca delante o se queda donde está. Se hace avanzar el índice que señala el primer elemento no ordenado. Se coge el tercero y se coloca en su lugar correcto respecto a los dos anteriores. Se hace avanzar el índice y se repite el proceso hasta que todos los elementos ocupen su lugar correcto, con lo cual el array quedará ordenado según las especificaciones de partida.

A continuación, se implementará un algoritmo que realice el proceso descrito, para lo cual se mantendrá la misma interfaz considerada en los métodos anteriores.

PSEUDOCÓDIGO

```
INSERCIÓN (entero ARRAY [], entero tam)
Inicio
 Entero I, J
  Para (I=1 , mientras I<tam, incremento 1)
   Para (J=I, mientras ARRAY[J-1]>ARRAY[J] Y J>0, decremento 1)
         //intercambio de elementos
         aux = array[j];
         array[j] = array[j-1];
         array[j-1] = aux;
   Finpara
  Finpara
Fin-inserción
```

Al implementar en Java este algoritmo observa que esta expresión aunque correcta hace que se desborde el índice cuando tiene que ordenar los dos primeros elementos `ARRAY[J-1]>ARRAY[J] Y J>0`, esto es porque Java evalúa las expresiones en cortocircuito y por tanto si la primera es falsa no evaluará la segunda, si j esta en cero, j-1 es -1. Implementadlo así, comprobar el error en ejecución y corregir después.

Ejemplo: Con el mismo array del ejemplo anterior se seguirá la traza del algoritmo descrito anteriormente. Observe que en este caso se va cogiendo uno a uno los elementos de la parte no ordenada para buscarle su "hueco" en la parte ordenada.

	ARRAY		ARRAY		ARRAY		ARRAY		ARRAY
[0]	35	[0]	23	[0]	11	[0]	5	[0]	5
[1]	23	[1]	35	[1]	23	[1]	11	[1]	11
[2]	11	[2]	11	[2]	35	[2]	23	[2]	13
[3]	5	[3]	5	[3]	5	[3]	35	[3]	23
[4]	13	[4]	13	[4]	13	[4]	13	[4]	35
	original		Primera iteración		Segunda iteración		Tercera iteración		Cuarta iteración

4. CLASIFICACIÓN INTERNA: MÉTODOS AVANZADOS

Generalmente, los métodos avanzados están basados en los métodos sencillos, mejorando considerablemente sus prestaciones cuando se trata de ordenar arrays con un gran número de elementos.

El más conocido es el de ordenación rápida o Quick Sort.

4.1. Método de Ordenación rápida o Quick Sort

El algoritmo de ordenación rápida, basado en el de selección, se va a implementar estudiándolo de una manera recursiva. El enfoque recursivo aprovecha el hecho de que es más rápido y fácil, ordenar dos listas pequeñas que una mayor.

El nombre del método procede del hecho de que, en general, este procedimiento puede ordenar un array en un tiempo significativamente menor que cualquiera de los métodos comunes de ordenación. La estrategia básica es divide y vencerás.

Suponga que se dispone de un gran número de exámenes y se desea ordenar por nombre. Se puede usar el siguiente mecanismo de ordenación: elegimos un valor de división, por ejemplo la letra L, y dividimos el montón de exámenes en dos más pequeños, los que empiezan por A hasta L y los que van de M a Z. Tomamos el primero de estos nuevos montones y se elige un nuevo valor de división, para obtener otros dos nuevos montones, de A a F y de G a L. Se continúa el proceso con cada uno de los nuevos montones. Finalmente, todos los pequeños montones ordenados pueden apilarse uno sobre otro para obtener un conjunto ordenado de exámenes.

Se utilizará para implementar el método la estrategia recursiva descrita. Cada intento de ordenar una lista divide dicha lista en dos y usa el mismo método para ordenar cada una de las listas más pequeñas (un caso más pequeño). Este proceso continua hasta que las listas pequeñas no necesiten dividirse más (caso base).

INTERFAZ DEL MÉTODO RÁPIDO

Propósito: Método **rápido** de ordenación ascendente para ordenar un array (*lista*), desde la primera casilla (*primero*) hasta la última (*ultimo*), siendo *puntoParticion* la casilla por la que se divide el array.

Caso general: Realizar si *primero* es menor que *ultimo*:

- Cortar array por un punto.
- Ordenar utilizando el método **rápido** la mitad izquierda.
- Ordenar utilizando el método **rápido** la mitad derecha.

Caso base: Sería el “en-otro-caso” del si anterior, que no existe explícitamente, es decir, si *primero* es mayor o igual que *ultimo* no hacer nada, con lo que se acaban las llamadas recursivas. Existe una versión “Turbo” del método rápido que cuando la distancia entre primero y último es pequeña llama a un método directo para ordenarla.

Entradas: un array (*lista*), indice del *primero* y *ultimo* elemento.

Precondiciones: el array (*lista*) no debe estar vacío.

Salida: el mismo array (cambiado).

Postcondiciones: lista [0],...,lista[N-1] está ordenado ascendentemente.

PSEUDOCÓDIGO DEL MÓDULO RÁPIDO

Versión recursiva en dos módulos: *rapido* y *partir*.

```
rapido(entero lista[ ], primero, ultimo)
     entero puntoParticion
    si (primero < ultimo)
        /* divide el array según las especificaciones de
        partir dejando puntoParticion en su sitio*/
        puntoParticion = partir(lista, primero, ultimo)
        //ordena las dos mitades.
        <rapido> (lista, primero, puntoParticion - 1)
        <rapido> (lista, puntoParticion + 1, ultimo)
    finsi
fin rapido
```

Verificación del algoritmo del programa principal:

1. ¿*Existe un caso base no recursivo?*: sí, cuando **primero** >= **ultimo**. De hecho, no llegará a ser **primero** mayor que **ultimo**, sino que acabará cuando sea igual. Hay un elemento en la lista, en cuyo caso no se llama a **rapido**.

2. ¿*Cada llamada recursiva supone un caso más pequeño del problema?*: sí, **partir** divide el array en dos trozos menores que el original y cada uno de esos trozos también se ordenará "*rápidamente"*. Aún en el caso en que **puntoParticion** sea el primer o último elemento del array los dos trozos, son también menores que el original.

3. *Suponiendo que las llamadas recursivas funcionan, ¿el algoritmo completo funciona?*. La primera llamada ordena todos los elementos que son menores o iguales al que está en **puntoParticion**, y la segunda llamada lo hace con los que son mayores. El valor **puntoParticion** está en su sitio, luego el array está ordenado. Por tanto, puede concluirse que el diseño funciona correctamente si es correcto el diseño del módulo **partir**.

A continuación, se detalla el algoritmo de **partir**.

INTERFAZ DEL MÓDULO PARTIR

Propósito: El objetivo es escoger un valor de partición, **valorParticio**n, y reordenar el array de manera que se coloquen delante de ese valor todos los elementos del array que son menores o iguales a dicho valor de partición y detrás a todos los que son mayores. Se devolverá el punto de partición, índice, de la tabla donde ha quedado situado dicho valor de partición. Es decir, puesto que el objetivo es ordenar el array de forma ascendente, el algoritmo elige el **valorParticion** y rehace el array de forma que:

- tabla [ppio], ...,tabla[puntoParticion – 1] <= valorParticion
- tabla [puntoParticion] = valorParticion

- tabla [puntoParticion + 1], ...,tabla[fin] > valorParticion

Entradas: un array, el índice de la primera y de la última casilla.

Precondiciones: el array no debe estar vacío.

Salida: el mismo array (cambiado) y el punto por donde se ha partido el array.

Postcondiciones: Las siguientes,

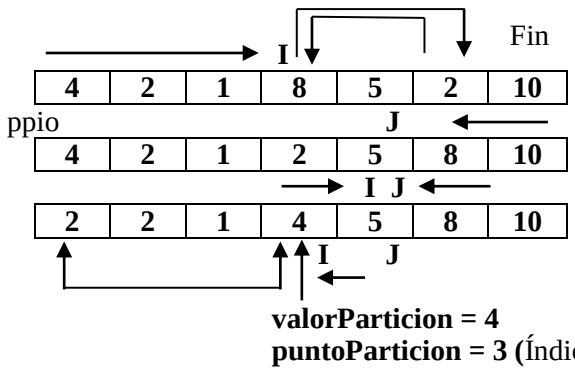
- Según las especificaciones, array[ppio], ..., array[fin] quedará dividido respecto a valorParticion.
- Asociado al nombre del subprograma se devuelve el lugar por el que se divide el array. Por tanto, el subprograma se diseñará como función.

PSEUDOCÓDIGO DE LA FUNCIÓN PARTIR

```
entero partir (entero tabla[ ], ppio, fin)
        /* Estas dos variables no son necesarias, se usan
         para que el algoritmo sea más fácil de leer, i
         recorre el array de  arriba hacia abajo, o de
         izquierda a derecha , y j desde abajo hacia arriba, o
         de derecha a izquierda */

      entero valorParticion, i, j, aux
        /* Se elige por comodidad como valor de partición el
         elemento que se encuentra en la primera casilla */

      valorParticion = tabla[ppio]

        /*En el siguiente Para no aparece la variación de los
         índices porque se incrementan o decrementan dentro */

      Para(i = ppio+1, j = fin; mientras (i< j); )

        /* A continuación, se saltan todos los elementos del
         array que sean menores o iguales a valorParticion
         empezando por la primera casilla. Se utiliza la
         segunda condición i<j para controlar el
         desbordamiento de rango del array y para que no se
         crucen los índices  */

        Para( ;mientras((tabla[i]<=valorParticion)Y(i<j));i++)
        Finpara
        /* i apunta al primer elemento encontrado mayor que
         valorPartición */

        /* Se saltan todos los elementos que son mayores que
           valorParticion empezando por la última casilla */

        Para( ;mientras((tabla[j]>valorParticion)Y(i<j));j--)
```

```
        Finpara
        /*j apunta al primer elemento que ha encontrado menor
          o igual que valorPartición, salvo cuando i =j */

        /* A continuación intercambiamos el primer elemento
          encontrado mayor que valorParticion, tabla [i], con
          el primer elemento encontrado menor, tabla[j]*/

         /*Se podría añadir un if para intercambiar sólo en el
          caso en que i sea distinto de j, pero serían
          iguales sólo una vez en cada llamada, por tanto, no
          vale la pena. Incluir el if iría en detrimento de
          la velocidad de ejecución del algoritmo */
        aux = tabla [ i ]
        tabla [ i ] = tabla [ j ]
        tabla [ j ] = aux
      FinPara

        /* Decrementamos i puesto que se ha quedado apuntando
          a la misma casilla que j y debemos llevar
          valorParticion a la casilla anterior a la que ocupa
          el primer valor mayor que él y devolver la posición
          donde se ha colocado */

      SI (valorParticion <= tabla[i])
        i = i - 1
      FINSI
      // Intercambio para poner en su sitio valorParticion.
      tabla[ppio] = tabla[i]
      tabla [ i ] = valorParticion
      Devolver i
FINPartir
```

Ejemplo: Se analizará el algoritmo siguiendo la traza con el array de la figura siguiente, donde las flechas señalan el movimiento de los índices.



Se dispone originalmente de un array de enteros formado por los elementos que aparecen en la figura. Se llama al algoritmo **rapido** para ordenar ese array con los siguientes argumentos en la primera llamada:

```
rapido (array, 0, 6)
```

Como primero (0) es menor que último (6), se llama a **partir** con los siguientes argumentos, el array, *ppio* que vale 0 y *fin* que vale 6.

```
puntoPartición = partir (array, 0, 6)
```

De esta llamada se obtiene en *puntoPartición* el lugar donde ha quedado el elemento colocado en su lugar adecuado.

A continuación, se llama a *rapido* para ordenar los dos arrays que quedan:

```
rapido (array, 0, puntoParticion - 1)
```

Cuando se acabe de ordenar los trozos que resultan de las sucesivas llamadas a *rapido* en este tramo se pasará a resolver la otra rama:

```
rapido (array, puntoParticion + 1, 6)
```

Se repetirán las tres últimas sentencias hasta que **primero** sea igual a **ultimo**.

Como el algoritmo recursivo funciona, se realiza la traza al algoritmo de partir.

— Se comienza eligiendo como valor de partición el elemento que se encuentra en primer lugar, es decir, 4. Esto es decisión del programador que implementa el método, nada impide haber elegido el último o cualquier otro. Existen algoritmos para elegir el mejor punto de partición en primera instancia.

— Primer *para*:

- Primera iteración:
  - En la figura, i se mueve hacia la derecha saltando todos los elementos menores o iguales a 4 y queda apuntando a la casilla donde se encuentra 8.
  - j se mueve desde el final hacia la izquierda, saltando todos los elementos mayores a 4 y queda apuntando a la casilla donde se encuentra 2.
  - Se intercambian 8 y 2, para que queden los menores a 4 a un lado y los mayores a otro.
- Segunda iteración:
  - i sigue moviéndose hacia la derecha, saltando todos los elementos menores o iguales a 4 y queda apuntando a 5.

- j sigue moviéndose hacia la izquierda saltando todos los elementos mayores a 4, pero queda apuntando a 5, puesto que cuando llega aquí, i no es menor que j, sino igual.

- Se intercambia 5 consigo mismo. Ya hemos comentado que no se incluye el *if* para evitar esto por rapidez en el código.

▪ Tercera iteración: No se realiza puesto que i no es menor que j.

— A continuación, sigue la línea de código ejecutando el *SI*: como valor de partición, 4, es menor que tabla[i], 5, se decrementa i para hacerle hueco al valor de partición.

— Se intercambia el valor de partición 4 por 2.

Ha finalizado esta primera llamada a ***partir*** colocando el valor de partición precisamente en el lugar que le corresponde dentro de la serie de elementos del array y devolviendo al subprograma **rapido** la casilla donde se encuentra el valor de partición, o sea, 3 en este caso.

Se realizarán llamadas recursivas hasta que **primero** sea igual a **ultimo**.

Explicación de:

```
SI (valorParticion <= tabla[i])
    i = i - 1
FINSI
```

— Caso en que *valorPartición* sea menor que tabla[i]: suponiendo que la serie de elementos antes de llegar al *SI* fuera 3,1,1,2,4,7,5,6, i y j quedan apuntando a 4, de manera que debe decrementarse i para intercambiar el valor de partición 3 por 2.

— Caso en que *valorPartición* sea igual que tabla[i]: suponiendo que la serie de elementos antes de llegar al *SI* fuera 3,1,1,2,3,7,5,6, i y j quedan apuntando al 3 posterior, de manera que hay que decrementar i para intercambiar el valor de partición 3 por 2. Observe que los elementos iguales quedan consecutivos.

— Caso en que *valorPartición* sea mayor que tabla[i]: suponiendo que la serie de elementos antes de llegar al *SI* fuera 3,1,1,2,2,7,5,6, i y j quedan apuntando al 2 que aparece más a la derecha, de manera que no hay que decrementar i, pasando a intercambiar el valor de partición 3 por el 2 de más a la derecha.

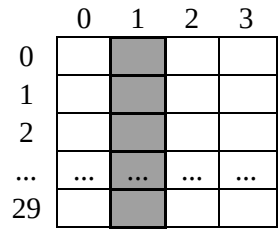
6. ORDENACIÓN BAJO CONDICIONES COMPLEJAS

Es frecuente tener que ordenar arrays que contienen varios campos de información, por ejemplo, arrays bidimensionales, y tener que hacerlo por uno o varios de sus campos. En estos casos, se dice que se ordena por una o varias *claves*, siendo una de ellas la primaria y única, y las demás secundarias y repetibles. Una *clave de ordenación* es un campo del array cuyo valor se usa para ordenar el array completo. En estos casos, tan sólo habría que modificar el algoritmo de ordenación para adaptarlo al tipo de dato del array en particular.

Tenga en cuenta que cuando se desea ordenar arrays bidimensionales, al realizar el intercambio, deben intercambiarse todos los elementos de la fila en cuestión, aunque el criterio de ordenación se esté aplicando a una característica en particular de alguna columna concreta del array.

Suponga que el siguiente array debe ordenarse por un criterio que compete exclusivamente a la columna 1, la comparación se haría con los elementos de dicha columna, pero el intercambio afecta a todos los elementos de la fila, para no perder consistencia respecto a la información referente al objeto de dicha fila. Por ejemplo, si se dispone de un array donde se guardan las notas de los 30 alumnos de un aula junto a su código, de manera que en la columna 0 se guarda el código y en las restantes las notas, si se aplica el método de la burbuja para ordenar ascendentemente por la nota que se almacena en la columna 1 se escribiría

```
......
Si (array[1][j]<array[1][j-1])
   intercambiar(array[0][j],array[0][j-1])
   intercambiar(array[1][j],array[1][j-1])
   intercambiar(array[2][j],array[2][j-1])
   intercambiar(array[3][j],array [3][j-1])
Finsi
......
```



7. BÚSQUEDA

En algunas ocasiones, el acceso a componentes de una estructura de datos se limita a elementos que ocupan una posición determinada dentro de la estructura. Por ejemplo, si se desea el quinto elemento de un array se encuentra en *array [4]*. Sin embargo, es habitual en otras ocasiones, que se desee acceder a un elemento usando algún valor clave o alguna característica en particular. Por ejemplo, si en una lista de los alumnos de un grupo queremos buscar aquél cuyo nombre es *"Pablo Luján Marchena"*. Se necesita algún método de búsqueda que, de un modo eficaz, permita obtener el elemento deseado.

Al igual que los métodos de ordenación, existen métodos de búsqueda interna, cuyo objetivo es buscar elementos en estructuras de datos que están completamente almacenados en memoria principal y métodos de búsqueda externa para acceder a

elementos de estructuras almacenadas en memoria secundaria. En este capítulo se hablará de la búsqueda interna.

La búsqueda va a consistir en saber si el elemento forma parte o no de la estructura, generalmente, indicando la posición que ocupa dentro de ella si fuera encontrado y un indicador que refleje que el elemento no está en caso de no ser encontrado.

7.1. Búsqueda secuencial o lineal

La búsqueda secuencial es la técnica más sencilla que existe. Empezando por la primera casilla del array, se irá examinando secuencialmente cada una de las casillas restantes, hasta encontrar el elemento buscado o finalizar el array. Esta técnica no sólo es apropiada para arrays, sino que también es aplicable a listas enlazadas y a ficheros secuenciales, estén o no ordenadas dichas estructuras de datos, aunque el método puede mejorarse aumentando su eficiencia si la estructura está ordenada.

INTERFAZ

Propósito: localizar un elemento en un array unidimensional de elementos enteros de tamaño N.

Entradas: un array, el elemento a buscar (*dato*).

Precondiciones: el array debe no estar vacío.

Salida: un entero con la posición si encuentra el dato ó -1 en caso contrario.

Postcondiciones: asociado al nombre de la función se devuelve la posición del elemento buscado ó – 1 si no se encuentra.

PSEUDOCÓDIGO

```
Entero busqueda_sec (entero array[], entero dato)
 inicio
   entero i, encontrado = -1
   Para (i=0, mientras encontrado = -1 y i < array.lenght, i++)
       Si (array[i]==dato)
          encontrado = i
       Fin_si
   Fin_para
   devolver (encontrado)
FinBus-Sec
```

7.2. Búsqueda binaria

La ventaja del algoritmo de búsqueda secuencial es su sencillez, en oposición a la búsqueda binaria, que aunque más eficaz en arrays ordenados, tiene un proceso más

complejo. Aplicando el algoritmo secuencial en el peor de los casos se realizan *N* comparaciones, puesto que sólo se examina una casilla en cada pasada y pudiera ser que el elemento buscado se encuentre en la última posición del array. Sin embargo, si el array estuviera ordenado, se puede mejorar el tiempo de búsqueda, aumentando la eficiencia del algoritmo, aunque sacrificando la sencillez del código.

El algoritmo, **sólo aplicable a arrays ordenados**, consiste en lo siguiente: el array se dividirá en dos mitades y se buscará el dato en la mitad que corresponda, siguiendo el mismo método hasta encontrarlo o hasta estar seguros de que el elemento no se encuentra en dicho array, lo cuál no implica recorrer el array completo. En principio, se compara el dato a buscar con el elemento que ocupa la posición mitad. Si este elemento no es igual al buscado, se reduce el intervalo de búsqueda a la mitad inferior o a la mitad superior, dependiendo del lugar donde sea posible encontrar dicho elemento buscado. Por ejemplo, si el array está ordenado ascendentemente y el elemento buscado es menor que el que se encuentra en la casilla mitad, habrá que buscarlo entre las posiciones anteriores a dicha mitad.

Si el número de elementos del array es impar, se tomará la parte entera de la mitad para realizar la búsqueda.

Este método es más eficaz que la búsqueda secuencial cuando se trata de buscar en arrays ordenados con una gran cantidad de elementos. Es posible que, en arrays pequeños, sea incluso menos rápido que el secuencial, pues aunque realiza menos iteraciones y esto supone una disminución de tiempo de ejecución, también efectúa más comparaciones y esto significa un aumento considerable de tiempo de ejecución.

INTERFAZ

Propósito: buscar un elemento en un array ordenado, unidimensional, de enteros y de tamaño N.

Entradas: un array, el elemento a buscar (*dato*).

Precondiciones: el array no debe estar vacío y si ordenado.

Salida: un entero con la posición si se encuentra el dato ó -1 en caso contrario.

Postcondiciones: Asociado al nombre de la función se devuelve la posición del elemento en la estructura ó –1 si no se encuentra dentro de ella.

PSEUDOCÓDIGO

```
Entero busquedaBinaria(entero array[], entero dato)
 Inicio
  entero inicio=0, fin=n-1, med=p.e ((ini +fin)/2),
  encontrado=-1   //No encontrado
  Mientras (inicio<= fin y encontrado= -1)
          Si(array[med] = dato)
```

```
            encontrado = med
            /*Mover inicio y fin para dividir el área de
              búsqueda*/
          En otro caso
            Si(array[med]>dato) /*El dato estará del principio
                                  a la mitad*/
                fin = med – 1
            En otro caso //El dato estará de la mitad al final
                inicio = med + 1
            Fin_si
            med = p.e. ((inicio + fin)/ 2) /*p.e. parte entera
                                             de la división*/
          Fin_si
  Fin_mientras
  Devolver (encontrado)
FinBus-Bin
```

8. INSERCIÓN

Es posible que se desee añadir nuevos elementos a los ya existentes en un array. Habrá de tenerse en cuenta si el array está ordenado o no. En el segundo de los casos, son muchas las alternativas que se presentan y el algoritmo dependerá de la interfaz que se establezca a tal efecto. Si el array está ordenado, caben dos alternativas, insertar el elemento después del último y, a continuación, ordenar el array de nuevo, lo cual supone una forma bastante ineficaz de solucionar el problema, o buscar el lugar que debe ocupar en la lista el nuevo elemento y mover los demás con objeto de "hacerle hueco" y almacenarlo.

Dado que las alternativas para construir el algoritmo de inserción son múltiples, se presentará una interfaz amplia para que el lector construya la que más se atenga a sus necesidades.

INTERFAZ BÁSICO

Propósito: insertar un elemento en un array unidimensional de enteros de tamaño *N*. El array puede estar ordenado o no, los elementos pueden estar repetidos o no. Se puede informar del éxito o fracaso de la operación. El fracaso puede ser debido a que el elemento ya está en el array y no están permitidas las repeticiones, o porque el array está lleno y no caben más elementos.

Entradas: un array, la última casilla con dato válido o su tamaño (se dispone de él si se implementa en Java) y el elemento que se desea incorporar a la estructura.

Precondiciones/restricciones: Según la filosofía de diseño de Java las responsabilidades asociadas a las precondiciones debe asumirlas las funcionalidades

por lo que se controlará que el array no debe estar lleno, e informar de ello en caso de no poder insertar por esta causa.

Salida: el mismo array, cambiado o no, y un indicador de éxito si se ha añadido o fracaso si no se ha añadido el elemento. Se actualizará también el nuevo tamaño del array o la última casilla con dato significativo.

Postcondiciones: el array queda con un elemento más si se ha insertado o sin cambios si ha fracasado la operación de inserción. Asociado al nombre de la función, se devuelve un valor en razón del éxito o fracaso de la operación. Este valor puede ser entero, carácter o cualquier otro que represente el tipo de dato lógico.

Algoritmo de inserción en un array ordenado

Dado que el array está ordenado, en primer lugar se buscará en dicho array el lugar que le corresponde al elemento que se desea insertar. A continuación, en caso de que el orden sea ascendente, deben desplazarse todos los valores mayores al nuevo una casilla hacia abajo en el array para hacer sitio al nuevo elemento.

INTERFAZ

Propósito: insertar un elemento en un array ordenado ascendentemente, unidimensional, de enteros y de tamaño *N* donde no se permiten elementos repetidos.

Entradas: un array, el elemento a insertar (*dato*) y posición de la última casilla con dato válido.

Precondiciones: el array debe estar ordenado. Por tanto, es responsabilidad del programa que llama, asegurarse que se cumplen la precondición antes de realizar la llamada.

Salida: un entero y el mismo array modificado si se insertó con éxito.

Postcondiciones: asociado al nombre de la función se dispone de –1 si no se ha insertado satisfactoriamente ó la nueva longitud del array en otro caso.

PSEUDOCÓDIGO

```
entero insertarOrden(entero array[],entero dato, Entero ultCas)
 Inicio
  entero i, s, nuevaLong = -1 //fracaso
  para (i=0; array[i]<dato && i<ultCas; i++)
  finpara /*i queda en el lugar de inserción, o donde se
            encuentra el dato buscado*/
  Si (array[i]!=dato)
      Para (s = ultCas; s>=i; s--)
          array[s+1] = array[s];
      finPara
```

```
      array[i]=dato;
      ultCas++; //se actualiza la última casilla con dato
      válido
      nuevaLong = ultCas;  //éxito
   Fin_si

   Devolver (nuevaLong)
 fin InsertarOrden
```

NOTA: Los algoritmos desarrollados en pseudocódigo deberán ser implementados en Java con las interfaces correspondientes, incluyéndolos en clases arrays o en clases de utilidades. Es aconsejable implementarlos con tipos genéricos.

ALGORITMOS DE ORDENACIÓN BAILADOS

Burbuja

http://www.youtube.com/watch?v=lyZQPjUT5B4

Inserción

http://www.youtube.com/watch?v=EdIKIf9mHk0

Selección

http://www.youtube.com/watch?v=Ns4TPTC8whw

Quick Sort

http://www.youtube.com/watch?NR=1&v=kDgvnbUIqT4&feature=endscreen

Shell

http://www.youtube.com/watch?v=CmPA7zE8mx0

Merge

http://www.youtube.com/watch?v=XaqR3G_NVoo

JAVA

http://www.docjar.com/docs/api/java/util/package-index.html

ANEXOS:

OTROS ALGORITMOS DE ORDENACIÓN SIMPLES

OTROS ALGORITMOS DE ORDENACIÓN AVANZADA

ORDENACIÓN EN JAVA

BÚSQUEDA EN JAVA

NOTA Interfaz Comparable

```
package java.lang;

public interface Comparable<T>{

    int compareTo (T o);

}
```

- **compareTo** establece un orden natural para los objetos.
- Debe coincidir en criterio con el método **equals**.
- Valor entero devuelto:
    - Negativo si *this* es menor que *o*, es decir, está antes en el orden.
    - Cero si *this* es igual a *o*, es decir, el método *equals* entre ellos devuelve *true*.
    - Positivo si *this* es mayor que *o*, es decir, está después en el orden.