

## 4.5. Algoritmos de Búsqueda en Memoria Principal

En esta sección vamos a estudiar los algoritmos de búsqueda de un elemento en un arreglo (¿dónde está un elemento dado?) y los algoritmos de búsqueda en general de distintas características en un conjunto de valores (por ejemplo, ¿cuántas veces aparece un valor en un arreglo?), los cuales necesitamos para escribir los métodos que implementan los requerimientos funcionales del caso de estudio.

### 4.5.1. Búsqueda de un Elemento

El proceso de búsqueda de un elemento en un arreglo depende básicamente de si éste se encuentra ordenado. En caso de que no haya ningún orden en los valores, la única opción que tenemos es hacer una búsqueda secuencial utilizando para esto el patrón de recorrido parcial estudiado en cursos anteriores.

#### Tarea 13



**Objetivo:** Escribir el método que busca un elemento en un arreglo.

Para la clase `Muestra` escriba dos versiones del método que busca un elemento: en el primero, haga una búsqueda secuencial suponiendo que la muestra no está ordenada. En el segundo, haga también una búsqueda secuencial pero suponga que la muestra está ordenada ascendentemente. ¿Qué cambia de un algoritmo a otro?

```
public class Muestra
{
    public boolean buscarSecuencial( int valor )
    {

    }
}
```

```
public class Muestra
{
    public boolean buscarSecuencial( int valor )
    {

    }
}
```

Si el arreglo en el que queremos buscar el elemento se encuentra ordenado, podemos utilizar una técnica muy eficiente de localización que se denomina la **búsqueda binaria**. La idea de esta técnica, que se ilustra en el ejemplo 12, es localizar el elemento que se encuentra en la mitad del arreglo. Si ese elemento es mayor que el valor que estamos buscando, podemos descartar en el proceso de búsqueda la mitad final del arreglo (¿para qué buscar allí si todos los

valores de esa parte del arreglo son mayores que aquél que estamos buscando?). Si el elemento de la mitad es menor que el valor buscado, descartamos la mitad inicial del arreglo. Piense que sólo con lo anterior ya bajamos el tiempo de ejecución del método a la mitad. Y si volvemos a repetir el mismo proceso anterior con la parte del arreglo que no hemos descartado, iremos avanzando rápidamente hacia el valor que queremos localizar.

### Ejemplo 12



**Objetivo:** Mostrar el funcionamiento del algoritmo de búsqueda binaria.

En este ejemplo se muestra cada una de las etapas por las que pasa el algoritmo de búsqueda binaria para localizar un elemento en un arreglo ordenado de valores.

0	1	2	3	4	5	6	7	8	9	10	11
21	24	54	55	57	58	75	81	87	88	90	95



Éste es el arreglo ordenado sobre el que vamos a hacer las búsquedas. Tiene 12 valores.

0	1	2	3	4	5	6	7	8	9	10	11
21	24	54	55	57	58	75	81	87	88	90	95
↑				↑							↑
início				medio		fin					

21	24	54	55	57	58	75	81	87	88	90	95
espacio de búsqueda						descartado					



Vamos a buscar el valor 55. En la primera iteración marcamos las posiciones de inicio (0) y fin (11) de la parte del arreglo en la que queremos buscar. Con esos valores localizamos la posición del medio (5). Puesto que 55 (valor que estamos buscando) es menor que el valor de la mitad (58), descartamos la segunda mitad del arreglo en la búsqueda.

0	1	2	3	4	5	6	7	8	9	10	11
21	24	54	55	57	58	75	81	87	88	90	95
↑											
início			medio		fin						

21	24	54	55	57	58	75	81	87	88	90	95
----	----	----	----	----	----	----	----	----	----	----	----



En la segunda iteración los valores de fin (4) y medio (2) han sido recalculados para contemplar únicamente la parte del arreglo en la que podría aparecer el valor buscado. Puesto que estamos localizando el 55, y éste es mayor que el valor que se encuentra en la posición del medio (54), descartamos de la búsqueda los elementos que están entre inicio y medio.

0	1	2	3	4	5	6	7	8	9	10	11
21	24	54	55	57	58	75	81	87	88	90	95
		início	medio	fin							



Repetimos el proceso recalculando los valores de inicio, fin y medio. Puesto que el valor que se encuentra en la posición del medio es el valor buscado (55), terminamos el proceso.

0	1	2	3	4	5	6	7	8	9	10	11
21	24	54	55	57	58	75	81	87	88	90	95
↑					↑						↑
início				medio		fin					

21	24	54	55	57	58	75	81	87	88	90	95
descartado						espacio de búsqueda					



Vamos a buscar ahora un valor inexistente en el arreglo para ver el comportamiento del algoritmo. Busquemos entonces el 92. En la primera iteración descartamos los valores de la primera mitad, puesto que el 92 es mayor que el valor del medio (58).

0	1	2	3	4	5	6	7	8	9	10	11
21	24	54	55	57	58	75	81	87	88	90	95
						↑		↑		↑	
						inicio		medio		fin	

En la segunda iteración descartamos de nuevo la primera parte de la zona de búsqueda, puesto que el 92 es mayor que el elemento del medio (87).

0	1	2	3	4	5	6	7	8	9	10	11
21	24	54	55	57	58	75	81	87	88	90	95
									↑	↑	↑
									inicio	medio	fin

En la tercera iteración reducimos la zona de búsqueda a un solo elemento (95), puesto que el 92 es mayor que aquél que se encuentra en la posición del medio (90).

0	1	2	3	4	5	6	7	8	9	10	11
21	24	54	55	57	58	75	81	87	88	90	95
										↑	
										inicio	medio
										fin	

Puesto que el único elemento que queda en la zona de búsqueda es el 95 y es diferente del valor buscado, podemos afirmar en este punto que el 92 no está en el arreglo.

Fíjese que en sólo cuatro iteraciones nos pudimos dar cuenta de que un valor no existía en un arreglo de 12 posiciones. Nada mal, ¿no?

A continuación se muestra el código del método de la clase `MuestraOrdenada` encargado de buscar un elemento usando la técnica de búsqueda binaria:

```
public boolean buscarBinario( int valor )
{
    boolean encuentre = false;
    int inicio = 0;
    int fin = valores.length - 1;
    while( inicio <= fin && !encontre )
    {
        int medio = ( inicio + fin ) / 2;
        if( valores[ medio ] == valor )
        {
            encuentre = true;
        }
        else if( valores[ medio ] > valor )
        {
            fin = medio - 1;
        }
        else
        {
            inicio = medio + 1;
        }
    }
    return encuentre;
}
```

Vamos a utilizar tres variables enteras (`inicio`, `medio` y `fin`) para indicar en cada iteración la posición donde comienza la zona de búsqueda, la posición media y la posición en la que termina.

Inicialmente "`inicio`" comienza en 0 y "`fin`" en la última posición del arreglo.

La posición media se calcula sumando las posiciones de `inicio` y `fin`, y dividiendo el resultado entre 2.

Luego, hay tres casos: (1) si el valor es igual al de la mitad, ya lo encontramos; (2) si el valor es menor que el de la mitad, reposicionamos la marca final de la zona de búsqueda, y (3) si el valor es mayor, movemos el inicio de la zona de búsqueda.

Este proceso se repite mientras no hayamos encontrado el elemento y mientras exista algún elemento en el interior de la zona de búsqueda.

**Tarea 14**

**Objetivo:** Medir el tiempo de ejecución de los algoritmos de búsqueda e intentar identificar diferencias entre ellos.

Localice en el CD el ejemplo `n7_muestra`, cópielo al disco duro y ejecútelo. Siga luego las instrucciones que aparecen a continuación.

Algoritmo	5000	10000	20000	40000	60000	80000	100000	200000	Llene la tabla de tiempos de ejecución de los dos algoritmos de búsqueda, para muestras de distintos tamaños. Utilice una muestra entre 1 y 200.000.
Secuencial									
Binaria									

¿Qué se puede concluir de la tabla anterior?

#### 4.5.2. Búsquedas en Estructuras Ordenadas

En esta sección planteamos, en términos de tareas, los métodos de búsqueda necesarios para completar el programa del caso de estudio.

**Tarea 15**


**Objetivo:** Escribir los métodos que se necesitan en el caso de estudio, para calcular algún valor sobre la muestra ordenada.


Escriba los métodos que se piden a continuación en la clase `MuestraOrdenada`. Asegúrese de utilizar en su algoritmo el hecho de que los valores de la muestra se encuentran ordenados, para así tratar de encontrar una solución lo más eficiente posible.


```
public int contarOcuurrencias( int valor )
{
    ...
}
```

Cuenta el número de veces que aparece un valor en la muestra ordenada.

```
public int contarElementosEnRango( int inf, int sup )
{
```

 Cuenta el número de valores distintos que hay en la muestra ordenada.

 Cuenta el número de elementos que hay en un rango de valores (incluidos los extremos).

 Retorna el valor más frecuente en la muestra ordenada. Si hay varios números con la misma frecuencia, retorna el menor de ellos.