# Algoritmos de búsqueda y ordenación

Los arrays son uno de los medios principales para almacenar datos en programas. Debido a esta causa, existen operaciones fundamentales cuyo tratamiento es imprescindible conocer. Estas **operaciones** son:

* la **búsqueda** de un elemento en un array
* la **ordenación** de un array

Los algoritmos que vamos a estudiar, son algoritmos generales que se pueden implementar con cualquier lenguaje de programación que maneje arrays.

## **Búsqueda**

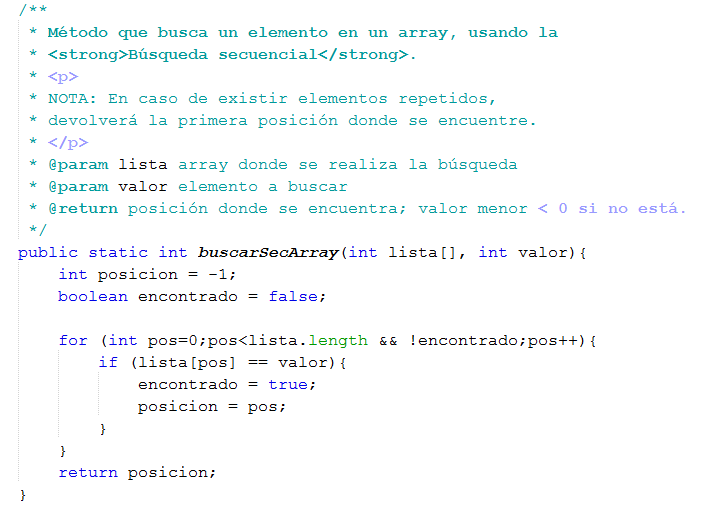
La búsqueda de un elemento dado en un array, es una operación muy usual en cualquier trabajo de programación. Dos algoritmos típicos que realizan esta tarea son:

* la **búsqueda secuencial** o lineal
* la **búsqueda binaria** o dicotómica.

### Búsqueda secuencial

Supongamos una lista de elementos almacenados en un array, la búsqueda secuencial consiste en recorrer la lista desde el primer elemento al último, comparando cada elemento de la lista con el elemento a localizar, pudiendo ocurrir:

1. que el **elemento** **exista** en cuyo caso se indicará que el elemento existe y devolverá la posición donde ha sido encontrado.
2. que el **elemento** **NO exista** en cuyo caso se indicará que el elemento no existe, devolviendo -1.



fun busquedaSecuencial(numeros: Array<Int>, elementoBuscado: Int): Int {

var numero = -1

for (i in numeros.indices) {

if (numeros[i] == elementoBuscado) {

numero = i // Se encontró el elemento, se devuelve su índice

}

}

return numero // Si no se encuentra, se devuelve -1

}

val numeros = arrayOf(4, 7, 2, 9, 1, 5, 8, 3, 6)

val elementoBuscado = 5

val indiceEncontrado = busquedaSecuencial(numeros, elementoBuscado)

### Búsqueda binaria

Este tipo de búsqueda solo se puede aplicar en **listas o arrays ordenados**.

Se trata de una búsqueda similar a la de una palabra en un diccionario, donde aprovechamos el orden alfabético para dirigirnos con rapidez al lugar donde puede estar la palabra buscada.

Se trata de **dividir** **el espacio de búsqueda en sucesivas mitades** hasta encontrar el elemento buscado, o hasta que ya no pueda hacer más mitades.

* Primero hallamos el **índice de la mitad del array** y miramos si el elemento coincide con él.
* Si no coincide, averiguamos donde debería estar el elemento buscado, si en la **sublista de la derecha o de la izquierda**, y dentro de esa mitad hago lo mismo sucesivamente.

**Funcionamiento:**

Para su realización usamos 3 variables (IZQ, CEN y DER) que nos indican en qué zona de lista nos encontramos.

* A la lista de **N elementos ordenados ascendentemente** la llamamos **Lista** (donde N es una constante declarada).
* Al **dato a buscar** (introducido por teclado) lo denominamos **X**, que debe ser del mismo tipo que los elementos de la lista.

| 4 | 8 | 9 | 10 | 14 | 25 | 84 | 99 |
| --- | --- | --- | --- | --- | --- | --- | --- |

IZQ=0 CEN=4 DER=7

El algoritmo de búsqueda binaria o dicotómica consiste en **repetir** el siguiente bucle:

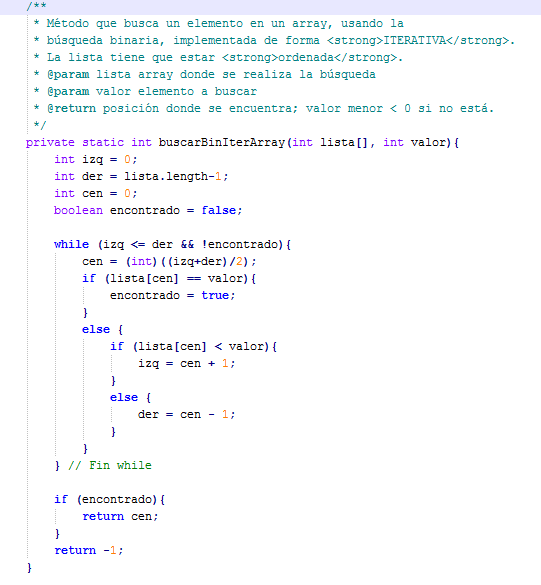
**Mientras** exista el intervalo de búsqueda (o sea, IZQ sea menor o igual que DER) y **NO se haya encontrado**.

* Calcular el elemento central de la lista CEN (conociendo IZQ y DER)



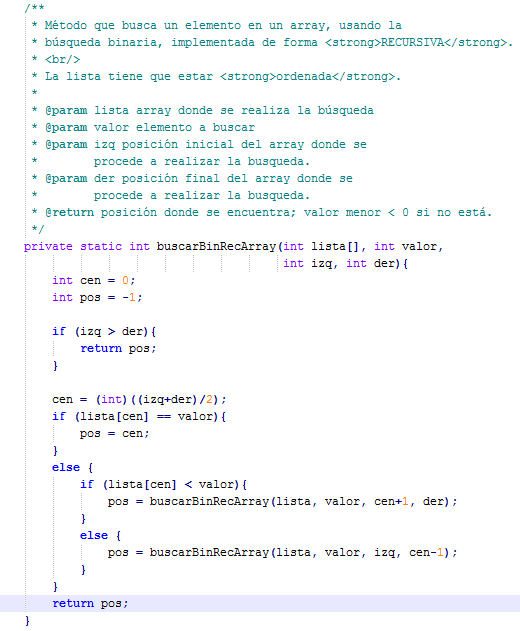
* Si Lista(CEN) es **distinto** de X, ver si es mayor o menor que X:
* Si es mayor, fijar **DER** en el elemento **anterior** al central, para buscar a la derecha de CEN
* Si es menor, fijar **IZQ** en el elemento **siguiente** al central, para buscar ala izquierda de CEN

A continuación, se muestra una implementación **ITERATIVA** en Java, reescríbelo en kotlin:

****

Existe otra solución de la búsqueda binaria, que se puede plantear de forma ligeramente diferente, tal que, se puede **sustituir la repetición (bucle) por llamadas recursivas**; es decir, se llama al método a sí mismo cada vez con un rango más pequeño, la sublista izquierda o derecha desde el elemento central.

A continuación, se muestra una implementación **recursiva** en Java, reescríbelo en kotlin:



## **Algoritmos de ordenación**

La ordenación es el proceso de **organización de un conjunto de elementos similares** de información, en orden creciente o decreciente. Los algoritmos de ordenación han sido profusamente analizados y son bien comprendidos.

En general tenemos dos clases de ordenación:

* **Ordenación Interna**: consiste en clasificar un array en la memoria principal.
* **Ordenación Externa**: consiste en clasificar ficheros sin utilizar la memoria principal, utilizando el almacenamiento secundario.

El tipo de ordenación que nos interesa en esta unidad es la **ordenación interna**. Estudiaremos algunos **algoritmos de ordenación interna** para listas almacenadas en memoria (arrays).

Este proceso es mucho más rápido que la ordenación de ficheros al estar todos los datos en el almacenamiento principal, y por tanto está exento de las operaciones de E/S.

Desde los comienzos de la informática, el problema de la ordenación ha atraído a bastantes investigadores debido, tal vez, a la **complejidad de resolverlo eficientemente** a pesar de su planteamiento simple y familiar. Por ejemplo, el algoritmo de la Burbuja o **BubbleSort**, fue analizado desde 1956.

Aunque muchos puedan considerarlo un problema resuelto, nuevos algoritmos de ordenación y variantes de algunos existentes, se continúan desarrollando.

Ejemplos de algoritmos modernos son:

* **Library Sort** u Ordenación de la biblioteca. Se publicó por primera vez en el 2006.

Información: <http://en.wikipedia.org/wiki/Library_sort>

* **Timsort**, un híbrido derivado del *Merge sort* e *Insertion sort*. Es el algoritmo de ordenación que utiliza la plataforma **Android, Python** y **Java SE 7** para la ordenación de arrays que no sean de tipos simples.

Información: <https://en.wikipedia.org/wiki/Timsort>

* **K-sort**, nueva versión del Quicksort (2007).

Información: <http://arxiv.org/abs/1107.3622>

### Algoritmo de la Burbuja (BubbleSort)

Es el método más conocido por su nombre pegadizo y su simplicidad. Sin embargo, para una ordenación de propósito general, es uno de los **peores algoritmos en cuanto a eficiencia**:

* Su ***complejidad*** es de **O(n2)**. Es decir, el tiempo crece **exponencialmente** al aumentar el nº de elementos.

Es una **ordenación por** **intercambio** que involucra repetidas comparaciones y, si es necesario, intercambios de elementos adyacentes. Los elementos son como *“burbujas”* en un tanque de agua donde cada burbuja va subiendo quedando clasificada en su nivel.

La filosofía de este método es ir comparando los elementos del array de 2 en 2 y si no están colocados correctamente intercambiarlos, así hasta que tengamos el array ordenado.

**Funcionamiento**

* Se compara la posición 1 y la 2 y si no están ordenadas las intercambio.
* Después la posición 2 y la 3
* Así sucesivamente hasta que comparo las últimas posiciones.



**Coloca un solo elemento**

Con el **primer recorrido** lograremos que quede colocado el **último elemento** del array.

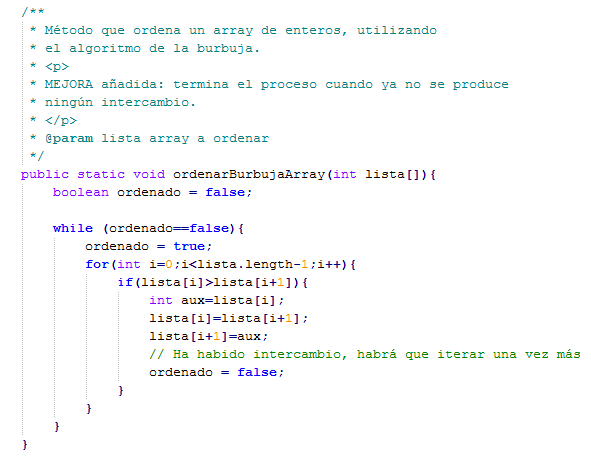
En el peor de los casos, en cada recorrido colocaremos un elemento, por lo que tendríamos que hacer **N-1 recorridos**.

* **Optimización**: En caso de que en un recorrido no se hagan cambios, es que el array está ordenado y se podrá finalizar.

**Ejemplo:** Para ver la forma en que funciona este algoritmo, supongamos que el array a ordenar es: **10**  **8** **2** **5**

| Inicial: | 10 | 8 | 2 | 5 |  |
| --- | --- | --- | --- | --- | --- |
| Después del Recorrido 1 | 8 | 2 | 5 | **10** | ***queda colocado el 10*** |
| Después del Recorrido 2 | 2 | 5 | **8** | **10** | ***queda colocado el 8*** |
| Después del Recorrido 3 | ***No ha habido ningún intercambio*** | | | | |
| **Finaliza** |  | | | | |

**Implementación** en **Java, reescríbelo en kotlin**:



### Algoritmo “Rápido” (QuickSort)

El ordenamiento rápido *(quicksort en inglés)* es un algoritmo basado en la técnica que sigue el principio de *“divide y vencerás”*, que permite ordenar n elementos en un tiempo mucho menor que el de la Burbuja porque dicho tiempo **NO crece exponencialmente.**

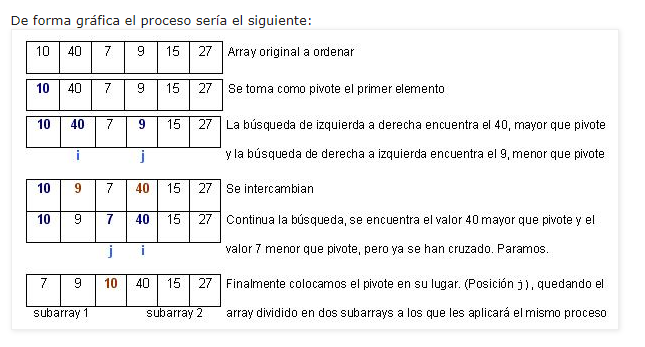
Este algoritmo de ordenación y sus variantes existentes, es el más rápido conocido. Fue desarrollado por C. Antony R. Hoare en 1960, pero existen variantes más modernas.

Aunque el algoritmo original es **recursivo**, existen implementaciones iterativas para mejorar su rendimiento (los algoritmos recursivos son en general más lentos que los iterativos por las múltiples llamadas al método).

**Funcionamiento**

El algoritmo fundamental es el siguiente:

* Elegir un elemento de la lista de elementos a ordenar, al que llamaremos **pivote**. La elección del pivote, afectará a la eficiencia del algoritmo.
* **Resituar los demás elementos** de la lista a cada lado del pivote, de manera que a un lado queden todos los menores que él, y al otro los mayores. En este momento, **el pivote ocupa exactamente el lugar que le corresponderá** en la lista ordenada.
* **La lista queda separada en dos sublistas**, una formada por los elementos a la izquierda del pivote, y otra por los elementos a su derecha.
* **Repetir este proceso de forma recursiva para cada sublista** mientras éstas contengan más de un elemento. Una vez terminado este proceso todos los elementos estarán ordenados.

****

**Implementación** en **Java (recursivo), reescríbelo en kotlin**:

