ESTRUCTURAS DE DATOS

La **estructura de datos en Kotlin** es el sistema mediante el cual se organizan un conjunto de datos bajo un mismo nombre, para que se puedan usar de manera eficiente.

Existen estructuras de datos **estáticas** y **dinámicas**:

- Una estructura de datos **ESTÁTICA** es aquella que el tamaño ocupado en la memoria se define antes de que el programa se ejecute y no puede ser modificado durante la ejecución del programa.
  Son estructuras estáticas: Arrays y Matrices.

- Las estructuras de datos **DINÁMICAS** son aquellas en las que el tamaño podrá modificarse durante la ejecución del programa; teóricamente no hay límites a su tamaño, salvo el que impone la memoria disponible en la computadora.
  Son estructuras dinámicas: Listas, Pilas, Colas, Árboles, Grafos

LOS ARRAYS

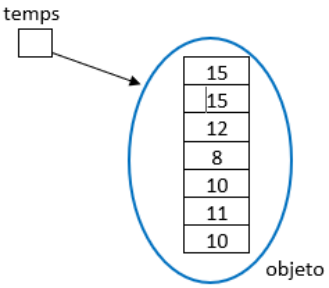
Arrays unidimensionales

Un array es una estructura de datos que permite almacenar un número indeterminado de variables del mismo tipo, que son referenciadas por el mismo identificador.

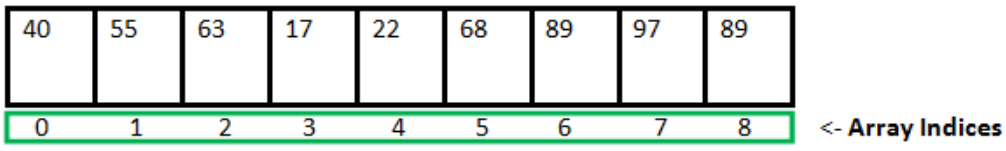
Características de un array:

- Se puede acceder a cada elemento mediante su **posición**
- La posición es un **número entero** que normalmente se le llama **índice**.
- Es importante tener en cuenta que el **primer elemento** de un array, tiene como índice el número cero.
- Dentro del array, cada valor ocupa una posición y se puede recuperar o modificar su valor combinando el identificador y dicha posición.
- Un array es una referencia a la estructura donde se almacenan sus valores (o sus referencias si es un array de objetos), por lo que, si se pasa como parámetro a un método, se pasará como **referencia**.

Ejemplo: Representación en memoria de un array de 7 elementos.



Como se puede observar en la siguiente figura, la primera posición del array es la posición 0. Como consecuencia, la última posición del array será igual a la longitud del mismo menos uno.



Array Length = 9
First Index = 0
Last Index = 8

Declaración y creación del array

Declarar un array es declarar una **referencia** al array que posteriormente se creará. El array se declara igual que una variable, indicando que es una variable o una constante y el tipo de dato a guardar.

```
var nombre: Array<tipo>
```

Si quieres crear y poblar arrays en Kotlin, utiliza la función **arrayOf** para crearlos, y las propiedades y métodos dentro de la clase **Array** para trabajar con los valores contenidos.

```
var miArray: Array<Int> = arrayOf(1, 2, 3, 4, 5)
```

Por ejemplo, para crear un array de enteros con 5 elementos.

```
var otroArray: Array<Int> = Array(5) { 0 }
```

Crea un array de 5 elementos inicializados en 0.

Recuerda que en Kotlin los arrays son objetos, pero el tipo de los elementos debe ser especificado entre los signos de menor < y mayor >.

Diferentes ejemplos de creación de Arrays:

Crear un array de cadenas (Strings) con elementos predefinidos:

```
val nombres: Array<String> = arrayOf("Juan", "María", "Carlos")
```

Crear un array de booleanos con valores predeterminados:

```
var booleanos: Array<Boolean> = Array(3) { false }
```

Crear un array de tamaño fijo con elementos inicializados en un valor específico:

```
val arrayInicializado: Array<Int> = Array(7) { 100 } // Crea un array de 7
```

Crear un array de números enteros con valores por defecto, que da como resultado este array: {"0", "1", "4", "9", "16"}

```
val numeros: Array<Int> = Array(5) { i -> (i * i) }
```

Kotlin tiene clases especializadas para representar arrays de tipos primitivos para evitar el coste de empaquetar y desempaquetar. Las funciones *booleanArrayOf, byteArrayOf, shortArrayOf, charArrayOf, intArrayOf, longArrayOf, floatArrayOf, y doubleArrayOf* crea un array de los tipos asociados (*BooleanArray, ByteArray, ShortArray*, etc.).

Crear un array de números flotantes utilizando la función floatArrayOf():

```
val flotantes: FloatArray = floatArrayOf(1.5f, 2.3f, 5.6f)
```

Crear un array de caracteres (chars) utilizando la función charArrayOf():

```
val caracteres: CharArray = charArrayOf('a', 'b', 'c', 'd')
```

Crear un array que contiene nulos. Es interesante que, aunque el array contiene sólo valores nulos, también tienes que elegir un tipo de datos particular para el array. Después de todo, es posible que no contenga valores nulos para siempre, y el compilador necesita saber qué tipo de referencia planeas agregarle.

```
val miArray: Array<String> = arrayOfNulls<String>(5)
```

Para asignar valor a cada uno de los espacios que se han reservado se puede acceder al array e indicar la posición para la que se quiere asignar

```
telefonos[0] = "976123456";
```

Para recorrer todos los elementos de un array se suele utilizar un bucle **for** junto con un índice que va desde 0 hasta el tamaño del array menos 1.

Por tanto, se puede recorrer el array, conociendo el número de elementos, para ello contamos con una propiedad que se llama **size**.

También es posible inicializar los valores de un array aprovechando esta estructura de bucle **for**, de forma que se puede asignar un valor a cada elemento del mismo, posición por posición.

```
// Declaración del array de Strings con tamaño 5
val telefonos = arrayOfNulls<String>(5)

// Inicialización de cada elemento del array con "..."
var i = 0
while (i < telefonos.size) {
   telefonos[i] = " . . . "
   i++
}
```

También podemos usar la propiedad llamada **indices**, que devuelve una lista con las posiciones válidas del array.

```
        val telefonos = arrayOfNulls<String>(5)

        // Inicialización de cada elemento del array con "..."
        for (i in telefonos.indices) {
           telefonos[i] = " . . . "
        }
```

Como hemos visto, normalmente se itera sobre un array utilizando el bucle **for-in** estándar, pero si desea acceder tanto a los valores como al índice, podemos usar la función **withIndex** de Array:

```
val strings = arrayOf("this", "is", "an", "array", "of", "strings")
for ((index, value) in strings.withIndex()) {
        println("En la posición: $index se encuentra el valor: $value")
}
```

Hay que tener en cuenta que se pueden crear arrays de tipos primitivos, de clases de kotlin, y también de cualquier clase de las que hayas programado en el proyecto.

```
val misMotos = arrayOfNulls<Moto>(10)
```

Ejemplo de creación de array de la clase Kotlin: String:

```
        // Declaro y creo array de 4 objetos String (cadenas)
        var ciudades = arrayOfNulls<String>(4)
        ciudades[0] = "París"
        ciudades[1] = ""
        ciudades[2] = "Madrid"
        ciudades[3] = ciudades[0]

        // Escribe el contenido del array
        for (i in 0 until ciudades.size) {
            println("${ciudades[i]}")
        }
        for (ciudad in ciudades) {
            println(ciudad)
        }

        // Escribirá: París Madrid París de dos formas distintas
```

Utilización del array

La utilización del array es muy sencilla. Cada elemento del mismo, una vez indicada la posición, se puede tratar como una variable del mismo tipo que el array, por lo que las operaciones que se pueden hacer con cada elemento son las mismas que se pueden hacer sobre variables individuales.

Ejemplo: Si tenemos un array de **int :**

```
var temps: Array<Int> = arrayOf( 0,2,4,6,8)
```

Cada elemento del array se puede utilizar exactamente igual que cualquier otra variable, es decir, se le puede asignar un valor o se puede usar dentro de una expresión.

En nuestro ejemplo, cada elemento del array es un entero, temps[0] es un entero y se podrá utilizar su valor de la misma manera que si fuera una variable entera:

```
var suma = temps[0] + temps[1]
```

Ejemplo: Tenemos un array de una clase Moto que hemos programado en el proyecto. Podemos invocar al método de cada elemento del array, de esa clase Moto:

```
var misMotos:Array<Motos> = arrayOf(m1,m2,m3,m4,m5)

// recorrer el array . . .
for (i in 0 until misMotos.size) {
    misMotos[i].pasarRevision()
}
```

También se puede declarar una variable para tomar la referencia del elemento del array. Esto a veces es más cómodo para acceder a los métodos del objeto.

```
var misMotos:Array<Motos> = arrayOf(m1,m2,m3,m4,m5)
// Poblar el array . . .
. . .
for (i in 0 until misMotos.size) {
    var moto:Motos = misMotos[i]
    if (!moto.estaRevisada()) {
        moto.pasarRevision()
    }
}
```

Los arrays se pueden pasar como parámetro a un método, y como se ha dicho, un array es una referencia a la estructura donde se almacenan sus valores (o sus referencias, si es un array de objetos), por lo que, si se pasa como parámetro a un método, se pasará como referencia, por lo que es posible modificar los datos del array desde el método que se pasa como parámetro. La excepción está en los objetos de la clase String, que como sabemos, estos objetos no pueden modificarse.

```
fun main() {
    val numeros = IntArray(5)

    for (i in numeros.indices) {
        numeros[i] = 10
    }
    incrementar(numeros)
    println(numeros[3]) // Imprimirá 11
}

fun incrementar(valores: IntArray) {
    for (i in valores.indices) {
        valores[i]++
    }
}
```

También debemos de tener en cuenta, que **los arrays NO pueden crecer,** por lo que, si necesitamos añadir un nuevo elemento a un array, tendremos que crear un nuevo array.

Vemos a continuación, un ejemplo de cómo podríamos hacerlo. Tenemos un método que toma un **array por parámetro** y **un nuevo elemento**. Veremos cómo, crea y **devuelve** un nuevo array

```
/**
* Crea un nuevo array con un elemento más y guarda en él,
* el valor pasado por parámetro.
*
*/
fun agregarElementoArray(lista: IntArray, valor:   Int): IntArray {
    // Creo un nuevo array con 1 elemento más
    val nuevo = IntArray(lista.size + 1)

    // Copio todos los elementos al nuevo array
    for (i in lista.indices) {
        nuevo[i] = lista[i]
    }

    // Agrego el nuevo valor en la última posición
    nuevo[nuevo.size - 1] = valor

    return nuevo
}
```

Una llamada válida al método anterior:

```
var lista = intArrayOf(3, 4, 5)

// Añade un elemento más a la lista con un 10
lista = agregarElementoArray(lista, 10)
```

Al guardar la referencia devuelta al nuevo array, en la **misma referencia** que se usaba para el array original, el array original deja de ser referenciado y será eliminado de memoria cuando pase el Garbage Collector (**gc**).

Copia de un array

Si necesitamos copiar un array, se hace igual que con dos variables del mismo tipo:

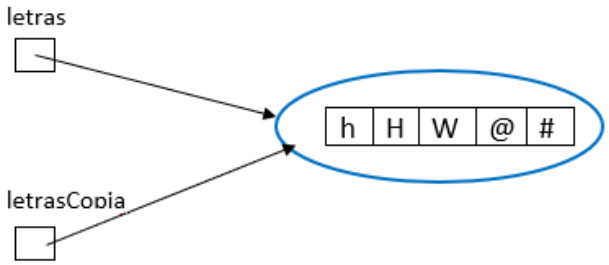
Ejemplo: Tenemos los siguientes arrays:

```
val letras = charArrayOf('h', 'H', 'W', '@', '#')
var letrasCopia: CharArray
```

La siguiente sentencia no copia este array en uno nuevo llamado letrasCopia:

```
letrasCopia = letras // Asigna el array letras a letrasCopia
```

El resultado de la instrucción anterior es que tanto letras como letrasCopia, referencian al mismo array (**se copió la referencia al array, no se hizo una copia de los datos**).

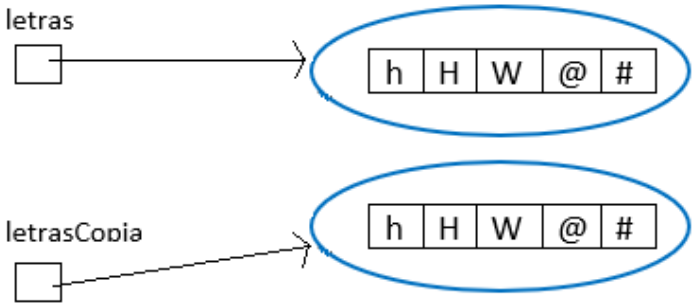


Esta situación hace que cualquier cambio que se haga en letrasCopia, cambie también en letras, puesto que **solo hay uno**.

La forma correcta de copiar un array de tipos primitivos, es la siguiente:

```
// Creo un nuevo array con el mismo tamaño
val letras = charArrayOf('h', 'H', 'W', '@', '#')
var letrasCopia = CharArray(letras.size)
// Creo un nuevo array con el mismo tamaño
// Recorro el array copiando cada elemento
for (i in letras.indices) {
        letrasCopia[i] = letras[i]
}
```

NOTA: Kotlin también tiene el método letras.**copyOf**()



NOTA: Si se quisiera copiar un array de objetos, se deberían "clonar" esos objetos. Esto es algo que se verá más adelante.

Comparación de arrays

Un array es igual que otro, si todos sus elementos son iguales. Cuidado si se utiliza el operador **==** para comparar arrays, lo que se **comparan son las referencias** y **NO el contenido** de los arrays.

Ejemplo

```
val letras = charArrayOf('h', 'H', 'W', '@', '#')
val letras2 = charArrayOf('h', 'H', 'W', '@', '#')
var letras3: CharArray

letras3 = letras2

if (letras == letras2){  // NO se cumple
    ...
}

if (letras3 == letras2){  // SI se cumple
    ...
}
```

Para comparar dos arrays hay que hacer un recorrido de ambos arrays, e ir comparando dato a dato, aunque si el tamaño no coincide, no es necesario realizar el recorrido puesto que ya NO son iguales.

```
val letras = charArrayOf('h', 'H', 'W', '@', '#')
val letras2 = charArrayOf('h', 'H', 'W', '@', '#')
var iguales = true

if (letras.size != letras2.size) {
    iguales = false
}

var i = 0
while (i < letras.size && iguales) {
    if (letras2[i] != letras[i]) {
        iguales = false
    }
    i++
}

if (iguales) {
  println("Los arrays letras y letras2 tienen el mismo contenido.")
} else {
 println("Los arrays letras y letras2 tienen contenido diferente.")
}
```

NOTA: Kotlin también tiene el método letras.**contentEquals**(letras2).

Reasignación de arrays

Un array en Kotlin puede ser inicializado en varias ocasiones, al igual que ocurre con todas las variables en Kotlin:

```
var notas = IntArray(5)

notas = IntArray(8)
```

Primero, notas tiene 5 elementos y después tiene 8. Pero, ¿QUÉ implicaciones tiene el código anterior?

Es fácil ver que el segundo **crea un nuevo array**, o lo que es lo mismo, un nuevo objeto y su referencia se guarda en notas.

Por tanto, el anterior objeto queda **sin referencia**, entonces ya no puede utilizarse, pero sigue ocupando espacio.

En Kotlin un **recolector de basura** se encarga cada cierto tiempo de liberar la memoria que ha quedado sin referencia, librando al programador de esta tarea.

Clase Array

En Kotlin, se puede trabajar con algunas funciones y extensiones útiles para operaciones con arrays dentro del paquete **kotlin.collections**. Aunque no hay una clase específica como java.util.Arrays en Java, muchas de las operaciones básicas que se realizan con arrays pueden llevarse a cabo utilizando funciones de extensión o funciones de alto nivel en Kotlin.

Algunas funciones útiles para trabajar con arrays en Kotlin se encuentran en el paquete **kotlin.collections** y son extensiones de las clases de **Array** estándar. Algunas de estas funciones son:

- **forEach**: Permite realizar operaciones en cada elemento del array.
- **filter**: Filtra los elementos del array basándose en un predicado (devuelve una lista)
- **sortedArray**: Ordena los elementos del array.
- **sum**: Devuelve un Int cuyo valor es la suma de todos los elementos del array
- **count**: Devuelve un Int, con el número de los elementos que cumplen cierta condición.
- **map**: Transforma cada elemento del array en otro valor según una función proporcionada (devuelve una lista)
- **reduce:** Aplica una operación a cada elemento de una colección de izquierda a derecha y al final retorna el valor acumulado.

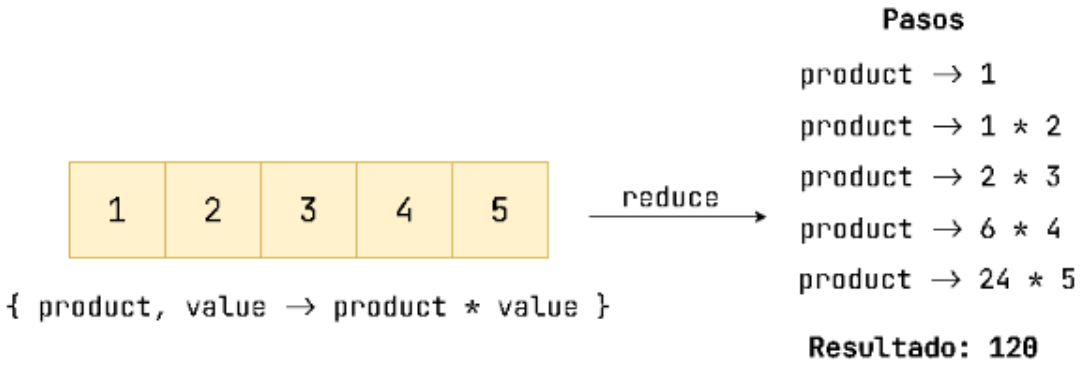
Por ejemplo:

```
val numbers = arrayOf(-3, 10, 4, 1, 5, 9, 2, -6)

numbers.forEach { println(it) } // Imprime cada elemento del array

val multiplesOfFive = numbers.filter { x -> x % 5 == 0 }
val multiplesOfFive2 = numbers.filter { it % 5 == 0 }

val sortedArray = numbers.sortedArray() //copia ordenada del array

val sum = numbers.sum() // Suma todos los elementos del array

val num = numbers.count { it < 0 } // 2

val difficultyLevels = arrayOf("fácil", "intermedio", "difícil")
val diffLevelsUpper = difficultyLevels.map{ it.uppercase() }

val oneToFive = arrayOf(1, 2, 3, 4, 5)
val sequenceProduct = oneToFive.reduce { product, value -> product
* value }
println("El producto es $sequenceProduct") //120
```

Nota: Puedes usar la referencia **it** si no tienes problema con la legibilidad en el contexto de la función lambda

Con respecto a la función **reduce**, toma como argumento una operación, cuyos parámetros son el valor acumulado (product) y el elemento actual(value).

En la siguiente imagen se muestra como funciona la función **reduce()** sobre el rango entero 1..5. La idea es multiplicar cada número con su consecutivo:



El primer valor de *product* es el elemento de la posición inicial del rango. En el segundo paso *value* será el elemento consecutivo 2 y así sucesivamente, hasta que *producto* acumule el valor 120 para el número 5.

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:

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

```kotlin
/**
 * Método que busca un elemento en un array, usando búsqueda secuencial
 * Devuelve la posición donde se encuentra el valor.
 * Devuelve -1 si no encuentra el valor
 * En el caso de existir elementos repetidos,
 * devolverá la primera posición donde se encuentre.
 */
fun buscarSecArray(lista: Array<Int>, valor: Int): Int {
    var posicion:Int = -1
    var booEncontrado = false

    var pos=0
    while((pos<lista.size) && !booEncontrado){
        if(lista[pos] == valor){
            booEncontrado = true
            posicion = pos
        }
        pos++
    }

    return posicion
}
```

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.

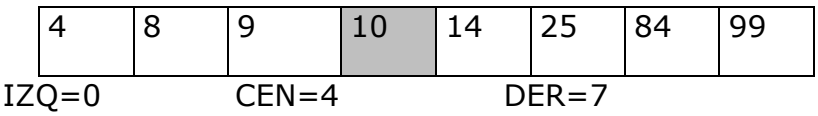
Se trata de **dividir el espacio de búsqueda en sucesivas mitades** hasta encontrar el elemento buscado, o hasta que ya no pueda hacer más mitades.

- 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.



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

Repetición

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 Kotlin:

```kotlin
/** Método que busca un elemento en un array, ...*/
fun buscarBinIterArray(lista: Array<Int>, valor: Int): Int {
    var posicion: Int = -1
    var booEncontrado = false

    var izq = 0
    var der = lista.size -1
    var cen: Int

    while((izq <= der) && !booEncontrado){
        cen = (izq + der)/2
        if (lista[cen] == valor){
            booEncontrado = true
            posicion = cen
        }
        else if (lista[cen] < valor){
            izq = cen + 1
        }
        else {
            der = cen -1
        }
    }
    return posicion
}
```

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 Kotlin

```kotlin
/**
 * Método que busca un elemento en un array,
 * usando la búsqueda binaria, implementada de forma recursiva
 * Es requisito indispensable que el array esté ordenado
 * Devuelve la posición donde se encuentra el valor.
 * Devuelve -1 si no encuentra el valor
 */
fun buscarBinRecArray(lista: Array<Int>, valor: Int, izq: Int, der:Int): Int {
    var posicion: Int = -1
    var cen:Int

    if (izq <= der){
        cen = (izq + der)/2
        if (lista[cen] == valor){
            posicion = cen
        }
        else if (lista[cen] < valor){
            posicion=buscarBinRecArray(lista, valor,  izq: cen+1, der)
        }
        else {
            posicion=buscarBinRecArray(lista, valor, izq,  der: cen-1)
        }
    }
    return posicion
}
```

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 profundamente 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 **$O(n^2)$**. 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

Un Recorrido

- 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 **Kotlin**:

```kotlin
/** Método que ordena un array de enteros ...*/
fun ordenarBurbujaArray(lista: Array<Int>) {
    var booOrdenado = false

    while (!booOrdenado) {
        //pensamos que este es el último recorrido
        booOrdenado = true

        //recorremos desde la primera posicion a la ultima
        for (i in 0 ≤ until < lista.size-1) {
            if (lista[i] > lista[i + 1]) {
                //intercambiar datos entre posicion i/i+1
                var aux = lista[i]
                lista[i] = lista[i + 1]
                lista[i + 1] = aux

                //ha habido intercambio, continuar la ordenacion
                booOrdenado = false
            }
        }
    }
}
```

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.

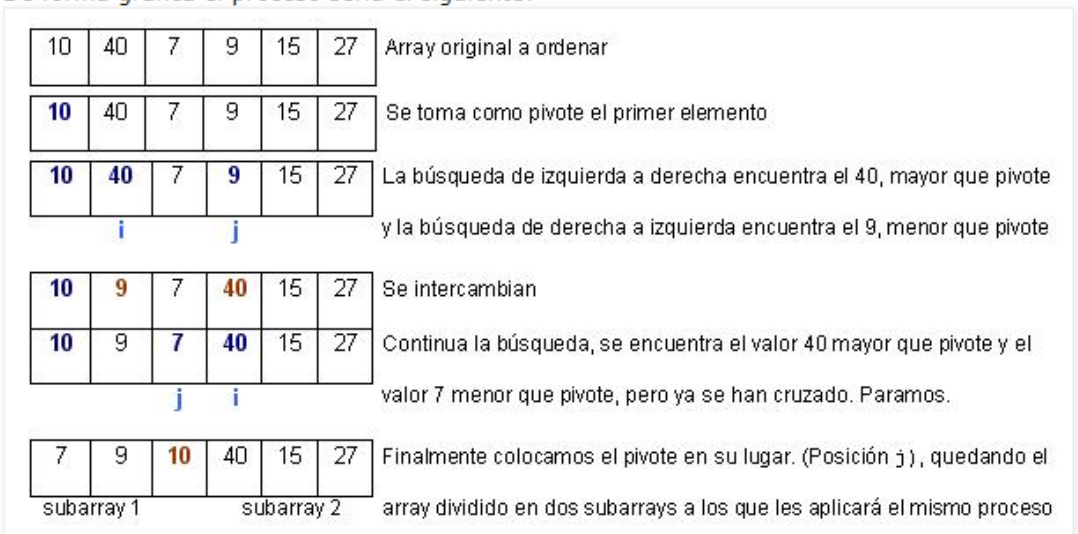
Aunque el algoritmo original es **recursivo**, existen implementaciones iterativas para mejorar su rendimiento (los algoritmos recursivos son en general más lentos que los iterativos por las múltiples llamadas al método).

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.

De forma gráfica el proceso sería el siguiente:



Implementación en **Kotlin (recursivo)**:

```
fun ordenarQuicksortArray(lista: Array<Int>, izq:Int, der:Int) {
    var i = izq
    var j = der
    var pivote = lista[(izq+der)/2] //se toma como pivote el centro

    do {
        //busca la posicion del primer valor > que el pivote
        while (lista[i] < pivote) {
            i++
        }
        //busca la posicion del primer valor < que el pivote
        while (lista[j] > pivote) {
            j--
        }

        if (i <= j) {
            //intercambiamos valores
            var aux = lista[i]
            lista[i] = lista[j]
            lista[j] = aux
            //avanzamos posiciones
            i++
            j--
        }
    }while (i<=j)

    if (izq < j){
        ordenarQuicksortArray(lista, izq, j)
    }
    if (i < der){
        ordenarQuicksortArray(lista, i, der)
    }
}
```

Arrays bidimensionales o MATRICES

Un array bidimensional utiliza dos índices para localizar cada elemento. Podemos ver este tipo de dato como un array que, a su vez, contiene otros arrays. También se puede ver como una cuadrícula en la que los datos quedan distribuidos en filas y columnas, de ahí que los arrays bidimensionales, también sean llamados **matrices**.

Al poderse interpretar como un array de dos dimensiones, **la primera dimensión hará referencia a las filas, y la segunda a las columnas.**

Se declaran e inicializan de la misma manera que los arrays unidimensionales, pero esta vez hay que tratar con dos dimensiones.

Para introducir u obtener cualquier elemento en la estructura, hay que indicar la posición de la fila y de la columna**:**

```
// Acceder al elemento en la fila 2 y columna 3
val valor = telefonos[1][2]

// Modificar un elemento en la fila 4 y columna 5
telefonos[3][4] = "555-1234"
```

Para recorrer una matriz, será necesario anidar dos bucles for para acceder a las dos dimensiones del array. En el siguiente ejemplo tendremos un array de 5 filas, cada una con 6 columnas.

```
val telefonos = Array(5){ Array(6) { "" } }

for (i in telefonos.indices) {
    // Recorrer filas
    for (j in telefonos[i].indices) {
        // Recorrer columnas de cada fila
        telefonos[i][j] = " . . . "
        println("Posicion[$i][$j] : ${telefonos[i][j]}")
    }
}
```

También se puede utilizar la función **size**, teniendo en cuenta que si llamamos a la función teléfonos.size, devolverá el número de filas de la matriz, en este caso devolverá 5. Y si llamamos a telefonos[i].size devolverá el número de columnas de la fila, en este caso 6.

Otra forma de recorrer una matriz, sin saber la posición, es usando for each:

```
for (fila in tablero) {
    //por cada fila
    for (celda in fila) {
        //por cada columna de la fila
        print(celda)
        print(" | ")
    }
    println()
}
```

También es posible crear un array bidimensional asignando directamente los valores que queremos que contenga el mismo. En ese caso se declara un array con la capacidad justa para almacenar todos los valores especificados, crearemos un Array<Array<String>>.

```
val telefonos = arrayOf(

    arrayOf("976123456", "91123456", "986789876"),

    arrayOf("926213040", "926547814", "926503021"),

    // ... otras filas

)
```

Y, por supuesto, también podemos crear arrays bidimensionales de objetos:

```
val misMotos = Array(10) { Array<Moto?>(2) { null } }
```

Los arrays bidimensionales se utilizan con frecuencia para situar objetos en un plano, como por ejemplo las piezas de ajedrez en un tablero, o un personaje de videojuego en un laberinto.

```
var m = Array(5){Array(6){7}}
```

Adyacentes a una casilla dada.

Adyacentes en un vector.

z	z+1		i-1	i	i+1			j-1	j

Se define como adyacente, a las casillas contiguas que se tiene antes y después de una casilla dada.

Tenemos que dos tipos de casillas, la **centrales** tendrán adyacentes por delante y detrás de la casilla dada, como en el ejemplo de la casilla i y las casillas de los **laterales** que no tienen todos los adyacentes, ejemplo casilla z y j.

El error que se puede producir en el cálculo de adyacentes se produce en la posición inicial y en la final.

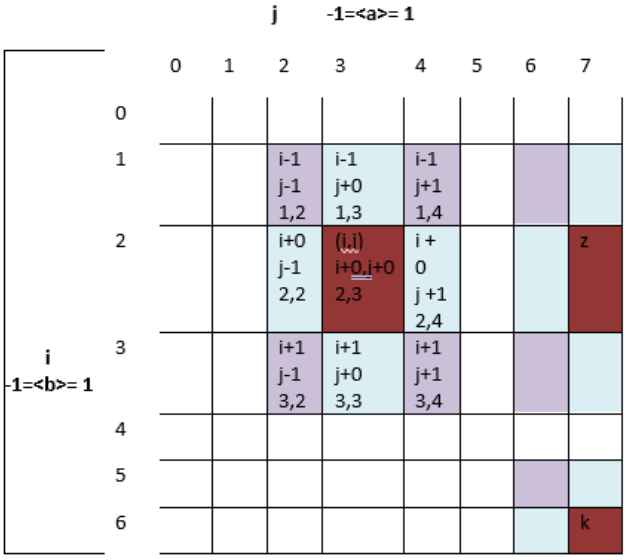
```
fun devolverAdyacentes(a:Array<Int>,posicion:Int) {
    if ((posicion >= 0) && (posicion < a.size)) {
        if (posicion == 0) {
            println("${a[posicion + 1]}")
        } else if (posicion == (a.size - 1)) {
            println("${a[posicion - 1]}")
        } else {
            println("${a[posicion - 1]} - ${a[posicion + 1]}")
        }
    }
}
```

Adyacentes de una matriz.

Igual que en los vectores, en las matrices tenemos adyacentes, la diferencia es que tenemos más.

Los podemos diferenciar en adyacentes de vértices (los violetas) y adyacentes de lado (los azules).

También tendremos casillas centrales con todos los adyacentes y casillas laterales donde no se tienen todos los adyacentes.




```kotlin
fun adyacentesMatriz(m: Array<Array<Int>>, x:Int, y:Int){

    var i = -1
    var j= -1

    while (i<=1){
        j=-1
        while (j<=1) {
            var adjX = x + i
            var adjY = y + j
            if ((adjX >= 0) && (adjX < m.size)
                && (adjY >= 0) && (adjY < m.size)){
                if ((i != 0) || (j != 0)) {
                    print(" Posicion[$adjX][$adjY] : ${m[adjX][adjY]}")
                }
            }
            j++
        }
        println()
        i++
    }
}
```