

# Práctica Evaluable 2

Análisis y Diseño de Algoritmos

Nombre: Emilio Gómez Esteban

Grupo: 2ºD (Grado Matemáticas + Ingeniería Informática)

- **MÉTODOS *ordRapidaRec* Y *partir* DE LA CLASE *OrdenacionRapida*:**

```
public static <T extends Comparable<? super T>> void ordRapidaRec(T v[], int
izq, int der) {
    if(izq<der) {
        int p = partir (v, v[izq], izq, der);
        ordRapidaRec(v,izq,p-1);
        ordRapidaRec(v,p+1,der);
    }
}

public static <T extends Comparable<? super T>> int partir(T v[], T pivote,
int izq, int der) {
    int i = izq+1;
    int j = der;

    do {
        while((i<=j) && (v[i].compareTo(pivote)<=0)) {
            i++;
        }

        while ((i<=j) && (v[j].compareTo(pivote)>0)) {
            j--;
        }

        if(i<j) {
            Ordenacion.intercambiar(v, i, j);
        }
    } while (i<j);

    Ordenacion.intercambiar(v,izq,j);

    return j;
}
```

En este caso, se intenta implementar un programa que ordene un array de elementos, los cuales son de un tipo que admite comparaciones entre elementos. Para ello, trataremos de dividir el problema en diferentes partes, es decir, se usará la técnica de “Divide y Vencerás” en la que se echa mano además de la recursividad.

Ahora bien, el método *partir* se encarga de dividir el array  $v[izq, \dots, der]$ , de forma que todos los elementos de  $v[izq, \dots, p-1]$  son menores o iguales que  $v[p]$  y todos los

elementos de  $v[p+1, \dots, \text{der}]$  son mayores o iguales que  $v[p]$  (donde  $v[\text{izq}]$  es pasado como el parámetro pivote inicialmente, es decir, el valor que delimitará los dos subarrays). La forma de hacerlo, es recorriendo el array por la izquierda y por la derecha mientras el elemento sea menor o mayor que el pivote respectivamente. Para realizar este recorrido se utilizará el método *compareTo*, ya que los elementos del array serán del tipo genérico *T*, los cuales implementan el interfaz *Comparable*. Cuando una de las condiciones mencionadas falle, se intercambiarán los valores del array donde se hayan parado los contadores de posición (el método *intercambiar* viene dado por la clase *Ordenacion*).

Al acabar el bucle *do\_while* (es decir, cuando ya se han cruzado los contadores), *j* es la posición del elemento con valor menor o igual que el pivote que está más a la derecha. Por ello, intercambiamos  $v[\text{izq}]$ , que tiene el pivote, con  $a[j]$ , para que a su izquierda los elementos sean menores o iguales y a su derecha los elementos sean mayores o iguales. Esta *j* será la *p*.

A continuación, será usado en el método *ordRapidaRec*. Este parte el array en dos subarrays que tendrán la forma  $v[\text{izq}, \dots, p-1]$  y  $v[p+1, \dots, \text{der}]$ , donde *p* vendrá dado por el método *partir*. Llamando de forma recursiva se ordenan los arrays nombrados llamando a *ordRapidaRec*.

- **MÉTODOS ordenar Y barajar DE LA CLASE OrdenacionRapidaBarajada:**

```
// Implementación de QuickSort con reordenación aleatoria inicial (para  
comparar tiempos experimentalmente)
```

```
public static <T extends Comparable<? super T>> void ordenar(T v[]) {  
    barajar(v);  
    ordRapidaRec(v, 0, v.length-1);  
}  
  
    // reordena aleatoriamente los datos de un vector  
private static <T> void barajar(T v[]) {  
  
    for(int ind=0; ind<v.length; ind++) {  
        int k = aleat.nextInt(v.length);  
        //nextInt(int n) Devuelve un pseudoaleatorio de tipo int  
        //comprendido entre cero (incluido) y el valor especificado (excluido).  
  
        Ordenacion.intercambiar(v, ind, k);  
    }  
}
```

Para este método, lo que se intenta hacer es usar la misma forma de ordenación rápida por recurrencia barajando antes los elementos del array y cambiando sus posiciones aleatoriamente.

Para ello, se tienen estos dos métodos. En la superclase *Ordenacion* se crea una variable de la clase *Random* llamada *aleat*. Esta puede implementar el método *nextInt(int)*, el cual devuelve un valor de tipo *int*, elegido al azar, comprendido entre cero, que se incluye, y el valor *int* pasado como parámetro, no incluido. De esta forma se crea un índice comprendido en el conjunto de índices del array que tenemos, de forma

aleatoria. A continuación, lo usaremos para llamar al método intercambiar que cambiará la posición de los elementos del array de la inicial a la posición dada aleatoriamente.

Por último, la función ordenar de esta clase, barajará el vector inicialmente y después lo ordenará de la forma *ordRapidaRec*.

- **MÉTODOS *kesimo* Y *kesimoRec* DE LA CLASE *BuscaElem*:**

```
public static <T extends Comparable<? super T>> T kesimo(T v[], int k) {
    return kesimoRec(v, 0, v.length-1, k);
}

public static <T extends Comparable<? super T>> T kesimoRec(T v[], int izq,
int der, int k) {
    if(izq < der) {
        int p = OrdenacionRapida.partir(v, v[izq], izq, der);

        if(p == k) {
            return v[k];
        } else if(k < p) {
            return kesimoRec(v, izq, p-1, k);
        } else {
            return kesimoRec(v, p+1, der, k);
        }
    }
    return v[k];
}
```

En este caso, se pide aprovechar el método *partir* usado en el primer apartado, para encontrar el elemento k-ésimo del array si este estuviera ordenado, pero sin ordenarlo. El método *partir* permite dividir el problema. Lo que se hará será ir encerrando la posición k-ésima para hallar el valor que debería estar en esta posición si el array estuviera ordenado.

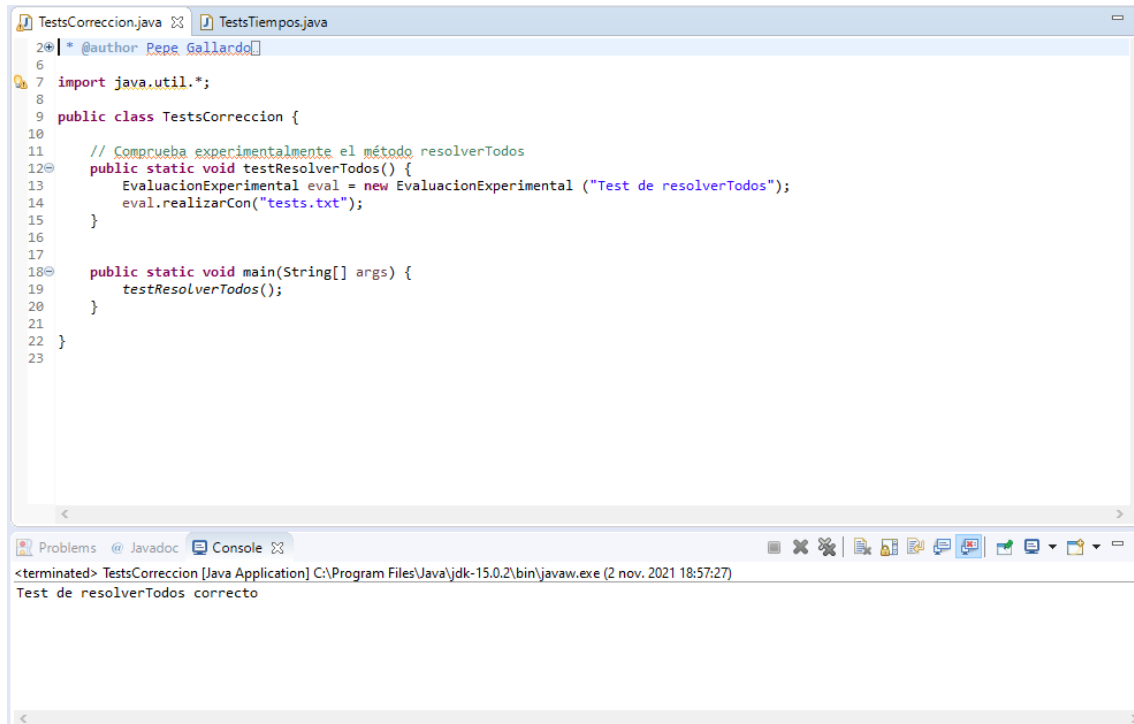
Como caso base está aquel en el que k es el pivote de partir, por tanto, se resolverá el problema y ya se habrá encontrado el elemento. Se irá partiendo el vector como hemos explicado en el apartado primero. No se ordenará el vector explícitamente, pero a un lado y a otro del pivote, los elementos serán mayores o menores que este. Si se reduce el razonamiento hasta tener vectores de dos elementos, este se ordenará teóricamente. Si k está a la derecha del pivote entonces el nuevo ínfimo será pivote+1, y en caso contrario, el supremo será p. Así sucesivamente hasta que k esté “encerrado” entre el ínfimo y el supremo, es decir, que todos coincidan. Se devuelve el valor que habría en esa posición del array ordenado.

- **MÉTODO *ordenar* DE LA CLASE *OrdenacionJava*:**

```
public static <T extends Comparable<? super T>> void ordenar(T v[]) {
    Arrays.sort(v);
}
```

Se elige el método *sort* de la clase *Arrays*, para el apartado en que nos pide implementar el método de *ordenar* en la clase *OrdenacionJava*, usando alguna de las estructuras de datos predefinidas en el paquete *java.util* de la biblioteca estándar de Java. En las gráficas que se ven más adelante se puede comprobar que la eficiencia de esta implementación es bastante mayor a la de los demás métodos que realizan la ordenación (gráfica de color amarillo) en cualquiera de los tres casos.

Tomando la clase *TestsCorreccion*, se puede realizar una comprobación del algoritmo usado. En este caso, la respuesta conseguida es la siguiente:



```
TestsCorreccion.java TestsTiempos.java
24 | * @author Pepe Gallardo
6
7 import java.util.*;
8
9 public class TestsCorreccion {
10
11     // Comprueba experimentalmente el método resolverTodos
12     public static void testResolverTodos() {
13         EvaluacionExperimental eval = new EvaluacionExperimental ("Test de resolverTodos");
14         eval.realizarCon("tests.txt");
15     }
16
17
18     public static void main(String[] args) {
19         testResolverTodos();
20     }
21
22 }
23
```

Problems @ Javadoc Console

<terminated> TestsCorreccion [Java Application] C:\Program Files\Java\jdk-15.0.2\bin\javaw.exe (2 nov. 2021 18:57:27)

Test de resolverTodos correcto

Además, con las clases *Temporizador* y *Gráfica* suministradas, se pueden obtener gráficas para comparar experimentalmente (para distintos tamaños de vector) el rendimiento de su implementación. Para ello, se ejecuta el método *main* de la clase “*TestsTiempos*”, cuya salida es:

