Práctica 2. QuickSort

Autor: Marcos Hidalgo Baños **Fecha:** 14 de octubre del 2022

Descripción del problema.

El objetivo de esta práctica es implementar un total de tres algoritmos de ordenación de vectores de valores en el lenguaje de programación JAVA. Aunque se nos proporcionan numerosos archivos auxiliares, sólamente deberemos modificar las siguientes clases:

- **OrdenacionRapida.java** pretende ordenar el array de manera ascendente aplicando técnicas de Divide y Vencerás para reducir la complejidad temporal del algoritmo.
- **OrdenacionRapidaBarajada.java** es una variante de la clase anterior que realiza una mezcla aleatoria en la posición de los valores antes de llamar al método implementado.
- **BuscaElem.java** realiza una ordenación del array de entrada para facilitar la búsqueda del elemento que debería estar en la posición k, reutilizando el método partir.

Código implementado y decisiones de diseño.

Siguiendo las indicaciones de los profesores de la asignatura, en la clase OrdenacionRapida podemos realizar la implementación de los métodos suponiendo que el pivote se encuentra en el propio array, pero **no sabemos la posición del mismo**.

Por este motivo, el método ordRapidaRec además de emplear Divide y Vencerás para dividir el esfuerzo computacional, utiliza el método partir como medio principal para la ordenación. El intercambio entre pares de valores se hará o bien si durante el recorrido de la búsqueda se encuentran **valores desordenados**, o bien si al final de la ordenación el pivote no lo está.

En lo relativo a las otras dos clases JAVA que conforman la práctica, la estructura y toma de decisiones se realiza de manera similar a la recién explicada puesto que se reutilizan métodos .

OrdenacionRapida.java

ordRapidaRec solo efectuará algún cambio si el array está por ordenar (o está siendo ordenado) puesto que el parámetro izq es inferior a der. Si no fuera el caso, la ordenación habría acabado.

```java
// Debe ordenar ascendentemente los primeros @n elementos del vector @v con
// una implementación recursiva del método de ordenación rápida.
public static <T extends Comparable<? super T>> void ordRapidaRec(T v[], int izq, int der) {
    // A completar por el alumno
    if (izq < der) {
        int p = partir(v, v[izq], izq, der);
        ordRapidaRec(v, izq,  der: p-1);
        ordRapidaRec(v,  izq: p+1, der);
    }
}
```

En el método partir, moveremos el puntero inf hasta encontrar un valor mayor o igual que el pivote, y el puntero sup hasta encontrar un valor menor que el pivote para después intercambiar.

```java
public static <T extends Comparable<? super T>> int partir(T v[], T pivote, int izq, int der) {
    int inf = izq;
    int sup = der;

    while (inf < sup) {
        while (v[inf].compareTo(pivote) <= 0 && inf < v.length-1) {
            inf++;
        }
        while (0 < v[sup].compareTo(pivote) && 0 < sup) {
            sup--;
        }
        if (inf < sup) {
            intercambiar(v, inf, sup); // metodo de Ordenacion.java
        }
    }

    if (0 < pivote.compareTo(v[sup])) {
        intercambiar(v, izq, sup); // metodo de Ordenacion.java
    }

    return sup;
}
```

OrdenacionRapidaBarajada.java

```java
public class OrdenacionRapidaBarajada extends OrdenacionRapida {

    // Implementación de QuickSort con reordenación aleatoria inicial (para comparar tiempos experimentalmente)
    public static <T extends Comparable<? super T>> void ordenar(T v[]) {
        // A completar por el alumno
        barajar(v);
        ordRapidaRec(v, izq: 0, der: v.length-1); // metodo de la clase padre
    }

    // reordena aleatoriamente los datos de un vector
    private static <T> void barajar(T v[]) {
        // A completar or el alumno
        for (int k = v.length-1; k > 0 ; k--) {
            intercambiar(v, k, aleat.nextInt( i: k+1));
        }
    }
}
```

BuscaElem.java

Se basa en el cálculo del pivote de OrdenacionRapida.java para encontrar el elemento k-esimo.

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

public static <T extends Comparable<? super T>> T kesimoRec(T c[], int izq, int der, int k) {
    // A IMPLEMENTAR POR EL ALUMNO
    if (izq < der) {
        int p = OrdenacionRapida.partir(c, c[izq], izq, der);
        if (k == p) {
            return c[k];
        } else if (k < p) {
            return kesimoRec(c, izq,  der: p-1, k);
        } else {
            return kesimoRec(c,  izq: p+1, der, k);
        }
    }
    return c[k];
}
```

Análisis de la complejidad del programa.

Debido a que en esta práctica hemos realizado modificaciones a diversos archivos, deberemos realizar un análisis individual a cada una de las funciones descritas anteriormente.

OrdenacionRapida.java

Mientras que el método **ordRapidaRec** en sí no aporta grandes problemas en aspectos de complejidad temporal por la simplicidad de su contenido, son las constantes llamadas al método **partir** desde el mismo las que hacen que aumente la complejidad temporal.

El peor de los casos será aquel que realice el mayor número de llamadas a dicho método, siendo este en el que la partición resulte en dos mitades de tamaños extremadamente asimétricos, donde su complejidad temporal será del orden cuadrático **$O(n^2)$**.

En el mejor de los casos, la partición será exactamente por la mitad, reduciendo un orden de complejidad el programa un grado, hasta ser **O(n)**. En un caso normal (sin que ocurra nada extraño) el algoritmo se comportará con orden cuasi-logarítmico.

OrdenacionRapidaBarajada.java

Gracias a que esta clase basa su funcionamiento en gran medida en la clase padre, la complejidad temporal será muy similar. Sin embargo, como añadimos un método para barajar los valores del array, la expresión será de la manera: **n * O(ordRapidaRec)** puesto que tendremos que recorrer una vez completa el array de entrada para barajar los valores.

BuscaElem.java

El método kesimoRec también se basa en el método partir, además de poseer cierto grado de recurrencia, por lo que su comportamiento será muy parecido al de OrdenacionRapida.java.

Sin embargo, gracias a que ahora estamos trabajando sobre un array ordenado, podemos elegir qué lado del pivote buscar. Esto reducirá en gran medida los costes computacionales, **reduciendo las invocaciones recurrentes a la mitad** (en el peor de los casos, pudiendo retornar un valor si coincide con el pivote elegido).