Analisis y Diseño de Algoritmos — Práctica 1

En esta práctica, el alumno estudiará y desarrollará diversos algoritmos de ordenación en *Java* para ordenar ascendentemente un vector de valores.

Se suministra un conjunto de fuentes para la práctica, así como un fichero de pruebas sobre los cuales deberá trabajar el alumno (`qs.zip`). Se incluye la implementación de diversos métodos `ordenar` mediante diversas subclases herederas de la clase `Ordenacion`. (El proyecto proporciona también otras clases y métodos auxiliares cuyo funcionamiento interno no es necesario entender para realizar la práctica).

Para la realización de la práctica podrá utilizarse Eclipse (en cuyo caso habrá de crearse un proyecto con todos los ficheros que se proporcionan) o bien la línea de comandos para la compilación y ejecución.

a) En la clase `OrdenacionRapida` implementar el método `ordRapidaRec` que tome como parámetros un vector `v` y dos enteros, `izq` y `der`, y ordene los elementos de dicho vector desde `izq` a `der` (los restantes elementos deberán quedar como en el vector original). Tal como se ha definido la cabecera del método, será posible ordenar vectores con cualquier tipo de elementos, siempre que dichos elementos implementen el interfaz `Comparable`; es decir, siempre que definan el método `compareTo`. En la cabecera, `T` es un tipo genérico que denota el tipo de los elementos del vector. Para poder comparar elementos de tipo `T` el alumno tendrá que usar el método `compareTo`, que se invoca sobre un objeto y que toma como parámetro otro objeto, devolviendo 0 si ambos son iguales, menor que 0 si el objeto sobre el que se invoca es menor al objeto parámetro, o mayor que 0 en caso contrario.

Para ordenar un vector completo, habría que invocar el método `ordRapidaRec` del siguiente modo:

```
public static void main (String args[]) {

        Integer vEnt[] = {3,7,6,5,2,9,1,1,4};
        ordenar(vEnt);
        System.out.println(vectorAString(vEnt));

        Character vCar[] = {'d','c','v','b'};
        ordenar(vCar);
        System.out.println(vectorAString(vCar));
}
```

ya que tanto `Integer` como `Character` tienen predefinidos el método `compareTo`. En el código, el método `length` devuelve el número total de elementos del vector y el método `vectorAString`, que convierte un vector en su representación como cadena de caracteres, está implementado en el proyecto (en la superclase `Ordenacion`).

b) Implementar una variante del método de ordenación rápida, en la clase `OrdenacionRapidaBarajada`, de manera que previamente a la ordenación se mezclen

aleatoriamente los elementos del vector. Comprobar ahora el funcionamiento de esta variante del método.

c) Aprovechando el método partir implementado en el primer apartado, completar en la clase `BuscaElem` la implementación eficiente del método `kesimoRec` que tome como parámetros un vector `v` (desordenado) y tres enteros, y encuentre el elemento que debería estar en la posición k (si este estuviera ordenado, pero sin ordenar el vector) de dicho vector desde `izq` a `der`.

d) 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. Comparar la eficiencia de esta implementación con los demás métodos de ordenación.

Pruebas

Con objeto de que el alumno compruebe experimentalmente el funcionamiento de la implementación del algoritmo de ordenación rápida, se suministra la clase `TestsCorrección`, que comprueba que la implementación es correcta para un conjunto de instancias solucionadas y almacenadas en el fichero "`tests.txt`". Para una implementación correcta, la salida por pantalla tras ejecutar este método debe ser:

```
java TestsCorreccion
Test de resolverTodos correcto
```

Comparar gráficamente el rendimiento de sus algoritmos con otros algoritmos de ordenación suministrados en el esqueleto (método de la burbuja, inserción, selección, y ordenación por mezcla). Para ello se debe ejecutar el programa

```
java TestsTiempos
```

Entrega

Una vez finalizada la practica, entregar los tres primeros apartados para su corrección de forma automática mediante el programa *Siette*. Para ello, seleccionar la actividad correspondiente en el Campus Virtual y enviar, una vez modificados, los ficheros:

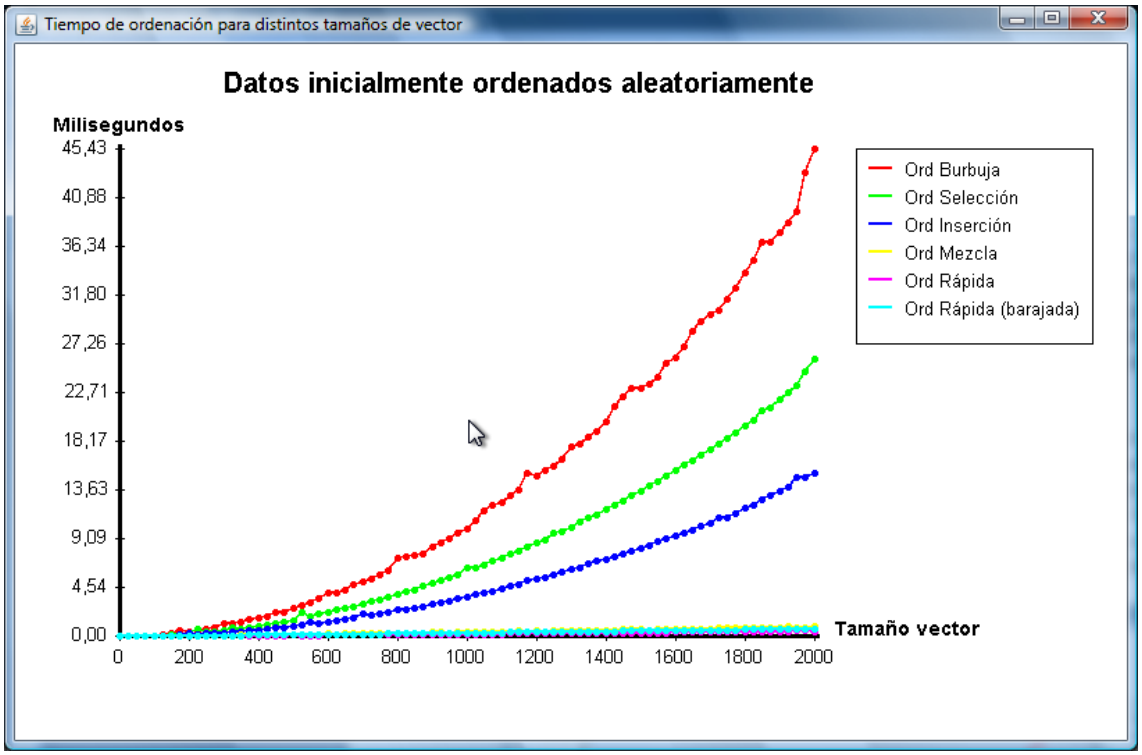
```
OrdenacionRapida.java
OrdenacionRapidaBarajada.java
BuscaElem.java
```

Atención: Es muy importante que vuestra solución, para pasar correctamente los tests de SIETTE NO incorporen paquetes.

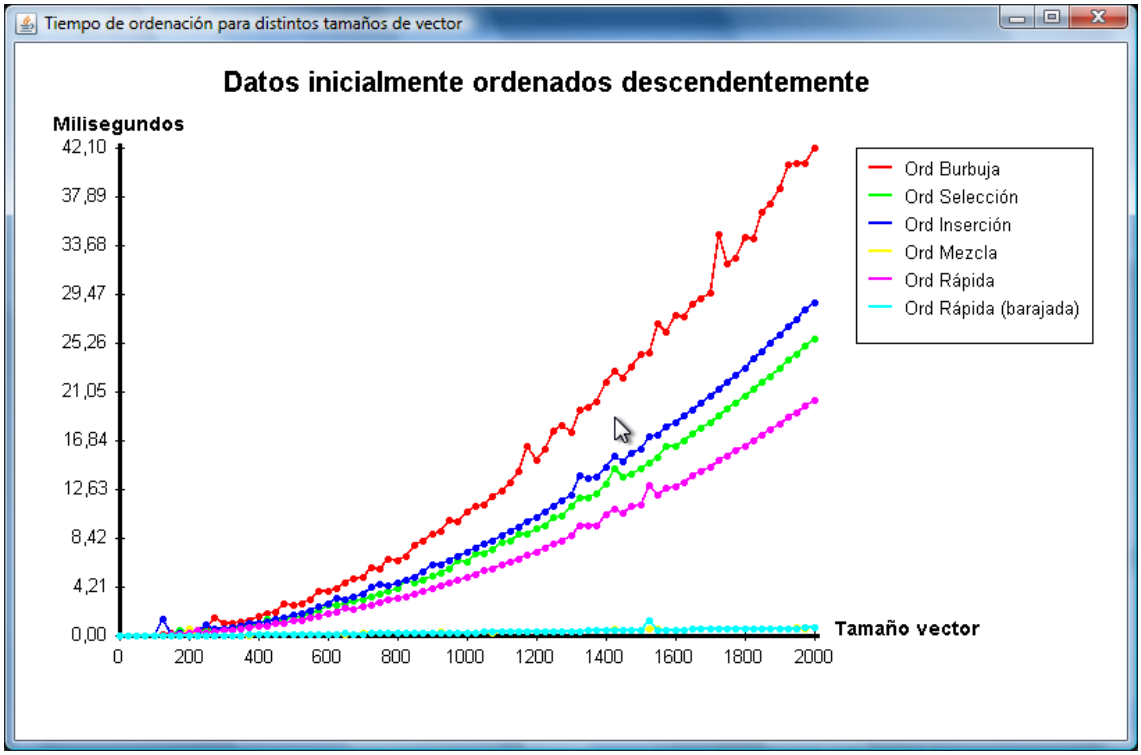
Comparación gráfica del rendimiento del algoritmo de Quicksort.

Con la clases `Temporizador` y `Gráfica` suministradas, el alumno podrá obtener gráficas para comparar experimentalmente (para distintos tamaños de vector) el rendimiento de su implementación. Para ello, tendrá que ejecutar el método `main` de la clase `TestsTiempos`. El alumno deberá obtener gráficas similares a las siguientes, que se obtuvieron en un Pentium Core2 6300.

Obsérvese cómo el tiempo de ejecución de ordenación rápida crece bastante menos rápido que el de ordenación por inserción y burbuja para datos inicialmente distribuidos de forma aleatoria. Esto es consistente con el análisis teórico de la complejidad, ya que la complejidad de ordenación rápida es $O(n \lg n)$ mientras que la de los otros métodos es $O(n^2)$. Obsérvese también que el método de la burbuja es el peor de los comparados en este caso, ya que realiza $O(n^2)$ comparaciones y $O(n^2)$ intercambios.



En la siguiente gráfica puede observarse que si los datos se encuentran ordenados inicialmente en orden inverso, el rendimiento de la versión simple del algoritmo de ordenación rápida se degrada (ya que éste es uno de los peores casos para este algoritmo y la complejidad se eleva a $O(n^2)$, aunque la constante oculta sigue siendo menor que la de los algoritmos de ordenación por inserción, selección y burbuja). Sin embargo, una versión mejorada de la ordenación rápida (que baraja de modo aleatorio los datos antes de comenzar la ordenación) no es sensible a este caso y mantiene su buen rendimiento junto con la ordenación por mezcla.



En la última gráfica, vemos que si los datos están inicialmente casi ordenados todos los algoritmos salvo ordenación por selección y la versión simple de la ordenación rápida funcionan bastante bien.