Memoria Práctica 2: Análisis y Diseño de Algoritmos

En esta práctica se realiza la implementación de cuatro clases: *OrdenacionRapida*, *OrdenacionRapidaBarajada*, *BuscaElem* y *OrdenacionJava*. Las dos primeras y la última heredan de la clase ordenación e implementan distintos métodos para ordenar un array de elementos de cualquier tipo, siempre que permitan una comparación entre ellos.

Veamos la primera de ellas, OrdenacionRapida:

```
public class OrdenacionRapida extends Ordenacion {

     public static <T extends Comparable<? super T>> void
ordenar(T v[]) {
          ordRapidaRec(v, 0, v.length-1);
     }

     public static <T extends Comparable<? super T>> void
ordRapidaRec(T v[], int izq, int der) {
          if(izq<der) {
           int pivote = partir(v, v[izq], izq, der);

           ordRapidaRec(v, izq, pivote-1);
           ordRapidaRec(v, pivote+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) {
                     intercambiar(v,i,j);
               }
         }while(i<j);
         intercambiar(v,izq,j);
         return j;
   }

     public static void main (String args[]) {

          Integer vEnt[] = {3,8,6,5,2,9,1,1,4};
          ordenar(vEnt);
          System.out.println(vectorAString(vEnt));
```

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

      }

}
```

Veamos cada parte del código por separado:

```
public static <T extends Comparable<? super T>> void ordenar(T
v[]) {
          ordRapidaRec(v, 0, v.length-1);
      }
```

Esta clase implementa tres métodos. El método ordenar se encarga de pasarle al método ordRapidaRec los argumentos necesarios para ordenar un array por completo.

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

            ordRapidaRec(v, izq, pivote-1);
            ordRapidaRec(v, pivote+1, der);
          }
      }
```

Por otro lado, el método ordRapidaRec, toma como argumentos un array v y dos enteros izq y der y si izq<der, siguiendo la teoría de QuickSort actúa de forma recursiva llamando a la función *partir*.

```
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) {
                      intercambiar(v,i,j);
                }
          }while(i<j);
```

```
            intercambiar(v,izq,j);
            return j;
      }
```

El método partir toma como base el funcionamiento de QuickSort, visto en clase, que mediante el uso de un bucle pasa los elementos menores que el pivote a la izquierda de este y los elementos mayores a la derecha.

Tratemos a continuación la clase OrdenacionRapidaBarajada:

```
public class OrdenacionRapidaBarajada extends OrdenacionRapida {

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

     private static <T> void barajar(T v[]) {
      for(int i=0; i<v.length; i++) {
            int n= aleat.nextInt(v.length-1);
            intercambiar(v,n,i);
      }
     }
}
```

Esta clase hereda de la clase OrdenacionRapida, y se diferencia de esta en que implementa un método adicional barajar, que mezcla de forma aleatoria los elementos del array antes de invocar el método Ordenar de la clase de la que hereda.

Veamos también la clase BuscaElem:

```
import java.util.Scanner;

public final class 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) {
            int indice= OrdenacionRapida.partir(v, v[izq], izq,
der);
            if(indice==k) {
                  return v[k];
            }else if(k<indice) {
                  return kesimoRec(v,izq, indice-1, k);
```

```
            }else {
                  return kesimoRec(v,indice+1,der,k);
            }
      }


      public static void main(String[] args) {
            int maxvector;
            int i,k;

            Scanner sc = new Scanner(System.in);
            System.out.print("Introduce el numero de posiciones
del vector: ");
            maxvector=sc.nextInt();
            Integer v[]=new Integer[maxvector];

            System.out.print("Introduce "+maxvector+" enteros
separados por espacios: ");
            for (i=0;i<maxvector;i++)
                  v[i]=sc.nextInt();
            System.out.print("Introduce la posicion k deseada (de
1-"+maxvector+"): ");k=sc.nextInt();
            Integer elem=kesimo(v,k-1);
            System.out.print("El elemento "+k+"-esimo es:
"+elem);

            sc.close();
      }

}
```

Este primer método llama al método kesimoRec y le pasa como argumentos el vector, el índice 0 y el último, así como el índice k cuyo elemento del vector queremos encontrar.

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

El método kesimo aplica por primera vez el método partir y guarda en un entero índice el que devuelve el método partir. En caso de que coincida con el índice k cuyo elemento queremos encontrar ya hemos terminado ya que esa posición en concreto ya está en su posición final. En caso de que no coincidan y sea menor que índice se aplica de forma recursiva con el trozo izquierdo y en caso de que sea mayor con el trozo izquierdo, hasta encontrar el índice k.

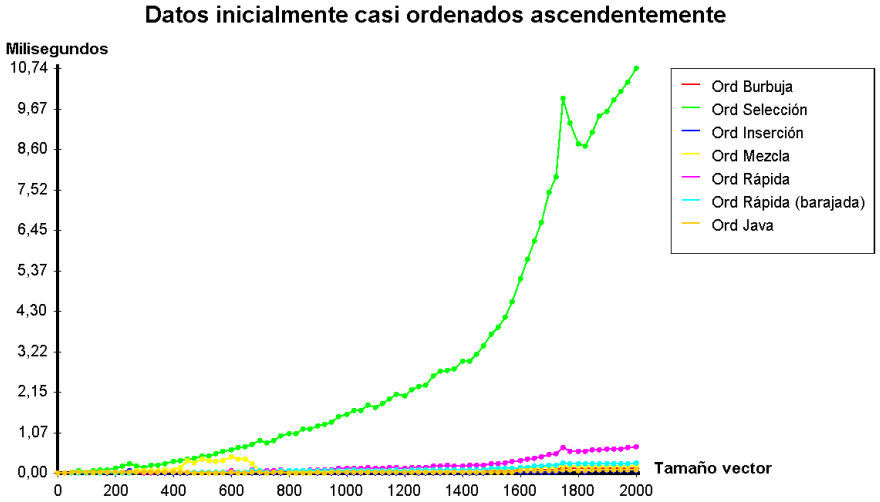
Por último estudiemos la clase OrdenacionJava:

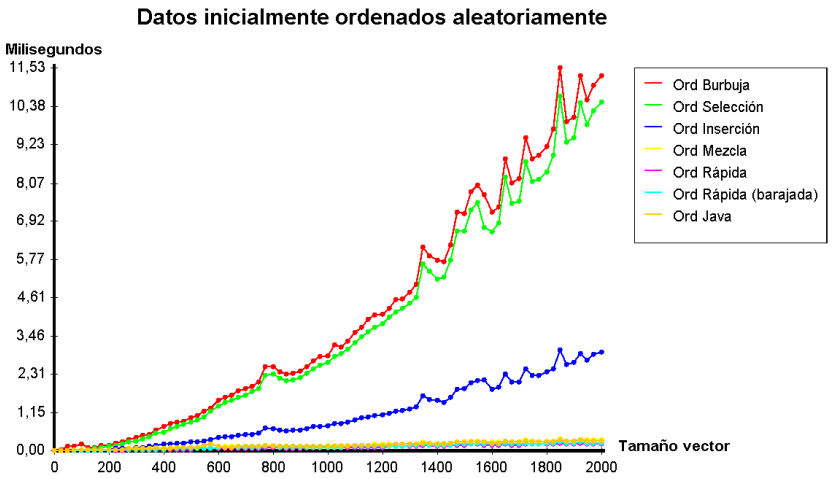
```
import java.util.ArrayList;
```

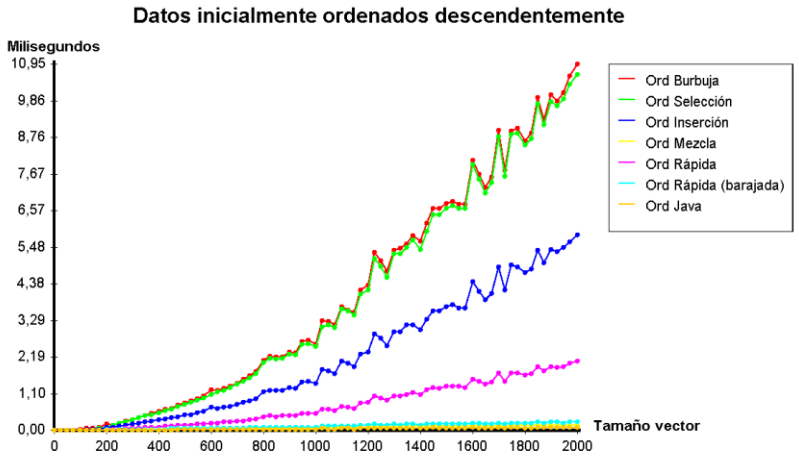
```
import java.util.Collections;
public class OrdenacionJava extends Ordenacion {


   public static <T extends Comparable<? super T>> void
ordenar(T v[]) {
        ArrayList<T> lista= new ArrayList<>();
        for(int i=0; i<v.length; ++i) {
             lista.add(v[i]);
        }
        Collections.sort(lista);
        v= lista.toArray(v);
   }


     public static void main (String args[]) {

          // Un vector de enteros
          Integer vEnt[] = {3,8,6,5,2,9,1,1,4};
          ordenar(vEnt);
          System.out.println(vectorAString(vEnt));

          // Un vector de caracteres
          Character vCar[] = {'d','c','v','b'};
          ordenar(vCar);
          System.out.println(vectorAString(vCar));

     }

}
```

Esta clase implementa el método *Ordenar*, que mediante el uso de un ArrayList auxiliar lista, al que le pasamos todos los elementos del array a ordenar. Posteriormente utilizamos el método *sort* de la biblioteca java.util.Collections que devuelve el ArrayList de menor a mayor. Por último devolvemos el array con los elementos ordenados.

Estudiemos la complejidad de OrdenacionJava mediante las pruebas suminstradas:







Como podemos ver en las gráficas, independientemente de cómo se encuentren los datos a ordenar, la complejidad de OrdenacionJava es la que mejor se comporta en todos los casos manteniendo una complejidad casi constante.