

Práctica 2

D&V: Estudio experimental de la eficiencia de *Merge Sort* y *Quick Sort*

Departamento de Sistemas Informáticos y Computación
Universitat Politècnica de València

Índice

1. Introducción: los métodos de ordenación <i>Merge Sort</i> y <i>Quick Sort</i>	1
1.1. Actividad #1: Organización de paquetes y clases	2
2. Implementación eficiente del algoritmo <i>Merge Sort</i>	2
2.1. Actividad #2: Método eficiente de ordenación por <i>Merge Sort</i> : <code>mergeSort2</code>	3
2.2. Actividad #3: Validación de la nueva versión de <i>Merge Sort</i>	3
3. Comparación de los tres métodos DyV de la clase <code>Ordenacion</code>	3
3.1. Actividad #4: Estudio del análisis experimental de la eficiencia de <code>mergeSort2</code>	3
3.2. Actividad #5: Ajuste y representación gráfica de los resultados	4
3.3. Actividad #6: Estudio del crecimiento del coste de la comparación individual	4
4. Otras actividades	4

1. Introducción: los métodos de ordenación *Merge Sort* y *Quick Sort*

A la hora de elegir un algoritmo para la ordenación *in situ* de los datos de un array se deben de tener en cuenta, básicamente, tres aspectos:

1. Su eficiencia expresada en función de la talla del problema, lo que conduce a elegir algoritmos de ordenación directa ($O(n^2)$), como Inserción Directa, cuando el número de datos a ordenar es pequeño y algoritmos rápidos ($O(n \log n)$), como *Merge Sort* o *Quick Sort*, cuando el número de datos a ordenar es muy grande.
2. El número y coste efectivo de las comparaciones y movimientos de datos (intercambios o copias) que realiza. Así, dado que *Merge Sort* realiza un número de comparaciones bastante menor que *Quick Sort* pero un número de movimientos bastante mayor, . . .
 - a) será preferible usar *Merge Sort* cuando el coste de comparar los datos sea notorio y, como sucede en Java, el de moverlos resulte irrelevante por no afectar a los datos sino únicamente a sus referencias (*Pointer sorting*);
 - b) será preferible usar *Quick Sort* cuando el coste de mover los datos sea tanto o más relevante que el de compararlos porque, como sucede en C++, dicho movimiento sí afecta directamente a los datos.
3. La estabilidad (o preservación del orden inicial de los datos idénticos), característica que solo resulta irrelevante cuando la ordenación se realiza siguiendo un único criterio, como sucede con los datos de tipo primitivo. Dado que *Merge Sort* es estable y *Quick Sort* no, *Merge Sort* será preferible a la hora de ordenar objetos comparables según diversos criterios.

Este análisis permite entender la implementación ofrecida en el paquete `java.util` para los métodos de ordenación de datos de tipo primitivo (por ejemplo, `static void Arrays.sort(int[] a)`) y la de objetos *Comparable* (por ejemplo, `static void Arrays.sort(Object[] a)`): mientras que los primeros implementan *Quick Sort*, los segundos implementan *Merge Sort* con el fin de garantizar la estabilidad y favorecer las situaciones en las que el coste efectivo de las comparaciones sea importante.

Un ejemplo de objetos cuya comparación es mucho más costosa que su movimiento es el caso de las cadenas de ADN. Los humanos tenemos unos 3200 millones de pares de bases de ADN, que se representan por la letra inicial de las cuatro bases nitrogenadas (Tiamina, Citosina, Adenina y Guanina) esto representa si lo representásemos en formato DIN A4 en letra Times New Roman 10p, más de 600000 páginas de unas 5330

letras cada una aproximadamente. De estas páginas dos humanos cualesquiera diferirían en un 0.1 %, es decir, que unas 600 páginas tendrían alguna letra distinta. Como puede verse el intercambio de referencias en este caso tiene un coste infinitamente menor que la comparación.

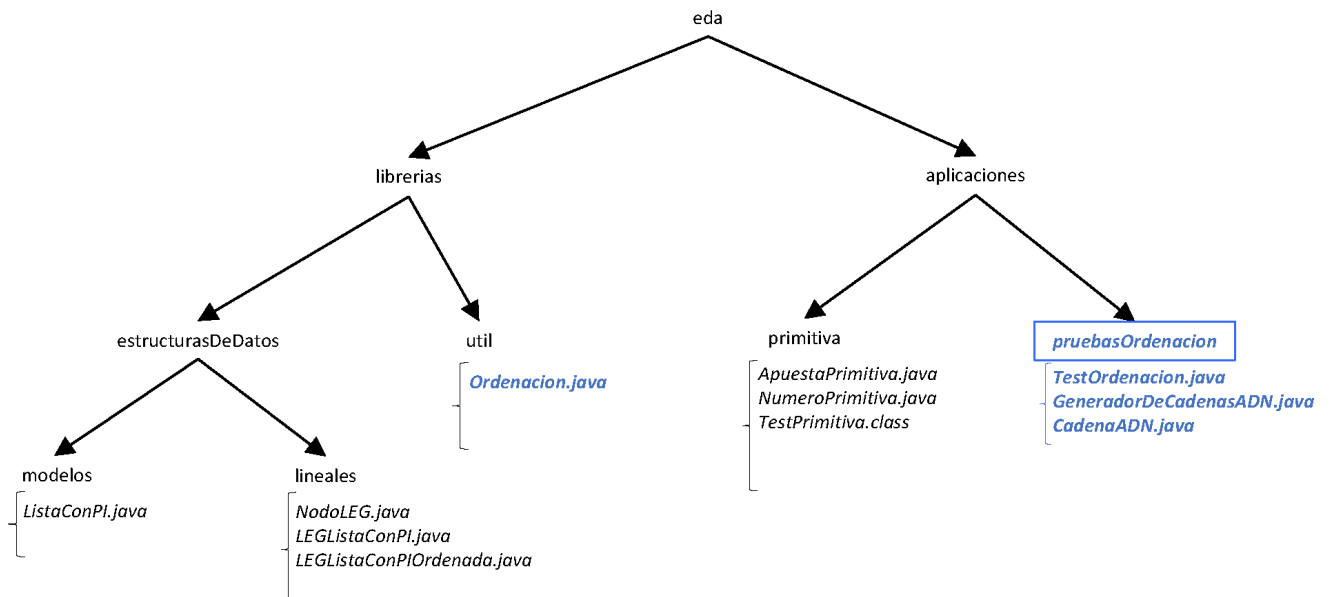
En esta práctica se implementará una versión eficiente de *Merge Sort* y se efectuará la comparación con la de *Quick Sort* realizada en clase de teoría, analizando la influencia que sobre su coste temporal puede tener el incremento del coste de la comparación de dos elementos. Para ello, y en concreto, se utilizarán arrays de dos clases que implementan la interfaz **Comparable**: **Integer** (del estándar de Java) y **CadenaADN** ya implementada. Como se recordará, los valores de tipo **Integer** se comparan entre sí en tiempo constante, mientras que para los de tipo **CadenaADN** contienen un array que debe compararse componente a componente, por lo que el tiempo de la comparación individual depende del número de componentes iguales iniciales que tengan los objetos a comparar; así, inicializando adecuadamente los objetos **CadenaADN** a ordenar, se podrá comprobar cómo cambia el tiempo de ejecución de los dos algoritmos en estudio cuando crece el tiempo de ejecución de cada comparación individual.

1.1. Actividad #1: Organización de paquetes y clases

El alumno debe crear los subpaquetes **util** en **librerias** y **pruebasOrdenacion** en **aplicaciones**, en los que trabajará en esta práctica. Para la realización de las distintas actividades se usaran las tres clases siguientes, disponibles en *PoliformaT*:

- **Ordenacion**, que se ubicará en el subpaquete **util**: contiene distintos métodos estáticos para ordenar arrays de datos de tipo genérico y **Comparable**, además del método auxiliar **sonIguales** que permite comprobar si los arrays que se le pasan como parámetros coinciden elemento a elemento.
- **TestOrdenacion**, que se ubicará en el subpaquete **pruebasOrdenacion**: contiene los métodos necesarios para realizar el análisis empírico de la eficiencia de los métodos de ordenación de la clase **Ordenacion**, valga la redundancia. Entre otros, los de creación de arrays de tallas distintas sobre los que se ejecutan los métodos a analizar y el de medición/tabulación de sus tiempos promedio de ejecución sobre dichos arrays.
- **GeneradorDeCadenasADN**, que se ubicará también en el subpaquete **pruebasOrdenacion**: es un generador random de objetos de tipo **CadenaADN** con las primeras **n** bases nitrogenadas (caracteres) iguales.
- **CadenaADN**, que se ubicará también en el subpaquete **pruebasOrdenacion**: implementa a **Comparable** y contiene las bases nitrogenadas de la cadena de ADN almacenadas en un array de caracteres.

Se recomienda por tanto que, antes de llevar a cabo las actividades, se compruebe que la estructura de paquetes y ficheros del proyecto *BlueJ eda* sea la que muestra la siguiente figura:



2. Implementación eficiente del algoritmo *Merge Sort*

La versión del algoritmo *Merge Sort* que se proporciona en la clase **Ordenacion**, la presentada en teoría, es mejorable si en la operación de fusión o mezcla (método **merge1**) se evita copiar dos veces cada elemento del

array a fusionar: los elementos fusionados se escriben en un array auxiliar **aux** (dos primeros bucles de **merge1**) y luego, cuando dicho array se ha generado, se copian de nuevo al array a ordenar (en el último bucle de **merge1**).

2.1. Actividad #2: Método eficiente de ordenación por *Merge Sort*: **mergeSort2**

En esta actividad se pide añadir a la clase **Ordenacion** una nueva versión de *Merge Sort* que evita esta doble copia y, por tanto, es más eficiente que la actual. Para ello, los pasos a dar son:

- I. Modificar el método de mezcla **merge1** para que devuelva como resultado el array auxiliar **aux**, con lo que tan solo se copia cada elemento una vez. Por ello, el perfil del método resultante será:

```
private static <T extends Comparable<T>> T[] merge2(T[] v1, T[] v2)
```

- II. Modificar el método **mergeSort1(T[], int, int)** para que, en lugar de modificar el array original, devuelva el array resultante de la ordenación y, obviamente, para que pueda usar **merge2(T[], T[])**. El perfil del método resultante será:

```
private static <T extends Comparable<T>> T[] mergeSort2(T[] v, int i, int f)
```

Además, en el diseño de este método se deben definir dos casos base de la recursión, para evitar en lo posible la generación excesiva de arrays de un solo elemento que se produce en **merge1**: uno para arrays de talla 1 y otro para arrays de talla 2.

- III. Modificar el método **mergeSort1(T[])** para que use **mergeSort2(T[], int, int)** y, al mismo tiempo, mantenga el perfil típico de un método de ordenación *in situ*; es importante destacar que **mergeSort2(T[], int, int)** devuelve un array ordenado y el tipo del resultado de un método de ordenación *in situ* es **void**, no **T[]**. El perfil del método resultante será:

```
public static <T extends Comparable<T>> void mergeSort2(T[] v)
```

2.2. Actividad #3: Validación de la nueva versión de *Merge Sort*

Antes que nada, es necesario verificar que la nueva versión de *Merge Sort* implementada, **mergeSort2(T[])**, ordena correctamente; para ello, lo más sencillo es comprobar si su resultado, ante diversas entradas, es el mismo que el conseguido por cualquier otro método que se sepa que es correcto.

En esta actividad se pide completar el diseño del método **comprobar** de la clase **TestOrdenacion** para que, haciendo uso de los métodos **sonIguales** y **quickSort** de la clase **Ordenacion**, compruebe si **mergeSort2(T[])** es, en efecto, un método correcto.

3. Comparación de los tres métodos DyV de la clase **Ordenacion**

Para poder comparar los tres métodos DyV de la clase **Ordenacion**, primero es necesario realizar el análisis experimental de su eficiencia. El método **temporizar** de la clase **TestOrdenacion** ya realiza dicho análisis para los métodos **quickSort** y **mergeSort1**; específicamente, como se puede observar en su código, las condiciones bajo las que **temporizar** realiza las medidas del tiempo de ejecución de ambos métodos son las siguientes:

- Los arrays a ordenar contienen valores de tipo **Integer** generados aleatoriamente, como resultado de invocar al método **crearAleatorioInteger**.
- Las medidas de tiempo de ejecución se realizan para arrays de tallas (crecientes), desde 10.000 a 100.000 elementos con pasos de 10.000 en 10.000.
- Para garantizar resultados significativos, para cada talla se promedian los resultados de unas 100 ejecuciones, ordenando en cada una de ellas un array diferente.
- El reloj utilizado para medir los tiempos de ejecución es **nanoTime()**, el método estático de la clase **java.lang.System** que devuelve el valor **long** actual del temporizador más preciso empleado en el sistema en nanosegundos (aunque la resolución real suele ser menor).

Observando el método **temporizar**, puede observarse que, también tabula los tiempos medios de ejecución de **quickSort(T[])** y **mergeSort1(T[])**.

3.1. Actividad #4: Estudio del análisis experimental de la eficiencia de **mergeSort2**

Añadir al método **temporizar** las líneas de código correspondientes a la temporización de **mergeSort2(T[])**.

Tras compilar y ejecutar **TestOrdenacion**, se podrán extraer las conclusiones correspondientes sobre cuál de los tres métodos analizados es el más rápido ordenando números enteros.

3.2. Actividad #5: Ajuste y representación gráfica de los resultados

Se ha de utilizar `gnuplot` para ajustar (mediante el comando `fit`) y representar gráficamente los resultados y sus ajustes (mediante el comando `plot`). Si es necesario, se puede consultar el documento “Resumen de `gnuplot`” disponible en *PoliformaT*, que contiene los comandos más habituales de `gnuplot` así como un ejemplo de una sesión con su uso.

3.3. Actividad #6: Estudio del crecimiento del coste de la comparación individual

La clase `GeneradorDeCadenasADN` permite generar objetos de tipo `CadenaADN` con sus primeras `n` bases nitrogenadas iguales. Por ejemplo, si se quieren generar dos `CadenaADN` con sus 30 primeras bases iguales con ayuda de esta clase, las instrucciones a emplear son:

```
// Construir un objeto GeneradorDeCadenasADN con un prefijo de 30 bases iguales:
GeneradorDeCadenasADN g = new GeneradorDeCadenasADN(30);
CadenaADN ejem1 = g.generar();
CadenaADN ejem2 = g.generar();
// ejem1 y ejem2 tienen iguales sus 30 primeras bases (caracteres)
```

Se proporcionan las dos clases en código abierto `CadenaADN` y `GeneradorDeCadenasADN` cuyo contenido puede consultarse y probarse. La Actividad consiste en repetir la temporización de los tres métodos `DyV` de la clase `Ordenacion` que se efectuó en la Actividad #4, si bien en este caso para arrays de tipo `CadenaADN` con las 30 primeras bases nitrogenadas (internamente representados por caracteres) iguales. A continuación, se debe realizar el ajuste y la representación gráfica de los resultados obtenidos como en la Actividad #5 y, finalmente, deben extraerse las conclusiones correspondientes sobre cuál de los tres métodos analizados es el más rápido en este caso y por qué.

4. Otras actividades

Si se desea, resultaría de interés resolver alguna de las siguientes actividades para continuar analizando los problemas planteados en esta práctica:

- Repetir la última temporización realizada pero usando ahora arrays de tipo `CadenasADN` que tengan, respectivamente, sus primeras 1 y 50 bases nitrogenadas iguales. Interpretar los resultados obtenidos.
- Repetir los estudios de coste realizados añadiendo en una última columna los resultados obtenidos usando el método `sort` de la clase `java.util.Arrays`.