# GUION DE LA PRÁCTICA 2 (directorio p2)

## 0)INTRODUCCIÓN

Abordaremos en la práctica 2 el problema de la ordenación, esto es, dados n elementos en cualquier orden aplicarles un algoritmo que logre ordenarlos por una clave predeterminada. Los elementos pueden ser de cualquier tipo; aquí serán de tipo entero, pero los mismos algoritmos ordenarían reales, strings u objetos de cualquier tipo sobre los que haya una relación de orden que permita saber si un objeto dado es menor, mayor o igual a cualquier otro.

Cuando hay un cierto volumen de datos, tenerlos ordenados hace más rápido el acceso para hacer cualquier búsqueda (u otras operaciones básicas) sobre ellos. Por ello, ordenar es sin duda una de las operaciones que se realiza de forma más frecuente en cualquier tratamiento de información y de ahí su interés.

Es uno de los problemas que tiene más algoritmos capaces de resolverlo, si bien aquí, en esta sesión, vamos a centrarnos solamente en cuatro de ellos.

En esta práctica, los tiempos que nos piden los tomaremos sin el optimizador (SIN_OPTIMIZACIÓN) porque si bien hemos visto que serán más altos, evitamos sorpresas que suceden cuando el optimizador JIT actúa y esos tiempos no se correspondan con la complejidad temporal de los diferentes algoritmos.

En las tablas de tiempos que nos piden, si cualquier tiempo se alarga más de 1 minuto pondremos Fuera de Tiempo (FdT) y si fuese menor a 50 milisegundos pondremos Falta de Fiabilidad (FdF) y no es necesario repetir el proceso a medir varias veces (10,100, …), como se ha visto, en una sesión anterior, que había que proceder para medir tiempos por debajo de los milisegundos.

## 1)INTERCAMBIO DIRECTO o BURBUJA

La clase **Vector.java** tiene unas operaciones básicas que permiten generar un vector ordenado, en orden inverso, en cualquier orden o aleatorio y escribirlo.

Asimismo, en la clase **Burbuja.java** se implementa ese algoritmo y se comprueba que funciona correctamente. Hay que estudiar sobre el papel su funcionamiento para algún caso ejemplo y después analizar su complejidad temporal para los casos de elementos inicialmente ya ordenados, en orden inverso y en orden aleatorio.

Para finalizar, en la clase **BurbujaTiempos.java** se va incrementando el tamaño del problema y calculando los tiempos de ese algoritmo para los tres supuestos vistos.

```
javac *.java

dir

java -Djava.compiler=NONE  p2.Burbuja  n
// n es un entero que dimensiona el problema

java -Djava.compiler=NONE  p2.BurbujaTiempos caso
// caso = [ordenado  or inverso or aleatorio]
```

**SE LE PIDE:**

**Tras medir los tiempos, rellenar la tabla:**

***TABLA 1= ALGORITMO BURBUJA***

*(tiempos en milisegundos y SIN_OPTIMIZACIÓN):*

**Pondremos "FdT" para tiempos superiores al minuto y "FdF" los menores a 50 msg.**

| *n* | *t ordenado* | *t inverso* | *t aleatorio* |
|---|---|---|---|
| 10000 | | | |
| 2*10000 | | | |
| 2**2*10000 | | | |
| 2**3*10000 | | | |
| 2**4*10000 | | | |

**Razone si los diferentes tiempos obtenidos concuerdan con lo esperado, según la complejidad temporal estudiada.**

# 2)SELECCIÓN

En la clase **Seleccion.java** se implementa ese algoritmo y se comprueba que funciona correctamente. Hay que estudiar sobre el papel su funcionamiento para algún caso ejemplo y después analizar su complejidad temporal para los casos de elementos inicialmente ya ordenados, en orden inverso y en orden aleatorio.

```
java -Djava.compiler=NONE  p2.Seleccion  n
// n es un entero que dimensiona el problema
```

**SE LE PIDE:**

**Implementar una clase SeleccionTiempos.java que tras ser ejecutada sirva para rellenar la siguiente tabla:**

***TABLA 2= ALGORITMO SELECCIÓN***

*(tiempos en milisegundos y SIN_OPTIMIZACIÓN):*

**Pondremos "FdT" para tiempos superiores al minuto y "FdF" los menores a 50 msg.**

| *n* | *t ordenado* | *t inverso* | *t aleatorio* |
|---|---|---|---|
| 10000 | | | |
| 2*10000 | | | |
| 2**2*10000 | | | |
| 2**3*10000 | | | |
| 2**4*10000 | | | |

**Razone si los diferentes tiempos obtenidos concuerdan con lo esperado, según la complejidad temporal estudiada.**

# 3)INSERCIÓN

En la clase **Inserción.java** se implementa ese algoritmo y se comprueba que funciona correctamente. Hay que estudiar sobre el papel su funcionamiento para algún caso ejemplo y después analizar su complejidad temporal para los casos de elementos inicialmente ya ordenados, en orden inverso y en orden aleatorio.

```
java -Djava.compiler=NONE  p2.Insercion  n
// n es un entero que dimensiona el problema
```

**SE LE PIDE:**

**Implementar una clase InsercionTiempos.java que tras ser ejecutada sirva para rellenar la siguiente tabla:**

## *TABLA 3= ALGORITMO INSERCIÓN*

*(tiempos en milisegundos y SIN_OPTIMIZACIÓN):*

**Pondremos "FdT" para tiempos superiores al minuto y "FdF" los menores a 50 msg.**

| *n* | *t ordenado* | *t inverso* | *t aleatorio* |
|---|---|---|---|
| 10000 | | | |
| 2*10000 | | | |
| 2**2*10000 | | | |
| 2**3*10000 | | | |
| 2**4*10000 | | | |
| 2**5*10000 | | | |
| ….. | | | |
| 2**13*10000 | | | |

**Razone si los diferentes tiempos obtenidos concuerdan con lo esperado, según la complejidad temporal estudiada.**

# 4)RÁPIDO

En la clase **Rapido.java** se implementa el algoritmo Rápido (el genial Quicksort fue inventado por C.A.R. Hoare en el año 1960) y se comprueba que funciona correctamente. Hay que estudiar sobre el papel su funcionamiento para algún caso ejemplo y después analizar su complejidad temporal para los casos de elementos inicialmente ya ordenados, en orden inverso y en orden aleatorio.

```
java -Djava.compiler=NONE  p2.Rapido  n

// n es un entero que dimensiona el problema
```

**SE LE PIDE:**

**Implementar una clase RapidoTiempos.java que tras ser ejecutada sirva para rellenar la siguiente tabla:**

## ***TABLA 4= ALGORITMO RÁPIDO***

*(tiempos en milisegundos y SIN_OPTIMIZACIÓN):*

**Pondremos "FdT" para tiempos superiores al minuto y "FdF" los menores a 50 msg.**

| *n* | *t ordenado* | *t inverso* | *t aleatorio* |
|---|---|---|---|
| 250000 | | | |
| 2*250000 | | | |
| 2**2*250000 | | | |
| 2**3*250000 | | | |
| 2**4*250000 | | | |
| 2**5*250000 | | | |
| 2**6*250000 | | | |

**Razone si los diferentes tiempos obtenidos concuerdan con lo esperado, según la complejidad temporal estudiada.**

**Tras ver lo que tarda el rápido en ordenar 16 millones de elementos inicialmente en orden aleatorio, calcule (a partir de las complejidades y los datos de las tablas anteriores), ¿cuántos días tardaría cada uno de esos tres métodos (Burbuja, Selección e Inserción) en hacer lo mismo?**

# 5)RÁPIDO + INSERCIÓN

A pesar de los excelentes tiempos del apartado anterior, vamos a intentar mejorarlos (al menos, intentarlo). La idea es sencilla, en el rápido visto se está llamando recursivamente hasta llegar a subvectores de tamaño cero (caso de parada). Pues bien, se propone que cuando los tamaños de esos subvectores sean menores o iguales a un parámetro k llamar a inserción (esta idea se basa en que esos subvectores pueden estar ya bastantes ordenados y la inserción se comportaba bien (O(n)) en ese caso).

**SE LE PIDE:**

**Reimplementar el método de inserción, para que en lugar de ordenar el vector que se le pasa en todas sus posiciones [0 .. longitud-1], lo ordene solo entre las posiciones [inferior .. superior], posiciones que se pasarán como parámetros.**

**Implementar una clase RapidoInsercion.java que ordene los elementos del vector según lo apuntado antes.**

**Ahora se propone hacer una toma de tiempos, que aunque no sea concluyente y determinante para compararlos, sí será orientativa. Para ello, implementar una clase RapidoInsercionTiempos.java que tomará tiempos (para un tamaño de**

**n=16 millones de elementos generados inicialmente de forma aleatoria) para rellenar la siguiente tabla:**

## ***TABLA 5= ALGORITMO RÁPIDO+INSERCIÓN (n=16 M. y aleatorio)***

*(tiempos en milisegundos y SIN_OPTIMIZACIÓN):*

**Pondremos "FdT" para tiempos superiores al minuto y "FdF" los menores a 50 msg.**

| *Caso* | *t aleatorio* |
|---|---|
| Rápido | |
| Rápido+Inserción (k=5) | |
| Rápido+Inserción (k=10) | |
| Rápido+Inserción (k=20) | |
| Rápido+Inserción (k=30) | |
| Rápido+Inserción (k=50) | |
| Rápido+Inserción (k=100) | |
| Rápido+Inserción (k=200) | |
| Rápido+Inserción (k=500) | |
| Rápido+Inserción (k=1000) | |

**Razone conclusiones obtenidas a partir de la tabla anterior.**

Se ha de entregar en un **.pdf** el trabajo que se le pide y además las clases **.java** que ha programado. Todo ello lo pondrá en una carpeta, que es la que entregará comprimida en un fichero **practica2ApellidosNombre.zip**.

El **plazo límite** de entrega de esta práctica es **un día antes** de su sesión de prácticas de la semana siguiente (los de martes antes de que acabe el próximo lunes, los de miércoles antes de que acabe el próximo martes, los del jueves antes de que acabe el próximo miércoles y los del viernes antes de que acabe el próximo jueves), en la tarea que se creará a tal efecto en el Campus Virtual.