ALGORITMIA

Grado en Ingeniería Informática del Software - Curso 2025-2026

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 alto 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 centro de proceso de datos 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 (algunas veces) cuando el optimizador JIT actúa y esos tiempos no se corresponden 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).

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 pequeño 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 (según lo pedido en la TABLA1 de tiempos) y calculando los tiempos de ese algoritmo para los tres supuestos vistos.

Tras hacer lo visto en sesiones anteriores:

```
javac *.java

dir

java  -Xint  p2.Burbuja  n
// n es un entero que dimensiona el problema

java -Xint p2.BurbujaTiempos  caso  repeticiones
// caso = [ordenado o inverso o aleatorio]

// repeticiones = [1 o 10 o 100 o 1000 o …]
```

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

n	*t ordenado*	*t inverso*	*t aleatorio*
10000			
20000			
40000			
80000			
160000			
320000			
640000			
1280000			

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 pequeño caso ejemplo y después analizar su complejidad temporal para los casos de elementos inicialmente ya ordenados, en orden inverso y en orden aleatorio.

```
java  -Xint  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

n	*t ordenado*	*t inverso*	*t aleatorio*
10000			
20000			
40000			
80000			
160000			
320000			
640000			
1280000			

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 sencillo caso ejemplo y después analizar su complejidad temporal para los casos de elementos inicialmente ya ordenados, en orden inverso y en orden aleatorio.

```
java -Xint  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

n	*t ordenado*	*t inverso*	*t aleatorio*
10000			
20000			
40000			
80000			
160000			
320000			
640000			
1280000			

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 pequeño caso ejemplo y después analizar su complejidad temporal para los casos de elementos inicialmente ya ordenados, en orden inverso y en orden aleatorio.

```
java  -Xint  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)

N	*t ordenado*	*t inverso*	*t aleatorio*
10000			
20000			
40000			
80000			
160000			
320000			
640000			
1280000			

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

Proyecte, a partir de las complejidades y los datos de las tablas anteriores, ¿cuántos días (1 día= 86ı400.000 milisegundos) tardaría cada uno de los cuatro métodos (Burbuja, Selección, Inserción y Rápido) en ordenar un vector de n= 81ı920.000 (o sea, n=80000*2^4*2^6=80000*2^{10}), en el caso inicialmente en orden aleatorio?

5)RÁPIDO vs. INSERCIÓN para tamaños bajos

Vamos a ver que es posible que un algoritmo de mayor complejidad tenga tiempos de ejecución más bajos que otro de menor complejidad; eso sí, para tamaños bajos del problema (no suficientemente grandes).

SE LE PIDE:

Implementar dos clases (RapidoTiemposBajos.java e InsercionTiemposBajos.java), que tras ser ejecutadas sirvan para rellenar la siguiente tabla:

TABLA 5= RÁPIDO vs. INSERCIÓN PARA TIEMPOS BAJOS

(tiempos en milisegundos y SIN_OPTIMIZACIÓN y para caso ALEATORIO)

n	*t rapido*	*t insercion*	*t rapido/t inserción*
10			
15			
20			
25			
30			
35			
40			
…			
100			

Razone a partir de qué valor de n comienza a tardar menos tiempo el algoritmo de menor complejidad (rápido) que el de mayor complejidad (inserción).

De forma **opcional**, a partir del análisis de los resultados obtenidos en el apartado anterior, puede diseñar e implementar un algoritmo que combine el método rápido con el método inserción, teniendo como objetivo bajar los tiempos obtenidos por el rápido y reflejados en la TABLA4.

Calcular los tiempos del algoritmo diseñado y compararlos con los del rápido, para acabar concluyendo en qué grado se consiguió el objetivo antes enunciado.

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 **p2ApellidosNombre.zip**.

La entrega de esta práctica se realizará, en tiempo y forma, según las indicaciones dadas por el profesor de prácticas.