Lógica y Algorítmica
Práctica 7
Análisis de Algoritmos

Resumen

En esta práctica vamos a implementar varios algoritmos y a realizar un **análisis empírico** de su tiempo de ejecución. Los pasos a seguir son:

1. **Implementar** el algoritmo propuesto en el ejercicio en Java con Eclipse.
2. Realizar diferentes pruebas del método implementado **aumentando el tamaño de la entrada** n.
3. Para cada tamaño de n considerado, tenemos que realizar **10 ejecuciones** y guardar el tiempo empleado en cada una de ellas. Después calculamos el **tiempo de ejecución medio** a partir de esos valores (descartamos el tiempo máximo y calculamos la media con los 9 valores restantes).
4. Realizar el Análisis de Datos y los ejercicios propuestos.
5. ENTREGA: el **código** se entrega al final de la sesión de prácticas en el repositorio GIPP. Los ejercicios del **análisis de datos**, tablas/gráficas, etc, se guardan en un documento pdf que hay que subir en una tarea en el aula virtual.

Cómo medir el tiempo de ejecución

Para medir el tiempo de ejecución de un método necesitamos obtener **dos valores de tiempo**, uno antes de iniciar su ejecución y otro cuando termina. El resultado será la **diferencia** entre ambos.
No debemos contabilizar el tiempo de las operaciones de lectura o escritura de datos.

Básicamente, estos son los pasos a seguir:

Dado un tamaño n del problema:

1. Instanciar objetos, inicializar variables,...
2. Generar valores aleatorios , leer datos de entrada, etc ..
3. REPETIR 10 veces:
   a) tiempo_inicio ← ObtenerTiempo();
   b) **MÉTODO(n)**
   c) tiempo_final ← ObtenerTiempo();
   d) **tiempo_ejecuc$_i$ ← tiempo_final - tiempo_inicio**
4. Calcular **tiempo_ejecucion_medio**
5. Imprimir tiempo medio, resultados, etc ..

Recursos que necesitamos

- Estudiar la API de las siguientes clases:
  - Clase `System`:
    - El método `currentTimeMillis()` para obtener el tiempo en milisegundos.
    - El método `nanoTime()` para obtener el tiempo en nanosegundos.
  - Clase `Random` (en `java.util`) para obtener números aleatorios o `Math.random()`.

Sesión 06. Ejercicio 1. Ordenar una matriz por filas

Dada una matriz cuadrada de $n \times n$ números enteros aleatorios, implementar el método `matrizOrdenadaPorFilas()` que ordene cada fila de la matriz en orden ascendente .
Ejemplo:

- Entrada :

10	4	2	3
50	40	10	20
11	10	20	0
10	2	3	5

- Para cada fila de la matriz: la copia en un array y la ordena con el método de Inserción.

- Salida :

2	3	4	10
10	20	40	50
0	10	11	20
2	3	5	10

Obtener de forma experimental el orden de complejidad del método `matrizOrdenadaPorFilas()` realizando los ejercicios del análisis de datos de la siguiente sección.

Clases que necesitamos:

- El paquete para este ejercicio es: `org.lya.sesion06`
- Clase `MatrizEnterosCuadrada.java` : Propiedades:
  - campo privado con la matriz de enteros `matriz`
  - Dos constructores:

    - `MatrizEnterosCuadrada(int numeroFilasCol)` que crea la matriz con valores aleatorios entre 0 y 50.

    - `MatrizEnterosCuadrada(int[][]m)` , que crea la matriz a partir de otra matriz de entrada.
  - Y los métodos:

    - de ordenación por inserción (estático) : `insercion(int[] array)`

    - el que queremos evaluar: `matrizOrdenadaPorFilas()`

    - `muestraMatriz()` (este código se proporciona en el repositorio)
- Clase `TiemposMatrizEnterosCuadrada.java` : con el `main()` y las pruebas para diferentes tamaños de matriz:
  - Cada experimento consistirá en crear una matriz de $n \times n$ elementos de forma aleatoria y repetir 10 veces la ejecución del método `matrizOrdenadaPorFilas()` midiendo su tiempo de ejecución. A partir de los los 10 tiempos obtenidos se calcula el **tiempo medio** para ese tamaño de matriz.
  - Repetimos el proceso anterior para diferentes valores de n, comenzamos con n=32 y vamos doblando el tamaño n=64, 128, 256, etc.. hasta n=8192.
- Se recomienda usar como referencia el código java implementado en las sesiones 9 (MatrizEnteros.java) y 10 (OrdenacionBidimensional.java)de la asignatura Introducción a la Programación.

Análisis de los datos: Obtener el orden de `matrizOrdenadaPorFilas()`

1. Realizar los experimentos **duplicando** el tamaño de la matriz: partimos de n=32, y continuamos con 64, 128, 256, etc.. hasta el máximo tamaño que nos permita nuestro ordenador. Elaborar una tabla con el **tiempo medio** obtenido para cada n:

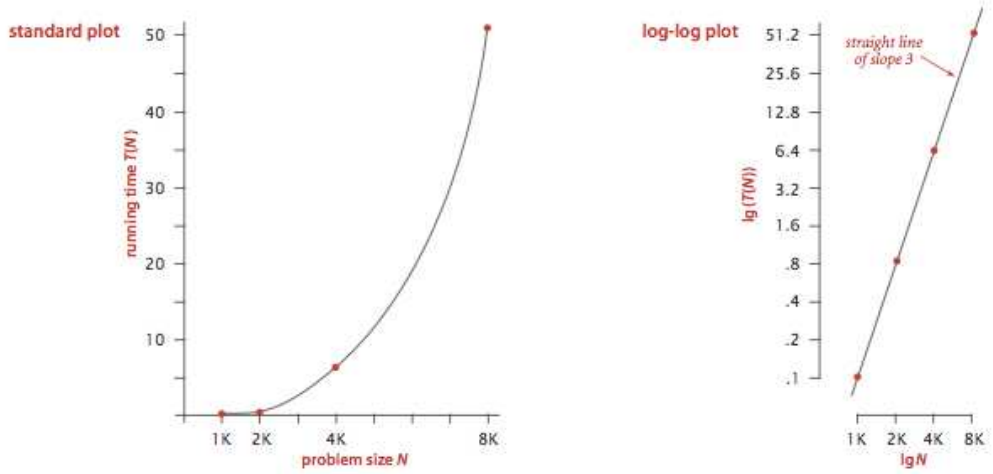
n	$\bar{T}(n)$
32	-
64	-
128	-
256	-
...	...
4096	-
8192	-

2. A partir de esta tabla obtener una gráfica en la que representamos:
tiempo de ejecución *vs* tamaño de la entrada.

3. Añadir dos columnas más en las que calculamos los logaritmos de los valores de n y de $T(n)$:

n	$\bar{T}(n)$	$\log(n)$	$\log(\bar{T}(n))$
32	-		
64	-		
...	—		
4096	-		
8192	-		

4. Obtener una gráfica representando los valores de logaritmos de las dos últimas columnas. Debemos obtener una **recta** cuya pendiente es la tasa de crecimiento del algoritmo.



Analysis of experimental data (the running time of `ThreeSum.count()`)

5. El orden de complejidad de `matrizOrdenadaPorFilas()` es polinómico, es de $O(n^k)$. Obtener el valor de k a partir de la gráfica anterior. (Nota: es la pendiente de la recta).

6. Estimar la constante c dependiente de la implementación a partir de la primera tabla. Es la constante que verifica que $\bar{T}(n) \leq cn^k$.
Entonces, ¿cuál sería la expresión para $T(n)$?

7. En base a la fórmula obtenida en el apartado anterior, ¿cuál sería teóricamente el tiempo de ejecución del algoritmo si se usa una matriz de 9000 filas?
Ejecutar el programa y comparar el tiempo teórico y el experimental (medio). ¿Coinciden?

Sesión 07. Ejercicio 2. Problema: Subsecuencia de suma máxima.

Dados n enteros cualesquiera $a_1, a_2, \cdots, a_n$ (posiblemente negativos) necesitamos encontrar el valor de la expresión:

$$\max_{1 \le i \le j \le n} \left\{ \sum_{k=i}^{j} \right\}$$

que calcula el **máximo de las sumas parciales de elementos consecutivos**.

Ejemplo 1:

- Entrada : $A =$

-2	11	-4	13	-5	-2
a_1	a_2	a_3	a_4	a_5	a_6

- Salida : valor máximo = 20 $\rightarrow$ desde a_2 hasta a_4 :

-2	**11**	**-4**	**13**	-5	-2

Ejemplo 2:

- Entrada : $A =$

1	-3	4	-2	-1	6
a_1	a_2	a_3	a_4	a_5	a_6

- Salida : valor máximo = 7 $\rightarrow$ desde a_3 hasta a_6 :

1	-3	**4**	**-2**	**-1**	**6**

Algoritmos: Implementar tres algoritmos que resuelven este problema (ver Weiss. Capítulo 5.3) y obtener su orden de complejidad, que en los tres casos es de la forma $O(n^k)$:

1 Algoritmo básico de fuerza bruta: Examina todas las posibles subsecuencias.

2 Algoritmo mejorado.

3 Algoritmo lineal.

Clases que necesitamos:

- **Paquete para este ejercicio:** `org.lya.sesion07` .
- `Subsecuencia.java` : con los 3 algoritmos y el array de enteros que se genera de forma aleatoria
  - campo privado con el array de enteros `array` .
  - campos privados para valores de los índices `Primer`, `Ultimo` y el valor de la `Suma` .
  - Dos constructores:
    - `Subsecuencia (int numeroElementos)` que crea el array con valores aleatorios entre -50 y 50.
    - `Subsecuencia(int[] array)` , que crea el array a partir de otro.
  - Métodos de acceso a los campos anteriores: `getSuma()`, `getPrimer()` y `getUltimo()`.
  - Un método para cada algoritmo: `SubsecuenciaFuerzaBruta()` , `SubsecuenciaMejorado()` , `SubsecuenciaLineal()`.
- `TiemposSubsecuencia.java` : con el main para probar los 3 métodos
  - Método `main()`: Crear un programa principal que permita:
    - Crear el array con la secuencia de tamaño n de **forma aleatoria**.
    - **Para una MISMA secuencia**: Ejecutar **10 veces cada algoritmo** y calcular el tiempo medio de ejecución que ha empleado cada uno de ellos. Es decir, **para cada secuencia de tamaño n obtenemos 3 tiempos medios**.

Estudio de los algoritmos y Análisis de los datos:

1. Estudiar **teóricamente** cada algoritmo (consultando el libro) e indicar su orden de complejidad.
2. Comparar los tiempos empleados por cada algoritmo para un mismo vector de entrada. Realizar las pruebas como en el ejercicio 1, repitiendo 10 veces cada experimento (mismo vector) con secuencias de longitud 64, 128, 256, 512, 1024,2048 y 8192 ... y calcular los tiempos medios. Elaborar una tabla con los tiempos medios obtenidos:

n	$\bar{T}(n)$ algor. Fuerza Bruta	$\bar{T}(n)$ algor. Mejorado	$\bar{T}(n)$ algor. Lineal
64	-	-	-
128	-	-	-
....	-	-	-
1024	-	-	-
2048	-	-	-
4096	-	-	-
8192	-	-	-

3. A partir de esta tabla, obtener una gráfica representando los resultados experimentales del tiempo de ejecución de los tres algoritmos (las tres curvas en la misma gráfica).
4. Crear una nueva gráfica comparativa tomando los logaritmos de los datos, y calcular la tasa de crecimiento para cada método (con las pendientes de las rectas), y a partir de ella obtener $O(n^k)$ de cada uno de los métodos.
5. ¿Coinciden las tasas de crecimiento experimentales obtenidas en el apartado anterior con las obtenidas del libro en el primer apartado? Justifica la respuesta.

Trabajo autonomo: BENCHMARKING

Buscar en Internet (si es en wikipedia usar la versión inglesa) información sobre estos términos:

- ¿Qué es un *benchmark* para computadores?
- Principales campos de aplicación
- ¿Qué tipo de pruebas realiza Linpack? ¿Qué es un MFLOP?
- ¿Qué es SPEC? ¿En qué campos desarrolla Benchmarks?

Entrega de Material

- El código java se sube a vuestro repositorio personal de esta asignatura. Debe pasar los juegos de pruebas.
- Los ejercicios se contestan en el mismo orden que en el enunciado, usando un máximo de 6 folios para las respuestas y gráficas (y uno más si se pone portada). Todo se incluye en un ÚNICO documento .pdf.
  El documento se llamará Practica7-DNI, es decir debe incluir el DNI del autor/autora. Se subirá al aula virtual dentro de una tarea que se propondrá para su entrega. No se aceptarán trabajos fuera de plazo.
- Los ejercicios copiados tendrán automáticamente la calificación de suspenso en Algorítmica.