Análisis de Algoritmos

Práctica A1: sesiones 01 y 02

Lógica y Algorítmica

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** los algoritmos propuestos en los ejercicios en Java usando el IDE Eclipse.
2. Realizar diferentes pruebas de ejecución de los métodos **doblando el tamaño de la entrada** n, partiendo de un valor inicial (que se indica en cada ejercicio). En una **tabla** iremos recogiendo los distintos valores de n y el tiempo medio empleado en la ejecución del método.

   Repetimos estos pasos:

   a) Generamos una entrada de tamaño n_i, con la que tenemos que realizar **10 ejecuciones** y guardamos el tiempo empleado en cada una de ellas: $t_1, t_2, ..., t_{10}$.

   b) Después calculamos el **tiempo medio de ejecución**, $\bar{T}(n_i)$, a partir de esos valores: descartamos el tiempo máximo y calculamos la media con los 9 valores restantes.

   c) Guardamos en la tabla el valor de n_i y el tiempo medio de ejecución obtenido $\bar{T}(n_i)$.

   d) Repetimos para el siguiente tamaño n_{i+1} que será el doble que el anterior, $n_{i+1} = 2n_i$.
3. Realizar el Análisis de Datos y los ejercicios propuestos.
4. ENTREGA: el **código** se entrega al final de la sesión de prácticas en el repositorio svn. Los ejercicios del **análisis de datos**, tablas/gráficas, etc, se contestan en el mismo orden del enunciado, y se suben en formato pdf a una actividad 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 tiempo empleado por el método 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:

Dada una instancia de tamaño n_i:

1. REPETIR 10 veces:
   a) tiempo_inicio ← ObtenerTiempo();
   b) Invocar **MÉTODO(n_i)**
   c) tiempo_final ← ObtenerTiempo();
   d) $\mathbf{t_k}$ ← **tiempo_final - tiempo_inicio**
2. Calcular **tiempo_ejecucion_medio** a partir de $t_1, t_2, ..., t_{10}$
3. Salida: $<$ n_i, **tiempo_ejecucion_medio**$>$

Métodos java que necesitamos:

- El método `System.currentTimeMillis()` para obtener el tiempo en milisegundos.
- El método `System.nanoTime()` para obtener el tiempo en nanosegundos.

1. Sesión 01. Método a evaluar: Ordenar una matriz por filas

1. Implementar el método `matrizOrdenadaPorFilas()` que toma como entrada una matriz cuadrada de $n \times n$ números enteros aleatorios, ordena cada fila en orden ascendente y devuelve una nueva matriz con las filas ordenadas. Por ejemplo:

   - Matriz de Entrada :

   $$\begin{pmatrix} 10 & 4 & 2 & 3 \\ 50 & 40 & 10 & 20 \\ 11 & 10 & 20 & 0 \\ 10 & 2 & 3 & 5 \end{pmatrix} \quad (1)$$

   - Matriz de Salida :

   $$\begin{pmatrix} 2 & 3 & 4 & 10 \\ 10 & 20 & 40 & 50 \\ 0 & 10 & 11 & 20 \\ 2 & 3 & 5 & 10 \end{pmatrix} \quad (2)$$

   - Cada fila de la matriz se ordena con el método de **burbuja mejorado**.
   - Se crea un nuevo objeto con la nueva matriz de salida. **La matriz de entrada NO se modifica**.

2. Implementar método `main()`: para obtener **de forma experimental** el orden de complejidad del método anterior es necesario obtener su tiempo de ejecución medio para diferentes tamaños de la matriz n. Por tanto, seguiremos estos pasos en el método `main()`:
   - Crear una matriz de $n \times n$ elementos de forma aleatoria de valores de tipo entero ≤ 80.
   - Con la misma matriz, Repetir 10 veces la ejecución del método `matrizOrdenadaPorFilas()` . En cada ejecución se mide el tiempo empleado y se guarda este tiempo en un array.
   - Una vez completadas las diez ejecuciones con la misma matriz $n \times n$ se descarta el valor del tiempo máximo, y 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.

     ☞ Con los tiempos medios obtenidos completamos la tabla que se indica en el ejercicio 1 de la sección de Análisis de Datos.

Clases que necesitamos implementar: paquete `org.lya.sesion01`

1. Clase `MatrizEnterosCuadrada.java` : Contiene la matriz de enteros, los getters y los siguientes métodos:
   - campo privado con la matriz : `int [][] matriz`
   - Dos constructores para crear la matriz:

     - `MatrizEnterosCuadrada(int numeroFilasCol)`: le asigna valores aleatorios entre 0 y 80.

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

     - `static void burbujaMejora(int[] array)` : ordenación por burbuja mejorado

     - `MatrizEnterosCuadrada` **matrizOrdenadaPorFilas()** : el método que queremos evaluar. Devuelve un nuevo objeto con la nueva matriz ordenada.

     - `String toString()` : método toString para pasar los elementos de la matriz a un String. Se recomienda revisar el fichero de test para comprobar qué resultado se debe obtener.

2. Clase `TiemposMatrizEnterosCuadrada.java`: con el `main()` y las pruebas para diferentes tamaños de matriz que se muestran en la tabla del ejercicio 1 de la siguiente sección.

☞ El código debe pasar las pruebas de test. El fichero se encuentra en el repositorio de la asignatura.

☞ Se recomienda consultar el código java implementado en la asignatura Introducción a la Programación.

2. Sesión 01. Análisis de los datos

Queremos obtener el orden de complejidad del método `matrizOrdenadaPorFilas()` de forma experimental.

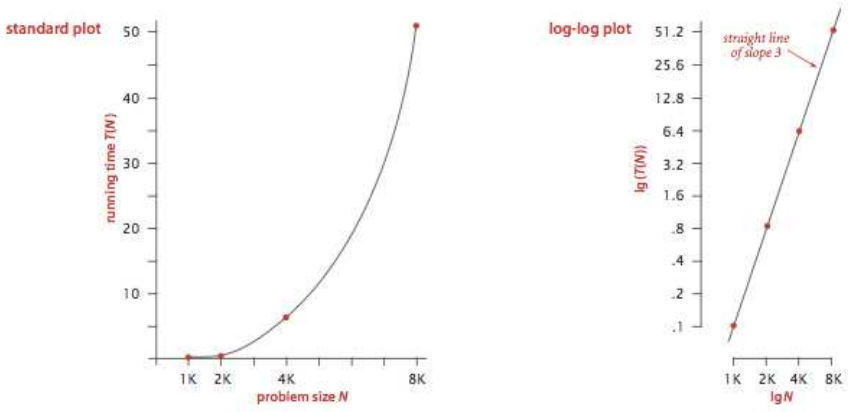
1. Realizar los experimentos **duplicando** el tamaño de la matriz: partimos de n=32, y continuamos con 64, 128, 256, etc.. hasta el máximo tamaño que nos permita nuestro ordenador. Elaborar una tabla con el **tiempo medio** obtenido para cada n:

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 en base 2 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 orden de complejidad del método de forma experimental calculando el valor de k a partir de la gráfica anterior (k es la pendiente de la recta).
6. Indicar por qué el orden teórico de este algoritmo es $O(n^3)$, es decir, $k = 3$.
7. Estimar la constante c dependiente de la implementación a partir de la primera tabla. Es la constante que verifica que $\bar{T}(n) \leq cn^k$.
8. En base a que $T(n)$ es $O(n^3)$ y el valor de c obtenido en el apartado anterior, ¿cuál sería **teóricamente** el tiempo de ejecución del algoritmo si se usa una matriz cuadrada de tamaño n=100000?

3. Sesión 02. Métodos a evaluar: Resolver el Problema de la "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_0	a_1	a_2	a_3	a_4	a_5

  - Salida :

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

valor máximo $= 20 \rightarrow$ desde a_1 hasta a_3

- Ejemplo 2:
  - Entrada : $B =$

1	-3	4	-2	-1	6
a_0	a_1	a_2	a_3	a_4	a_5

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

valor máximo $= 7 \rightarrow$ desde a_2 hasta a_5

1. 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 $O(n^k)$:
   a) Algoritmo básico de fuerza bruta: Examina todas las posibles subsecuencias.
   b) Algoritmo mejorado.
   c) Algoritmo lineal.

2. **Clases que necesitamos implementar:** Paquete `org.lya.sesion02` .
   a) `Subsecuencia.java` : con los 3 algoritmos y el array de enteros que se genera de forma aleatoria. Contiene los siguientes campos y métodos:
   - campo privado con el array de enteros: `int[] array` .
   - campos privados para valores de los índices `primer`, `ultimo` y el valor de la `suma` .
   - Dos constructores:
     - `Subsecuencia (int numeroElementos)`: crea `array` con valores aleatorios entre -80 y 80.
     - `Subsecuencia(int[] arrayEnteros)`: crea `array` igual que el que se pasa por parámetro.
   - Métodos de acceso a los campos anteriores: `getPrimer()`, `getUltimo()` y `getSuma()`.
   - Un método para cada algoritmo: `void SubsecuenciaFuerzaBruta()` , `void SubsecuenciaMejorado()` , `void SubsecuenciaLineal()`.

   b) `TiemposSubsecuencia.java` : con el main para probar los 3 métodos
   - Método `main()`: Crear un programa principal (similar a la sesión 01) que permita:
     - Crear el array con la secuencia de tamaño n de **forma aleatoria**.
     - **Para un MISMO array o secuencia**: Ejecutar **10 veces cada algoritmo** y calcular el tiempo medio de ejecución que ha empleado cada uno de ellos.
     Es decir, **para cada array de tamaño n, calculamos 3 tiempos medios**.

     ☞ Con los tiempos medios obtenidos completamos la tabla que se indica en el ejercicio 2 de la sección de Análisis de Datos.
     ☞ El código debe pasar las pruebas de test. El fichero se encuentra en el repositorio de la asignatura.

4. Sesión 02. Análisis de los datos

1. Estudiar **teóricamente** cada algoritmo (consultando el libro) y explicar su orden de complejidad.
2. Comparar los tiempos empleados por cada algoritmo para un mismo array de entrada. Realizar las pruebas como en el ejercicio 1, repitiendo 10 veces cada experimento (mismo array) 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 en base 2 de los datos, y calcular de forma experimental el orden $O(n^k)$ de cada uno de los métodos (con las pendientes de las rectas).
5. ¿Coinciden los órdenes experimentales obtenidos en el apartado anterior con los órdenes teóricos del libro? Justifica la respuesta.

5. Trabajo autónomo: 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?
- Recoge en una tabla las características de los 10 primeros supercomputadores de la **TOP500 list** de noviembre de 2021. Indica también para qué se usan esos supercomputadores y cuánta energía consumen.

Entrega de Material

- El **código java** se sube al **repositorio personal svn** de esta asignatura. **Debe pasar los juegos de pruebas**.
- Los ejercicios se deben **contestar en el mismo orden que están propuestos**, y se entregan las dos sesiones en un ÚNICO documento pdf con el nombre PA1_Apellido1Apellido2Nombre.pdf.
  El **documento pdf** se subirá al **aula virtual** dentro de una tarea que se propondrá para su entrega. No se aceptarán trabajos fuera de plazo.
- **Las prácticas se defenderán en clase. Los ejercicios copiados tendrán automáticamente la calificación de suspenso en Algorítmica, tanto para la convocatoria ordinaria como para la extraordinaria.**