Análisis y Diseño de Algoritmos — Práctica Tema 1

Complejidad experimental

Desde el punto de vista teórico, el análisis de un algoritmo nos permite determinar cuál es su complejidad teórica. En la práctica, nos encontramos también con muchos casos en los que determinar esta complejidad de un programa complejo de forma teórica es difícil, o incluso imposible si no disponemos del código fuente. Aun así, la experimentación con distintos valores de entrada nos ofrece información sobre la posible complejidad de un programa. En esta práctica nos proponemos diseñar un programa en Java para medir experimentalmente la complejidad de otro programa ya compilado (es decir, a posteriori).

Supongamos que queremos medir experimentalmente la complejidad de un algoritmo implementado en una clase denominada `Algoritmo.class`, que contiene la implementación de ese algoritmo mediante el método:

```
public class Algoritmo {
	public static synchronize void f(long n) {
		...
	}
}
```

en donde el argumento n es la variable de la que depende el tiempo de ejecución. Supongamos inicialmente que la complejidad del algoritmo solo puede ser $\Theta(n)$, o bien $\Theta(n^2)$; o bien $\Theta(n^3)$. En principio cabria esperar que el tiempo de ejecución cambie en cada caso de la siguiente manera:

Tabla 1

n	1	2	3	4	5	6	7	8	9	10
$f(n)=n$	1	2	3	4	5	6	7	8	9	10
$f(n)=n^2$	1	4	9	16	25	36	49	64	81	100
$f(n)=n^3$	1	8	27	64	125	216	343	512	729	1000

Para medir el tiempo de ejecución de un método en Java, simplemente es necesario obtener la fecha antes y después de la ejecución de la función y hallar la diferencia. (También se puede utilizar la clase auxiliar `Temporizador.java` que se proporciona como fichero auxiliar).

```
public class Analizador {
	public static void main(String arg[]) {
		long t1, t2;
		for(int n=1; n<=10; n++) {
			t1 = System.currentTimeMillis();
			Algoritmo.f(n);
			t2 = System.currentTimeMillis();
			System.out.println("T("+n+")="+(t2-t1));
		}
		/* ... modificar y completar el código ... */
	}
}
```

Aunque en la práctica, si medimos el tiempo en milisegundos probablemente obtendríamos unos resultados muy diferentes a los esperados. En primer lugar, tendremos que buscar una escala de tiempo adecuada al rango de milisegundos[1], ya que si se ejecuta con valores pequeños no se obtienen resultados:

[1] Los procesadores actuales realizan muchas operaciones elementales por milisegundo

Tabla 2

n	1	2	3	4	5	6	7	8	9	10
$T_0(n)$	1	0	0	0	0	0	0	0	0	0

Para solucionar este problema, simplemente hay que utilizar valores multiplicados por una constante.
El siguiente problema que podemos encontrar, es que si repetimos varias veces las mismas pruebas, en cada caso obtendremos resultados diferentes, tales como por ejemplo:

Tabla 3

n	$1*10^4$	$2*10^4$	$3*10^4$	$4*10^4$	$5*10^4$	$6*10^4$	$7*10^4$	$8*10^4$	$9*10^4$	$10*10^4$
$T_1(n)$	4	9	20	34	54	77	103	135	169	237
$T_2(n)$	5	9	20	36	53	78	102	134	170	209
$T_3(n)$	4	9	19	35	52	77	103	136	170	207
$T_4(n)$	4	9	20	36	53	79	103	137	169	207
$T_5(n)$	4	11	20	36	55	81	107	139	176	215

Esto es debido a que al ejecutar el programa sobre un sistema operativo multitarea, el tiempo real de ejecución depende de la carga del sistema. Para minimizar el efecto de la carga del sistema se pueden seguir dos estrategias: (1) Hallar la media de los tiempos de ejecución para cada valor de *n*, ya que así se compensan los efectos de los procesos subyacentes; (2) Hallar el mínimo tiempo de ejecución para cada valor de *n*, ya que este valor estará más próximo al tiempo de ejecución real del método que nos interesa analizar. Supongamos que optamos por la primera estrategia, en ese caso el tiempo de ejecución empírico es:

Tabla 4

n	$1*10^4$	$2*10^4$	$3*10^4$	$4*10^4$	$5*10^4$	$6*10^4$	$7*10^4$	$8*10^4$	$9*10^4$	$10*10^4$
$\overline{T(n)}$	4,2	9,4	19,8	35,4	53,4	78,4	103,6	136,2	170,8	215

Podemos comparar los resultados obtenidos con los resultados teóricos de la tabla 1, simplemente dividiendo ambas series:

Tabla 5

n	$1*10^4$	$2*10^4$	$3*10^4$	$4*10^4$	$5*10^4$	$6*10^4$	$7*10^4$	$8*10^4$	$9*10^4$	$10*10^4$
$\frac{\overline{T(n)}}{n}$	4,20	4,70	6,60	8,85	10,68	13,07	14,80	17,03	18,98	21,50
$\frac{\overline{T(n)}}{n^2}$	4,20	2,35	2,20	2,21	2,14	2,18	2,11	2,13	2,11	2,15
$\frac{\overline{T(n)}}{n^3}$	4,20	1,18	0,73	0,55	0,43	0,36	0,30	0,27	0,23	0,22

Recordando la teoría, sabemos que si T(n) tiene la misma complejidad que f(n), entonces:

$$\exists k \in \Re, k > 0 \qquad \lim_{n->\infty}\left(\frac{T(n)}{f(n)}\right) = k$$

Observando la tabla 5 vemos que la primera fila correspondiente a una complejidad lineal cada vez tiene valores mayores, mientras que la segunda se mantiene en valores cercanos a 2 y la tercera va disminuyendo acercándose a cero.

También se pueden comparar los ritmos de crecimiento de una y otra función, fijándonos para ello en valores múltiplos de 2.

Tabla 6

n	$1*10^4$	$2*10^4$	$3*10^4$	$4*10^4$	$5*10^4$	$6*10^4$	$7*10^4$	$8*10^4$	$9*10^4$	$10*10^4$
$\frac{2n}{n}$	2	2	2	2	2	2	2	2	2	2
$\frac{(2n)^2}{n^2}$	4	4	4	4	4	4	4	4	4	4
$\frac{(2n)^3}{n^3}$	8	8	8	8	8	8	8	8	8	8

Como puede observarse, si la función es lineal al duplicar el valor de *n*, se duplica el tiempo de respuesta, si la función es cuadrática el tiempo teóricamente se multiplica por 4 y si es cúbica, se multiplica por 8. Basta observar lo que ocurre en el caso del tiempo de ejecución del algoritmo que se está analizando para llegar a la conclusión de que si hay que decidir entre una de las tres opciones, la más probable es la complejidad $\Theta(n^2)$.

Tabla 7

n	$1*10^4$	$2*10^4$	$4*10^4$	$8*10^4$	$16*10^4$	$32*10^4$	$64*10^4$	$128*10^4$	$256*10^4$	$512*10^4$
$\overline{T(n)}$	4,2	9,4	19,8	35,4	136,4	537,2	2111,8	8266,6	31530,0	122375,2
$\frac{\overline{T(2n)}}{\overline{T(n)}}$	2,24	2,11	1,79	3,85	3,94	3,93	3,91	3,81	3,88	...

En efecto, el algoritmo que se ha implementado era el siguiente:

```
public class Algoritmo {
	long l = 0L;
	public static synchronized void f(long n) {
		for (int j= 0; j < n; j++) {
		   for (int i= 0; i < n; i++) {
		       l += 1L;
		    }
		}
	}
}
```

Observamos no obstante que para valores relativamente pequeños de n tales como $1*10^4$ y $2*10^4$, el algoritmo implementado no se comporta como de forma cuadrática ; sino más bien de forma lineal. Esto es debido a los costes inherentes a la carga de la función y a la ejecución de procesos básicos añadidos por el compilador en el programa objeto. En la práctica existen casos aún más complicados en los que la función de complejidad no alcanza la asíntota hasta valores de *n* mayores. Por ejemplo, otro algoritmo de complejidad $\Theta(n^2)$ podría dar como tiempos de ejecución los siguientes, en vez de los que aparecen en la tabla 7, obteniéndose unos ratios diferentes:

Tabla 8

n	$1*10^4$	$2*10^4$	$4*10^4$	$8*10^4$	$16*10^4$	$32*10^4$	$64*10^4$	$128*10^4$	$256*10^4$	$512*10^4$
$\overline{T(n)}$	53,0	63,2	126,2	298,4	841,4	2737,0	9391,0	33423,4	126634,4	489038,6
$\frac{\overline{T(2n)}}{\overline{T(n)}}$	1,19	2,00	2,36	2,82	3,25	3,43	3,56	3,79	3,86	...

En este caso, el algoritmo con valores bajos de *n* parece tener un coste computacional casi constante o lineal, y hasta que probamos con valores más altos no se muestra su verdadera complejidad.

Se pide:

Implementar un programa `Analizador.java` capaz de determinar de forma automática la complejidad experimental de la función `f(int n)` de distintas clases denominadas `Algoritmo.class` que se proporcionan ya compiladas.

A priori se sabe que la complejidad de las funciones es siempre una de las ocho que se muestran en la siguiente tabla. La ejecución del programa `java Analizador` dará como resultado una de las siguientes etiquetas:

Tabla 9

Complejidad	Salida de `java Analizador`
$\Theta(1)$	**1**
$\Theta(log(n))$	**LOGN**
$\Theta(n)$	**N**
$\Theta(nlog(n))$	**NLOGN**
$\Theta(n^2)$	**N2**
$\Theta(n^3)$	**N3**
$\Theta(2^n)$	**2N**
$\Theta(n!)$	**NF**

La evaluación de la práctica se hará de forma automática en dos partes:

(1) Se proporcionarán 20 clases diferentes compiladas con el nombre `Algoritmo.class`. Para cada una de ellas, el alumno deberá seleccionar mediante una pregunta de opción múltiple una de estas ocho complejidades.(*)

(*) Aunque la clase se llame siempre `Algoritmo.class`, y el metodo `f()` tenga la misma forma, en cada una de las preguntas se proporciona un fichero diferente. El test no tiene un tiempo máximo, pero debe contestar a cada pregunta en el orden en que aparece. Una vez enviada una respuesta, no es posible modificarla. Si abandona el test puede reanudarlo. El test continuará por la última pregunta que dejó sin contestar. Cada pregunta acertada vale 1 punto, y cada pregunta fallada resta (1/7) puntos, para compensar el efecto de la respuesta al azar.

(2) Como resultado de la práctica se enviara un fichero `Analizador.zip` que contenga todos los ficheros fuente necesarios para compilar y ejecutar este ejercicio (salvo los ficheros `Algoritmo.java`, y `Algoritmo.class`, que se incluirán automáticamente en el proceso de corrección automática, y no deben estar incluidos en el .zip).
Para compilar y ejecutar el programa se usaran las siguientes instrucciones:

```
javac *.java
java Analizador
```

Al realizar corrección automática es imprescindible que la salida sea exactamente la que se indica en la tabla 9. El programa debe ser capaz de determinar la complejidad de cualquier algoritmo en menos de 10 seg.

Algunas pistas para realizar la practica.

* No utilice paquetes de Java, coloque todas las clases en un mismo directorio.

* La salida del programa ejemplo debe ser `"N2"`, pero se deben probar otras clases, para ello en vez de usar la clase compilada, se deben escribir distintas clases `Algoritmo.java` de las que se conozca su complejidad , compilarlas (con `javac Algoritmo.java`), y probarlas con el programa que se haya implementado (con `java Analizador`).

* La complejidad de un algoritmo es independiente de la potencia de la máquina en la que se ejecuta, aunque distintas máquinas darán distintos tiempos de ejecución los límites o los ratios entre dos ejecuciones en la misma máquina son perfectamente validos para determinar la complejidad.

* Distintos algoritmos pueden requerir distintos valores de `n` para determinar su complejidad. En ocasiones es suficiente probar con valores 1,2,3,4,etc. otras será necesario probar con valores 10,20,40,80, 160, etc. y otras con 1000000,2000000,4000000,16000000,256000000, etc. Una buena practica consiste en probar inicialmente con valores pequeños, y si el tiempo de ejecución es pequeño, ir aumentando hasta conseguir determinar claramente la complejidad, o bien alcanzar tiempos de ejecución del orden de segundos. (para cumplir la restricción de no superar los 10 segundos de ejecución).

* Para realizar este ejercicio NO ES NECESARIO UTILIZAR ECLIPSE, aunque se puede hacer si se quiere y se sabe como. Para compilar el código que se entrega solo es necesario poner los tres ficheros en una misma carpeta, abrir una terminal y utilizar las instrucciones:

```
>javac *.java
>java Analizador
N2
```

* Eclipse, cuando compila un proyecto Java, por defecto mete los fuentes en un directorio y la clases compiladas en otro. Si se quiere usar Eclipse debe hacerse lo siguiente:

- Crear una carpeta "classes" dentro del proyecto.
- Meter "Algoritmo.class" dentro de esa carpeta.
- Indicarle a Eclipse, en Preferencias de proyecto -> Java Build Path -> Libraries -> Add Class folder -> Añadir la carpeta "classes".
- Comprobar que Eclipse usa JRE 1.7. (si no obtendreis una Excepcion... major minor version)Para eso hay que hacer dos cosas:
    - .... en Preferencias de proyecto -> Java Build Path -> Libraries -> Seleccionar el JRE + Edit y cambiar a JRE 1.7.
    - ... en Preferencias de proyecto -> Java Compilar -> Poner la variable “Compiler Compliance level” a 1.7. [Esto es necesario porque la clase que se entrega esta compilada con JDK 1.7]