Análisis y Diseño de Algoritmos — Práctica Complejidad

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 cual 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:

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

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

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 mas 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:

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

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 {
        public static synchronized void f(long n) {
                long l = 0L;
                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 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.

1 Ejercicios

Para completar la práctica se proponen una serie de tareas, que deberán llevar a cabo cada alumno individualmente. El resultado de estas tareas dará lugar al desarrollo de una aplicación en Java y de una memoria explicativa de los resultados obtenidos. Ambos deberán entregarse a través del campus virtual antes de la fecha límite, a través de la tarea creada al efecto y en el formato solicitado. No se corregirán productos enviados por otros medios o que se envíen en un formato distinto al solicitado.

El objetivo de la práctica consiste en desarrollar un programa que sea capaz de determinar empíricamente, mediante la toma de tiempos, la complejidad de un conjunto de algoritmos. Concreatemente, las tareas a llevar a cabo son:

1. Descargar el archivo comprimido del campus con los códigos de ayuda y estudiar su contenido.
2. Ejecutar el archivo "compile.bat" desde una consola de comandos para compilar el código de ejemplo.
3. Ejecutar el archivo "run.bat" para ejecutar el código de ejemplo.
4. Modificar el método "findComplexityOf", de la clase "Analyzer", para que devuelva la complejidad del algoritmo pasado como argumento. No debe emplearse más de "maxExecutionTime" milisegundos en determinar esa complejidad. En caso de sobrepasar ese tiempo, se considerará que el analizador no ha sido capaz de determinar la complejidad del algoritmo. Pueden añadirse todos los métodos que se consideren oportunos a esa clase, pero no deben modificarse los existentes (salvo el método "findComplexityOf", evidentemente).

2 Entrega

Todo el código desarrollado en esta práctica deberá estar incluido en el fichero "Analyzer.java". Será éste el único archivo que se envíe a través de la tarea habilitada al efecto en el campus virtual. El fichero debe enviarse sin comprimir.

3 Evaluación

Todos los programas enviados por los alumnos serán ejecutados en uno de los ordenadores del laboratorio y se anotará el número de algoritmos clasificados correctamente por el código de cada alumno. La calificación de la práctica se otorgará en función de la precisión alcanzada. Además de los algoritmos de ejemplo suministrados en el material de prácticas, para la evaluación se podrán usar otros algoritmos no suministrados a los alumnos. El tiempo máximo permitido para el análisis de cada algoritmo será de diez segundos.