Análisis de Algoritmos – Complejidad

José A. Mañas

Dept. de Ingeniería de Sistemas Telemáticos

10.2.2017

Índice

1 Introducción

La resolución práctica de un problema exige por una parte un algoritmo o método de resolución y por otra un programa o codificación de aquel en un ordenador real. Ambos componentes tienen su importancia; pero la del algoritmo es absolutamente esencial, mientras que la codificación puede muchas veces pasar a nivel de anécdota.

algoritmo

serie de pasos que nos llevan a resolver efectivamente un problema; podemos decir que un algoritmo es una idea

programa

un algoritmo escrito usando un lenguaje de programación (por ejemplo, java)

Ante un mismo problema, puede haber mejores ideas y peores ideas acerca de cómo afrontar su solución. En lo que sigue sólo nos ocuparemos de algoritmos correctos, es decir, que nos llevan a una solución válida del problema planteado. Dentro de lo que es correcto, el análisis de algoritmos nos lleva a poder decir si una idea es mejor que otra.

Con un mismo algoritmo podemos escribir un programa mejor o peor. No se puede escribir un programa correcto basado en un algoritmo incorrecto; pero dentro de la corrección de la idea, hay programadores y compiladores que son mejores que otros.

1.1 El tamaño

Para cada problema determinaremos una medida N de su tamaño (por número de datos) e intentaremos hallar respuestas en función de dicho N. El concepto exacto que mide N depende de la naturaleza del problema. Así, para un vector se suele utilizar como N su longitud; para una matriz, el número de elementos que la componen; para un grafo, puede ser el número de nodos (a veces es más importante considerar el número de arcos, dependiendo del tipo de problema a resolver); en un fichero se suele usar el número de registros, etc. Es imposible dar una regla general, pues cada problema tiene su propia lógica de coste.

1.2 Recursos

A efectos prácticos o ingenieriles, nos deben preocupar los recursos físicos necesarios para que un programa se ejecute. Aunque puede haber muchos parámetros, los mas usuales son el tiempo de ejecución y la cantidad de memoria (RAM). Ocurre con frecuencia que ambos parámetros están fijados por otras razones y se plantea la pregunta inversa: ¿cuál es el tamaño del mayor problema que puedo resolver en T segundos y/o con M bytes de memoria?

En lo que sigue usaremos esta notación

- T(n) – tiempo de ejecución en función del tamaño n del problema
- E(n) – espacio (RAM) en función del tamaño n del problema

En lo que sigue nos centraremos casi siempre en tiempo de ejecución, si bien las ideas desarrolladas son fácilmente aplicables a otro tipo de recursos.

1.3 Tiempo de ejecución

Una medida que suele ser útil conocer es el tiempo de ejecución de un programa en función de N, lo que denominaremos T(N). Esta función se puede medir físicamente

(ejecutando el programa, reloj en mano), o calcularse sobre el código contando instrucciones a ejecutar y multiplicando por el tiempo requerido por cada instrucción. Así, un trozo sencillo de programa como

```
S1;
for (i= 0; i < N; i++)
  S2;
```

requiere

$$T(N) = t1 + t2*N$$

siendo t1 el tiempo que lleve ejecutar la serie "S1" de sentencias, y t2 el que lleve la serie "S2".

Prácticamente todos los programas reales incluyen alguna sentencia condicional, haciendo que las sentencias efectivamente ejecutadas dependan de los datos concretos que se le presenten. Esto hace que más que un valor T(N) debamos hablar de un rango de valores

$$Tmin(N) \leq T(N) \leq Tmax(N)$$

los extremos son habitualmente conocidos como "caso peor" y "caso mejor". Entre ambos se hallara algún "caso promedio" o más frecuente.

Cualquier fórmula T(N) incluye referencias al parámetro N y a una serie de constantes "Ki" que dependen de factores externos al algoritmo como pueden ser la calidad del código generado por el compilador y la velocidad de ejecución de instrucciones del ordenador que lo ejecuta. Dado que es fácil cambiar de compilador y que la potencia de los ordenadores crece a un ritmo vertiginoso[1], intentaremos analizar los algoritmos con algún nivel de independencia de estos factores; es decir, buscaremos estimaciones generales ampliamente válidas.

1.4 Asíntotas

Por una parte, necesitamos analizar la potencia de los algoritmos independientemente de la potencia de la máquina que los ejecute e incluso de la habilidad del programador que los codifique. Por otra, este análisis nos interesa especialmente cuando el algoritmo se aplica a problemas grandes. Casi siempre los problemas pequeños se pueden resolver de cualquier forma, apareciendo las limitaciones al atacar problemas grandes. No debe olvidarse que cualquier técnica de ingeniería, si funciona, acaba aplicándose al problema más grande que sea posible: las tecnologías de éxito, antes o después, acaban llevándose al límite de sus posibilidades.

Las consideraciones anteriores nos llevan a estudiar el comportamiento de un algoritmo cuando se fuerza el tamaño del problema al que se aplica. Matemáticamente hablando, cuando N tiende a infinito. Es decir, su comportamiento asintótico.

[1] Se suele citar la Ley de Moore que predice que la potencia se duplique cada 2 años.

2 Metodología

Para enfocar la comparación de algoritmos seguiremos los siguientes pasos:

1. Averiguar la función f(n) que caracteriza los recursos requeridos por un algoritmo en función de tamaño n de los datos a procesar.
2. Dadas dos funciones f(n) y g(n) definir una relación de orden entre ellas que llamaremos complejidad y que nos dirá cuándo una función es más compleja que la otra o cuándo son equivalentes.
3. Seleccionar una serie de funciones de referencia para situaciones típicas.
4. Definir conjuntos de funciones que llamaremos órdenes de complejidad.

2.1 Paso 1 – función característica

Decidimos cómo medir N.

Decidimos el recurso que nos interesa

tiempo de ejecución = f(N)

memoria necesaria = f(N)

A partir de aquí analizaremos f(n).

Como veremos más adelante, calcular la fórmula analítica exacta que caracteriza un algoritmo puede ser bastante laborioso.

Con frecuencia nos encontraremos con que no es necesario conocer el comportamiento exacto, sino que basta conocer una cota superior, es decir, alguna función que se comporte "aún peor". De esta forma podremos decir que el programa práctico nunca superará una cierta cota.

2.2 Paso 2 – complejidad

Dadas dos funciones f(n) y g(n) diremos que f(n) es más compleja que g(n) cuando

$$\lim_{n\to\infty}\left(\frac{f(n)}{g(n)}\right)=\infty$$

Diremos que f(n) es menos compleja que g(n) cuando

$$\lim_{n\to\infty}\left(\frac{f(n)}{g(n)}\right)=0$$

Y diremos que f(n) es equivalente a g(n) cuando

$$\lim_{n\to\infty}\left(\frac{f(n)}{g(n)}\right)=K$$

siendo $K \neq 0$ y $K \neq \infty$

Nótese que esta definición nos permite un análisis algorítmico conociendo la formulación de la función, y también un análisis experimental observando los recursos consumidos para valores crecientes de N.

2.3 Paso 3 – funciones de referencia

Se suelen manejar las siguientes

f(n) = 1	constante
f(n) = log (n)	logaritmo
f(n) = n	lineal
f(n) = n × log(n)	
f(n) = n^2	cuadrática
f(n) = n^a	polinomio (de grado a > 2)
f(n) = a^n	exponencial (a > 1)
f(n) = n!	factorial

¿Por qué estas y no otras? Simplemente porque (1) son sencillas y (2) la experiencia dice que aparecen con mucha frecuencia y que es muy poco frecuente que necesitemos otras.

2.4 Paso 4 – órdenes de complejidad

A un conjunto de funciones que comparten un mismo comportamiento asintótico le denominaremos un orden de complejidad. Habitualmente estos conjuntos se denominan O, existiendo una infinidad de ellos; pero nos centraremos en unos pocos de uso frecuente.

Para cada uno de estos conjuntos se suele identificar un miembro f(n) que se utiliza como representante de la clase.

conjuntos u órdenes de complejidad	
O (1)	orden constante
O (log n)	orden logarítmico
O (n)	orden lineal
O (n log n)	
O (n^2)	orden cuadrático
O (n^a)	orden polinomial (a > 2)
O (a^n)	orden exponencial (a > 1)
O (n!)	orden factorial

La definición matemática de estos conjuntos debe ser muy cuidadosa para involucrar los dos aspectos antes comentados:

- identificación de una familia y
- posible utilización como cota superior de otras funciones menos malas.

Dícese que el conjunto O(f(n)) es el de las funciones de orden de f(n), que se define como

O(f(n)) = { g: INTEGER → REAL+
tales que existen las constantes K y M

tales que para todo $N > M$, $g(N) \leq K * f(N)$ }

o, dicho de otra forma,

$$O(f(n)) = \{ g(n), \lim_{n\to\infty} \left(\frac{g(n)}{f(n)}\right) < \infty \}$$

en palabras,

> O(f(n)) está formado por aquellas funciones g(n) que crecen a un ritmo menor o igual que el de f(n).

De las funciones "g" que forman este conjunto O(f(n)) se dice que "están dominadas asintóticamente" por "f", en el sentido de que, para N suficientemente grande, y salvo una constante multiplicativa "K", f(n) es una cota superior de g(n).

Si un algoritmo A se puede demostrar de un cierto orden O(…), es cierto que también pertenece a todos los órdenes superiores (la relación de orden *"cota superior de"* es transitiva); pero en la práctica lo útil es encontrar la "menor cota superior", es decir el menor orden de complejidad que lo cubra.

2.5 Interpretación práctica

La siguiente tabla muestra el comportamiento de diferentes funciones para valores crecientes de n. Observe cómo las funciones de mayor orden de complejidad crecen de forma más desmesurada:

n	O(1)	O(log n)	O(n)	O(n log n)	O(n^2)	O(n^5)	O(5^n)	O(n!)
10	1	2	10	23	100	1e+05	1e+07	4e+06
20	1	3	20	60	400	3e+06	1e+14	2e+18
30	1	3	30	102	900	2e+07	9e+20	3e+32
40	1	4	40	148	1,600	1e+08	9e+27	8e+47
50	1	4	50	196	2,500	3e+08	9e+34	3e+64
60	1	4	60	246	3,600	8e+08	9e+41	8e+81
70	1	4	70	297	4,900	2e+09	8e+48	1e+100
80	1	4	80	351	6,400	3e+09	8e+55	7e+118
90	1	4	90	405	8,100	6e+09	8e+62	1e+138
100	1	5	100	461	10,000	1e+10	8e+69	9e+157

La misma tabla puede resultar más intuitiva si hablamos de tiempos medidos en microsegundos:

n	O(1)	O(lg n)	O(n)	O(n lg n)	$O(n^2)$	$O(n^5)$	$O(5^n)$	O(n!)
10	1µs	2µs	10µs	23µs	100µs	100ms	10s	4s
20	1µs	3µs	20µs	60µs	400µs	3s	3a	63 mil años
30	1µs	3µs	30µs	102µs	900µs	20s	28 millones de años	
40	1µs	4µs	40µs	148µs	1,6ms	100s		
50	1µs	4µs	50µs	196µs	2,5ms	300s		
60	1µs	4µs	60µs	246µs	3,6ms	800s		
70	1µs	4µs	70µs	297µs	4,9ms	33m		
80	1µs	4µs	80µs	351µs	6,4ms	50m		
90	1µs	4µs	90µs	405µs	8,1ms	1h40m		
100	1µs	5µs	100µs	461µs	10ms	2h46m		

Disculpe el lector que no hayamos calculado los números más grandes; pero es que no tenemos palabras para tiempos tan fuera de nuestra capacidad de imaginación.

Claramente esos programas son inviables a todos los efectos prácticos.

3 Propiedades de los conjuntos O (f)

No entraremos en muchas profundidades, ni en demostraciones, que se pueden hallar en los libros especializados. No obstante, algo hay que saber de cómo se trabaja con los conjuntos O() para poder evaluar los algoritmos con los que nos encontremos.

Para simplificar la notación, usaremos O(f) para decir O(f(n)).

Las primeras reglas sólo expresan matemáticamente el concepto de jerarquía de órdenes de complejidad:

A. La relación de orden definida por

$f < g \Leftrightarrow f(n) \in O(g)$

es reflexiva: $f(n) \in O(f)$

y transitiva: $f(n) \in O(g)$ y $g(n) \in O(h) \Rightarrow f(n) \in O(h)$

B. $f \in O(g)$ y $g \in O(f) \Leftrightarrow O(f) = O(g)$

Las siguientes propiedades se pueden utilizar como reglas para el cálculo de órdenes de complejidad. Toda la maquinaria matemática para el cálculo de límites se puede aplicar directamente:

C. $\lim_{n\to\infty}\left(\frac{f(n)}{g(n)}\right) = 0 \Rightarrow f \in O(g)$

$\Rightarrow g \notin O(f)$

$\Rightarrow O(f) \subset O(g)$

D. $\lim_{n\to\infty}\left(\frac{f(n)}{g(n)}\right) = K \Rightarrow f \in O(g)$

$\Rightarrow g \in O(f)$

$\Rightarrow O(f) = O(g)$

E. $\mathrm{L}\lim_{n\to\infty}\left(\frac{f(n)}{g(n)}\right) = \infty \Rightarrow f \notin O(g)$

$\Rightarrow g \in O(f)$

$\Rightarrow O(f) \supset O(g)$

Las que siguen son reglas habituales en el cálculo de límites:

F. Si $f, g \in O(h) \Rightarrow f + g \in O(h)$

G. Sea k una constante, $f(n) \in O(g) \Rightarrow k * f(n) \in O(g)$

H. Si $f \in O(h1)$ y $g \in O(h2) \Rightarrow f + g \in O(h1+h2)$

I. Si $f \in O(h1)$ y $g \in O(h2) \Rightarrow f * g \in O(h1*h2)$

J. Sean los reales $0 < a < b \Rightarrow O(n^a)$ es subconjunto de $O(n^b)$

K. Sea P (n) un polinomio de grado k $\Rightarrow P(n) \in O(n^k)$

L. Sean los reales a, b > 1 $\Rightarrow$ O (log_a) = O (log_b)

La regla [L] nos permite olvidar la base en la que se calculan los logaritmos en expresiones de complejidad.

La combinación de las reglas [K, G] es probablemente la más usada, permitiendo de un plumazo olvidar todos los componentes de un polinomio, menos su grado.

Por último, la regla [H] es la básica para analizar el concepto de secuencia en un programa: la composición secuencial de dos trozos de programa es de orden de complejidad el de la suma de sus partes.

4 Reglas Prácticas

Aunque no existe una receta que siempre funcione para calcular la complejidad de un algoritmo, si es posible tratar sistemáticamente una gran cantidad de ellos, basándonos en que suelen estar bien estructurados y siguen pautas uniformes.

Los algoritmos bien estructurados combinan las sentencias de alguna de las formas siguientes

4.1 Sentencias sencillas

Nos referimos a las sentencias de asignación, entrada/salida, etc. siempre y cuando no trabajen sobre variables estructuradas cuyo tamaño esté relacionado con el tamaño N del problema. La inmensa mayoría de las sentencias de un algoritmo requieren un tiempo constante de ejecución, siendo su complejidad O(1).

4.2 Secuencia (;)

La complejidad de una serie de elementos de un programa es del orden de la suma de las complejidades individuales, aplicándose las operaciones arriba expuestas.

4.3 Decisión (if)

La condición suele ser de O(1), complejidad a sumar con la peor posible, bien en la rama THEN, o bien en la rama ELSE. En decisiones múltiples (ELSIF, CASE), se tomará la peor de las ramas.

4.4 Bucles

En los bucles con contador explícito, podemos distinguir dos casos: que el tamaño N forme parte de los límites o que no. Si el bucle se realiza un número fijo de veces, independiente de N, entonces la repetición sólo introduce una constante multiplicativa que puede absorberse.

```
for (int i= 0; i < K; i++)
  algo_de_O(1);
```

$\Rightarrow K * O(1) = O(1)$

Si el tamaño N aparece como límite de iteraciones, tenemos varios casos:

caso 1:

```
for (int i= 0; i < N; i++)
  algo_de_O(1);
```

$\Rightarrow N * O(1) = O(n)$

caso 2:

```
for (int i= 0; i < N; i++)
 for (int j= 0; j < N; j++)
   algo_de_O(1);
```

$\Rightarrow N * N * O(1) = O(n^2)$

caso 3:

```
for (int i= 0; i < N; i++)
 for (int j= 0; j < i; j++)
   algo_de_O(1);
```

el bucle exterior se realiza N veces, mientras que el interior se realiza 1, 2, 3, ... N veces respectivamente. En total, tenemos la suma de una serie aritmética:

$1 + 2 + 3 + ... + N = N * (1+N) / 2 \rightarrow O(n^2)$

A veces aparecen bucles multiplicativos, donde la evolución de la variable de control no es lineal (como en los casos anteriores)

```
int c = 1;
while (c < N) {
 algo_de_O(1);
 c*= 2;
}
```

El valor inicial de "c" es 1, siendo "2^k" al cabo de "k" iteraciones. El número de iteraciones es tal que

$2^k \geq N \Rightarrow k = \lceil \log_2(N) \rceil$

$\lceil x \rceil$ es el entero inmediato superior a x

y, por tanto, la complejidad del bucle es O(log n).

```
c = N;
while (c > 1) {
 algo_de_O(1);
 c/= 2;
}
```

Un razonamiento análogo nos lleva a log(N) iteraciones y, por tanto, a un orden O(log n) de complejidad.

```
for (int i = 0; i < N; i++) {
 c = i;
 while (c > 0) {
   algo_de_O(1);
   c/= 2;
 }
}
```

tenemos un bucle interno de orden O(log n) que se ejecuta N veces, luego el conjunto es de orden O(n log n).

4.5 Llamadas a procedimientos

La complejidad de llamar a un procedimiento viene dada por la complejidad del contenido del procedimiento en sí. El coste de llamar no es sino una constante que podemos obviar inmediatamente dentro de nuestros análisis asintóticos.

El cálculo de la complejidad asociada a un procedimiento puede complicarse notablemente si se trata de procedimientos recursivos. Es fácil que tengamos que aplicar técnicas propias de la matemática discreta, tema que queda fuera de los límites de esta nota técnica.

5 Relaciones de recurrencia

En el cálculo de la complejidad de un algoritmo a menudo aparecen expresiones recursivas para estimar el tiempo que se tarda en procesar un problema de un cierto tamaño N.

Por ejemplo, podemos encontrarnos con un problema que para resolver un problema de tamaño N resuelve 2 problemas de tamaño N/2 y luego combina las soluciones parciales con una operación de complejidad O(n).

La relación de recurrencia es

$T(n) = 2\ T(n/2) + O(n)$

$T(1) = O(1)$

Y el problema es cómo deducir a partir de esa definición la complejidad del algoritmo completo.

Para resolverlo, vamos a ir aplicando pasos sucesivos de recursión

$T(n) = 2\ T(n/2) + n$

$T(n) = 2\ (2\ T(n/4) + n/2) + n = 4\ T(n/4) + 2n$

$T(n) = 8\ T(n/8) + 3n$

… hasta que veamos un patrón …

$T(n) = 2^k\ T(n/2^k) + k\ n$

Esta fórmula es válida siempre, en particular cuando $n/2^k = 1$ y podemos escribir

$T(n) = 2^k\ T(1) + k\ n$

y como sabemos la condición de parada de la recursión

$T(n) = 2^k + k\ n$

El valor de k que termina la recursión es

$n/2^k = 1 \Rightarrow k = \log_2(n)$

sustituyendo

$T(n) = 2^{\log_2(n)} + \log_2(n)\ n$

$T(n) = n + n \log(n) \Rightarrow O(n \log(n))$

5.1 Relaciones de recurrencia habituales

El método descrito permite resolver numerosas relaciones de recurrencia. Algunas son muy habituales:

relación	complejidad	ejemplos
$T(n) = T(n/2) + O(1)$	$O(\log n)$	búsqueda binaria
$T(n) = T(n-1) + O(1)$	$O(n)$	búsqueda lineal factorial bucles for, while
$T(n) = 2\ T(n/2) + O(1)$	$O(n)$	recorrido de árboles binarios: preorden, en orden, postorden
$T(n) = 2\ T(n/2) + O(n)$	$O(n \log n)$	ordenación rápida (*quick sort*)
$T(n) = T(n-1) + O(n)$	$O(n^2)$	ordenación por selección ordenación por burbuja

$T(n) = 2\,T(n-1) + O(1)$	$O(2^n)$	torres de hanoi

6 Ejemplo: evaluación de un polinomio

Vamos a aplicar lo explicado hasta ahora a un problema de fácil especificación: diseñar un programa para evaluar un polinomio $P(x)$ de grado N. Sea coef el vector de coeficientes:

```
double evaluaPolinomio1(double[] coef, double x) {
     double total = 0;
     for (int i = 0; i < coef.length; i++) {
          // bucle 1
          double xn = 1.0;
          for (int j = 0; j < i; j++)
               // bucle 2
               xn *= x;
          total += coef[i] * xn;
     }
     return total;
}
```

Como medida del tamaño tomaremos para N el grado del polinomio, que es el número de coeficientes. Así pues, el bucle más exterior (1) se ejecuta N veces. El bucle interior (2) se ejecuta, respectivamente

$$1 + 2 + 3 + \ldots + N \text{ veces} = N * (1+N) / 2 \Rightarrow O(n^2)$$

Intuitivamente, sin embargo, este problema debería ser menos complejo, pues repugna al sentido común que sea de una complejidad tan elevada. Se puede ser más inteligente a la hora de evaluar la potencia x^n:

```
double potencia(double x, int y) {
     double t;
     if (y == 0)
          return 1.0;
     if (y % 2 == 1)
          return x * potencia(x, y - 1);
     else {
          t = potencia(x, y / 2);
          return t * t;
     }
}

double evaluaPolinomio2(double[] coef, double x) {
     double total = 0;
     for (int i = 0; i < coef.length; i++)
          total += coef[i] * potencia(x, i);
```

```
        return total;
    }
```

El análisis del método potencia es delicado, pues si el exponente es par, el problema tiene una evolución logarítmica; mientras que, si es impar, su evolución es lineal. No obstante, como si "j" es impar entonces "j-1" es par, el caso peor es que en la mitad de los casos tengamos "j" impar y en la otra mitad sea par. El caso mejor, por contra, es que siempre sea "j" par.

Un ejemplo de caso peor sería x ^ 31, que implica la siguiente serie para j:

31 30 15 14 7 6 3 2 1

cuyo número de términos podemos acotar superiormente por

$2 * \lceil \log_2 (j) \rceil$

donde $\lceil r \rceil$ es el entero inmediatamente superior (este cálculo responde al razonamiento de que en el caso mejor visitaremos $\lceil \log_2 (j) \rceil$ valores pares de "j"; y en el caso peor podemos encontrarnos con otros tantos números impares entremezclados).

Por tanto, la complejidad de potencia es de orden O(log n).

Otra forma de calcularlo es replantear la función recursiva como

```
double potencia(double x, int y) {
    if (y == 0)
        return 1.0;
    if (y == 1)
        return x;
    if (y % 2 == 1)
        return x * potencia(x * x, y / 2);
    else
        return potencia(x * x, y / 2);
}
```

Visto así, tenemos una función de recurrencia

$T(n) = O(1) + T(n/2)$

cuya solución es

$O(\log n)$

Insertado el método auxiliar potencia en el método evaluaPolinomio, la complejidad compuesta es del orden O(n log n), al multiplicarse por N un sub-algoritmo de O(log n).

Así y todo, esto sigue resultando extravagante y excesivamente costoso. En efecto, basta reconsiderar el algoritmo almacenando las potencias de "X" ya calculadas para mejorarlo sensiblemente:

```
double evaluaPolinomio3(double[] coef, double x) {
```

```
        double xn = 1;
        double total = coef[0];
        for (int i = 1; i < coef.length; i++) {
             xn *= x;
             total += coef[i] * xn;
        }
        return total;
   }
```

que queda en un algoritmo de O(n).

Habiendo N coeficientes distintos, es imposible encontrar ningún algoritmo de un orden inferior de complejidad.

En cambio, si es posible encontrar otros algoritmos de idéntica complejidad:

```
double evaluaPolinomio4(double[] coef, double x) {
     double total = 0;
     for (int i = coef.length - 1; i >= 0; i--)
          total = total * x + coef[i];
     return total;
}
```

No obstante ser ambos algoritmos de idéntico orden de complejidad, cabe resaltar que sus tiempos de ejecución pudieran ser notablemente distintos. En efecto, mientras el último algoritmo ejecuta N multiplicaciones y N sumas, el penúltimo requiere 2N multiplicaciones y N sumas. Si el tiempo de ejecución es notablemente superior para realizar una multiplicación, cabe razonar que el último algoritmo ejecutará en la mitad de tiempo que el anterior, aunque eso no cambie el orden de complejidad.

La tabla siguiente muestra algunos tiempos de ejecución. Para calcularlo, se han utilizado unos coeficientes aleatorios, y se ha ejecutado cada método 10.000 veces:

N =	**10**	**100**	**500**	**1.000**
evaluaPolinomio1	4ms	138ms	3.306ms	12.221ms
evaluaPolinomio2	8ms	57ms	474ms	1.057ms
evaluaPolinomio3	2ms	6ms	15ms	26ms
evaluaPolinomio4	2ms	9ms	21ms	38ms

Nótese que para polinomios pequeños todos los algoritmos son equivalente, mientras que al crecer el número de coeficientes, van empeorando según lo previsto. También puede observarse que entre los algoritmos 3 y 4 la diferencia es contraria a lo previsto, pudiendo pensarse que el ordenador tiene un co-procesador matemático que multiplica a la misma velocidad que suma y que el compilador genera mejor código para el algoritmo 3 que para el 4. Esto son sólo conjeturas.

7 Cálculo de números de Fibonacci

La serie de Fibonacci es una serie de números naturales muy conocida y estudiada. Su definición más habitual es

fib(1) = 1

fib(2) = 1

fib(n) = fib(n-1) + fin(n-2), para n > 2

7.1 Algoritmo recursivo

Si aplicamos ciegamente la definición matemática:

```
int fibo1(int n) {
    if (n < 2)
        return 1;
    else
        return fibo1(n - 1) + fibo1(n-2);
}
```

Calcular su complejidad no es fácil, pues la función del tiempo de ejecución en función del valor de n es

$T(n) = T(n-1) + T(n-2)$

Podemos hacer una hipótesis e intentar corroborarla.

Supongamos que T(N) responde a una función del tipo

$T(n) = k\,a^n$

entonces debe satisfacerse que

$k\,a^n = k\,a^{n-1} + k\,a^{n-2}$

simplificando

$a^2 = a + 1$

de donde se despeja

$a = \frac{1+\sqrt{5}}{2} \sim 1{,}618$ [2]

o sea

$T(n) \sim 1{,}6^n$

o, en otras palabras,

$T(n) \in O(1{,}6^n)$

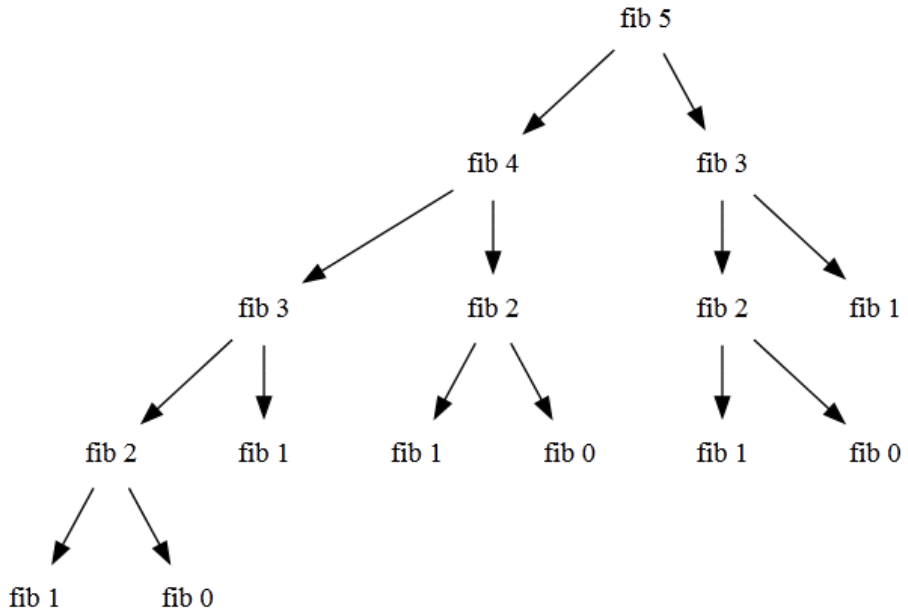
En cuanto al espacio requerido en RAM, baste observar que el algoritmo profundiza por dos ramas, n-1 y n-2, siendo n-1 la mayor, lo que lleva a una profundidad de recursión de n pasos (puede ver un diagrama en la siguiente sección). En consecuencia

$E(n) \in O(n)$

[2] Es precisamente la proporción áurea.

7.2 Algoritmo recursivo con memoria

El algoritmo anterior es un poco 'tonto' porque recalcula el mismo valor una y otra vez

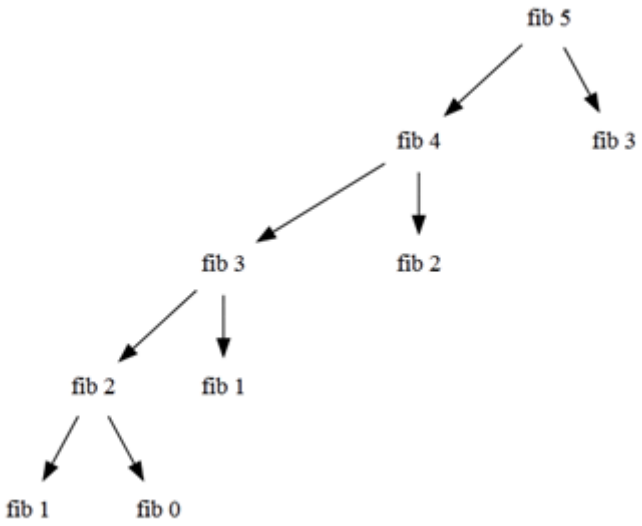


Esto lo podemos arreglar si vamos memorizando (cache) los datos que vamos calculando y antes de calcular un nuevo dato miramos si ya lo tenemos memorizado:

```
Map<Integer, Integer> memoria =
    new HashMap<Integer, Integer>();

int fibo2(int n) {
    if (n < 2)
        return 1;
    Integer resultado = memoria.get(n);
    if (resultado != null)
        return resultado;
    resultado = fibo2(n - 1) + fibo2(n - 2);
    memoria.put(n, resultado);
    return resultado;
}
```

Así visto, reducimos radicalmente el grafo de llamadas. En la primera bajada vamos calculando y luego reutilizamos



De forma que el programa se convierte en una simple secuencia de llamadas

$$T(n) \in O(n)$$

No obstante, cabe destacar que hemos introducido un coste en espacio en memoria RAM: los valores memorizados. En términos de memoria RAM necesaria, este programa también requiere O(n) datos memorizados. Comparado con fibo1, esta forma de programarlo en vez de usar la RAM para guardar llamadas recursivas la utiliza para memorizar datos.

Cabe destacar que esta técnica de memorizar datos se puede aplicar en muchísimos casos pues es muy general.

7.3 Algoritmo con memoria limitada

Una variante del algoritmo con memoria es memorizar sólo los datos que más se repiten de forma que el espacio requerido en RAM es fijo e independiente de N:

```
private static int[] tabla = new int[20];

public static int fibo3(int n) {
    if (n < 2)
        return 1;
    if (n < tabla.length) {
        int resultado = tabla[n];
        if (resultado > 0)
            return resultado;
        resultado = fibo3(n - 1) + fibo3(n - 2);
        tabla[n] = resultado;
        return resultado;
    }
    return fibo3(n - 1) + fibo3(n - 2);
}
```

El efecto en la complejidad del tiempo de ejecución es difícil de calcular pues depende del tamaño de la memoria respecto de los valores N que necesitemos.

7.4 Algoritmo iterativo

Si pensamos en fin(n) como el n-ésimo término de la serie de Fibonacci

1 1 2 3 5 8 13 21 34 …

podemos programarlo como

```
int fibo4(int n) {
    int n0= 1;
    int n1= 1;
    for (int i= 2; i <= n; i++) {
        int ni= n0 + n1;
        n0=n1;
        n1= ni;
    }
    return n1;
}
```

En este caso tenemos un simple bucle

$$T(n) \in O(n)$$

En cuanto a memoria RAM, el espacio ocupado es constante; o sea

$$E(n) \in O(1)$$

7.5 Algoritmo directo

Se conoce la fórmula de Binet (1786 – 1856)

$$fib(n) = \frac{(1+\sqrt{5})^n - (1-\sqrt{5})^n}{2^n \sqrt{5}}$$

que podemos programar como

```
static final double SQRT_5 = Math.sqrt(5);

int fibo5(int n) {
    if (n < 2)
        return 1;
    n += 1;
    double t1 = Math.pow((1 + SQRT_5) / 2, n);
    double t2 = Math.pow((1 - SQRT_5) / 2, n);
    double t = (t1 - t2) / SQRT_5;
    return (int) Math.round(t);
}
```

lo que nos deja un algoritmo de tiempo constante

$$T(n) \in O(1)$$

En cuanto a memoria RAM, el espacio ocupado es constante; o sea

$$E(n) \in O(1)$$

7.6 Datos experimentales

Realizamos una serie de medidas experimentales (ver sección 9 más adelante). Obtenemos estos resultados, que parecen bastante explícitos de las mejoras de cada caso:

N	fibo1	fibo2	fibo3	fibo4	fibo5
10	812.001	2.020.159	699.112	6.569	158.460
20	3.397.022	4.701.252	723.332	6.979	154.766
30	392.534.843	5.567.034	8.739.927	6.979	153.945
40		6.339.219	57.582.944	7.800	153.944
50		6.283.388	2.947.764.873	8.621	153.945

Analizando estos datos, no olvide que la estructura de memoria, fibo2(), es mucho más pesada que la simple tabla en forma de array, fibo3(). Es por eso que, para valores de N por debajo del tamaño de la memoria, fibo3() es preferible a fibo2(),

También puede verse cómo se dispara la ejecución de fibo3() para valores de N superiores a 20 que es donde se termina la tabla de memoria prevista en el ejemplo usado. Nótese que tenemos que elegir: o ponemos más memoria o tardamos más tiempo en ejecutar. Estos equilibrios entre tiempo y RAM son frecuentes en la práctica.

Nótese también el efecto del cálculo con números reales en fibo5. El coste del cálculo es constante, O(1), pero costoso. Este número depende fuertemente de la capacidad de la máquina de pruebas para trabajar con números reales.

Por último, podríamos intentar validar experimentalmente la supuesta complejidad $O(1{,}6^n)$ del algoritmo fib1(). Para ello calcularemos los valores del cociente entre el tiempo medido T(N) y la función de referencia $1{,}6^n$:

N	T(N)	1,618^n	T(N)/1,618^n
10	33.662	123	274
12	72.661	322	226
14	197.868	843	235
16	626.448	2.206	284
18	1.627.697	5.776	282
20	3.227.889	15.121	213
22	8.685.293	39.585	219
24	22.499.166	103.630	217
26	56.970.218	271.295	210
28	151.812.614	710.229	214

donde podemos ver que el cociente f(n)/g(n) parece tender a un valor constante que ni es 0 ni es infinito, como queríamos mostrar.

8 Problemas P, NP y NP-completos

Hasta aquí hemos venido hablando de algoritmos. Cuando nos enfrentamos a un problema concreto, habrá una serie de algoritmos aplicables. Se suele decir que el orden de complejidad de un problema es el del mejor algoritmo que se conozca para

resolverlo. Así se clasifican los problemas, y los estudios sobre algoritmos se aplican a la realidad.

Estos estudios han llevado a la constatación de que existen problemas muy difíciles, problemas que desafían la utilización de los ordenadores para resolverlos. En lo que sigue esbozaremos las clases de problemas que hoy por hoy se escapan a un tratamiento informático.

Clase P.-

Los algoritmos de complejidad polinómica se dice que son tratables en el sentido de que suelen ser abordables en la práctica. Los problemas para los que se conocen algoritmos con esta complejidad se dice que forman la clase P.

Aquellos problemas para los que la mejor solución que se conoce es de complejidad superior a la polinómica, se dice que son problemas intratables. Sería muy interesante encontrar alguna solución polinómica (o mejor) que permitiera abordarlos.

Clase NP.-

Algunos de estos problemas intratables pueden caracterizarse por el curioso hecho de que puede aplicarse un algoritmo polinómico para comprobar si una posible solución es válida o no. Esta característica lleva a un método de resolución no determinista consistente en aplicar heurísticos para obtener soluciones hipotéticas que se van desestimando (o aceptando) a ritmo polinómico.

Los problemas de esta clase se denominan NP (la N de no-deterministas y la P de polinómicos).

Ejemplo.

Dado un conjunto de enteros, averiguar si existe un subconjunto tal que la suma de sus miembros sea 0.

Sea el conjunto

{ -2, -3, 15, 14, 7, -10 }

la mejor forma que se conoce para resolver el caso es ir probando todos los subconjuntos posibles hasta encontrar alguno que sume 0. Si se nos acaban las combinaciones y no lo encontramos la respuesta es NO.

En el ejemplo anterior, la respuesta es SÍ porque existe ese subconjunto

{ -2, -3, -10, 15 }

Clase NP-completos.-

Se conoce una amplia variedad de problemas de tipo NP, de los cuales destacan algunos de ellos de extrema complejidad. Gráficamente podemos decir que algunos problemas se hayan en la "frontera externa" de la clase NP. Son problemas NP, y son los peores problemas posibles de clase NP. Estos problemas se caracterizan por ser todos "iguales" en el sentido de que, si se descubriera una solución P para alguno de ellos, esta solución sería fácilmente aplicable a todos ellos. Actualmente hay un premio de prestigio equivalente al

Nobel reservado para el que descubra semejante solución ... ¡y se duda seriamente de que alguien lo consiga!

Es más, si se descubriera una solución para los problemas NP-completos, esta sería aplicable a todos los problemas NP y, por tanto, la clase NP desaparecería del mundo científico al carecerse de problemas de ese tipo. Realmente, tras años de búsqueda exhaustiva de dicha solución, es hecho ampliamente aceptado que no debe existir, aunque nadie ha demostrado, todavía, la imposibilidad de su existencia.

9 Análisis experimental

A veces no es evidente cómo analizar el código para aplicar reglas matemáticas y calcular la complejidad de un programa y una porción de programa.

1. porque no tenemos el código fuente (por ejemplo, usamos una librería de otros)
2. porque hacemos un uso combinado de varias partes de un objeto; por ejemplo, cuando tenemos una estructura de datos en la que hacemos varias operaciones y el rendimiento global depende de la proporción de operaciones de cada clase
3. etc.

En estas circunstancias podemos recurrir al método experimental:

1. programamos el código
2. lo ponemos a ejecutar con varios valores de N
3. tabulamos los tiempos medidos en función n, n^2, n log(n), etc.
4. analizamos a qué grupo de complejidad se ajusta mejor

A fin de hacer medidas tenemos que tener en cuenta varios aspectos prácticos

1. realizar la operación un número elevado de veces, con datos aleatorios, para quedarnos con un valor promedio o tendencia
2. [en el caso de java] ejecutar el programa una serie de veces antes de medir para asegurarnos de que el compilador ha generado código optimizado
   (ver JIT – *Java just in time compiler optimization*)

Supongamos que tenemos el programa que queremos medir encapsulado en una clase java

Datos.java

```
import java.util.Random;

public class Datos {

    public Datos(int n) {
        // crea un conjunto de datos de tamano N
    }

    public void algoritmo() {
        // aplica el algoritmo a los datos
    }
}
```

El esqueleto de un banco de pruebas puede ser algo así

```
public static void main(String[] args) {
    for (int i = 0; i < 500; i++) {
        int N = 100;
        Datos datos = new Datos(N);
        datos.algoritmo();
    }

    int[] tamanos = {100, 200, 500, 1000, 2000, 5000, 10000};
    int casos = 1000;
    for (int N : tamanos) {
        long t0 = System.currentTimeMillis();
        for (int i = 0; i < casos; i++) {
            Datos datos = new Datos(N);
            datos.algoritmo();
        long t2 = System.currentTimeMillis();
        System.out.printf("N= %6d: %,5dms%n", N, (t2 - t0));
    }
}
```

Si no es relevante saber si son milisegundos o cualquier otra magnitud, podemos recurrir a la máxima precisión del reloj del ordenador de pruebas y simplemente comparar la evolución de tiempos:

```
public static void main(String[] args) {
    for (int i = 0; i < 500; i++) {
        int N = 100;
        Datos datos = new Datos(N);
        datos.algoritmo();
    }

    int[] tamanos = {100, 200, 500, 1000, 2000, 5000, 10000};
    int casos = 1000;
    for (int N : tamanos) {
        long t0 = System.nanoTime ();
        for (int i = 0; i < casos; i++) {
            Datos datos = new Datos(N);
            datos.algoritmo();
        long t2 = System.nanoTime ();
        System.out.printf("%d:%d%n", N, (t2 - t0));
    }
}
```

Un punto que suele ser delicado es el constructor de Datos pues al estar dentro del bucle de medición estaremos midiendo tanto el tiempo de preparación de los datos de prueba como el tiempo de ejecución del algoritmo.

9.1 Ejemplo

Queremos medir la función de ordenación de arrays de la librería de java. Para ello necesitaremos arrays de tamaño N con datos aleatorios.

Para evitar medir los tiempos de creación del array y de carga de valores aleatorios, podemos precalcular un cierto número de datos aleatorios y luego usar una fracción de N de ellos. Algo así

```
import java.util.Arrays;
import java.util.Random;

public class Datos {
    public static final int MAX = 1000000;
    private static final Random RANDOM;
    private static final int[] datosAleatorios;
    private static final int[] datosPrueba;
    private final int N;

    static {
        RANDOM = new Random();
        datosAleatorios  = new int[MAX];
        for (int i = 0; i < MAX; i++)
            datosAleatorios[i] = RANDOM.nextInt();
        datosPrueba  = new int[MAX];
    }

    public Datos(int N) {
        this.N = N;
        int n0 = RANDOM.nextInt(MAX - N);
        System.arraycopy(datosAleatorios, n0, datosPrueba, 0, N);
    }

    public void algoritmo() {
        Arrays.sort(datosPrueba, 0, N);
    }
}
```

Ejecutando las pruebas obtenemos estas medidas:

```
100 5099023
200 10049849
500 27268133
1000 59154985
2000 124892599
5000 342082804
10000 729927088
20000 1546868027
50000 4161772150
100000 8817641280
200000 18540020616
```

```
500000 49852784330
```

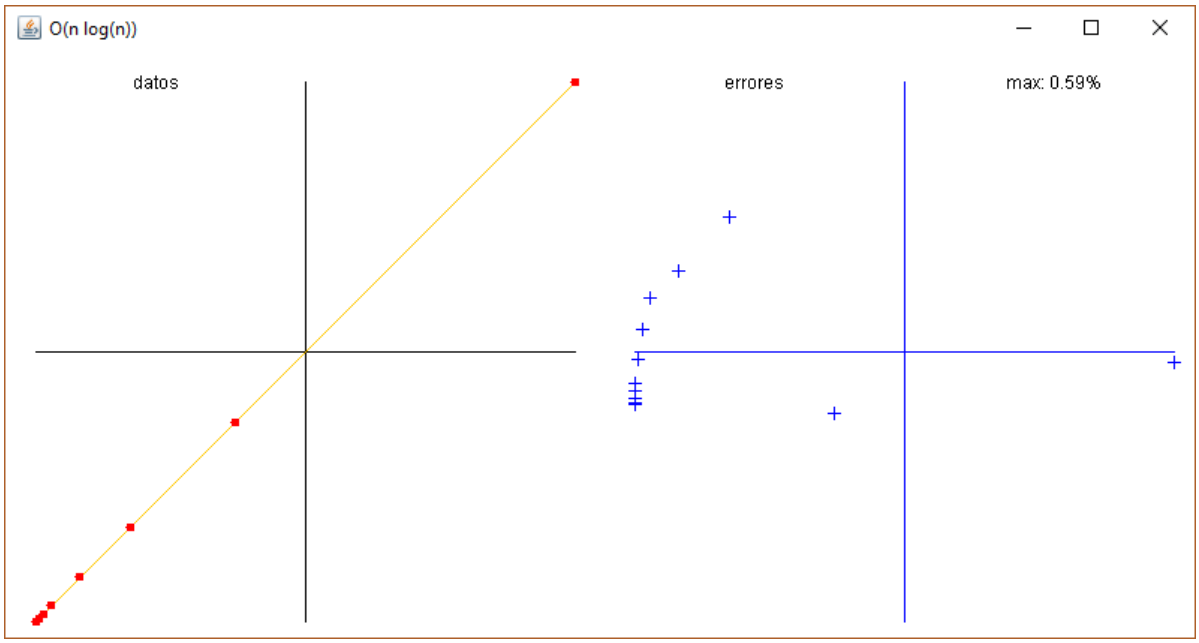
que, analizadas por regresión, nos ofrecen estas sugerencias:

Correlator (2.3.2016)

```
100 5099023
200 10049849
500 27268133
1000 59154985
2000 124892599
5000 342082804
10000 729927088
20000 1546868027
50000 4161772150
100000 8817641280
200000 18540020616
500000 49852784330
```

complejidad	a	b	r^2
O(log(n))	3.7e+09	-2.5e+10	0.49
O(n)	9.9e+04	-3.3e+08	1.0
O(n log(n))	7.6e+03	2.1e+07	1.0
$O(n^a)$	1.1	3.3e+04	1.0
$O(a^n)$	1.0	1.6e+08	0.51

Veamos O(n log n):



que parece corroborar que el algoritmo que tiene java implementado es O(n log n). La siguiente sección comenta esta conclusión.

9.2 Estimación de la complejidad

Aunque el grado de complejidad no se puede precisar experimentalmente, si podemos hacer alguna estimación al respecto o, para ser más precisos, podemos usar pruebas experimentales para corroborar o refutar una hipótesis.

A continuación, se presentan someramente diferentes técnicas experimentales, no excluyentes.

9.2.1 Opción 1

Vimos anteriormente una forma de hacerlo al hablar de que el cálculo de números de Fibonacci es $O(1{,}6^n)$. Para corroborar la hipótesis calculamos la evolución en función de N de la proporción

$$\frac{tiempo\ de\ ejecución\ medido}{1{,}6^n}$$

y constatamos experimentalmente que ese cociente se mantiene constante.

9.2.2 Opción 2

Si sospechamos que un código se ajusta a $O(n^2)$, la gráfica que presenta los tiempos en función de n^2 debe ser una línea recta. Si no es una línea recta, es que no es de $O(n^2)$.

9.2.3 Opción 3

Otro ejemplo. Si sospechamos que sea de orden polinómico (n^p) pero no estamos seguros del exponente, podemos dibujar el tiempo en función del log(n) y calcular la pendiente de la línea: esa pendiente es el valor experimental de p.

9.2.4 Caso práctico

Veamos un caso práctico. Hay muchos algoritmos de ordenación, analizados en muchos libros, que nos dicen su complejidad.

algoritmo	**complejidad**
inserción	$O(n^2)$
selección	$O(n^2)$
burbuja	$O(n^2)$
quick sort	O(n log(n))

Supongamos que codificamos estos algoritmos y medimos tiempos de ejecución para varios valores de n:

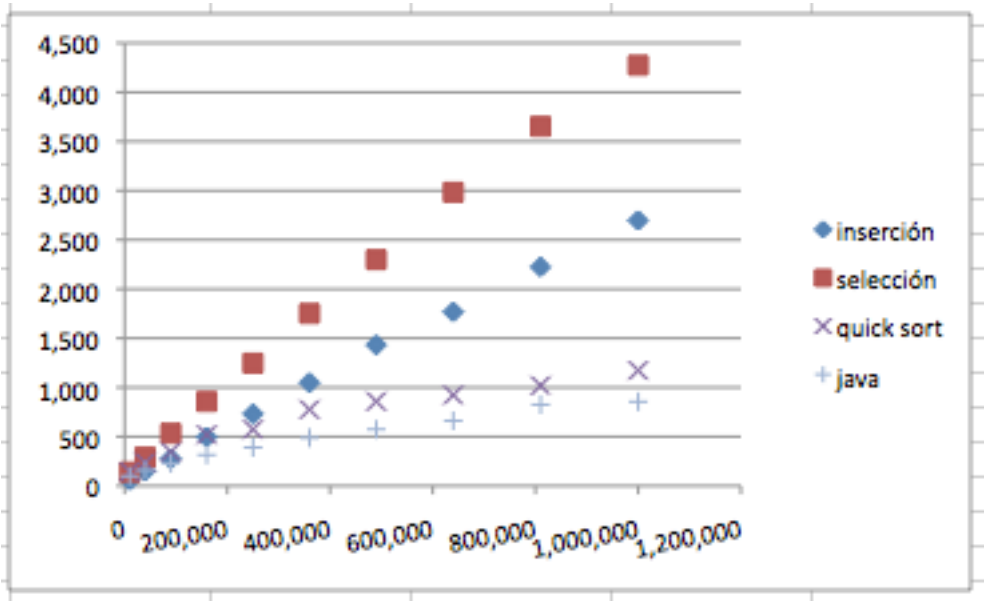
n	inserción	selección	burbuja	quick sort	java
100	57	140	380	144	99
200	152	291	1.399	235	159
300	277	534	3.202	345	230
400	496	858	5.711	520	315
500	732	1.247	8.660	574	390
600	1.048	1.752	12.694	776	489
700	1.432	2.299	17.572	858	574
800	1.768	2.984	22.751	923	661
900	2.224	3.653	28.644	1.019	825
1.000	2.697	4.274	34.010	1.173	855

Donde la última columna es un programa java cuyo código fuente no tenemos para analizar y queremos descubrir su complejidad.

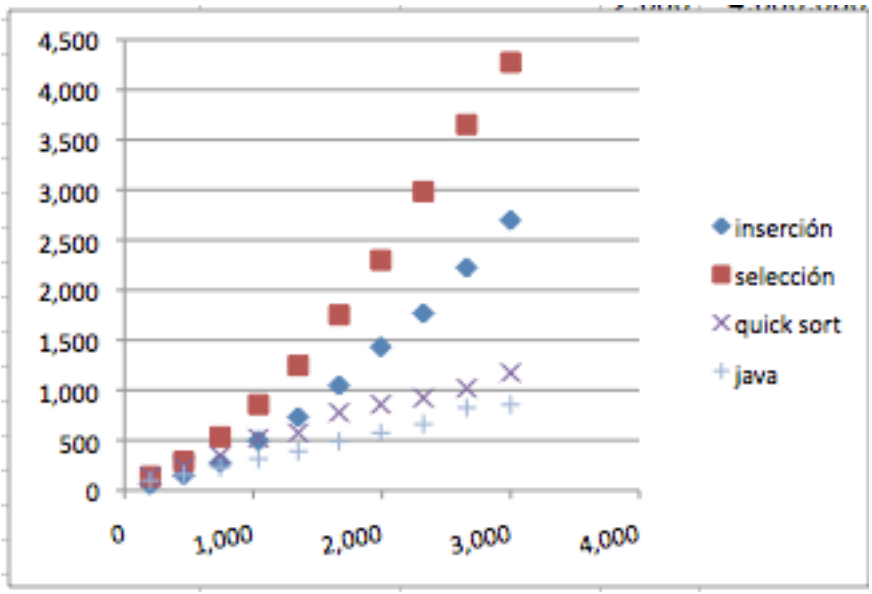
Para analizar experimentalmente si unos tiempos se ajustan a un orden de complejidad, hacemos una transformación de variable en el eje X:

orden	cambio de variable	
O(log n)	x = log(n)	y = t
O(n)	x = n	y = t
O(n log n)	x = n * log(n)	y = t
$O(n^2)$	x = n * n	y = t

Si representamos y = t, $x = n^2$, tendremos algo así:



Y si representamos y = t, x = n.log(n):



donde vemos que los algoritmos de inserción y selección no se ajustan a un O(n log(n)) pero sí se ajustan razonablemente los algoritmos *quick sort* y el desconocido.

No olvide que los órdenes de magnitud son cotas superiores cuando N tiende a infinito. Los datos experimentales deben satisfacer esa cota; pero a veces el ruido de otros factores lo oscurecen notablemente.

10 CONCLUSIONES

Antes de realizar un programa conviene elegir un buen algoritmo, donde por bueno entendemos que utilice pocos recursos, siendo usualmente los más importantes el tiempo que lleve ejecutarse y la cantidad de espacio en memoria que requiera. Es engañoso pensar que todos los algoritmos son "más o menos iguales" y confiar en nuestra habilidad como programadores para convertir un mal algoritmo en un producto eficaz. Es asimismo engañoso confiar en la creciente potencia de las máquinas y el abaratamiento de las mismas como remedio de todos los problemas que puedan aparecer.

En el análisis de algoritmos se considera usualmente el caso peor, si bien a veces conviene analizar igualmente el caso mejor y hacer alguna estimación sobre un caso promedio. Para independizarse de factores coyunturales tales como el lenguaje de programación, la habilidad del codificador, la máquina soporte, etc. se suele trabajar con un cálculo asintótico que indica cómo se comporta el algoritmo para datos muy grandes y salvo algún coeficiente multiplicativo. Para problemas pequeños es cierto que casi todos los algoritmos son "más o menos iguales", primando otros aspectos como esfuerzo de codificación, legibilidad, etc. Los órdenes de complejidad sólo son importantes para grandes problemas.

11 Bibliografía

Es difícil encontrar libros que traten este tema a un nivel introductorio sin caer en amplios desarrollos matemáticos, aunque también es cierto que casi todos los libros que se precien dedican alguna breve sección al tema. Un libro sencillo y extremadamente claro es

L. Goldschlager and A. Lister
Computer Science, A Modern Introduction
Series in Computer Science. Prentice-Hall Intl., London (UK), 1982.

y también es muy recomendable por la cantidad de algoritmos que presenta y analiza

S.S. Skiena
The Algorithm Design Manual
November 14, 1997

Siempre hay algún clásico con una presentación excelente, pero entrando en mayores honduras matemáticas como

A. V. Aho, J. E. Hopcroft, and J. D. Ullman
Data Structures and Algorithms
Addison-Wesley, Massachusetts, 1983.

Existe un libro en castellano con una presentación muy buena, si bien esta escrito en plan matemático y, por tanto, repleto de demostraciones (un poco duro)

Carmen Torres
Diseño y Análisis de Algoritmos
Paraninfo, 1992

Hay un libro interesante, con ejemplos en Pascal, que se limita al análisis numérico de algunos algoritmos, contabilizando operaciones costosas. Sirve para comparar algoritmos; pero no entra en complejidad:

N. Wirth
Algoritmos y Estructuras de Datos
Prentice-Hall Hispanoamericana, Mexico, 1987.

Para un estudio serio hay que irse a libros de matemática discreta, lo que es toda una asignatura en sí misma; pero se pueden recomendar un par de libros modernos, prácticos y especialmente claros:

R. Skvarcius and W. B. Robinson
Discrete Mathematics with Computer Science Applications
Benjamin/Cummings, Menlo Park, California, 1986.

R. L. Graham, D. E. Knuth, and O. Patashnik
Concrete Mathematics
Addison-Wesley, 1990.

Para saber más de problemas NP y NP-completos, hay que acudir a la "biblia", que en este tema se denomina

M. R. Garey and D. S. Johnson
Computers and Intractability: A Guide to the Theory of NP-Completeness
Freeman, 1979.

Sólo a efectos documentales, permítasenos citar al inventor de las nociones de complejidad y de la notación O():

P. Bachmann
Analytische Zahlen Theorie
1894