Análisis de los algoritmos

Vicente Trigo Aranda
www.vicentetrigo.com

Como bien sabemos, un algoritmo es la secuencia de pasos que debemos seguir para resolver un problema; es decir, el conjunto de operaciones que nos permite hallar su solución. Por ejemplo, en la escuela nos enseñaron algoritmos (recetas, métodos, etc.) para averiguar la suma, resta, producto o división de varios números e, incluso, hallar su raíz cuadrada.

Claro que entonces no se nos habló de algoritmos, sino que se nos decía que íbamos a "aprender a multiplicar". De hecho, todavía la mayoría de la gente identifica el problema (hallar el producto de dos números) con el algoritmo para resolverlo que se nos explicó en el colegio.

Y es que, en general, no hay una única forma de resolver un problema; en otras palabras, es normal encontrar varios algoritmos que nos permitan alcanzar nuestro objetivo. En esta situación, ¿con cuál nos quedamos? La decisión más sensata sería analizarlos y compararlos de alguna manera, ¿no cree?

Esto es justo lo que vamos a ver en este artículo, deteniéndonos en el análisis de su eficiencia, una faceta que adquiere un gran interés, tanto científico como económico, cuando los algoritmos se implementan en los ordenadores.

Como resulta fácil imaginar, el estudio pormenorizado de esta cuestión no es nada sencillo, ni mucho menos, pero éste tampoco es el lugar para ello. Entonces, ¿qué vamos a ver en este artículo? Pues una introducción divulgativa a la eficiencia de los algoritmos, desde un punto de vista intuitivo y sin profundizar en cuestiones teóricas.

Unos ejemplos de algoritmos

Y como para ilustrar las cosas no hay nada mejor que un buen ejemplo, comencemos con uno muy sencillo.

Supongamos que, por el motivo que sea, nos interesa averiguar la suma de los números pares de 2 a 100. Resulta evidente que siempre podríamos sumar manualmente esos cincuenta números, pero es una tarea un tanto tediosa y nada exenta de errores. Además, seguir este algoritmo para resolver el problema sería prácticamente imposible cuando el número límite fuese muy alto (mil millones, por ejemplo, en lugar de 100), porque no acabaríamos con las operaciones en toda nuestra vida.

¿No podemos encontrar otro método más eficaz? Desde luego. Si nos fijamos en los números que debemos sumar, comprobaremos que el primero más el último (2+100) da el mismo resultado que el segundo más el penúltimo (4+98) y que el tercero más el antepenúltimo (6+96) y así sucesivamente.

Cada una de estas parejas suma 102 y, entre 2 y 100, hay 50 números pares; es decir, 25 de estas parejas. Ahora obtener el total buscado ya es algo elemental, pues basta con multiplicar 25 x 102. De esta forma tan sencilla, averiguamos que la suma de los números pares comprendidos entre 2 y 100 es 2.550.

Acabamos de comprobar que encontrar un buen algoritmo ahorra bastante tiempo y trabajo… pero, ¿cómo evaluamos la bondad de un algoritmo? Pues todo depende, como vemos a continuación.

Supongamos que necesitamos calcular el producto de dos números, 48 y 37, por ejemplo. Si aplicamos el algoritmo que nos enseñaron en el colegio hallamos el resultado, 1.776, sin mucho esfuerzo, ¿verdad? ¡Pues no! Ahora no nos cuesta nada, es cierto, pero en nuestra infancia tuvimos que dedicar muchas horas de esfuerzo a aprendernos de memoria las tablas de multiplicar, además del algoritmo en sí.

En otros países (Rusia, por ejemplo) se utilizaba otro algoritmo distinto para multiplicar, cuyo aprendizaje era mucho menos trabajoso, si bien no resultaba demasiado útil cuando aparecían números grandes. Este algoritmo, que se conoce habitualmente por "multiplicación rusa", es bastante sencillo:

1. Escribimos los dos números que deseamos multiplicar, 48 y 37 en nuestro ejemplo, uno al lado del otro, un poco separados.

2. Del número más grande calculamos su doble y del otro su mitad entera (nos olvidamos de los decimales) y escribimos estos valores debajo de los anteriores.

48	37
96	18

Figura 1. Primera multiplicación y división por 2

3. Repetimos el paso anterior (duplicar y dividir por dos) con los números obtenidos, hasta que alcancemos el 1 al dividir, al igual que ocurre en la figura 2.

48	37
96	18
192	9
384	4
768	2
1536	1

Figura 2. Tabla de la multiplicación

4. Nos olvidamos de los números de la columna izquierda que tienen a su derecha un número par y sumamos los restantes (48, 192 y 1.536). El resultado, 1.776, es justo el producto de 48 y 37.

48	37
96	18
192	9
384	4
768	2
1536	1
1776	

Figura 3. Fin del proceso

Este nuevo algoritmo para multiplicar es bastante cómodo y, a diferencia del que aprendimos en la escuela, no nos obliga a memorizar las tablas de multiplicar, aunque sí nos exige saber dividir por dos. ¿Cuál le parece mejor?

Es innegable que la "multiplicación rusa" resulta bastante pesada e incómoda si trabajamos con números de tres o cuatro cifras, y no digamos ya en el caso de valores mayores. No obstante, si alguien multiplicase y dividiese por dos rápidamente y supiese sumar,

seguro que empleaba este método para efectuar productos numéricos, aunque los factores fuesen muy grandes. ¡Lástima que nadie sea capaz de hacerlo! En efecto, pero las máquinas sí pueden.

Como los ordenadores trabajan en binario, para ellos duplicar cualquier número es muy sencillo, pues sólo tienen que añadir un cero a su derecha; además, hallar el cociente de dividir por 2 también les resulta trivial, porque basta con olvidarse de la última cifra de la derecha (en la figura 4 comprobamos estas afirmaciones). En otras palabras, si queremos enseñar a multiplicar a un ordenador, mejor nos olvidamos del método escolar tradicional y lo programamos con la "multiplicación rusa".

Número	En binario
48	110000
96	1100000
37	100101
18	10010

Figura 4. Multiplicar y dividir por 2 en binario

Con este último ejemplo ha quedado bastante patente que existen notables diferencias en la eficiencia de los algoritmos en función de que los vayamos a utilizar seres humanos o se implementen en el ordenador. Un buen algoritmo para las personas no tiene por qué serlo para el ordenador (de hecho, en general no lo será), ni viceversa.

En la sociedad actual, donde coste computacional equivale a coste económico, el caso verdaderamente importante es cómo trabaja el ordenador los algoritmos y, por tanto, es el que se ha estudiado más a fondo, tanto desde el punto de vista teórico como práctico.

Ordenadores y algoritmos

¿Qué interesa más en un algoritmo, aparte de que funcione correctamente, claro está? A primera vista parece evidente que, cuando se ejecute en el ordenador, dos parámetros muy a tener en cuenta serán su consumo de memoria y su rapidez. En este artículo nos centraremos únicamente en la eficiencia temporal de los algoritmos, es decir, en su tiempo de ejecución, y dejamos de lado el estudio del coste en espacio (cantidad de memoria que precisa).

Como es fácil deducir, por lo general siempre interesará utilizar el algoritmo más rápido. ¿Sólo por lo general? ¿Y por qué no siempre? Porque en determinadas situaciones la legibilidad es un aspecto muy importante y puede ser preferible optar por un algoritmo algo más lento (no mucho más, desde luego) que se lea mejor.

Así, cuando diseñamos un programa en equipo interesa hacerlo lo más claro posible, ya que puede necesitarse que otras personas lo retoquen en algún momento. Del mismo modo, si estamos realizando en el ordenador una simulación o determinada parte de una demostración, para validar científicamente nuestro trabajo es muy posible que deba ser supervisado por un comité y siempre es aconsejable facilitarles su tarea, ¿no cree?

Por ejemplo, supongamos que en un programa precisamos intercambiar los valores almacenados en dos variables *a* y *b*, algo muy habitual en informática. Generalmente utilizaremos otra variable auxiliar, *c*, para efectuar dicho intercambio, que se efectuará con las sentencias siguientes:

$$c=a,\ a=b,\ b=c$$

Como observamos en la figura 5, donde partimos de que en *a* hay almacenado un 7 y en *b* un 5, el algoritmo de intercambio es eficaz (al final en *a* hay un 5 y en *b* un 7) y bastante intuitivo. Cualquier persona que sepa algo de programación, al encontrarse con unas sentencias similares a las anteriores, reconocerá fácilmente que se están intercambiando dos variables.

Inicialmente	***a***	***b***	***c***
	7	5	
Tras hacer *c*=*a*	***a***	***b***	***c***
	7	5	7
Tras hacer *a*=*b*	***a***	***b***	***c***
	5	5	7
Tras hacer *b*=*c*	***a***	***b***	***c***
	5	7	7

Figura 5. Intercambio de los valores de dos variables

Ahora bien, ¿qué finalidad tienen las sentencias siguientes? ¿Verdad que no resulta sencillo averiguarlo a simple vista?

$$a=a+b,\ b=a-b,\ a=a-b$$

En realidad también estamos intercambiando los valores almacenados en las variables *a* y *b*, como vemos en la figura 6; además el coste en espacio de este segundo algoritmo se reduce con respecto al primero, pues nos ahorramos la variable auxiliar *c*. Pero, ¿merece la pena perder legibilidad a cambio de este pequeño ahorro?

Inicialmente	***a***	***b***
	7	**5**
Tras hacer *a*=*a*+*b*	***a***	***b***
	12	**5**
Tras hacer *b*=*a*-*b*	***a***	***b***
	12	**7**
Tras hacer *a*=*a*-*b*	***a***	***b***
	5	**7**

Figura 6. Intercambio de los valores de dos variables

Volviendo a la eficiencia temporal de los algoritmos, supongamos que tenemos a nuestra disposición varios que resuelven un mismo problema. ¿Cómo averiguamos cuál de ellos es más eficaz? Básicamente hay dos caminos para compararlos:

- Análisis teórico: estudiando el número de instrucciones elementales que exige la ejecución de un algoritmo es posible obtener una función que mida su tiempo de ejecución. Como, según el principio de invarianza, dos implementaciones distintas de un mismo algoritmo tienen tiempos de ejecución proporcionales, el estudio teórico nos da una estimación del coste temporal, independientemente del ordenador donde se ejecute el algoritmo y sin necesidad de implementarlo; tarea esta última que a veces supone un coste económico nada despreciable.
- Análisis práctico: haciendo pruebas empíricas, implementando los algoritmos en un mismo equipo, es muy sencillo medir sus tiempos de ejecución. Claro que los resultados variarán en función del equipo, el lenguaje de programación, los datos iniciales, etc.; sin embargo, nos darán una idea aproximada de lo que puede suceder en otras condiciones y, sobre todo, valores reales cuando el algoritmo debe implementarse en un ordenador concreto.

Seguidamente vamos a ver una introducción al análisis teórico, sin profundizar en los aspectos científicos y viendo unos ejemplos muy sencillos. Después, confrontaremos en la práctica algunos de los más conocidos algoritmos para ordenar listas, una de las acciones más habituales en programación.

Orden de los algoritmos

Comencemos con un juego muy elemental: una persona piensa un número del uno al cien y debemos adivinarlo. En el mejor de los casos lo acertaremos en el primer intento y en el peor en el último, ¿no? Eso sí, resulta evidente que, en general, el número de intentos que necesitaremos será proporcional al número de elementos que haya en la lista (100 en nuestro juego).

¿Y con qué caso nos quedamos? ¿El mejor, el peor o el promedio? Pues depende, aunque en general suele interesar sobre todo estudiar el peor de los casos, para tener limitado el tiempo máximo de ejecución. Por tanto, sería cuestión de acotar la función del coste del algoritmo tanto superior como inferiormente, para tener una idea más clara de su crecimiento. Como este estudio se saldría del entorno divulgativo del artículo, diremos simplemente que el algoritmo del juego, que viene a ser equivalente a buscar un dato en una tabla de *n* elementos, es de orden *n*, O(*n*), para denotar que su rapidez depende del valor de *n*.

Supongamos ahora que modificamos el juego anterior, de modo que la otra persona nos dice cada vez si el número que ha pensado es mayor o menor que nuestra elección. Enseguida nos damos cuenta de que podemos mejorar el algoritmo anterior, ¿verdad?

Basta comenzar probando con 50; si el número a encontrar es mayor, lo intentamos con 75 y, en caso contrario, con 25. Si seguimos dividiendo por dos el intervalo de búsqueda en cada ocasión, es fácil ver que con seis o siete intentos averiguamos el número.

Por tanto, en este segundo juego el orden de nuestro algoritmo será la potencia a la que debemos elevar 2 para llegar a 100. ¿Recuerda las matemáticas de bachillerato? Esa es precisamente la definición del logaritmo en base 2 de 100, cuyo valor es 6,64..., por eso necesitamos seis o siete intentos para encontrar el número pensado por la otra persona.

Si nos detenemos un momento en este juego modificado nos percataremos de que es equivalente a buscar un dato en una lista ordenada y, como acabamos de ver, el algoritmo de búsqueda resulta mucho más rápido cuando las lista está ordenada (por eso interesa tanto ordenar listas, cuestión que trataremos en el siguiente apartado). Tomando como referencia

dicho juego, podemos concluir que la búsqueda de un elemento en una lista ordenada de tamaño *n* es un algoritmo O(lg *n*).

El orden de algunos algoritmos es relativamente fácil de deducir intuitivamente; así, el cálculo del factorial de *n* (multiplicar todos los enteros comprendidos entre 1 y *n*) es elemental ver que es O(*n*). En cambio, averiguar el orden de otros algoritmos no resulta nada sencillo; por ejemplo, ordenar una lista de *n* elementos mediante el algoritmo de selección, que veremos más adelante, tiene orden n^2.

Claro que también podemos encontrarnos con algoritmos de orden exponencial, $O(a^n)$ con $a>1$, incluso en problemas aparentemente sencillos, como los mostrados en los ejemplos siguientes. Los algoritmos de estos órdenes se consideran intratables, porque el número de instrucciones a ejecutar es altísimo, incluso para valores no muy altos de *n*.

- La denominada sucesión de Fibonacci fue descrita por el matemático italiano Leonardo de Pisa (1170-1250), más conocido por Fibonacci, en su célebre *Liber abaci*. Al estudiar un problema de cría de conejos[1], obtuvo una sucesión que comenzaba en 0 y 1 y donde cada término era suma de los dos anteriores (0, 1, 1, 2, 3, 5, 8, 13, 21, ...). El cálculo del término que ocupa el lugar *n* en la sucesión de Fibonacci puede demostrarse que es un problema del orden mostrado en la figura 7. ¿Verdad que resulta curioso que un problema de reproducción de conejos esté tan íntimamente relacionado con el artístico número áureo?

$$\left(\frac{1+\sqrt{5}}{2}\right)^n$$

Figura 7. Orden en la sucesión de Fibonacci

- El famoso problema de las torres de Hanoi (figura 8) lo dio a conocer en 1883 el francés Edouard Lucas (1842-1891), en su obra *Récreations Mathématiques*: debemos pasar todos los discos de un pivote a otro de los dos, pudiendo utilizar el restante para traslados intermedios, pero teniendo en cuenta que en cada movimiento sólo está permitido tomar un disco y nunca se puede colocar un disco sobre otro de tamaño inferior[2]. En estas condiciones, resolver el problema con *n* discos resulta tener orden 2^n.

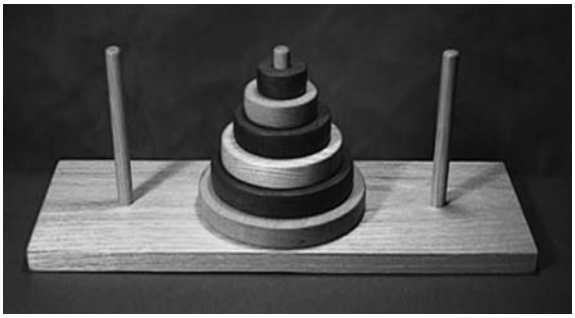

Figura 8. Puzle en madera basado en las torres de Hanoi

Volvamos de nuevo a los dos juegos de adivinación del principio del apartado. Gracias a ellos hemos comprobado empíricamente que el algoritmo lineal es menos eficiente que el logarítmico, ya que éste resultaba más rápido.

En general el número de instrucciones a ejecutar en un algoritmo de orden logarítmico crece a menor velocidad que uno de orden lineal; por tanto, como debe realizar menos operaciones es más rápido. Sin embargo, no debemos olvidar que su celeridad depende del número de datos.

Por ejemplo, supongamos que aplicamos a *n* datos un algoritmo que precisa ejecutar 2*n* instrucciones y otro que exige 100 lg *n*. Si trabajamos con un millón de datos, el primer algoritmo exigirá dos millo-

[1] Fibonacci quería averiguar cuántas parejas de conejos habría al cabo de un año en una granja, partiendo de una pareja y suponiendo que los animales se reproducen continuamente, el periodo de gestación dura un mes y sus descendientes (en cada parto nacen siempre un macho y una hembra) tardan un mes en alcanzar su madurez reproductiva. La respuesta, por si no quiere calcularla, es 144 parejas de conejos.

[2] En su momento Lucas comercializó un puzle con las torres de Hanoi y lo acompañó de un curioso texto en el que se afirmaba que, cuando Dios creó el mundo, situó en un templo de la ciudad sagrada de Benarés tres agujas de diamante, en la primera de las cuales insertó 64 discos de oro, todos de diferentes tamaño y apilados en orden decreciente (el menor arriba); los sacerdotes de ese templo tenían por única misión trasladar todos los discos a la tercera aguja de diamante, con las restricciones que ya conocemos. ¡Cuando finalizasen su tarea el mundo entero desaparecería!
¿Y por qué seguimos aquí? Pues la verdad es que, aunque fuese cierta la leyenda (sólo era un mero reclamo publicitario de Lucas para aumentar las ventas), tampoco deberíamos preocuparnos mucho. Si los sacerdotes del templo empleasen un segundo en cada movimiento y no parasen nunca, tardarían más de 5.800 millones de siglos en completar el traslado… y la edad actual de la Tierra se estima que ronda los 4.500 millones de años (no de siglos).

nes de operaciones y el segundo menos de dos mil, ¡una diferencia notable! Ahora bien, si sólo trabajamos con cien datos, el primer algoritmo precisa ejecutar 200 instrucciones y el segundo unas 664. De hecho, el algoritmo O(2*n*) será más rápido para valores de *n* inferiores a 439 (si no me he equivocado en los cálculos).

Dicho de otro modo, cuando comparemos dos algoritmos no está de más indicar sobre qué cantidad de datos se aplican, porque se trata de otro parámetro que también puede influir. No obstante, para valores suficientemente grandes de *n*, sí que se cumplen las siguientes relaciones de eficiencia:

O(lg *n*) más que O(*n*) más que O(*n* lg *n*) más que O(n^2) más que O(n^3) más que O(a^n), con a>1.

¿Y siempre aparece un único parámetro en el orden? No, pero sí. Me explico.

A veces, al calcular el orden de un algoritmo resulta que obtenemos expresiones de, por ejemplo, el tipo $5n^3 - 8n^2 + 7n + 134$. Según las anteriores relaciones de eficiencia, podemos olvidarnos de las constantes y de los términos de menor coste y diremos que ese algoritmo es de orden n^3. ¿Y si el orden fuese $4n^2 + 2^n$? Pues diríamos que es O(2^n).

Ordenación de listas

Un problema que aparece habitualmente en informática es la ordenación de listas. A continuación vamos a analizar algunos de los algoritmos más conocidos que nos permiten realizar esta tarea y, de forma empírica, compararemos su rapidez.

Al explicar los diferentes métodos de ordenación utilizaré una breve lista de sólo 8 números, la misma en los cuatro casos, para que si lo desea pueda seguirlos manualmente. Como es lógico, a la hora de implementar esos algoritmos en el ordenador se manejan listas de mayor tamaño y, para que los algoritmos trabajasen con los mismos datos, previamente he generado los valores al azar y los he guardado en archivos, desde donde se leen en cada caso. En la tabla final, donde no se especifican unidades de tiempo, pues todo depende del equipo en que se implementen los algoritmos, podremos comparar de forma empírica su eficiencia.

El método de la burbuja es el clásico algoritmo que se pone como ejemplo de chapuza, pues si bien es intuitivo resulta lentísimo: cada elemento se compara con el siguiente y se intercambian o no, según sea el caso, de modo que al terminar la primera pasada el mayor de ellos sube al final de la lista. Se reinicia el proceso considerando la lista disminuida en una unidad (pues ya sabemos que el mayor está al final), reiterándolo hasta que la lista queda ordenada.

Para ilustrar este método de la burbuja, partimos de la siguiente lista inicial y, debajo, se van detallando los pasos de que consta el proceso:

89 90 42 25 26 67 88 86 (lista inicial)

89 42 90 25 26 67 88 86

89 42 25 90 26 67 88 86

89 42 25 26 90 67 88 86

89 42 25 26 67 90 88 86

89 42 25 26 67 88 90 86

89 42 25 26 67 88 86,90 (el mayor ya está al final)

42 89 25 26 67 88 86,90

42 25 89 26 67 88 86,90

42 25 26 89 67 88 86,90

42 25 26 67 89 88 86,90

42 25 26 67 88 89 86,90

42 25 26 67 88 86,89 90

25 42 26 67 88 86,89 90

25 26 42 67 88 86,89 90

25 26 42 67 86 88 89 90 (lista ya ordenada)

El método de selección también es muy intuitivo y se basa en la búsqueda del elemento más pequeño (o más grande) de la lista e intercambiarlo con el primero (o el último), reiterándose el proceso con la lista restante hasta ordenarla completamente.

Utilizando de nuevo la misma lista, vamos a seguir el desarrollo del proceso:

89 90 42 25 26 67 88 86 (lista inicial)

25,90 42 89 26 67 88 86 (el menor al principio)

25 26,42 89 90 67 88 86

25 26 42,89 90 67 88 86

25 26 42 67,90 89 88 86

25 26 42 67 86,89 88 90

25 26 42 67 86 88,89 90 (lista ya ordenada)

El algoritmo de ordenación Shell (*Shell sort* en inglés) fue creado por Donald Shell, que lo dio a conocer en 1959, y se basa en colocar aproximadamente cada término de la lista donde puede estar al final. La idea clave es la noción de brecha o hueco (*gap*), que indica la distancia entre los elementos que se van comparando. Habitualmente se suele tomar como valor inicial del gap la mitad de la longitud de la lista y, tras cada pasada por la lista, se va reduciendo a su mitad entera.

El texto anterior resulta un tanto críptico, lo reconozco, pero con un ejemplo el proceso quedará meridiano.

89 90 42 25 26 67 88 86 (lista inicial)

1. Como el gap es 4, sólo debemos intercambiar dos parejas de valores (89 y 26, 90 y 67) de la lista inicial, que se transformará en:

   26 67 42 25 89 90 88 86

2. Cuando el gap es 2, las parejas a intercambiar en la lista anterior son 67 y 25, 89 y 88, 90 y 86, obteniéndose:

   26 25 42 67 88 86 89 90

3. Al ser el gap 1, las parejas a intercambiar en la última lista son 26 y 25, 88 y 86. Al finalizar el proceso ya hemos alcanzado nuestro objetivo.

   25 26 42 67 86 88 89 90 (lista ya ordenada)

Por último, el algoritmo de ordenación rápida (*quicksort* en inglés) fue desarrollado en 1960 por el británico C. A. R. Hoare y, como su nombre indica, se caracteriza por su rapidez. Se basa en una idea recursiva muy sencilla: dividir la lista original en dos listas, de modo que en la primera irán los elementos menores que uno dado (el que ocupa la posición central de la lista o el primero, por ejemplo) y en la segunda los mayores, volviéndose a aplicar el mismo procedimiento a cada una de las listas.

89 90 42 25 26 67 88 86 (lista inicial)

1. Si partimos del elemento central, que es el 4º (25), tenemos:

   1ª lista 25 (ya ordenada)

   2ª lista 90 42 89 26 67 88 86

2. Para ordenar esta última lista, hallamos su elemento central (26) y obtenemos:

   1ª lista 26 (ya ordenada)

   2ª lista 42 89 90 67 88 86

3. Ahora toca ordenar la segunda lista, cuyo elemento central es el 3º (90):

   2ª lista 90 (ya ordenada)

   1ª lista 42 89 86 67 88

4. Hay que ordenar esta última lista, cuyo elemento central es el 3º (86):

   1ª lista 42 67 (ya ordenada)

   2ª lista 89 88 (se ordena al siguiente intercambio)

5. De esta forma, la lista inicial se ha transformado en 25 26 42 67 86 88 89 90, que ya está ordenada.

¿Y cuál de los cuatro algoritmos es el más eficiente? Como queda patente en la tabla siguiente, que muestra los tiempos invertidos en ordenar las mismas listas de *n* números según los algoritmos comentados, el quicksort hace honor a su nombre y es el más rápido.

n	*Burbuja*	*Selección*	*Shell*	*Quicksort*
1.000	*3,19*	*0,99*	*0,11*	*0,05*
2.000	*12,57*	*3,90*	*0,22*	*0,11*
3.000	*28,62*	*9,79*	*0,39*	*0,16*
4.000	*50,86*	*15,54*	*0,60*	*0,22*
5.000	*79,04*	*24,28*	*0,82*	*0,28*
6.000	*114,41*	*34,94*	*0,93*	*0,32*
7.000	*154,78*	*47,56*	*1,10*	*0,39*
8.000	*205,43*	*62,12*	*1,48*	*0,49*
9.000	*259,41*	*78,60*	*1,65*	*0,55*
10.000	*314,55*	*98,64*	*1,92*	*0,60*
20.000	*1278,22*	*388,26*	*4,44*	*1,21*
30.000	*2881,50*	*875,68*	*7,08*	*1,93*

Observando la tabla comprobamos empíricamente que tanto el método de la burbuja como el de selección son claramente de orden n^2. Así, vemos que si multiplicamos por 2 el número de datos (cuando *n* vale 5.000 y 10.000, por ejemplo), el tiempo se cuadruplica aproximadamente; si *n* se multiplica por 10 (por ejemplo, para *n* igual a 3.000 y 30.000) los tiempos más o menos lo hacen por cien.

En cuanto a los otros dos algoritmos, de la tabla se concluye rotundamente que son mucho más eficientes, aunque deducir su orden no resulta una tarea sencilla ni mucho menos. Como curiosidad le diré que, en el peor de los casos, el método shell es $O(n \lg^2 n)$; por su parte, el quicksort es $O(n \lg n)$ en promedio.