Análisis de Algoritmos: Complejidad

José A. Mañas

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.

A efectos prácticos o ingenieriles, nos deben preocupar los recursos físicos necesarios para que un programa se ejecute. Aunque puede haber muchos parametros, los mas usuales son el tiempo de ejecución y la cantidad de memoria (espacio). Ocurre con frecuencia que ambos parametros están fijados por otras razones y se plantea la pregunta inversa: ¿cual es el tamano del mayor problema que puedo resolver en T segundos y/o con M bytes de memoria? En lo que sigue nos centraremos casi siempre en el parametro tiempo de ejecución, si bien las ideas desarrolladas son fácilmente aplicables a otro tipo de recursos.

Para cada problema determinaremos un 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 utizar 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 mas 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.

2. 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 (int 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 mas que un valor T(N) debamos hablar de un rango de valores

```
  Tmin(N)  <=  T(N)  <=  Tmax(N)
```

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

Cualquier fórmula T(N) incluye referencias al parámetro N y a una serie de constantes "Ti" 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 (en la actualidad, se duplica anualmente), intentaremos analizar los algoritmos con algun nivel de independencia de estos factores; es decir, buscaremos estimaciones generales ampliamente válidas.

3. Asintotas

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 problema 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 tecnologias 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.

Sean "g(n)" diferentes funciones que determinan el uso de recursos. Habra funciones "g" de todos los colores. Lo que vamos a intentar es identificar "familias" de funciones, usando como criterio de agrupación su comportamiento asintótico.

A un conjunto de funciones que comparten un mismo comportamiento asintótico le denominaremos un órden de complejidad'. Habitualmente estos conjuntos se denominan O, existiendo una infinidad de ellos.

Para cada uno de estos conjuntos se suele identificar un miembro f(n) que se utiliza como representante de la clase, hablándose del conjunto de funciones "g" que son del orden de "f(n)", denotándose como

```
        g IN O(f(n))
```

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".

La definición matemática de estos conjuntos debe ser muy cuidadosa para involucrar ambos aspectos: 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  N0 tales que
          para todo  N > N0,  g(N) <= k*f(N) }
```

en palabras, O(f(n)) esta 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).

3.1. Órdenes de Complejidad

Se dice que O(f(n)) define un **"orden de complejidad"**. Escogeremos como representante de este orden a la función f(n) más sencilla del mismo. Así tendremos

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 > 2)
O(n!)	orden factorial

Es más, se puede identificar una jerarquía de órdenes de complejidad que coincide con el orden de la tabla anterior; jerarquía en el sentido de que cada orden de complejidad superior tiene a los inferiores como subconjuntos. Si un algoritmo A se puede demostrar de un cierto orden O_1, es cierto que tambien pertenece a todos los órdenes superiores (la relación de orden çota 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.

3.1.1. Impacto Práctico

Para captar la importancia relativa de los órdenes de complejidad conviene echar algunas cuentas.

Sea un problema que sabemos resolver con algoritmos de diferentes complejidades. Para compararlos entre si, supongamos que todos ellos requieren 1 hora de ordenador para resolver un problema de tamaño N=100.

¿Qué ocurre si disponemos del doble de tiempo? Notese que esto es lo mismo que disponer del mismo tiempo en un odenador el doble de potente, y que el ritmo actual de progreso del hardware es exactamente ese:

"duplicación anual del número de instrucciones por segundo".

¿Qué ocurre si queremos resolver un problema de tamaño 2n?

O(f(n))	**N=100**	**t=2h**	**N=200**
log n	1 h	10000	1.15 h
n	1 h	200	2 h
n log n	1 h	199	2.30 h
$\mathbf{n^2}$	1 h	141	4 h
$\mathbf{n^3}$	1 h	126	8 h
$\mathbf{2^n}$	1 h	101	10^{30} h

Los algoritmos de complejidad O(n) y O(n log n) son los que muestran un comportamiento más "natural": prácticamente a doble de tiempo, doble de datos procesables.

Los algoritmos de complejidad logarítmica son un descubrimiento fenomenal, pues en el doble de tiempo permiten atacar problemas notablemente mayores, y para resolver un problema el doble de grande sólo hace falta un poco más de tiempo (ni mucho menos el doble).

Los algoritmos de tipo polinómico no son una maravilla, y se enfrentan con dificultad a problemas de tamaño creciente. La práctica viene a decirnos que son el límite de lo "tratable".

Sobre la tratabilidad de los algoritmos de complejidad polinómica habria mucho que hablar, y a veces semejante calificativo es puro eufemismo. Mientras complejidades del orden $O(n^2)$ y $O(n^3)$ suelen ser efectivamente abordables, prácticamente nadie acepta algoritmos de orden $O(n^{100})$, por muy polinómicos que sean. La frontera es imprecisa.

Cualquier algoritmo por encima de una complejidad polinómica se dice "intratable" y sólo será aplicable a problemas ridiculamente pequeños.

A la vista de lo anterior se comprende que los programadores busquen algoritmos de complejidad lineal. Es un golpe de suerte encontrar algo de complejidad logarítmica. Si se encuentran soluciones polinomiales, se puede vivir con ellas; pero ante soluciones de complejidad exponencial, más vale seguir buscando.

No obstante lo anterior ...

- ... si un programa se va a ejecutar muy pocas veces, los costes de codificación y depuración son los que más importan, relegando la complejidad a un papel secundario.

- ... si a un programa se le prevé larga vida, hay que pensar que le tocará mantenerlo a otra persona y, por tanto, conviene tener en cuenta su legibilidad, incluso a costa de la complejidad de los algoritmos empleados.
- ... si podemos garantizar que un programa sólo va a trabajar sobre datos pequeños (valores bajos de N), el orden de complejidad del algoritmo que usemos suele ser irrelevante, pudiendo llegar a ser incluso contraproducente.

  Por ejemplo, si disponemos de dos algoritmos para el mismo problema, con tiempos de ejecución respectivos:

algoritmo	**tiempo**	**complejidad**
f	100 n	O(n)
g	n^2	O(n^2)

  asintóticamente, "f" es mejor algoritmo que "g"; pero esto es cierto a partir de N > 100.
  Si nuestro problema no va a tratar jamás problemas de tamaño mayor que 100, es mejor solución usar el algoritmo "g".

  El ejemplo anterior muestra que las constantes que aparecen en las fórmulas para T(n), y que desaparecen al calcular las funciones de complejidad, pueden ser decisivas desde el punto de vista de ingeniería. Pueden darse incluso ejemplos más dramaticos:

algoritmo	**tiempo**	**complejidad**
f	n	O(n)
g	100 n	O(n)

  aún siendo dos algoritmos con idéntico comportamiento asintótico, es obvio que el algoritmo "f" es siempre 100 veces más rápido que el "g" y candidato primero a ser utilizado.

- ... usualmente un programa de baja complejidad en cuanto a tiempo de ejecución, suele conllevar un alto consumo de memoria; y viceversa. A veces hay que sopesar ambos factores, quedándonos en algún punto de compromiso.
- ... en problemas de cálculo numérico hay que tener en cuenta más factores que su complejidad pura y dura, o incluso que su tiempo de ejecución: queda por considerar la precisión del cálculo, el máximo error introducido en cálculos intermedios, la estabilidad del algoritmo, etc. etc.

3.2. 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 <=> 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) => f(n) IN O(h)

B. f IN O(g) y g IN O(f) <=> 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. $\lim_{(n->inf)} f(n)/g(n) = 0$ => f IN O(g)
=> g NOT_IN O(f)
=> O(f) es subconjunto de O(g)

D. $\lim_{(n->inf)} f(n)/g(n) = k$ => f IN O(g)
=> g IN O(f)
=> O(f) = O(g)

E. $\lim_{(n->inf)} f(n)/g(n) = INF$ => f NOT_IN O(g)
=> g IN O(f)
=> O(f) es superconjunto de O(g)

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

F. Si f, g IN O(h) => f+g IN O(h)
G. Sea k una constante, f(n) IN O(g) => k*f(n) IN O(g)
H. Si f IN O(h1) y g IN O(h2) => f+g IN O(h1+h2)
I. Si f IN O(h1) y g IN O(h2) => f*g IN O(h1*h2)
J. Sean los reales 0 < a < b => $O(n^a)$ es subconjunto de $O(n^b)$
K. Sea P(n) un polinomio de grado k => P(n) IN $O(n^k)$
L. Sean los reales a, b > 1 => $O(\log_a) = O(\log_b)$

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 basica 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 sistematicamente una gran cantidad de ellos, basandonos en que suelen estar bien estructurados y siguen pautas uniformes.

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

1. sentencias sencillas
2. secuencia (;)
3. decisión (if)
4. bucles
5. llamadas a procedimientos

4.0. Sentencias sencillas

Nos referimos a las sentencias de asignación, entrada/salida, etc. siempre y cuando no trabajen sobre variables estructuradas cuyo tamaño este 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).

4.1. 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.

4.2. 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 multiples (ELSE IF, SWITCH CASE), se tomara la peor de las ramas.

4.3. 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.

```
Ej.- for (int i= 0; i < K; i++) { algo_de_O(1) }      =>  K*O(1) =
O(1)
```

Si el tamaño N aparece como límite de iteraciones ...

```
Ej.- for (int i= 0; i < N; i++) { algo_de_O(1) }  =>  N * O(1) = O(n)
```

```
Ej.- for (int i= 0; i < N; i++) {
       for (int j= 0; j < N; j++) {
         algo_de_O(1)
       }
     }
```

tendremos N * N * O(1) = O(n^2)

```
Ej.- 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,

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

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

```
Ej.- c= 1;
     while (c < N) {
       algo_de_O(1)
       c= 2*c;
     }
```
El valor incial de "c" es 1, siendo "2^k" al cabo de "k" iteraciones. El número de iteraciones es tal que

$2^k >= N \Rightarrow k = eis(\log_2(N))$ [el entero inmediato superior]

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

```
Ej.- c= N;
     while (c > 1) {
       algo_de_O(1)
       c= c / 2;
     }
```
Un razonamiento análogo nos lleva a $\log_2(N)$ iteraciones y, por tanto, a un orden O(log n) de complejidad.

```
Ej.- for (int i= 0; i < N; i++) {
       c= i;
       while (c > 0) {
         algo_de_O(1)
         c= 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)

4.4. 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 notáblemente 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.

4.5. 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;

```
class Polinomio {
  private double[] coeficientes;

  Polinomio (double[] coeficientes) {
    this.coeficientes= new double[coeficientes.length];
    System.arraycopy(coeficientes, 0, this.coeficientes, 0,
```

```
                    coeficientes.length);
  }

  double evalua_1 (double x) {
    double resultado= 0.0;
    for (int termino= 0; termino < coeficientes.length; termino++) {
      double xn= 1.0;
      for (int j= 0; j < termino; j++)
        xn*= x;                // x elevado a n
      resultado+= coeficientes[termino] * xn;
    }
    return resultado;
  }
}
```

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

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

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 evalua_2 (double x) {
    double resultado= 0.0;
    for (int termino= 0; termino < coeficientes.length; termino++) {
      resultado+= coeficientes[termino] * potencia(x, termino);
    }
    return resultado;
  }

  private double potencia (double x, int n) {
    if (n == 0)
      return 1.0;
    // si es potencia impar ...
    if (n%2 == 1)
      return x * potencia(x, n-1);
    // si es potencia par ...
    double t= potencia(x, n/2);
    return t*t;
  }
```

El análisis de la función 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 seria x^{31}, que implica la siguiente serie para j:

```
31  30  15  14  7  6  3  2  1
```

cuyo número de terminos podemos acotar superiormente por

$2 * eis (\log_2(j)),$

donde eis(r) es el entero inmediatamente superior (este cálculo responde al razonamiento de que en el caso mejor visitaremos eis($\log_2$(j)) 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).

Insertada la función Potencia en la función EvaluaPolinomio, la complejidad compuesta es del orden O(n log n), al multiplicarse por N un subalgoritmo de O(log n).

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

```
  double evalua_3 (double x) {
    double xn= 1.0;
    double resultado= coeficientes[0];
    for (int termino= 1; termino < coeficientes.length; termino++) {
      xn*= x;
      resultado+= coeficientes[termino] * xn;
    }
    return resultado;
  }
```

que queda en un algoritmo de O(n).

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

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

```
  double evalua_4 (double x) {
    double resultado= 0.0;
    for (int termino= coeficientes.length-1; termino >= 0; termino--)
{
      resultado= resultado * x +
        coeficientes[termino];
    }
    return resultado;
  }
```

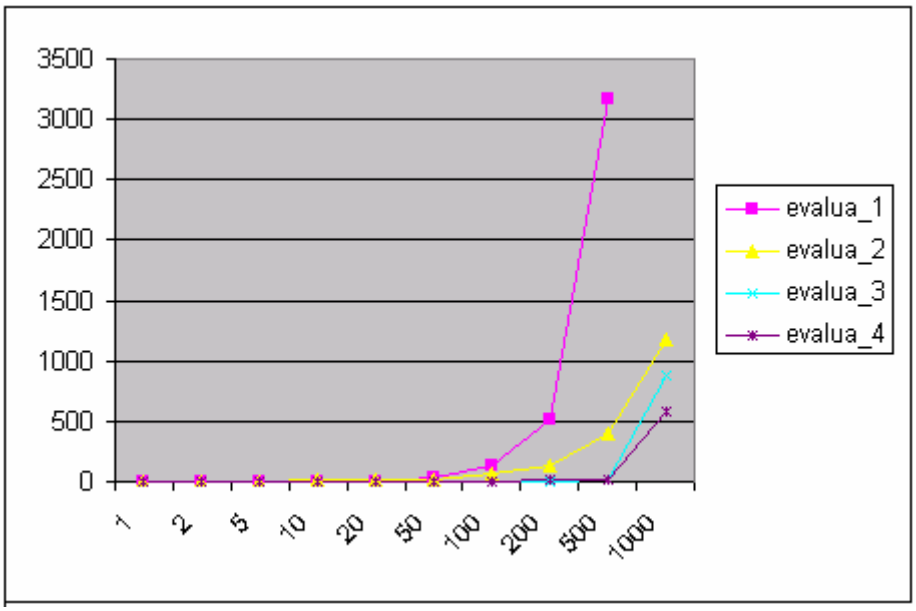
No obstante ser ambos algoritmos de idéntico orden de complejidad, cabe resaltar que sus tiempos de ejecución serán notablemente distintos. En efecto, mientras el último algoritmo ejecuta N multiplicaciones y N sumas, el penúltimo requiere 2N multiplicaciones y N sumas. Si, como es frecuente, el tiempo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

4.5.1. Medidas de laboratorio

La siguiente tabla muestra algunas medidas de la eficacia de nuestros algoritmos sobre una implementación en Java:

grado	evalua_1	evalua_2	evalua_3	evalua_4

1	0	10	0	0
2	10	0	0	0
5	0	0	0	0
10	0	10	0	0
20	0	10	0	0
50	40	20	0	10
100	130	60	0	0
200	521	140	0	10
500	3175	400	10	10
1000	63632	1171	872	580



5. 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 desafian 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. Seria 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).

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.

6. 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 como se comporta el algoritmo para datos muy grandes y salvo algun 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.

7. Bibliografia

Es difícil encontrar libros que traten este tema a un nivel introductorio sin caer en amplios desarrollos matemáticos, aunque tambien es cierto que casi todos los libros que se precien dedican alguna breve sección al tema. Probablemente uno de los libros que sólo dedican un capítulo; pero es extremadamente claro es

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

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.

Recientemente ha aparecido 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

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