

ALGORITMICA



Photo by Quino Al

Francisco Navarro Morales - GRG121

Segundo curso del Grado de Ingeniería Informática
Universidad de Granada
curso 2016-2017

Índice

1. Planteamiento general.	2
1.1. Problemas P y NP	2
1.2. Algorítmica	3
2. La eficiencia de los algoritmos	4
2.1. Medidas de la eficiencia	4
2.1.1. Ejemplos:	5
2.2. Calculando O grande de un algoritmo.	5
2.2.1. Ejercicio1: Demuestre las siguientes afirmaciones. ¿Qué quieren decir?	6
3. Calculo de eficiencia de funciones recursivas	6

1. Planteamiento general.

A la hora de buscar soluciones óptimas para un determinado problema es conveniente plantearse si dicho problema es resoluble mediante computación y, de ser así, diseñar un algoritmo que lo resuelva. Entendemos por algoritmo a una secuencia ordenada de pasos o instrucciones perfectamente definidas que, ejecutada correctamente soluciona el problema para el que fue diseñada en un tiempo finito. Este razonamiento nos lleva a clasificar los problemas en computables y no computables, pero además, podemos clasificar aquellos problemas computables según la eficiencia en tiempo que presentan.

1.1. Problemas P y NP

Denominaremos 'problemas de clase P' o simplemente P a aquellos problemas para los que existe al menos un algoritmo cuyo orden de eficiencia (temporal) es polinómico, y se considera que un tiempo polinómico es un tiempo eficiente. Aquellos problemas para los que no existe (o no se conoce) un algoritmo que lo resuelva en tiempo polinómico, y para los cuales la única opción viable de resolverlos eficientemente es realizando una etapa aleatoria (que incluye el azar y la estadística) seguida de la comprobación de si el resultado generado al azar es correcto en tiempo polinómico. Como es lógico, podría darse el caso de que se encontrara la respuesta buscada al primer intento o después de cientos de intentos, es por esto que al final la eficiencia temporal, aunque polinómica, nunca es todo lo eficiente que podríamos desear. Para que la eficiencia fuera realmente polinómica haría falta lo que denominamos máquina de Turing no determinista, que sería aquella capaz de clonarse y realizar tantas tareas en paralelo como quisiera (podría probar todas las posibles respuestas a la vez y comprobar si son ciertas en un instante). Sin embargo, esta máquina es sólo un concepto abstracto e imposible (de momento) de implementar.

Ahora bien, ¿es $P = NP$?

Podemos afirmar que $P \subseteq NP$ puesto que, cualquier problema para el que exista un algoritmo con eficiencia de orden polinómico podría ser resuelto por los métodos propios de algoritmos NP; esto es, una máquina de Turing no determinista podría ejecutar cualquier problema P simplemente no clonándose ninguna vez. Ahora bien, si $NP \subseteq P$ y, por tanto $P = NP$, entonces significaría que cualquier algoritmo de la clase NP podría realizarse en un orden de tiempo polinómico con una máquina de Turing determinista y, por tanto, las implicaciones serían cuantiosas.

Y es que, si por ejemplo, necesitáramos encontrar la clave para desencriptar un fichero, una máquina de Turing no determinista podría comprobar todas las claves posibles y determinar cual es la correcta en el mismo tiempo en que una máquina determinista comprobaría si una clave es correcta; sin embargo, dado que las máquinas no deterministas son sólo elementos teóricos, conseguir desencriptar un archivo en tiempo eficiente es imposible. Sin embargo, si $P = NP$, entonces existe un algoritmo tal que una máquina determinista (a las cuales sí tenemos acceso) podría realizar la labor que comentábamos propia de una no determinista en el mismo orden de tiempo, proporcionándonos la posibilidad de desencriptar cualquier archivo (que haya sido encriptado teniendo en cuenta las propiedades de los problemas NP) en un tiempo muy breve.

Esto supone aún más, si supiéramos que $P = NP$ significaría que es sólo cuestión de tiempo ir encontrando algoritmos polinómicos para problemas de la clase NP y, al encontrarlos, podríamos conseguir respuestas para las que esperamos durante meses o años, en cuestión de horas, minutos o segundos; es decir, podríamos obtener una respuesta para la que un supercomputador ha estado trabajando durante un año en menos de una hora sin necesidad de un ordenador especialmente potente, y obtener respuestas

que nos sería imposible obtener a día de hoy. Imagina una pregunta para la cual una computadora podría ofrecer una respuesta dentro de 1000 años, la única manera que tendríamos a día de hoy de saber esa respuesta antes de morir sería (aunque es casi tan improbable como vivir 1000 años) viajar en el tiempo para ver la respuesta que dará el ordenador dentro de 1000 años, y luego volver para poder compartirla. Sin embargo, si $P = NP$, entonces no necesitaríamos ese hipotético viaje en el tiempo, podríamos adelantar la respuesta a nuestro tiempo.

Además, esto supondría sin lugar a dudas una nueva revolución tecnológica puesto que permitiría el avance de numerosos estudios en muy poco tiempo y el desarrollo mucho más eficiente de nuevas tecnologías. No obstante, aunque aún es una cuestión abierta, los indicios actuales apuntan a que $P \neq NP$, que es la principal hipótesis de los que intentan encontrar una demostración matemática al respecto, ya que parece imposible que fuera real que $P = NP$. En caso de que la mayoría tuviera razón y $P \neq NP$, lo único que ganaríamos sería la certeza de que nuestros sistemas actuales de encriptación son seguros, y que es una pérdida de tiempo intentar encontrar soluciones eficientes a los problemas NP, pero esto no supondría ninguna revolución científica ni mucho menos. Es por esto que quizás sería conveniente no descartar del todo la posibilidad de que todos los problemas pertenezcan a un mismo grupo hasta que se demuestre lo contrario, pues si algún día se llegara a demostrar que esta hipótesis es correcta, se produciría un avance tecnológico que afectaría a todo el mundo.

1.2. Algorítmica

La algorítmica es la ciencia encargada de la construcción, validación, análisis y estudio empírico de los algoritmos.

La creación de un algoritmo es la parte más creativa del proceso, y no se puede dominar si no se conocen a la perfección las técnicas de diseño de algoritmos. Los algoritmos deben expresarse de forma clara y concisa, e independiente del lenguaje de programación con el que sean implementados.

Para saber si un algoritmo es válido (validación), el único método correcto es una validación formal (matemática) del mismo; no obstante, a veces es imposible dar esta validación o no es factible hacerlo, entonces nos limitamos a hacer las pruebas empíricas que sean necesarias para asegurar el correcto funcionamiento del algoritmo.

El análisis de algoritmos es el proceso por el cual se determina el consumo de tiempo y espacio de un algoritmo, y que permite comparar varios algoritmos a la hora de seleccionar uno entre varios posibles. Por lo general hoy en día se considera más importante el tiempo que el espacio.

El estudio empírico de algoritmos consiste en implementar el algoritmo en algún lenguaje de programación, probar y validar (depurar si es necesario) el programa y evaluar de forma empírica los requisitos de tiempo y espacio del algoritmo.

2. La eficiencia de los algoritmos

A la hora de seleccionar un algoritmo para resolver un problema concreto debemos atender a los recursos que requiere, estos son **tiempo** y espacio. Estos se distribuyen en diseño, implementación y explotación, y dependen del hardware dónde se vaya a emplear el algoritmo, de la calidad del código (el creado por el desarrollador y el generado por el compilador), del algoritmo en sí, y del tamaño de las entradas.

Principio de invarianza:

El principio de invarianza establece que dos implementaciones de un mismo algoritmo no diferirán entre sí en tiempo de ejecución más que por una constante significativa.

La eficiencia es la medida del uso de los recursos en función del tamaño de las entradas, principalmente usamos $T(n)$ para representar el tiempo empleado para una entrada de tamaño n .

2.1. Medidas de la eficiencia

El análisis de algoritmos puede hacerse a posteriori, es decir, implementar el algoritmo y comprobar cuánto tiempo emplea en ejecutarse. En este caso es especialmente importante especificar el ordenador donde se realizan los test, sistema operativo, condiciones de ejecución, opciones de compilación, etc. Hace falta indicar dichos factores externos porque influyen multiplicativamente en los tiempos de ejecución. Este análisis a posteriori suele complementarse con un estudio teórico y un contraste teórico/experimental en el que, tras haber deducido una posible función que se ajuste a la función temporal del algoritmo, encontrar basándonos en los resultados empíricos la regresión que mejor se ajuste a la función. Este contraste teórico/experimental puede servir para detectar posibles errores de implementación y extraer conclusiones que podrían ser difíciles de ver solo con el análisis a posteriori. Llamamos, por tanto, enfoque a priori al estudio previo a la implementación, necesario para el contraste teórico/experimental y útil para determinar si sale o no rentable implementar un algoritmo.

Por lo general nos interesa estudiar el comportamiento de la función de eficiencia del algoritmo cuando los tamaños de entrada tienden a infinito. Decimos que un algoritmo tiene un tiempo de ejecución de orden $f(n)$ para una función f si existe una constante positiva c y una implementación del algoritmo capaz de resolver cada caso del problema en un tiempo acotado superiormente por $cf(n)$, donde n es el tamaño de problema considerado.

Como es preferible un análisis ajeno a factores externos, utilizamos las notaciones asintóticas, que indican como crece n asintóticamente sin considerar constantes debidas a la implementación y/o la máquina en que se ejecute:

- $O(T)$ Orden de complejidad de T .
- $\Omega(T)$ Orden inferior de T u Omega de T .
- $\Theta(T)$ Orden exacto de T .

Decimos que una función $T(n)$ es $O(f(n))$ si $\exists c \in \mathbb{R}, \exists n_0 \in \mathbb{N} | T(n) \leq cf(n) \forall n \geq n_0$

Así mismo, denominaremos orden de complejidad de f al conjunto de todas las funciones de \mathbb{R} acotadas superiormente por un múltiplo real positivo de f para valores de n suficientemente grandes: $O(f) = \{t : \mathbb{N} \rightarrow \mathbb{R}^+ \mid \exists c \in \mathbb{R}, \exists n_0 \in \mathbb{N} | \forall n \geq n_0 t(n) \leq cf(n)\}$

$\Omega(f(n))$ y $\Theta(f(n))$ no se usan tanto como $O(f(n))$ pero conviene conocer su significado intuitivo y matemático. Ω es similar a 'O grande' pero acota inferiormente en lugar de superiormente, es decir: decimos que una función $T(n)$ es $\Omega(f(n))$ si $\exists c \in \mathbb{R}, \exists n_0 \in \mathbb{N} | T(n) \geq cf(n) \forall n \geq n_0$. Por su parte, $\Theta(f(n))$ es la combinación de las dos anteriores; es decir, la eficiencia de un algoritmo es del orden exacto de $f(n)$, es decir, tiene $\Theta(f(n))$ sí tiene $O(f(n))$ y $\Omega(f(n))$.

2.1.1. Ejemplos:

- ¿ $3n^2 \in O(n^2)$? Si, tomando $c = 3$ y $n_0 = 1$.
- ¿ $3n^2 \in \Omega(n^2)$? Si, tomando $c = 1$ y $n_0 = 1$.
- ¿ $3n^2 \in \Theta(n^2)$? Si, porque $3n^2 \in O(n^2)$ y $3n^2 \in \Omega(n^2)$
- ¿ $2^{n+1} \in O(2^n)$? Si, $2^{n+1} = 2 \times 2^{n+1}$.
- ¿ $O(n) \in O(n^2)$? Si.
- ¿ $n^2 \in O(n^3)$? Si.
- ¿ $n^2 \in \Omega(n^3)$? No. n^3 acota superiormente a n^2 .
- ¿ $2n^2 \in \Theta(n^2)$? No, porque $3n^2 \in O(n^2)$ pero $2n^2 \notin \Omega(n^2)$
- ¿ $(2+1)^n \in O(2^n)$? No.
- ¿ $(n+1)! \in O(n!)$? No. $(n+1)! = (n+1) \times n!$
- ¿ $n^3 \in O(n^2)$? No.
- ¿ $n^3 \in \Omega(n^2)$? Si.
- ¿ $n^3 \in \Theta(n^2)$? No, porque $n^3 \notin O(n^2)$
- ¿ $(2+1)^n \in \Omega(2^n)$? Si.
- ¿ $n^2 \in O(n!!)$? No, ni en broma.

2.2. Calculando O grande de un algoritmo.

De cara a calcular O de un determinado algoritmo, debemos contemplar la posibilidad de ir simplificando $T(n)$ conforme se va calculando teniendo en cuenta que al final queremos determinar $O(n)$, es decir, si mientras calculamos $T(n)$ encontramos que alguna de las funciones que la componen acota a las demás (por ejemplo, si un trozo de código tiene $T(n) = n^3 + 7n + 7\log(n)$ podemos simplificarlo a n^3 para facilitar el resto de cálculos. Además, tendremos en cuenta los siguientes factores:

- Propiedad de transitividad: si $f \in O(g)$ y $g \in O(h) \rightarrow f \in O(h)$. (Similar para Ω)
- Propiedad de pertenencia: $O(f) = O(g) \leftrightarrow f \in O(g)$ y $g \in O(f)$.
- Propiedad de contenido: $O(f) \subseteq O(g) \leftrightarrow f \in O(g)$.
- Regla de la suma: $O(f+g) = O(\max(f,g))$.
- Regla del producto: trozos de código anidados (no independientes), es decir, bucles, se calculan multiplicando el orden del cuerpo del bucle por el número de iteraciones que especifica la cabecera del bucle.
- **Operación elemental:** operación de un algoritmo cuyo tiempo de ejecución se puede acotar superiormente por una constante. Nota: no todas las instrucciones son operaciones elementales, un ejemplo: $x = \maxA[k]$, $1 \leq k \leq m$ es una única instrucción pero, dado que contiene una llamada a una función, su orden no es constante.

- Existen bucles homogéneos (todas las iteraciones son iguales) y no homogéneos (cada iteración puede tener distintos órdenes, por ejemplo, si hay dos bucles anidados y el número de operaciones del bucle interior es distinto para cada iteración del exterior, pues depende del índice del bucle exterior.)
- Cuando analizamos un bucle del tipo **while** que tiene varias condiciones tenemos que buscar cual es la condición que más iteraciones provocaría y calcular el orden de eficiencia suponiendo que se ejecutara el bucle todas esas iteraciones.
- Cuando analizamos trozos de código que tienen llamadas a funciones tenemos que calcular el orden de la función y tenerlo en cuenta para la instrucción en la que es llamada.
- Entendemos que las llamadas a funciones de librerías, salvo que se diga lo contrario, tienen orden $O(1)$.
- A la hora de calcular una sumatoria podemos 'aproximar' su resultado realizando la integral correspondiente.
- Los logaritmos son del mismo orden independientemente de la base.

2.2.1. Ejercicio1: Demuestre las siguientes afirmaciones. ¿Qué quieren decir?

Si $f(n) = O(\log a)$, $a > 0$, entonces $f(n) = O(\log n)$:

Dado que a es una constante, es menor que $g(n) = n$ y, si $f(n)$ es del orden de $\log(a)$, entonces también es de orden $f(\log(n))$ porque $\log(a) \in O(\log(n))$.

Si $f(n) = o(g(n))$ entonces $f(n) + g(n) = O(g(n))$ cuando n tiende a ∞ : La notación "o pequeña" es similar a la "O grande" salvo que implica que $f(n)$ es **estrictamente** menor que $g(n)$. Esto supone, por la regla de la suma, que $f(n) + g(n)$ esté acotada asintóticamente por $g(n)$.

Si $O(f(n)) < O(g(n))$ entonces $O(f(n) + g(n)) = O(g(n))$. (Aplicar el apartado b):

Al igual que en el caso anterior, como $O(f(n)) < O(g(n))$, $g(n)$ acota superiormente a $f(n)$ y, por tanto, aplicando la regla de la suma tenemos que $O(f(n) + g(n)) = O(g(n))$.

Si $f(n)$ es la función logarítmica, polinómica o suma y/o producto de ellas, y para n potencia de 2 tenemos que otra determinada función $g(n)$ es $O(f(n))$. Entonces, si $g(n)$ es creciente, es $O(f(n))$ para todo n :

Estamos diciendo que $g(n)$ es $O(f(n))$ en sus valores potencias de 2. Si además $g(n)$ es creciente, esto significa que, dadas dos potencias de 2 consecutivas, no puede haber un punto intermedio a estas que tome valores mayores que los de la mayor; y, dado que la mayor de estas potencias está acotada asintóticamente por $f(n)$, entonces todos los puntos que había entre las dos potencias consecutivas también lo están y $g(n) \in O(f(n)) \forall n$.

3. Calculo de eficiencia de funciones recursivas

Para analizar el tiempo de ejecución de un procedimiento recursivo, le asociamos una función de eficiencia desconocida, $T(n)$, y la estimamos a partir de $T(k)$ para distintos valores de k .

(...)

El mejor método para resolver ecuaciones de recurrencia es entender la resolución de las ecuaciones lineales y homogéneas, que tienen el formato: $a_0 t_n + a_1 t_{n-1} + \dots + a_k t_{n-k} = 0$, y reducir todos los

demás tipos de ecuaciones de recurrencia a estas para poder resolverlas como una ecuación lineal y homogénea.

La forma de resolver este tipo de ecuaciones pasa por suponer que las soluciones son del estilo: $t(n) = x^n$
Entonces una ecuación $a_0t(n) + a_1t(n - 1) + \dots + a_kt(n - k) = 0$ puede transformarse en