Notas de complejidad

Gonzalo Sainz-Trápaga (GomoX)

24 de Julio de 2008 - http://www.pcmasmas.com

Este documento es una breve reseña sobre los temas de complejidad de algoritmos con el objeto de servir de base para la charla sobre técnicas algorítmicas del 26 de Julio de 2008. A continuación se explica brevemente qué es la complejidad temporal o espacial de un algoritmo.

1. ¿Qué es la complejidad?

Cuando se dice que un algoritmo es $O(algo)$ (que es una manera de expresar su complejidad) se está intentando caracterizar de forma aproximada cuanto tiempo tarda el algoritmo en procesar una entrada de un cierto tamaño. Esto se conoce como complejidad temporal. La complejidad espacial habla sobre la cantidad de memoria que requiere el algoritmo para ejecutarse. En breve, la complejidad dice si un algoritmo es rápido o lento comparado con otros.

2. Notación *Big-O*

Cuando se dice que un algoritmo es $O(n)$ (en tiempo, ya que salvo que se aclare que se está hablando de la complejidad espacial, la complejidad es temporal), esto significa que la cantidad de tiempo que tarda el algoritmo en procesar una entrada de tamaño n es proporcional a n. Por lo tanto, si la entrada es 2 veces más grande, es de esperarse que el algoritmo tarde el doble de tiempo en procesarla. Por el contrario, si el algoritmo es $O(n^2)$, si el tamaño de la entrada se duplica el algoritmo tardará aproximadamente el cuádruple ($2^2 = 4$) de tiempo en procesar.

En general, cuando un algoritmo es $O(f(n))$ (con f alguna función), esto dice que el algoritmo tarda aproximadamente $k * f(n)$ segundos en computar. El valor de k puede ser cualquiera, pero debe ser fijo para cualquier n que uno elija. La complejidad con esta notación es independiente de la computadora en que se ejecute el programa, por lo tanto da una idea del rendimiento del algoritmo que se puede comparar sin tener en cuenta cuestiones de *hardware*.

3. Mejor y peor caso

Salvo que se indique lo contrario, siempre se considera la complejidad de peor caso del algoritmo. Por ejemplo, si un algoritmo de ordenamiento puede tardar $O(n)$ en ordenar un arreglo que ya está ordenado (ya que solo tiene que corroborar que está ordenado, lo cual puede hacerse en $O(n)$), pero para alguna entrada posible de ese tamaño n tardará $O(n^2)$ (por ejemplo, el arreglo ordenado al revés), se considera que el algoritmo es $O(n^2)$.

4. Ejemplos de complejidad

Función	$n = 10$	$n = 100$	$n = 1000$	Algoritmos
1	1	1	1	Tomar el primer elemento de una secuencia
$log(n)$	1	2	3	Búsqueda binaria
n	10	100	1000	Bucket sort, Búsqueda lineal
$n * log(n)$	10	200	3000	Merge Sort, Quick Sort
n^2	100	10000	1000000	Bubble Sort, Algoritmo de Dijkstra
n^5	10^5	10^{10}	10^{15}	
2^n	10^3	10^{30}	10^{301}	Hallar los subconjuntos de un conjunto
$n!$	10^6	10^{157}	10^{2567}	Hallar las permutaciones de una secuencia
n^n	10^{10}	10^{200}	10^{3000}	Pocas cosas!

A los efectos prácticos, $log(n)$ se considera excelente, n también y $n * log(n)$ es casi tan bueno como n. Las complejidades de la forma n^k son dichas polinomiales (ya que n^k es un polinomio) y se consideran buenas o malas en función del valor de k - para $k > 5$ pueden ser muy costosas. Finalmente, las complejidades que están debajo de n^5 en la tabla se dicen exponenciales y se consideran malas al punto de que un problema cuyo mejor algoritmo tiene esta complejidad se dice "no resuelto" computacionalmente.