Análisis de algortimos

Cuando hablamos de analizar un algoritmo nos referimos a estudiar la cantidad recursos que tomará, por ejemplo:

- Tiempo
- Memoria
- Ancho de banda
- Escrituras a disco

Esta lista no es exhaustiva y en que aspecto nos interesa enfocarnos dependerá del tipo de algoritmo. Los más habitual es hablar de complejidad temporal (analizar el tiempo de ejecución) y de complejidad espacial (memoria a utilizar).

Si nos concentramos en el tiempo de ejecución deberíamos medir todas las operaciones hechas, lo lleva un trabajo demasiado arduo, por eso se toman las "operaciones más representativas" solamente, ya que para un análisis nos alcanza. Por ejemplo para un algoritmo de búsqueda nos concentraremos en la cantidad de comparaciones hechas. Para un algoritmo de ordenamiento también es común enfocarse en la cantidad de comparaciones, sin embargo la cantidad de escrituras a memoria, al intercambiar elementos, puede ser factor decisivo si en un sistema (hardware) las escrituras llevan mucho mas tiempo que una comparación.

Dado un algoritmo llamaremos T(n) a la función que calcula el tiempo que le llevará al algoritmo concluir su trabajo con una entrada de datos de tamaño n. Conocer esta función exactamente puede ser muy complicado, pero se puede acotar. El proceso general es simplificar la función, no prestando atención ni a las constantes (multiplicativas o aditivas) y dejando solo los términos de mayor peso, por ejemplo en un polinomio el término de mayor grado.

Los algoritmos se clasifican, se agrupan por tipos de complejidad. Queremos saber como va a crecer el tiempo de ejecución en función de la cantidad de datos, pero armamos "familias" de funciones que crecen "parecido". Para esto recurrimos al concepto de acotar. Una función f está acotada superiormente por una función g, si a partir de un cierto valor n_0 se cumple que $f(n) \leq k.g(n)$

Diremos que un algoritmo es "de orden" g, donde g es una función, si la función T(n) del algoritmo está acotada superiormente por la función g. Utilicemos el caso de la búsqueda lineal como ejemplo. Con elementos tomados al azar y n elementos en los que buscar, a veces lo encontrará en el primer elemento, a veces en el último. En promedio la cantidad de comparaciones será n/2. El ½ como factor multiplicativo lo descartamos y decimos que es de orden n, usando la notación "O grande" de este modo: O(n)

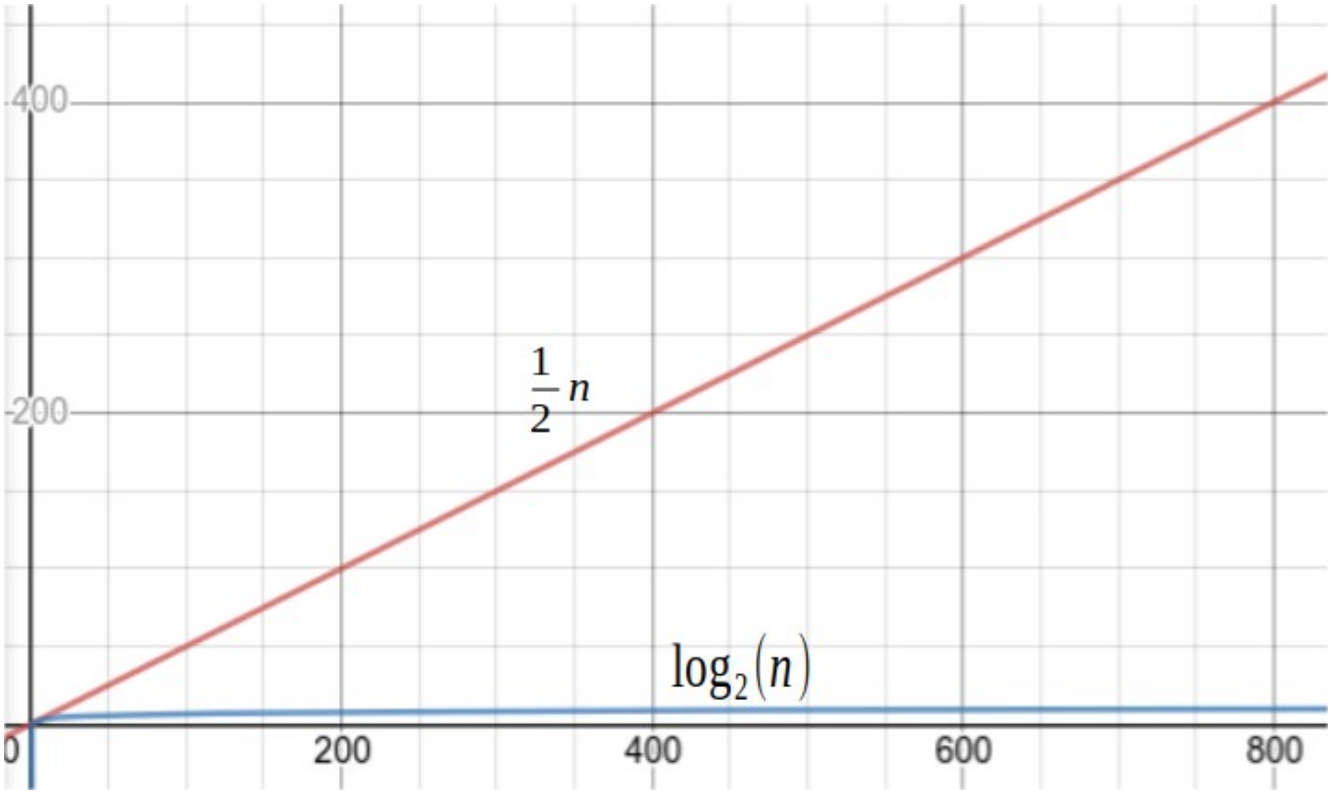
La cota superior no es la única que nos interesa, decimos que:

$f \in O(g)$ cuando f está acotada superiormente por g
$f \in \Omega(g)$ cuando f está acotada inferiormente por g
$f \in \Theta(g)$ cuando $f = O(g)$ y $f = \Omega(g)$.

El algoritmo de búsqueda binaria en cambio divide la cantidad de elementos en los que debe buscar por 2 en cada paso, por eso decimos que es O(log(n)) y el algunos casos, con un poco de abuso de notación simplemente se escribe O(log).

Siendo más específicos, Ω nos sirve para conocer el mejor caso, O para conocer el peor y Θ el caso promedio.

Comparación de búsqueda lineal vs binaria



Algoritmos de Ordenamiento

Algunas características que se estudian de los algoritmos de ordenamiento son:

Estabilidad: se dice que un algoritmo es estable si habiendo valores repetidos, no los intercambia al ordenar. Es decir, supongamos que hay un 4 en la posición 8 y otro en la posición 20, cuando termine de ordenar el 4 que estaba en la posición 8 estará en una posición menor que la del 4 que estaba en la posición 20.

Adaptabilidad: Se dice que un algoritmo es adaptable si la cantidad de operaciones puede variar de acuerdo a los datos. Por ejemplo el algoritmo de burbujeo puede detectar si el vector ya está ordenado y terminar sus ciclos antes, en cambio el algoritmo de selección no.

Comparación

Algoritmo	Mejor Caso	Caso Promedio	Peor Caso	Complejidad Espacial	Estable/ Adaptativo
Burbujeo	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$	O(1)	Si / Si
Selección	$\Omega(n^2)$	$\Theta(n^2)$	$O(n^2)$	O(1)	No / No
Inserción	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$	O(1)	Si / Si
Shell	$\Omega(n.\log n)$	$\Theta(n^{4/3})$	$O(n.\log^2 n)$	O(1)	No / Si
Quick	$\Omega(n.\log n)$	$\Theta(n.\log n)$	$O(n^2)$	O(n.log n)	No / No
Heap	$\Omega(n.\log n)$	$\Theta(n.\log n)$	O(n.log n)	O(1)	No / No

Esta tabla solo sirve para comparar en forma genérica. Sin embargo, en promedio, sabemos que el algoritmo de selección es más rápido que el de burbujeo y a su vez, el de inserción es mejor que el de

selección.

Por otra parte el algoritmo quick sort en promedio es mejor que heap sort, pero heap sort tiene un mejor peor caso. Shell sort no llega a ser logarítmico, pero es sub cuadrático.

Introducción a Quick Sort

Es un algoritmo que sigue el principio de dividir para conquistar. Divide el problema en partes más chicas, que resuelve (conceptualmente) en forma recursiva y luego compone esas soluciones parciales.

Los pasos son:

- Elige un elemento del vector a ordenar, al que llamaremos pivote. Como se elije ese elemento influye en desempeño del algoritmo y hay numerosas variantes al respecto. Lo idea sería elegir la mediana del conjunto.
- Separa en vector en 3 elementos, el pivote al medio, a su izquierda todos los elementos menores al pivote y la derecha todos los mayores.
- Aplica recursivamente lo mismo a las partes izquierda y derecha.