Ficha 21

Análisis de Algoritmos – Formalización

1.] Formalización de la notación *Big O*.

La notación *Big O* se usa para indicar un *límite superior* para el comportamiento esperado de un algoritmo en cuanto al tiempo de ejecución o el espacio ocupado o algún otro parámetro[1]. Si se dice que un algoritmo de ordenamiento tiene un tiempo de ejecución en el peor caso de $O(n^2)$, de alguna forma se está diciendo que ese algoritmo *no se comportará peor* que n^2 en cuanto al tiempo de ejecución. Puede decirse que la función *f* que calcula el tiempo de acuerdo al valor de *n*, siempre se mantendrá menor o igual que n^2 multiplicada por alguna constante *c* [1] [2].

Formalmente, decir que una función *f* está en el orden de otra función *g*, implica afirmar que eventualmente, para cualquier valor suficientemente grande de *n*, la función *f* siempre será *menor o igual* que la función *g* multiplicada por alguna constante *c* mayor a cero. En símbolos:

$$\text{Si} \quad f(n) \text{ es } O(g(n)) \quad \Rightarrow \quad f(n) \leq c*g(n)$$

(para todo valor *n* suficientemente grande y algún *c* > 0)

Para verlo mejor, analicemos la gráfica de la *Figura 1* (en *página 430*) [2]. En ella se supone que la función *f* es orden de *g*. A los efectos del ejemplo, no tiene importancia cuáles sean estrictamente las funciones *f* y *g*, sino sólo analizar lo que implica la relación *f(n) = O(g(n))*.

Si *f* es orden *g*, entonces podremos encontrar al menos una constante *c > 0* (que en la gráfica vale 2) tal que a partir de cierto número n_0 los valores de *g* multiplicados por *c* serán siempre mayores o iguales a los valores de *f*. Eso equivale a decir que la nueva función *c*g(n)* será mayor a ***f(n)*** para todo $n > n_0$. En el esquema gráfico, puede verse que a partir del valor n_0 la curva *2 * g(n)* se vuelve siempre mayor que la curva *f(n)*, con lo cual *f(n) = O(g(n))*.

Note que la relación *f(n) = O(g(n))* implica que *f* será menor o igual a *c * g* a partir de cierto valor n_0, y no necesariamente para valores pequeños de *n* (o sea, para valores de $n < n_0$). Es decir, lo que importa es lo que pasará para valores grandes o muy grandes de *n*, que es lo que se conoce como el *comportamiento asintótico de la función*.

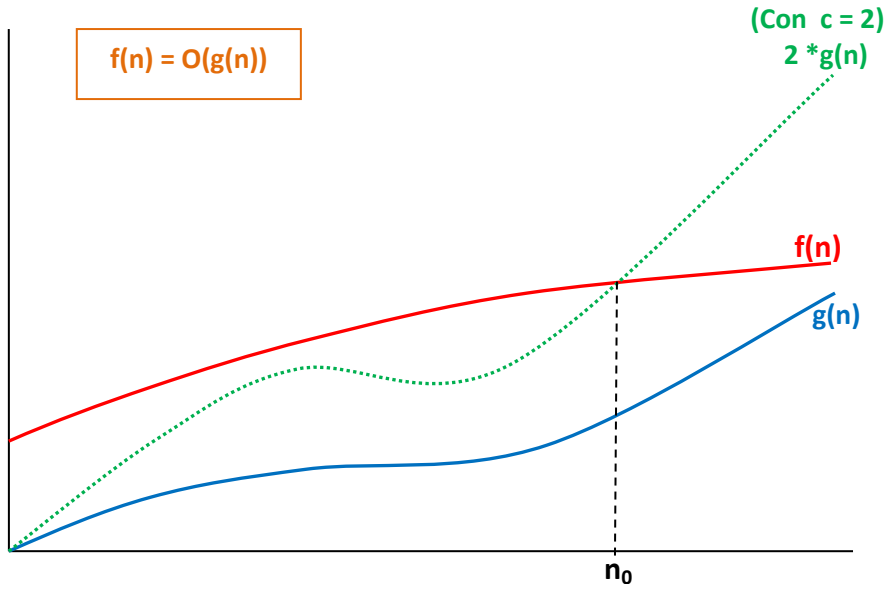
También note que la constante *c > 0* puede ser cualquiera y solo basta con encontrar una. Si puede encontrarse al *menos una constante c > 0* para la cual se pueda probar que *f(n) <= c*

[1] Parte del desarrollo de esta sección se basa en: Weiss, M. A. (2000). "*Estructuras de Datos en Java*". Madrid: Addison Wesley. ISBN: 84-7829-035-4 (página 103 y siguientes). También se han seguido ideas (sobre todo en el planteo de las gráficas) del material del curso "*Design and Analysis of Algorithms I*" – Stanford University (a cargo de *Tim Roughgarden*, Associate Professor): https://www.coursera.org/courses.

**g(n)* desde cierto punto n_0 en adelante, entonces *f* es orden de *g*. En la gráfica anterior, también hubiese sido válido suponer *c = 3* o *c = 4*, o cualquier valor de *c > 0* que cumpla la relación.

Si se observa con atención, lo que se está haciendo es intentar determinar la forma de variación de la función *f* de acuerdo a la forma en que varía otra función ***g*** ya conocida. Pero como en rigor no se usa necesariamente la propia ***g***, sino alguna función múltiplo de ***g*** (porque se multiplica a ***g*** por la constante *c*), entonces la expresión *O(g(n))* en realidad está implicando *una familia de funciones* cuyo comportamiento es caracterizado por el comportamiento de ***g***.

Figura 1: Idea general del significado de f(n) = O(g(n)).



En ese sentido, la expresión *O(g(n))* es lo que se conoce como un *orden de complejidad*: un conjunto o familia de funciones que se comportan asintóticamente de la misma forma. La función más característica de ese conjunto (la más simple, sin constantes ni términos independientes adicionales) es la que normalmente se usa para expresar la relación de orden, y se suele designar como *función representante* del conjunto (o *función característica* o también *función dominante*). Así, cuando decimos que la *búsqueda secuencial* tiene un tiempo de ejecución *t(n) = O(n)*, estamos diciendo que el tiempo *t* tiene la misma forma de variación asintótica que el conjunto de funciones representadas por *g(n) = n* (que es en este caso es la *función característica* de ese orden de complejidad). Como todas en *O(n)* se comportan de la misma forma que *g(n) = n*, no tiene relevancia entonces incluir constantes y términos independientes en la expresión de orden. No es necesario que el analista agregue detalles: decir (por ejemplo) *t(n) = O(2n + 5)* es asintóticamente lo mismo que *t(n) = O(n)*.

Y otro detalle a considerar: al expresar una relación de orden en notación *Big O*, se está indicando una *cota superior* para el comportamiento de una función *f* en términos del comportamiento de la familia de funciones *O(g)*, pero en general el analista buscará la *menor cota superior para f*. Está claro que si el tiempo de ejecución *t* de la búsqueda secuencial es *t(n) = O(n)*, es también cierto que $t(n) = O(n^2)$ (ya que como vimos $O(n) < O(n^2)$) y si se prosigue con ese argumento, resulta que prácticamente todos los algoritmos

conocidos son $O(2^n)$ (ya que la función exponencial o algún múltiplo de ella siempre será mayor que las demás para *n* grande). Pero al expresar una función *f* en notación *Big O*, lo que se quiere obtener es la familia de funciones que sirva como *mejor cota superior* para el comportamiento de *f*, y no cualquier familia que sea mayor o igual que *f* (ya que en este caso el análisis sería demasiado amplio y vago).

Todo lo anterior ayuda en el análisis de algoritmos: muchas veces un programador tiene un conocimiento estimado (o incluso intuitivo) del comportamiento de la función *f* que predice (por ejemplo) el tiempo de ejecución de un algoritmo en el peor caso, pero desconoce la forma analítica precisa de esa función (o a los efectos del análisis del peor caso no requiere de esa forma precisa)[2]. Si el programador puede probar que su función desconocida *f* es del orden de otra función conocida (y posiblemente más simple) *g*, entonces tendrá una *cota superior* para su función *f*: Sabrá que su algoritmo *nunca será peor* que *g* multiplicada por una constante dada. Esto puede parecer muy vago, pero en el análisis asintótico tiene mucha importancia: al fin y al cabo, saber que nuestro algoritmo nunca será peor que *n*log(n)* (por ejemplo) nos garantiza que ese algoritmo será subcuadrático incluso en el peor caso, aún si ignoramos los coeficientes precisos de la *verdadera* función *f*.

A partir de estos elementos, surgen algunas relaciones de orden básicas que debemos tener presentes, y que de alguna manera ayudan a justificar que en notación *Big O* se puede prescindir de las constantes y quedarse solo con el término o función dominante. Por otra parte, estas relaciones ayudan a poder estimar en forma rápida el comportamiento asintótico de un algoritmo sin entrar permanentemente en la necesidad de demostrar la relación de orden. No es necesario que estudie ni recuerde las demostraciones, pero viene bien tenerlas a mano [2]:

---

a.) Cualquier función polinómica en *n* de grado *k*, es orden n^k. Simbólicamente:

$$f(n) = a_k n^k + a_{k-1} n^{k-1} + a_2 n^2 + a_1 n + a_0 = O(n^k)$$

Demostración: si tomamos $c = |a_k| + |a_{k-1}| + |a_{k-2}| + \dots + |a_2| + |a_1| + |a_0|$ y $n_0 = 1$, entonces debemos probar que para todo $n > n_0 = 1$, se cumple que $f(n) <= c * n^k$. O sea:

$f(n) = a_k n^k + a_{k-1} n^{k-1} + a_2 n^2 + a_1 n + a_0$

$\Rightarrow$ $f(n) <= |a_k| * n^k + |a_{k-1}| * n^{k-1} + |a_{k-2}| * n^{k-2} + \dots + |a_2| * n^2 + |a_1| * n + |a_0|$

$\Rightarrow$ $f(n) <= |a_k| * n^k + |a_{k-1}| * n^k + |a_{k-2}| * n^k + \dots + |a_2| * n^k + |a_1| * n^k + |a_0| * n^k$

$\Rightarrow$ $f(n) <= (|a_k| + |a_{k-1}| + |a_{k-2}| + \dots + |a_2| * + |a_1| * + |a_0|) * n^k$

$\Rightarrow$ **$f(n) <= c * n^k$** que es lo que se quería probar.

Este resultado muestra que si sabemos que nuestra función *f* en *n* (desconocida) es polinómica de grado *k*, entonces *podemos prescindir de todas las constantes y de todos los términos que no sean de grado k*, y quedarnos sólo con el término dominante n^k, simplificando el análisis.

---

[2] En todo caso, buscar la forma analítica de una función que permita predecir el tiempo de ejecución de un algoritmo no es lo mismo que directamente predecir el futuro, como podía hacer el personaje de la película *Next* (del año 2007), dirigida por *Lee Tamahori* y protagonizada por *Nicolas Cage* y *Jessica Biel*. Se trata de la historia de Cris Johnson, un mago de Las Vegas que puede ver lo que ocurrirá en el futuro inmediato, pero sólo por un lapso de dos minutos en el futuro. Por causa de esa capacidad, el FBI lo busca y pretende detenerlo para que ayude a predecir y detener ataques terroristas, convirtiendo así la vida del mago en un martirio permanente.

b.) Sea $f(n) = 2^{n+10}$ y $g(n) = 2^n$. Entonces $2^{n+10} = O(2^n)$

Demostración: Debemos tomar dos constantes *c* y n_0 para las que se cumpla que $2^{n+10} <= c * 2^n$. Sabemos que:

$$2^{n+10} = 2^n * 2^{10} = 2^n * 1024$$

Por lo que si tomamos *c = 1024* sabemos que para *n0 = 1*, la relación postulada se cumple.

Procediendo en forma similar, se puede probar que $2^{n+k} = O(2^n)$ para cualquier $k >= 1$, lo cual también simplifica el análisis.

c.) En notación *Big O*, la base de los logaritmos puede obviarse, ya que la función $\log_b(n)$ es $O(\log_2(n))$ para cualquier base b > 1. Simbólicamente:

Sea $f(n) = \log_b(n)$ y $g(n) = \log_2(n) = \log(n)$. Entonces $\log_b(n) = O(\log(n))$ [con b > 1]

Demostración: Por defecto, asuma que la base 2 es implícita (o sea: asuma que $\log_2(n) = \log(n)$). Intentamos entonces probar que $\log_b(n) <= c * \log(n)$ para alguna constante cualquiera c > 0.

Sea $\log_b(n) = k$. Entonces (por definición de logaritmo): $b^k = n$. Tomemos la constante $p = \log(b)$. Entonces $2^p = b$.

Entonces, de los dos hechos anteriores resulta:

$n = b^k = (2^p)^k$
$n = 2^{pk}$
$\log(n) = p*k$
$\log(n) = p * \log_b(n)$
$\log_b(n) = 1/p * \log(n)$

Por lo tanto, si tomamos $c >= 1/p$, entonces $\log_b(n) <= c * \log(n)$ lo que prueba la relación 3. Hemos visto en fichas anteriores que este hecho resulta sumamente práctico cuando la relación de orden analizada incluye logaritmos: otra vez, se simplifica el análisis.

2.] Algunas consideraciones prácticas.

De ahora en adelante, se espera que el alumno sea capaz de al menos "intuir" una relación de orden para un algoritmo en el peor caso (es decir, dado un algoritmo, ser capaz de dar en notación *Big O* un orden para el peor caso de ese algoritmo en tiempo o espacio). Para ganar destreza en esa tarea, van algunos consejos prácticos:

i. Dado el algoritmo que debe analizar (expresado en forma de diagrama de flujo, o de pseudocódigo, o de programa fuente o incluso expresado en forma coloquial o intuitiva), tenga en claro *cuál será el factor de eficiencia* que quiere analizar: tiempo de ejecución o memoria empleada, por ejemplo).

ii. En ese algoritmo determine con exactitud el tamaño del problema (o volumen de datos a procesar). En muchos casos esto es simple de hacer (el tamaño *n* de un arreglo si se quiere ordenar ese arreglo o buscar en él) pero en algunos casos no es tan simple ni tan obvio (aunque no enfrentaremos casos extremos por ahora…) En algunos casos, el tamaño puede venir expresado con dos o más variables (por caso, si se pide procesar dos arreglos de

tamaños *m* y *n* respectivamente), y la expresión de orden final podría estar a su vez basada en dos o más variables (no es extraño encontrar expresiones de la forma *t(v, w) = O(v*log(w))* siendo *v* y *w* dos variables, o casos como $t(n, m) = O(m * n^2)$, por ejemplo).

iii. En el algoritmo analizado, determine claramente cuál es la *instrucción crítica* que ese algoritmo lleva a cabo. La instrucción crítica es la instrucción o bloque de instrucciones que se ejecuta más veces a lo largo de toda la corrida, y si está analizando el tiempo de ejecución del algoritmo, entonces la acumulación de los tiempos de ejecución de la operación crítica es lo que lleva al algoritmo a su tiempo final. En un ordenamiento, por lo general, la instrucción que más se ejecuta es la comparación (y lo mismo en una búsqueda) por lo que esa será la instrucción crítica.

iv. En este momento, tenga en cuenta que no es lo mismo realizar un conteo exhaustivo y riguroso de operaciones críticas, que hacer un análisis asintótico. Si lo que necesita es un *conteo exhaustivo*, entonces deberá extremar el análisis y tratar de expresar una fórmula con toda rigurosidad y detalle de constantes, términos no dominantes y términos independientes que indique cuántas operaciones críticas se ejecutan exactamente para un valor dado de *n*. Pero si lo que necesita es un *análisis asintótico*, entonces siga leyendo los puntos que vienen a continuación…

v. Para el *análisis asintótico*, si ya tiene una expresión de *conteo exhaustiva* identifique el *término dominante* en ella, y limítese a ese término. No diga que un algoritmo es $O(n^3 + n^2)$: por la *relación a.)* vista en *página 431*, en este caso bastará con decir $O(n^3)$. Entienda que para *n* suficientemente grande, el término n^2 será técnicamente despreciable frente a n^3. Si no tiene una expresión o fórmula exhaustiva, identifique en forma general la estructura del algoritmo que contiene a la operación crítica y deduzca en forma intuitiva. Valen los siguientes consejos para situaciones comunes, independientemente de otras situaciones que deberá resolver con otros criterios:

- Si la operación crítica es (por ejemplo) una comparación y está contenida dentro de dos ciclos anidados de *n* iteraciones cada uno, entonces su algoritmo es $O(n^2)$.
- Una operación crítica de tiempo de ejecución constante ($O(1)$) incluida dentro de *k* ciclos anidados de *n* iteraciones cada uno, tendrá tiempo de ejecución $O(n^k)$.
- Si su algoritmo consta de varios bloques de instrucciones independientes entre sí, entonces por la misma *relación a.)* de página 431 su algoritmo tendrá un tiempo de ejecución en el orden del que corresponda al bloque con mayor tiempo. Así, si su algoritmo tiene un primer bloque que ejecuta en tiempo $O(n)$ y luego dos bloques más que ejecutan en tiempos $O(n^2)$ y $O(log(n))$, entonces el tiempo completo sería $t(n) = O(n) + O(n^2) + O(log(n))$ pero esto es asintóticamente igual a $O(n^2)$, por lo que $t(n) = O(n^2)$.
- Si todo su algoritmo está compuesto sólo por un bloque de instrucciones de tiempo constante (asignaciones o condiciones, por ejemplo) sin ciclos ni procesos ocultos dentro de una función, entonces todo el algoritmo ejecuta en tiempo constante $t(n) = O(1)$ (ya que sería $t(n) = O(1) + O(1) + … + O(1)$ que es lo mismo que $t(n) = O(1)$ (por la misma relación a.) de *página 431*).
- Si su algoritmo se basa en tomar un bloque de *n* datos, dividirlo por 2, aplicar una operación de tiempo constante y luego procesar sólo una de las mitades de forma de volver a dividirla y continuar así hasta no poder hacer otra división, entonces su algoritmo ejecuta en tiempo $t(n) = O(log_2(n))$ que por la *relación c.* de *página 432*, es lo mismo que $t(n) = O(log(n))$ sin importar la base del logaritmo.

vi. No incluya *constantes* dentro de la expresión de orden, salvo aquellas que indiquen el exponente específico del término dominante. En general no dirá $O(2n)$ sino simplemente

O(n). La notación O le permite rescatar de un solo golpe de vista la forma de variación del término dominante, y en esa variación son en general despreciables las constantes.

vii. No se preocupe por la base del logaritmo si su expresión de orden incluye logaritmos. La relación 3 prueba que la base no es relevante en análisis asintótico. Y en todo caso, la base del logaritmo es ella misma una constante.

En el *punto iv* de estas consideraciones prácticas, hemos indicado que no es lo mismo un *conteo exhaustivo* que un análisis de *comportamiento asintótico*. El primero es mucho más riguroso. El segundo es más amplio. Tomemos por ejemplo el ya conocido *Ordenamiento de Selección Directa* para un arreglo *v* de *n* componentes, que en general se plantea así:

```
n = len(v)
for i in range(n-1):
    for j in range(i+1, n):
        if v[i] > v[j]:
            v[i], v[j] = v[j], v[i]
```

Si queremos hacer un *conteo exhaustivo* del número de operaciones críticas (en este caso, las *comparaciones*) que este algoritmo ejecuta, debemos ver que el proceso consiste en tomar la primera casilla del arreglo (designada como *pivot*), comparar su contenido con los *(n-1)* casilleros restantes y dejar el menor valor en la casilla *pivot*. Luego se toma la segunda casilla como *pivot*, se compara contra las *(n-2)* restantes, y se vuelve a dejar el menor de lo que quedaba del arreglo en la casilla *pivot*. Así, se hacen *(n-1)* pasadas, y cuando el *pivot* sea la casilla *(n-2)* se hará una única y última comparación más contra la casilla *(n-1)* y el arreglo quedará ordenado. Pero esto significa que la *cantidad de comparaciones* a realizar sale de:

```
Pasada 1:      n-1 comparaciones
Pasada 2:      n-2 comparaciones
Pasada 3:      n-3 comparaciones
...            ...
Pasada n-2:    2 comparaciones
Pasada n-1:    1 comparación
```

Lo anterior implica que el total de comparaciones *t(n)* a ejecutar para el total *n* de casilleros en el arreglo, será igual a la suma:

$$t(n) = 1 + 2 + \ldots + (n-3) + (n-2) + (n-1)$$

que se puede demostrar que es igual a:

$$t(n) = (n-1) * n / 2$$

y entonces:

$$t(n) = (n^2 - n) / 2$$

que tiene forma claramente cuadrática... Esto quiere decir que el algoritmo de *Selección Directa* hará una cantidad de comparaciones que será *exactamente* función de *n al cuadrado*, y por lo tanto el tiempo *t* demorado por ese algoritmo en ordenar el arreglo será *proporcional a n al cuadrado*. Hasta aquí, hemos desarrollado un *conteo exhaustivo* de operaciones críticas y la función obtenida expresa con todo detalle ese conteo. Si ahora se nos pide un análisis de *comportamiento asintótico*, el camino está allanado porque ya tenemos la fórmula precisa del *conteo exhaustivo*. El quinto punto de la lista de recomendaciones que hemos mostrado antes, nos lleva directamente a la expresión de orden: el tiempo de ejecución de este algoritmo, en notación *Big O*, es $t(n) = O(n^2)$,

prescindiendo de constantes y términos no dominantes. Pero aún sin tener la fórmula rigurosa del *conteo exhaustivo*, podríamos haber llegado a la misma conclusión simplemente analizando el código fuente (como también se indica en la quinta recomendación práctica): dos ciclos anidados de aproximadamente *n* repeticiones cada uno, y una instrucción condicional contenida en el ciclo más interno: $O(n^2)$.

Un análisis similar (tanto *exhaustivo* como *asintótico*) permite deducir que los otros algoritmos de *ordenamiento simples* o *directos* (como la ordenación por *Intercambio Directo* y la ordenación por *Inserción Directa* que ya hemos visto en la *Ficha 17*) también tienen un *comportamiento cuadrático* en el *peor caso*. El hecho es que el algoritmo de *Selección Directa* siempre hará la cantidad de comparaciones calculada más arriba, pero los otros dos podrían realizar una cantidad algo menor en *casos más favorables.* En otras palabras, *si sólo se consideran comparaciones* (y no por ejemplo la cantidad de veces que se hacen efectivamente intercambios), el método de *Selección* siempre cae en su peor caso, pero los otros dos podrían tener casos muy favorables dependiendo del estado inicial del arreglo. Los tres son de comportamiento cuadrático, y *asintóticamente pertenecen al mismo orden de complejidad*, pero las constantes que describen en forma analítica la función de *conteo exhaustivo* pueden ser diferentes.

En la siguiente tabla se muestran los tiempos de ejecución en notación *Big O* (comportamiento asintótico) para el peor caso de los algoritmos de ordenamiento vistos (o que veremos más adelante), más algunas consideraciones respecto de casos promedio o incluso mejores casos si esas situaciones son relevantes [3]:

Tabla 1: Comportamiento asintótico de los principales algoritmos de ordenamiento.

Algoritmo	Tiempo	Observaciones
Burbuja (Intercambio Directo)	$O(n^2)$ (peor caso)	Mejor caso: O(n) si el arreglo está ya ordenado.
Selección Directa	$O(n^2)$ (peor caso)	Siempre...
Inserción Directa	$O(n^2)$ (peor caso)	
Quicksort	**O(n*log(n)) (caso medio)**	Peor caso (depende de la implementación): **$O(n^2)$**.
Heapsort	O(n*log(n))	Caso medio: también O(n*log(n)).
Shellsort	$O(n^{1.5})$	Para la serie de incrementos mostrada en clase.

Los siguientes son problemas y/o algoritmos muy comunes para los que hacemos un breve análisis asintótico del *peor caso* en notación *Big O*. Intente rápidamente (en lo posible, sin mirar el análisis que hacemos para cada uno) hacer su propia estimación, a modo de ejercicio [4]:

- *Búsqueda secuencial de un valor x en un arreglo desordenado de n componentes*: La operación crítica es la comparación, y en el peor caso debe hacerse n veces (si x no está en el arreglo o está muy al final). Entonces para el peor caso resulta *t(n) = O(n)*.
- *Búsqueda secuencial de un valor x en un arreglo ordenado de n componentes*: La operación crítica es la comparación, y en el peor caso (otra vez) debe hacerse *n* veces (si *x* no está en el arreglo y es mayor a todos los elementos del mismo). Entonces para el peor caso también resulta *t(n) = O(n)*.
- *Búsqueda binaria de un valor x en un arreglo de n elementos (por supuesto, ordenado)*: La operación crítica es la comparación, y en el peor caso debe hacerse $log_2(n)$ veces (si *x* no está en el arreglo), ya que en ese caso se hacen $log_2(n)$ divisiones por 2, y una comparación por cada división. Entonces para el peor caso *t(n) = O(log(n))*.

- *Conteo de la cantidad de valores de un arreglo de tamaño n que son mayores al promedio de todos los valores del arreglo*: Hay un primer proceso en el cual deben hacerse n acumulaciones para calcular el promedio (hasta aquí, O(n)). Y luego un segundo proceso en el cual se hace n comparaciones para determinar cuántos valores son mayores que el promedio (con lo que se tiene otra vez O(n)). El tiempo total será $t(n) = O(n) + O(n) = O(2n) = O(n)$ con lo que $t(n) = O(n)$.
- *Multiplicación de matrices (suponga, para simplificar, que las matrices son cuadradas y del mismo orden n)*: Sin entrar en demasiados detalles, el algoritmo clásico emplea tres ciclos for de n iteraciones cada uno, y en cada giro realiza la acumulación de un producto. Por lo tanto, resulta $t(n) = O(n^3)$.
- *Conteo de las frecuencias de aparición de los n números de un conjunto, que pueden venir repetidos, sin conocer el rango en el que vienen esos números y usando un arreglo de objetos de conteo*: Cada uno de los *n* números debe buscarse secuencialmente en un arreglo que en el peor caso también llegará a tener *n* casilleros si todos los números fuesen diferentes. Como una búsqueda secuencia ejecuta en tiempo $t(n) = O(n)$, y debemos hacer *n* búsquedas, tenemos un tiempo final $t(n) = n * O(n)$ que es lo mismo que $t(n) = O(n * n) = O(n^2)$.
- *Conteo de las frecuencias de aparición de los n números de un conjunto, que pueden venir repetidos, conociendo el rango en el que vienen esos números y usando un arreglo de conteo directo*: Cada uno de los *n* números debe contarse en un arreglo de acceso directo (cada número se cuenta en la casilla cuyo índice coincide con el mismo número). Como un conteo de ese tipo ejecuta en tiempo constante $t(n) = O(1)$, y debemos hacer *n* conteos, tenemos un tiempo final $t(n) = n * O(1)$ que es lo mismo que $t(n) = O(n)$.
- *Fusión de dos arreglos ordenados (tamaños m y n) en un tercer arreglo ordenado*: En este caso, hay dos arreglos de entrada con tamaños *m* y *n* y hay que procesar todo el conjunto, por lo cual el tamaño del problema es $m + n$. El algoritmo esencial para hacer esta fusión, genera un arreglo de tamaño $m + n$ ordenado. Se recorren los dos arreglos originales, cada uno con su índice, se comparan dos elementos (uno de cada vector) y el menor se lleva al arreglo de salida. Como se hace una comparación por cada uno de los $m + n$ casilleros del arreglo de salida, el tiempo total resultante es $t(n) = O(m + n)$. Note que nada impide que la expresión de orden esté basada en dos o más variables.
- *Inserción y eliminación de un elemento en una pila o en una cola*: Sin importar cuántos elementos tenga la pila o la cola, la inserción y la eliminación proceden en tiempo constante (en nuestro caso, ambas estructuras han sido implementadas sobre un arreglo en Python, y en ambos casos, insertar o eliminar el elemento del frente o del fondo se hace en tiempo constante). Por lo tanto, para todas las operaciones pedidas, el tiempo de ejecución resulta ser $t(n) = O(1)$.

Anexo: Temas Avanzados

En general, cada Ficha de Estudios podrá incluir a modo de anexo un capítulo de *Temas Avanzados*, en el cual se tratarán temas nombrados en las secciones de la Ficha, pero que requerirían mucho tiempo para ser desarrollados en el tiempo regular de clase. El capítulo de *Temas Avanzados* podría incluir profundizaciones de elementos referidos a la programación en particular o a las ciencias de la computación en general, como así también explicaciones y demostraciones de fundamentos matemáticos. En general, se espera que el alumno sea capaz de leer, estudiar, dominar y aplicar estos temas aún cuando los mismos no sean específicamente tratados en clase.

a.) Otras notaciones típicas del análisis de algoritmos.

En ocasiones, se desea poder indicar no ya un *límite o cota superior*, sino (por ejemplo) un *límite o cota inferior* para una función *f* dada. Para ese y otros casos, existen otras notaciones de orden (que solo citaremos a modo documentativo, aunque no usaremos frecuentemente) [1] [2].

Si se quiere expresar que el tiempo de ejecución o el espacio de memoria ocupado por un algoritmo *no se comportará mejor* que cierta función *g*, se usa la notación *Omega (Ω)*. Si estamos analizando el tiempo de ejecución de un algoritmo, entonces al decir que el tiempo de ese algoritmo no se comportará mejor que cierta función *g*, queremos decir que los valores de tiempo de ejecución estarán siempre *por arriba* (o a lo sumo serán iguales) de los calculados para esa función límite. Así, por ejemplo, puede verse que si se considera la cantidad de comparaciones, el algoritmo de *Selección Directa* no sólo es $O(n^2)$, sino también $\Omega(n^2)$ (léase: *Omega n cuadrado*): simplemente, no lo hará ni mejor ni peor que algún múltiplos o submúltiplo de n^2.

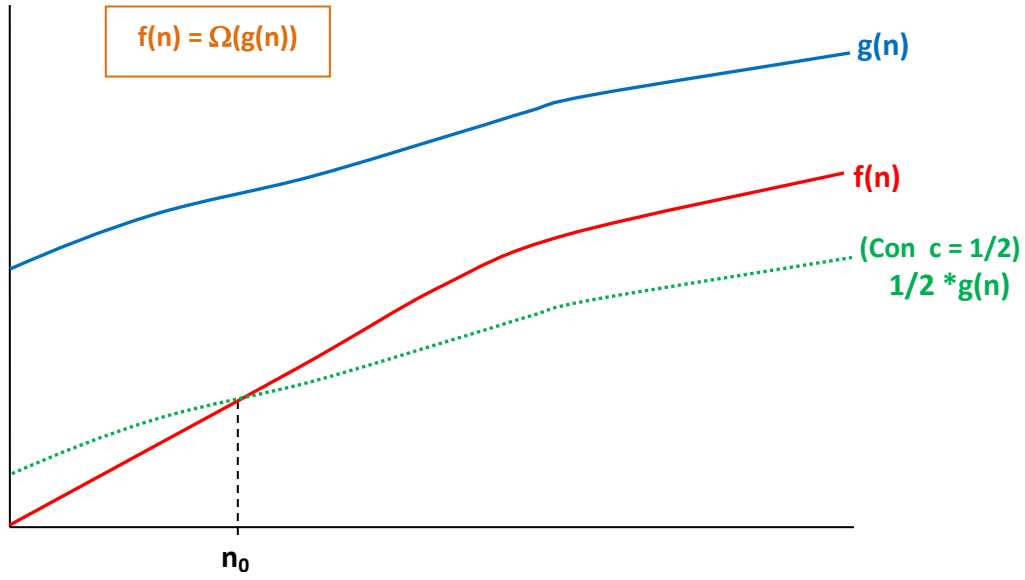
Formalmente, la función *f* que mide el tiempo o el espacio (o cualquier otro factor) de un algoritmo de acuerdo a los valores de *n*, es $\Omega(g(n))$ si se ajusta a la siguiente descripción:

$$\text{si } f(n) \text{ es } \Omega(g(n)) \Rightarrow f(n) \geq c*g(n)$$

(para n suficientemente grande y algún $c > 0$)

Gráficamente, si *f* es *omega g*, entonces podremos encontrar al menos una constante *c* mayor a cero para la cual los valores *f(n)* serán siempre mayores o iguales a *c * g(n)*, a partir de cierto número n_0:

Figura 2: Idea general del significado de f(n) = Ω(g(n)).



Por otra parte, si se quiere indicar que una función *f* se comporta tanto por arriba como abajo *de la misma forma* que múltiplos de otra función *g*, se puede usar también la notación *Theta (Θ)*: si *f(n)* es *Θ(g(n))*, significa que *f* puede acotarse tanto superior como inferiormente por múltiplos de *g*. En otras palabras:

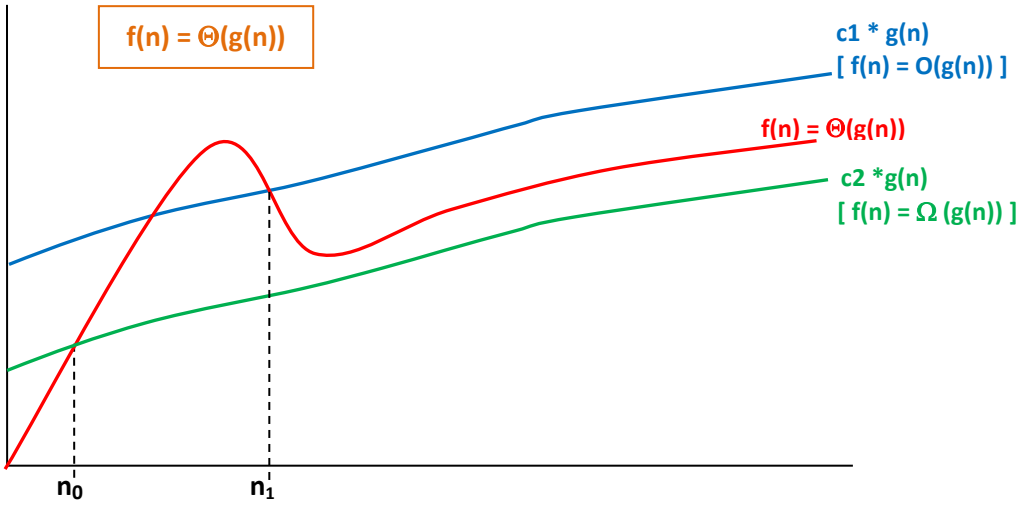
$$\text{Si } f(n) \text{ es } \Theta(g(n)) \Rightarrow f(n) \text{ es } O(g(n)) \text{ y } f(n) \text{ es } \Omega(g(n))$$

(para *n* suficientemente grande)

La notación *Theta* es muy precisa: no solo nos proporciona una cota superior para el análisis asintótico de una función, sino también *que también nos garantiza que esa cota superior es la mejor posible*. Note que este es el caso del algoritmo de Selección Directa respecto a $g(n) = n^2$ al considerar comparaciones: el algoritmo no será mejor que una función cuadrática y una función cuadrática es la mejor aproximación que puede darse tanto superior como inferior. Se dice entonces que ese algoritmo es $\Theta(n^2)$ (y se lee: *Theta n cuadrado*).

Gráficamente, si *f* es *Theta g*, entonces la gráfica de *f* podrá "encerrarse" entre dos curvas proporcionales a *g*, para dos constantes *c1* y *c2* diferentes:

Figura 3: Idea general del significado de f(n) = Θ(g(n)).



Finalmente, otra notación que también suele usarse, es la notación *o()* (léase: *o minúscula* o *little o*). Esta notación es similar a *O()*, pero considerando *sólo relación de menor estricto* (sin el signo igual). Es útil cuando se quiere indicar una función *estrictamente superior* para el comportamiento de un algoritmo. Así, si podemos decir que un algoritmo dado es (por ejemplo) $O(n^{1.5})$ en cuanto a cierta cantidad de operaciones críticas, entonces se puede indicar también a ese algoritmo como $o(n^2)$: es una forma elegante (aunque menos precisa) de decir que el algoritmo analizado es *subcuadrático*... En símbolos:

Si f(n) es o(g(n)) ⇒ f(n) < c*g(n)

(para *n* suficientemente grande y algún *c* > *0*)

Con el siguiente cuadro mostramos un resumen de las cuatro notaciones (tomado y adaptado del libro *Estructuras de Datos en Java*, de *Mark Allen Weiss – página 116*):

Tabla 2: Resumen de notaciones típicas del análisis de algoritmos.

Expresión	*Significado*
f(n) es O(g(n))	El crecimiento de f(n) es ≤ que el crecimiento de g(n)
f(n) es Ω(g(n))	El crecimiento de f(n) es ≥ que el crecimiento de g(n)
f(n) es Θ(g(n))	El crecimiento de f(n) es = que el crecimiento de g(n)
f(n) es o(g(n))	El crecimiento de f(n) es < que el crecimiento de g(n)

Créditos

El contenido general de esta Ficha de Estudio fue desarrollado por el *Ing. Valerio Frittelli* para ser utilizada como material de consulta general en el cursado de la asignatura *Algoritmos y Estructuras de Datos* – Carrera de Ingeniería en Sistemas de Información – UTN Córdoba, en el ciclo lectivo 2019.

Actuaron como revisores (indicando posibles errores, sugerencias de agregados de contenidos y ejercicios, sugerencias de cambios de enfoque en alguna explicación, etc.) en general todos los profesores de la citada asignatura como miembros de la Cátedra, que realizaron aportes de contenidos, propuestas de ejercicios y sus soluciones, sugerencias de estilo de programación, y planteo de enunciados de problemas y actividades prácticas, entre otros elementos.

Bibliografía

[1] M. A. Weiss, Estructuras de Datos en Java - Compatible con Java 2, Madrid: Addison Wesley, 2000.

[2] Coursera, "Take the world's best courses, online, for free.," 2012. [Online]. Available: https://www.coursera.org/. [Accessed 27 March 2013].

[3] R. Sedgewick, Algoritmos en C++, Reading: Addison Wesley - Díaz de Santos, 1995.

[4] V. Frittelli, Algoritmos y Estructuras de Datos, Córdoba: Universitas, 2001.