Universidad del Valle de Guatemala
Teoría de la Computación, CC2019
Docentes: Alan Reyes; Tomás Gálvez P.
Semestre 2, 2022

UVG
UNIVERSIDAD DEL VALLE DE GUATEMALA

Análisis de Algoritmos y Notaciones Asintóticas

El **análisis de algoritmos** se refiere a la estimación de los recursos (normalmente tiempo y memoria) que un algoritmo requiere para funcionar mediante el estudio de su estructura y sus operaciones, muchas veces con el objetivo de encontrar el algoritmo más eficiente o adecuado para resolver un problema específico.

Un algoritmo toma argumentos de entrada y produce un resultado dependiente de dichos argumentos. El consumo de recursos de un algoritmo depende de las características de estos argumentos, y suele describirse este consumo en función del "tamaño" de los argumentos. El tamaño puede tener diferentes interpretaciones, y ciertamente la interpretación que tomemos afectará el resultado de un análisis. Por ejemplo, podemos considerar el tamaño como la cantidad de elementos en un arreglo, la cantidad de bits con la que se representan números; o incluso el número de nodos y aristas en un grafo. Nuestros análisis miden cantidades de operaciones, que varían conforme al nivel de abstracción que tomemos.

Aunque se puede analizar un algoritmo buscando eficiencia en cuanto a memoria usada o energía eléctrica consumida, lo más común es que se busque determinar el **tiempo de ejecución** (***running time*** en inglés) que se mide como el número de pasos u operaciones primitivas realizadas. Las operaciones primitivas de una computadora varían en cuanto al tiempo que toma cada una, pero consideramos que cada instrucción realizada (o línea de pseudocódigo en un algoritmo) toma un tiempo constante.

La razón para tal imprecisión va desde que el verdadero tiempo de ejecución de una instrucción primitiva depende de la máquina física que la ejecuta, hasta que, en términos de eficiencia, lo más importante será(n) la(s) instrucción(es) que más peso tenga(n) sobre el cálculo del tiempo de ejecución. Si decidimos ejecutar un algoritmo que ordene un arreglo de números, no será tan importante la diferencia de tiempo de ejecución entre una computadora nueva y una de hace dos años. Sí podríamos percibir una diferencia significativa, sin embargo, entre los tiempos que tome ordenar un arreglo de diez números y uno de un millón.

Debemos observar que para un mismo tamaño de argumentos podemos tener variaciones en el tiempo de ejecución de un algoritmo. Estas variaciones ocurren debido a los diferentes posibles argumentos con igual tamaño que se pueden alimentar a un algoritmo, clasificados en tres escenarios: ***worst-case, average-case*** y ***best-case*** (peor caso, caso promedio y mejor caso, en español). Veamos un par de ejemplos para clarificar:

<u>Algoritmo del máximo</u>

1. Sean $j = n, k = n$ y $m = X[k]$
2. Si $k = 0$, terminar
3. Si $X[k] \leq m$ ir al paso 5
4. $j = k, m = X[k]$
5. $k = k - 1$
6. Ir al paso 2

¿Cuántas veces se ejecuta cada paso? Claramente, el paso 1 es una inicialización por lo que debe ocurrir una única vez. Dado que k es inicializado con n, y que decrece de uno en uno, se va a verificar el valor de k (paso 2) por cada elemento en el arreglo más una última verificación; o sea, $n + 1$ veces. Cuando se realice el paso 2 por última vez ya no será necesario realizar los pasos 3, 4, 5 y 6; por lo que la cantidad de veces que los pasos 3, 5 y 6 se ejecutarán será n.

El único problema lo presenta el paso 4 ya que, como al definir el algoritmo no conocemos los valores del arreglo X, no sabemos cuántas veces vamos a tener que actualizar nuestro resultado. En el mejor de los casos (*best-case scenario*) el paso 4 nunca se realiza ya que el valor con el que inicia nuestro algoritmo ($X[n]$) es el máximo, y por lo tanto no hay necesidad de actualizarlo. En el peor de los casos (*worst-case scenario*) el arreglo viene ordenado en forma descendente, y por lo tanto por cada vez que ocurra el paso 3 ocurrirá también el paso 4 (o sea, n veces).

Si designamos la cantidad de veces que se ejecuta el paso 4 con la variable a, el número de pasos de nuestro algoritmo será $T(n) = 1 + (n+1) + n + a + n + n = 4n + 2 + a$ en general. En el mejor caso a vale 0, y el resultado es $4n + 2$. En el peor caso es $5n + 2$ porque a vale n. Tomando en cuenta el tiempo constante c que suponemos que cuesta cada paso tendríamos que el peor tiempo de ejecución es $5cn + 2c$, y el mejor es $4cn + 2c$. Pero, como veremos más adelante, incluso esta abstracción se vuelve innecesaria. De momento podemos hacer la observación de que al comparar el mejor y el peor caso, $4cn + 2c < 5cn + 2c$, podremos eliminar la constante dividiendo ambos lados de la desigualdad dentro de c, lo cual expresa la independencia del análisis respecto al tiempo que toma cada paso.

Ejercicio de práctica A: explique cómo funciona el siguiente algoritmo de ordenamiento por inserción (*insertion sort*) y responda: ¿qué pasos cambiarán su cantidad de ejecuciones según los valores en el arreglo? ¿Cómo los podemos contar? Identifique sus *best-case* y *worst-case scenarios*. ***Hint***: vea el apéndice.

1. $for\ j = 2\ to\ n$
2. $\quad k = A[j]$
3. $\quad i = j - 1$
4. $\quad while\ i > 0\ and\ A[i] > k$
5. $\quad\quad A[i+1] = A[i]$
6. $\quad\quad i = i - 1$
7. $\quad A[i+1] = k$

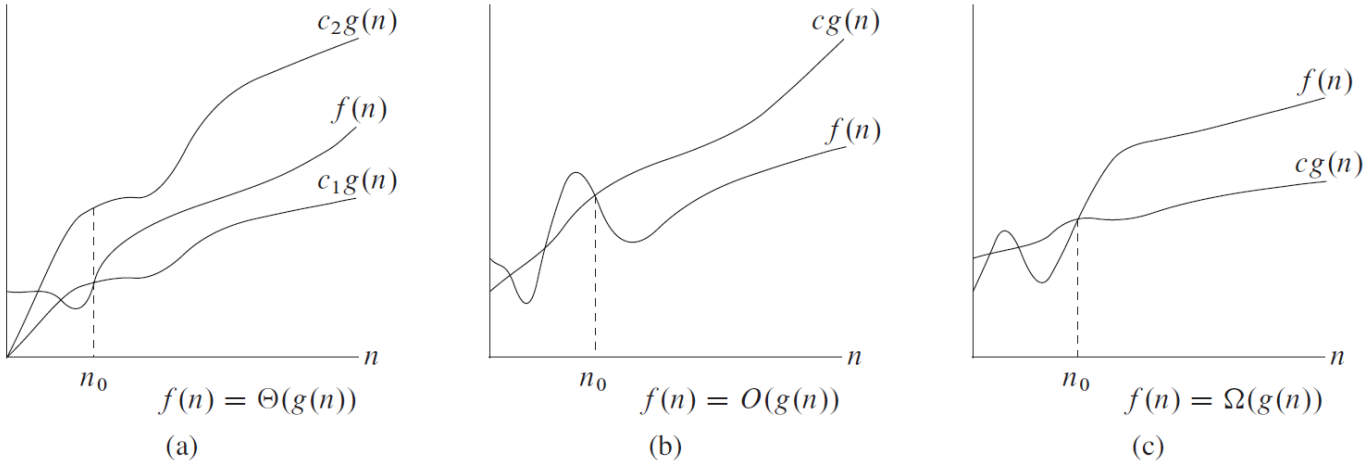
¿De aquí a diez años podría ser éste el único algoritmo necesario para ordenar arreglos? No. El incremento en capacidad del *hardware* es motivado por una demanda de mejor desempeño. En otras palabras, conforme más podemos/tenemos, más queremos. El tamaño de la entrada irá incrementando con la capacidad del *hardware*. Para un algoritmo ineficiente esto significa tener cada vez peor desempeño.

Si armamos una fórmula para contar las operaciones del *insertion sort*, a los coeficientes en el mejor caso los podemos resumir en constantes a y b, así como en el peor caso en a, b y c. Para el mejor caso tendremos una función lineal como tiempo de ejecución, mientras que para el peor caso tendremos una cuadrática. Lo que nos interesa para comparar dos algoritmos en cuanto a su tiempo de ejecución es precisamente el grado de la función que lo define, ya que éste determina su **tasa de crecimiento**.

La tasa de crecimiento es representada con **notación asintótica**, y la llamamos así porque expresa las funciones a las que una tasa de crecimiento se acerca (y que, a veces, alcanza) conforme el tamaño de la entrada del algoritmo crece más allá de cierto punto. La notación asintótica tiene tres formas populares:

- **Notación *theta*** ($\Theta(g(n))$): describe un conjunto de funciones tales que las imágenes de las mismas están, a partir de cierto n_0, inclusivamente entre algunos múltiplos positivos de la función $g(n)$. Es decir, $\Theta(g(n)) = \{f(n) : \exists c_1, c_2, n_0 > 0 \mid \forall n \geq n_0, 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)\}$
- **Notación *big-Oh*** ($O(g(n))$): describe el conjunto de funciones tales que, a partir de cierto n_0, las imágenes de las mismas están debajo de, o son iguales a, algún múltiplo positivo de la función $g(n)$. Es decir, $O(g(n)) = \{f(n) : \exists c, n_0 > 0 \mid \forall n \geq n_0, 0 \leq f(n) \leq cg(n)\}$
- **Notación *big-Omega*** ($\Omega(g(n))$): describe el conjunto de funciones tales que, a partir de cierto n_0, las imágenes de las mismas están siempre arriba de, o son iguales a, algún múltiplo positivo de la función $g(n)$. Es decir, $\Omega(g(n)) = \{f(n) : \exists c, n_0 > 0 \mid \forall n \geq n_0, 0 \leq cg(n) \leq f(n)\}$

La siguiente figura ilustra cada una de las notaciones:



Aunque lo correcto es decir que alguna función pertenece (*e.g.*, $f(n) \in O(g(n))$) a alguno de los conjuntos descritos por las notaciones, lo habitual es decir que la función "es" (*e.g.*, $f(n) = O(g(n))$) alguno de los conjuntos. Otra posible peculiaridad en la notación ocurrirá cuando digamos que algún tiempo de ejecución es $O(1)$ para expresar que es acotado por una constante.

Tomando en cuenta que

$$f(n) = \Theta(g(n)) \Leftrightarrow f(n) = O(g(n)) \wedge f(n) = \Omega(g(n))$$

podemos observar que el *worst-case scenario* del *insertion sort* es tanto $O(n^2)$ como $\Omega(n^2)$, pero el tiempo de ejecución del *insertion sort* en general (o, sea, tomando en cuenta el *worst-case* y el *best-case*) es $O(n^2)$ y $\Omega(n)$. Es importante que no asociemos las notaciones O y Ω directamente con el *best-case* y *worst-case* de un algoritmo. Las notaciones asintóticas son herramientas para expresar cotas al tiempo de ejecución de un algoritmo, y la expresión correcta depende del resultado de un análisis caso por caso.

Llamamos cota **ajustada** a aquella que es descrita por la notación Θ, indicándonos que múltiplos positivos de una misma función acotan al tiempo de ejecución (que es otra función) por arriba y por abajo. Cuando una misma función solo puede acotar por arriba o por abajo, esa función provee una cota no ajustada.

El algoritmo de ordenamiento *selection sort* busca el elemento más pequeño en un arreglo y lo intercambia con el elemento en la primera posición. Luego busca el segundo más pequeño y lo intercambia con el segundo elemento. Continúa así hasta ordenar el arreglo completo. Se presenta el pseudocódigo a continuación:

```
ssort(A):
1.    for i=1 to n-1:
2.          for j=i+1 to n:
3.                if A[j]<A[i]:
4.                      t=A[j]
5.                      A[j]=A[i]
6.                      A[i]=t
```

Ejercicio de práctica B: deduzca los tiempos de ejecución para el *best-case* y el *worst-case scenario* del *selection sort*, y expréselos en notación Θ.

Usar las notaciones asintóticas en ecuaciones y desigualdades expresa que en su posición puede ir cualquier función que pertenezca al conjunto que la notación describe. El uso de notaciones asintóticas en ecuaciones facilita el trabajo de análisis de tiempos de ejecución desconocidos, como en relaciones de recurrencia con la forma

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$$

Esta ecuación expresa que el tiempo de ejecución de su algoritmo correspondiente depende de dos ejecuciones recursivas de sí mismo sobre un *input* con la mitad del tamaño original, y que cada ejecución realiza un conjunto de tareas que toman un tiempo lineal adicional (sobre el tamaño de entrada original).

Las funciones $g(n)$ que se usan como "argumento" en las notaciones asintóticas constan únicamente del término de orden más grande, sin coeficientes, en la función $f(n)$. Esto debido a que, para los valores de n que son suficientemente grandes, las constantes y los términos de orden menor tienen comparativamente un efecto insignificante sobre el tiempo de ejecución. Esto provoca que en ocasiones un algoritmo con tasa de crecimiento de cierto orden sea más eficiente que otro con tasa de orden menor en las etapas tempranas (o sea para valores pequeños de n), pero mucho menos eficiente en las etapas avanzadas.

Ejercicio de práctica C: desarrolle implementaciones en Python para *insertion sort* y *merge sort*. Cree un *driver script* que:

- Genere un arreglo de 1000 enteros al azar.
- Ejecute *merge sort* e *insertion sort* sobre el arreglo, tomando sus tiempos de ejecución con `time.perf_counter()`.
- Genere un arreglo de 10 enteros al azar.

- Repita la ejecución y medida de tiempos con *merge sort* e *insertion sort*. Compare y comente sobre los tiempos de ejecución de ambos con las diferentes cantidades de números.

Nota: puede usar el lenguaje de programación que desee, aunque tendrá que buscar documentación sobre contadores de tiempo que emplear para las mediciones.

Una variación de las notaciones *big-Oh* y *big-Omega* son, respectivamente, la ***little-Oh* (*o*)** y ***little-Omega* (*ω*)**, y la diferencia es que las notaciones en minúscula proveen una cota estricta para las funciones que pertenecen a ellas.

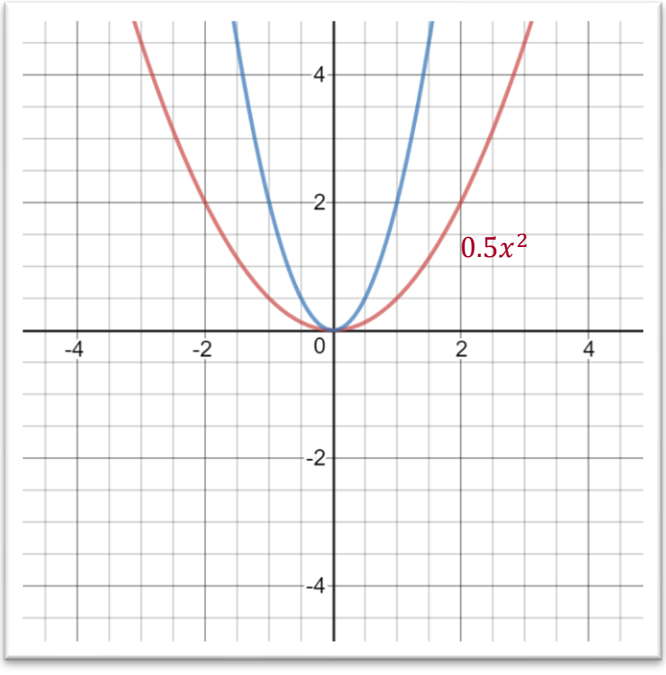
Se definen formalmente las notaciones *little* a continuación:

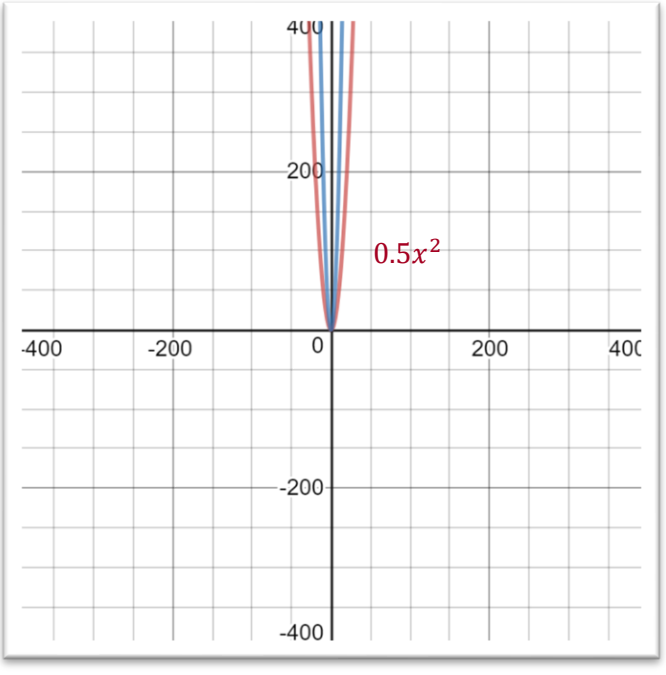
- $o\big(g(n)\big) = \{f(n) : \forall c > 0 : \exists n_0 > 0 \mid \forall n \geq n_0, 0 \leq f(n) < cg(n)\}$
- $\omega\big(g(n)\big) = \{f(n) : \forall c > 0 : \exists n_0 > 0 \mid \forall n \geq n_0, 0 \leq cg(n) < f(n)\}$

Obsérvese cómo las notaciones *little* imponen una cota estrictamente superior o inferior (*e.g.*, *little-oh* usa $<$ en lugar de $\leq$). Las definiciones indican que $g(n)$ crece/decrece a tal velocidad que no importa que tan cercana/lejana a cero elijamos a c, siempre encontraremos un $n_0 \in \mathbf{N}$ correspondiente a partir del cual $g(n)$ sea tan grande/pequeño que $cg(n)$ será mayor/menor que $f(n)$.

En las notaciones *little*, la notación asintótica expresa que la función $g(n)$ cambia a un ritmo diferente (mayor o menor, no parecido) que el de $f(n)$; conforme n se acerca a infinito, $f(n)$ se vuelve insignificante con respecto a $g(n)$ (*i.e.*, $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$) o viceversa.

Consideremos la función $f(x) = 2x^2$. Esta función es claramente $\Theta(x^2)$ dado que si $c_1 = 1$ y $c_2 = 3$ entonces $0 \leq c_1x^2 \leq f(x) \leq c_2x^2$ para todo $x_0 \geq 0$, y por lo tanto x^2 es una cota ajustada para $f(x)$. Para la función $h(x) = 2x$, por otro lado, x^2 no es una cota ajustada ya que, aunque $h(x) = O(x^2)$ (con $c = 1$ y $n_0 = 1$), $h(x) \neq \Omega(x^2)$. Esto último lo sabemos al suponer por contradicción que existen constantes c y x_0 tales que $0 \leq cx^2 \leq 2x, \forall x \geq x_0$. Al dividir dentro de x obtenemos $0 \leq cx \leq 2, \forall x \geq x_0$, lo que claramente es imposible ya que, por muy pequeña que sea la constante c, x alcanzará valores suficientemente altos para incumplir la desigualdad, conforme se acerca a infinito. También podemos asegurar que $h(x) = o(x^2)$, ya que para cualquier constante $c > 0$ que multiplique a x^2 siempre habrá un punto de intersección con $2x$ a partir del cual $2x < cx^2$. Esto no es posible con $f(x)$ (*i.e.*, $f(x) \neq o(x^2)$) porque fácilmente podemos elegir $c = 0.5$ y obtener lo siguiente, donde se ilustra que $f(x) \geq 0.5x^2, \forall x \in \mathbb{R}$:





Ejercicio de práctica D: use https://www.desmos.com/calculator para desarrollar gráficas interactivas de las funciones $f(x) = ax$ y $g(x) = bx^2$, donde a y b son constantes. Compruebe hasta donde pueda que no importa qué tan grande sea a, siempre habrá un b a partir del cual $f(x) < g(x)$. Puede comprobar también lo inverso (no importa qué tan pequeño sea a, siempre habrá b a partir del cual $f(x) > g(x)$).

Ejercicio de práctica E: demuestre que $f(n) + o(f(n)) = \Theta(f(n))$, donde $f(n)$ es asintóticamente positiva. Apóyese en la equivalencia $f(n) = \Theta(g(n)) \Leftrightarrow f(n) = \Omega(g(n)) \wedge f(n) = O(g(n))$, aunque ésta no haya sido demostrada todavía

Universidad del Valle de Guatemala
Teoría de la Computación, CC2019
Docentes: Alan Reyes; Tomás Gálvez P.
Semestre 2, 2022

UVG
UNIVERSIDAD DEL VALLE DE GUATEMALA

Apéndice: forma cerrada de $\sum_{j-1}^{n} j$ y $\sum_{j-1}^{n}(j-1)$

$$\sum_{j=1}^{n} j = 1 + 2 + \cdots + (n-1) + n$$

Lo cual, visto en orden invertido, es:

$$\sum_{j=1}^{n} j = n + (n-1) + \cdots + \big(n - (n-1)\big)$$

Entonces, al sumar ambas ecuaciones:

$$2\sum_{j=1}^{n} j = n + [(n-1)+1] + [(n-2)+2] + \cdots + \big[\big(n-(n-1)\big) + (n-1)\big] + n$$

$$= n * n + n = n(n+1)$$

Por lo tanto:

$$\sum_{j=1}^{n} j = \frac{n(n+1)}{2}$$

Luego veremos que:

$$\sum_{j=1}^{n}(j-1) = (1-1) + (2-1) + \cdots + \big((n-1)-1\big) + (n-1) = [1 + 2 + \cdots + (n-1) + n] + n(-1)$$

$$= 1 + 2 + \cdots + (n-1)$$

que es más fácil de comprender si observamos que $\sum_{j=1}^{n}(j-1) = \sum_{j=1}^{n} j - \sum_{j=1}^{n} 1$. Además:

$$\sum_{j=1}^{n}(j-1) = (n-1) + (n-2) + \cdots + \big(n-(n-1)\big) + 0$$

Por lo que, al sumar ambas formas de esta sumatoria y luego dividir entre 2, obtendremos:

$$\sum_{j=1}^{n}(j-1) = \frac{[1+(n-1)] + [2+(n-2)] + \cdots + [(n-1) + \big(n-(n-1)\big)]}{2} = \frac{n(n-1)}{2}$$

ya que cada corchete resulta en una n, y tenemos $n-1$ corchetes que sumar.

Fuentes:

- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms.* Massachusetts: The MIT Press.
- Knuth, D. E. (1997). *The Art of Computer Programming: Fundamental Algorithms.* Addison Wesley Longman.
- Massachusetts Institute of Technology. (2003). *16.070 Introduction to Computers & Programming.* Retrieved from Massachusetts Institute of Technology: http://web.mit.edu/16.070/www/lecture/big_o.pdf
- https://archive.org/details/handbuchderlehre01landuoft
- https://archive.org/details/dieanalytischeza00bachuoft