Universidad del Valle de Guatemala
Análisis y Diseño de Algoritmos
Docentes: Paulo Mejía; Tomás Gálvez P.
Semestre 1, 2023

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.

1. ¿Qué es un algoritmo? Explique sus propiedades.

Secuencia de pasos ordenados, precisos, definidos y finitos que resuelven un problema. La precisión se refiere a que el resultado de ejecutar los pasos es predecible. La definición a que cada paso debe ser un único paso en relación con los demás. Finito se refiere a la cantidad de pasos que conforman el algoritmo.

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.

Algoritmo de ordenamiento por inserción (*insertion sort*)

2. ¿Cómo funciona el siguiente algoritmo?

```
1.  for j = 2 to n
2.      k = A[j]
3.      i = j − 1
4.      while i > 0 and A[i] > k
5.          A[i + 1] = A[i]
6.          i = i − 1
7.      A[i + 1] = k
```

Este algoritmo ordena los números en un arreglo A de tamaño n recorriendo las posiciones ascendentemente, e insertando los valores que se encuentran en la posición pasada (o presente) que les corresponda, corriendo todos los números necesarios a la derecha.

Obviamente este algoritmo presenta un pseudocódigo diferente al del máximo, pero esto no es de importancia. Veremos que el paso 1 se realizará n veces ya que un ciclo for incluye la verificación de que su índice haya alcanzado el límite especificado. Aunque j sea inicializado con 2, la verificación extra que se hace cuando $j > n$ (en este ejemplo, al igual que en el anterior, suponemos que los índices del arreglo van de 1 a n). Igual que en el ejemplo anterior, la última verificación evita que se ejecuten los demás pasos, por lo que los pasos 2, 3 y 7 se realizarán $n - 1$ veces.

Otra vez desconocemos los valores del arreglo que se dio como argumento, entonces si un valor es insertado entre otros no sabemos cuántos valores se tuvieron que desplazar hacia la derecha para ello. Esto nos impide determinar con exactitud la cantidad de veces que se realizarán los pasos 4, 5 y 6. Podemos asignar la variable t_j al número de veces que se van a verificar las condiciones del ciclo $while$ en la iteración j del ciclo for. Entonces,

3. ¿Cuántas veces ocurrirá el paso 4, en total? ¿Cuántas veces ocurrirán los pasos 5 y 6?

El paso 4 ocurrirá $\sum_{j=2}^{n} t_j$ veces y, siguiendo la lógica de que las verificaciones de un ciclo siempre ocurren una vez más, los pasos 5 y 6 ocurrirán $\sum_{j=2}^{n}(t_j - 1)$ veces.

Pongámonos quisquillosos y determinemos que el costo de cada uno de los pasos es diferente. Para el paso i será c_i.

4. La fórmula del tiempo total de ejecución será entonces…

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^{n} t_j + c_5 \sum_{j=2}^{n} (t_j - 1) + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7(n-1)$$

5. Identifique el *best-case scenario* y deduzca su fórmula.

El *best-case scenario* se dará cuando el arreglo dado ya esté ordenado, ya que la verificación del ciclo $while$ se realizará sólo una vez por cada iteración (o sea, $t_j = 1$); lo que resulta en:

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_3(n-1) + c_4(n-1) + c_7(n-1) \\ &= (c_1 + c_2 + c_3 + c_4 + c_7)n - (c_2 + c_3 + c_4 + c_7) \end{aligned}$$

6. Identifique el *worst-case scenario* y deduzca su fórmula.

El *worst-case scenario* ocurre cuando el arreglo viene ordenado de forma descendente ya que por cada posición j que visitemos en el arreglo tendremos que ir a insertar su contenido hasta la primera posición, lo cual implica que el paso 4 realizará j veces la verificación $A[i] > k$ (es decir, $t_j = j$). En este caso:

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4\left(\frac{n(n+1)}{2} - 1\right) + c_5\left(\frac{n(n-1)}{2}\right) + c_6\left(\frac{n(n-1)}{2}\right) + c_7(n-1)$$

$$= \left(\frac{c_4}{2} + \frac{c_5}{2} + \frac{c_6}{2}\right)n^2 + \left(c_1 + c_2 + c_3 + \frac{c_4}{2} - \frac{c_5}{2} - \frac{c_6}{2} + c_7\right)n - (c_2 + c_3 + c_4 + c_5 + c_6 + c_7)$$

Esto se debe a que:

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

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. **Nota**: el "-1" incluido en el paréntesis que multiplica a la constante c_4, en la respuesta a la pregunta 6, se debe a que la sumatoria en el algoritmo inicia en realidad desde 2, no desde 1. En el cálculo de las sumatorias empezamos desde 1 porque es más fácil encontrarlas así en la literatura. Ese "-1" no aparece en los paréntesis que multiplican a las constantes c_5 y c_6 porque ¿qué pasa en esas sumatorias cuando $j = 1$?

7. ¿De aquí a diez años podría ser éste el único algoritmo necesario para ordenar arreglos? ¿Por qué?

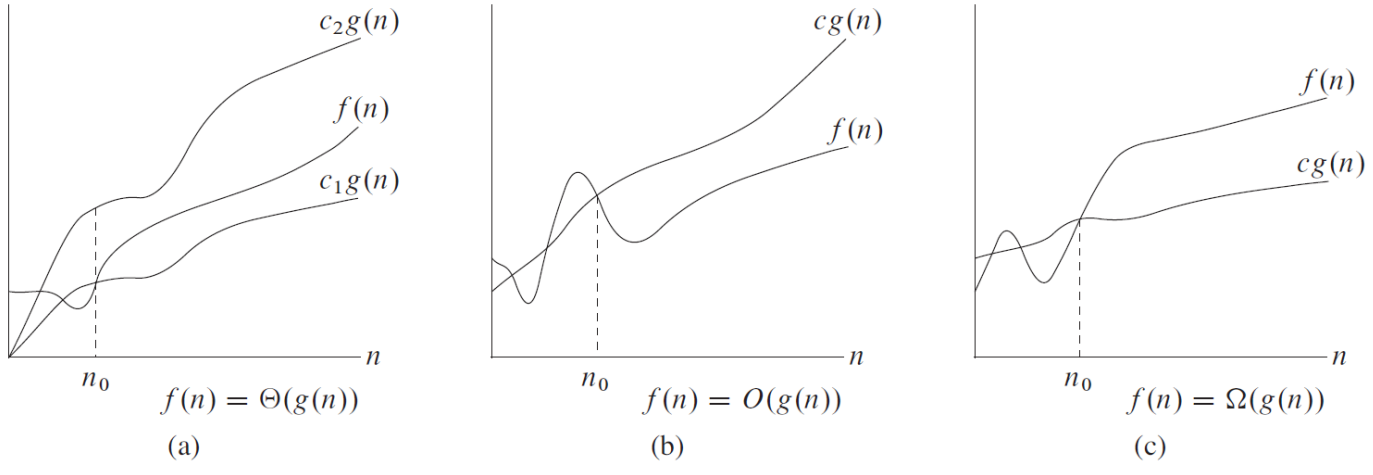
No. El incremento en capacidad del *hardware* es motivado por una demanda de mejor desempeño. En otras palabras, conforme más podemos/tenemos, más queremos. El tamaño de la entrada irá incrementando con la capacidad del *hardware.* Para un algoritmo ineficiente esto significa tener cada vez peor desempeño.

Si a los coeficientes en el mejor caso del *insertion sort* los resumimos 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 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, un 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, un 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:



8. ¿Qué significa que la tasa de crecimiento del *worst-case* del *insertion sort* sea cuadrática, en cuanto a la gráfica de su tiempo de ejecución?

Si x es el tamaño de la entrada, el algoritmo tendrá un tiempo de ejecución con comportamiento parecido a x^2 conforme x crezca.

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
```

9. Deduzca el tiempo de ejecución para el *best-case scenario* del *selection sort* y expréselo en notación Θ.

El paso 1 se realizará n veces. Para cada iteración del paso 1 se ejecutará $n - i + 1$ veces el paso 2, *i.e.,* $\sum_{i=1}^{n-1}(n - i + 1) = n(n-1) - \left[\frac{n(n+1)}{2} - n\right] + (n-1) = \frac{n(n-1)}{2} + (n-1) = \frac{(n+2)(n-1)}{2} = \frac{n^2+n-2}{2}$. El paso 3 se realizará $\frac{n(n-1)}{2}$ veces. En el mejor de los casos, el arreglo ya viene ordenado, entonces nunca se cumple el `if`. Esto provocaría que los pasos 4, 5 y 6 se realicen 0 veces. El tiempo de ejecución del *best-case scenario* será (con $c_1 = 1, c_2 = 2, n_0 = 1$):

$$n + \frac{(n+2)(n-1)}{2} + \frac{n(n-1)}{2} = n + \frac{(2n+2)(n-1)}{2} = n + (n+1)(n-1) = n^2 + n - 1 = \Theta(n^2)$$

10. Deduzca el tiempo de ejecución para el el *worst-case scenario* del *selection sort* y expréselo en notación Θ.

El tiempo de ejecución del *worst-case scenario* será:

$$n + \frac{(n+2)(n-1)}{2} + 4\left(\frac{n(n-1)}{2}\right) = n + \frac{(5n+2)(n-1)}{2} = n + \frac{5n^2-3n-2}{2} = \frac{5n^2-n-2}{2}$$

Como existen constantes $c_1 = \frac{1}{2}, c_2 = 5$ tales que $0 \leq \frac{n^2}{2} \leq \frac{5n^2-n-2}{2} \leq 5n^2, \forall n \geq 1$, este tiempo es $\Theta(n^2)$.

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 (ver ejemplo *sortingtimes.py*).

11. Escriba y explique una fórmula que describa el tiempo de ejecución de un *insertion sort* recursivo con el cual, para ordenar un arreglo $A[1..n]$, primero se ordena de forma recursiva el arreglo $A[1..(n-1)]$ y luego se inserta el elemento $A[n]$ en la posición correcta, el subarreglo ordenado.

$T(n) = T(n-1) + O(n)$. El uso de $T(n-1)$ denota la recursión y $O(n)$ indica que el tiempo máximo que se puede tardar el proceso de inserción de $A[n]$ es lineal (hallar la posición correcta para la inserción puede requerir recorrer todo el arreglo).

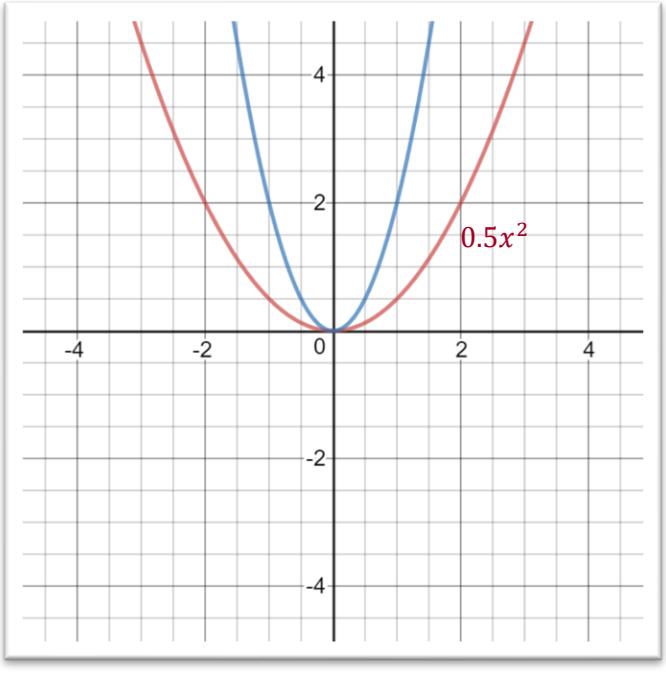
Una variación de las notaciones *big-Oh* y *big-Omega* son, respectivamente, la ***little-Oh* (o)** y ***little-Omega* (ω)**, y la diferencia es que las notaciones en minúscula proveen una cota estricta para las funciones que pertenecen a ellas.

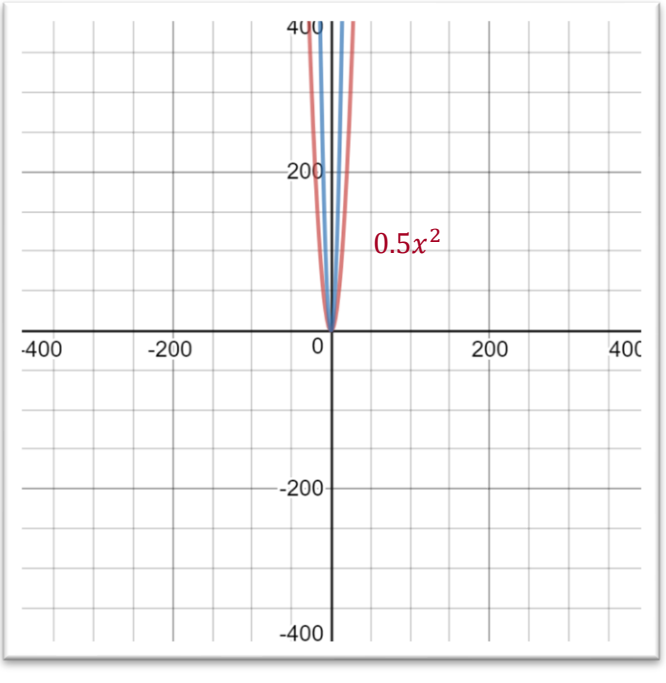
Se definen formalmente las notaciones *little* a continuación:

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

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 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 eventualmente alcanzará valores suficientemente altos para incumplir la desigualdad. 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 $2x < cx^2$ (ver ejemplo interactivo en: https://www.desmos.com/calculator/oszckpqrzo). 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:





12. Demuestre que $\boldsymbol{f(n) + o(f(n)) = \Theta(f(n))}$, donde $\boldsymbol{f(n)}$ es asintóticamente positiva.

Sea $h(n) = o(f(n)) \Rightarrow \forall c_o > 0, \exists n_o > 0 : 0 \leq h(n) < c_o f(n), \forall n \geq n_0$. Por otra parte, $0 \leq c_\Omega f(n) \leq f(n) + h(n)$ cuando $c_\Omega = 1$ y n_0 es cualquier n tal que $h(n) > 0$, entonces $f(n) + h(n) = \Omega(f(n))$. Esto significa que si tomamos $c_o = c_\Omega$, existe un n_0 a partir del cual $0 \leq h(n) < c_\Omega f(n)$. Como $f(n)$ es asintóticamente positiva, a partir de cierto n_f tendrá valores positivos, por lo que tomamos $n' = \max(n_o, n_f)$. Si sumamos $f(n)$ a la desigualdad que define $h(n) = o(f(n))$ obtenemos $0 \leq f(n) \leq h(n) + f(n) < c_\Omega f(n) + f(n) = (c_\Omega + 1)(f(n)), \forall n \geq n'$.

Para responder la pregunta anterior nos apoyamos en la equivalencia $f(n) = \Theta(g(n)) \Leftrightarrow f(n) = \Omega(g(n)) \wedge f(n) = O(g(n))$. Ésta no ha sido demostrada. Su demostración se requiere como parte de la tarea para este tema.

13. ¿Cómo se podría modificar casi cualquier algoritmo de manera que se obtenga un buen tiempo de ejecución para su *best-case scenario*? *Hint*: supongamos que se conoce la forma del resultado final, de modo que se puede verificar si un resultado propuesto es válido o "correcto".

En el caso de algoritmos de ordenamiento podríamos revisar si la lista ya está ordenada antes de cualquier operación. En general podríamos simplemente producir un resultado aleatorio del tipo necesitado y revisar si es el correcto. Ambos métodos completarían la ejecución del algoritmo en el tiempo que tome verificar el resultado, aunque la posibilidad de esto se restrinja a los mejores casos.

Universidad del Valle de Guatemala
Análisis y Diseño de Algoritmos
Docentes: Paulo Mejía; Tomás Gálvez P.
Semestre 1, 2023

UVG
UNIVERSIDAD DEL VALLE DE GUATEMALA

Ejercicios

1. Describa un algoritmo con tiempo de ejecución $O(n \log_2 n)$ tal que, dados un conjunto S de n números enteros y un entero arbitrario x, determine si existen o no dos números en S cuya suma sea exactamente x. Puede suponer que el arreglo está ordenado. ***Hint***: considere usar, como parte de su algoritmo, al algoritmo de búsqueda binaria, cuyo tiempo de ejecución es $O(\log_2 n)$.
2. La **Regla de Horner** dice que se puede evaluar un polinomio $P(x) = \sum_{k=0}^{n} a_k x^k$ de la siguiente manera:

   $$a_0 + x\big(a_1 + x(a_2 + \cdots + x(a_{n-1} + xa_n)\ldots)\big)$$

   El siguiente trozo de pseudocódigo implementa esta regla para un conjunto de coeficientes a_i dado:

   ```
   1. y=0
   2. for i=n downto 0:
   3.    y=a_i+x*y
   ```

   Calcule una cota ajustada para el tiempo de ejecución de este algoritmo.

3. Escriba código *naïve* para la evaluación de un polinomio (suponga que no hay una instrucción primitiva para calcular x^y). Compare las tasas de crecimiento de este código y el que implementa la Regla de Horner. **Nota:** una solución *naïve* se refiere a la implementación de una solución de la forma más simple posible, haciendo uso de funciones u operaciones fundamentales y conocidas. Por ejemplo, la forma *naïve* de encontrar un elemento en un diccionario es ver la primera palabra y determinar si es la que buscamos. Si no es, pasamos a la siguiente y repetimos el proceso.
4. Para dos funciones $f(n)$ y $g(n)$ demuestre que $\max\big(f(n), g(n)\big) = \Theta\big(f(n) + g(n)\big)$.
5. Argumente por qué, para constantes reales cualesquiera a y $b > 0$, $(n + a)^b = \Theta(n^b)$. ***Hint***: puede investigar o deducir la forma expandida $(n + a)^b$ para apoyar su respuesta.
6. ¿Es $2^{n+1} = O(2^n)$? ¿Es $2^{2n} = O(2^n)$?
7. Demuestre las siguientes propiedades:
   a. $f(n) = \Theta\big(g(n)\big) \Leftrightarrow f(n) = O\big(g(n)\big) \wedge f(n) = \Omega\big(g(n)\big)$.
   b. $o\big(g(n)\big) \cap \omega\big(g(n)\big) = \emptyset$.
   c. $f(n) = O\big(g(n)\big) \Rightarrow \log_2 f(n) = O(\log_2 g(n))$, donde sepamos que $\log_2 g(n) \geq 1$ y $f(n) \geq 1$ para n suficientemente grande (*i.e.*, para $n \geq n_0$ con algún n_0).

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