AA2021a-02-Analisis-Algoritmos 1

Abdiel E. Cáceres González

20 de febrero de 2021

Índice

Código de Honor

La UJAT espera que sus estudiantes muestren respeto por el orden, la moral y el honor de sus compañeros, sus maestros y su persona. Por ello, se establece este Código de Honor, con el propósito de guiar la vida escolar de los alumnos. Estos lineamientos no permiten todo acto que deshonre los ámbitos académicos. Las siguientes acciones son consideradas como violatorias al Código de Honor de:

1. Usar, proporcionar o recibir apoyo o ayuda no autorizada al presentar exámenes y al elaborar o presentar reportes, tareas y en general en cualquier otra forma por la que el maestro evalúe el desempeño académico del estudiante.
2. Presentar trabajos o exámenes por otra persona, o permitir a otro que lo haga por uno mismo.

Las sanciones podrán ser desde la reprobación con calificación 0 (cero) en la tarea por evaluar, hasta una calificación reprobatoria de 0 (cero) en la materia.

PARTE DOS
Análisis de algoritmos

1. Introducción

1.1. Fundamentos matemáticos

Hsy ciertos conceptos matemáticos que se deben utilizar a lo largo de este curso. El primero de ellos es el **suelo** y el **techo** de un número real. Decimos que «suelo de x», y lo escribimos como $\lfloor x \rfloor$, y es el mayor número entero que es menor o igual que x. Así $\lfloor 2.3 \rfloor = 2.0$ porque 2 es el máximo número entero que es menor o igual que 2.3; otro ejemplo, $\lfloor -6.7 \rfloor = -7.0$. El techo de un número entero se escribe $\lceil x \rceil$, y es el menor número entero que es máyor que x; así $\lceil 2.3 \rceil = 3.0$ y $\lceil -6.7 \rceil = -6.0$.

Utilizamos las funciones suelo y techo cuando necesitamos determinar cuántas veces se hace algo, y el valor depende de alguna fracción de los elementos. Por ejemplo, si comparamos un conjunto de n elementos en pares, donde el primer valor se compara con el segundo, el tercero se compara con el cuarto y así en adelante, el número de comparaciones es $\lfloor \frac{n}{2} \rfloor$. Si $n = 10$ se deben hacer $\lfloor \frac{10}{2} \rfloor = 5$ comparaciones, pero si son $n = 11$ elementos, también se hacen las mismas 5 comparaciones, ya que el último elemento no tiene pareja con la que se compara.

El número **factorial** de un número entero n se escribe $n!$, y es el producto de los primeros n números enteros positivos. Por ejemplo $3! = 1*2*3 = 6$; $6! = 1*2*3*4*5*6 = 720$. El número factorial crece muy rápidamente. Este algoritmo lo vamos a estudiar con detalle más adelante.

1.1.1. Funciones suelo y techo

Para cualquier número real x, denotamos $\lfloor x$ al entero más grande que es menor o igual que x [y se lee «el suelo de x»] y escribimos $\lceil x \rceil$ para referirnos al número entero más pequeño que es mayor o igual que x.

$$x - 1 < \lfloor x \rfloor \leq x \leq \lceil x \rceil < x + 1$$

Para cualquier número entero:

$$\left\lceil \frac{n}{2} \right\rceil + \left\lfloor \frac{n}{2} \right\rfloor = n,$$

y para cualquier número real $x \geq 0$ y enteros $a, b > 0$,

1. $\left\lceil \frac{\lceil x/a \rceil}{b} \right\rceil = \left\lceil \frac{x}{ab} \right\rceil$
2. $\left\lfloor \frac{\lfloor x/a \rfloor}{b} \right\rfloor = \left\lfloor \frac{x}{ab} \right\rfloor$
3. $\left\lceil \frac{a}{b} \right\rceil \leq \frac{a + (b-1)}{b}$
4. $\left\lfloor \frac{a}{b} \right\rfloor \leq \frac{a + (b-1)}{b}$

1.1.2. Logaritmos

Como los **logaritmos** juegan un papel muy importante en la práctica de análisis de algoritmos, hay algunas propiedades que debemos recordar y tener presente. El logaritmo base b de un número x, que se escribe $\log_b x$, es la potencia de b que debe producir el número x. Por ejemplo el logaritmo de 8 en base 2 se escribe $\log_2 8 = 3$, porque se debe elevar el 2 a la potencia 3 para obtener 8. Otro ejemplo, $\log_{10} 45 = 1.653$. Por convención, cuando no se especifica la base en una expresión log, la base por defecto es 10, entonces

$\log 45 = \log_{10} 45$. Otro convencionalismo es la notación ln, que representa el logaritmo natural que tiene base e, así $\ln 30 = \log_e 30 \simeq 3.4012$. En computación se utiliza la base 2, y se usa la notación lg para el logaritmo en base 2; entonces $\lg 16 = \log_2 16 = 4$.

El logaritmo es una función estrictamente creciente. Esto significa que para dos números x y y, $\forall b : x > y \iff \log_b x > \log_b y$. El logaritmo es una función inyectiva, esto significa que $\log_b x = \log_b y \iff x = y$. Otras propiedades importantes de los logaritmos son:

1. $\log_b 1 = 0$
2. $\log_b b = 1$
3. $\log_b(xy) = \log_b x + \log_b y$
4. $\log_b x^y = y \log_b x$
5. $\log_a x = \dfrac{\log_b x}{\log_b a}$

1.1.3. Árboles binarios

Un **árbol binario** es una estructura en donde cada nodo en el árbol puede tener a lo más dos nodos hijos. y cada nodo tiene exactamente un nodo padre. El nodo en la cima del árbol es el único nodo que no tiene padre y se le llama la «raíz» del árbol. Un árbol binario que tiene n nodos tiene al menos $\lfloor \lg n \rfloor + 1$ niveles, cuando todos los nodos del árbol tienes todos sus hijos, excepto los nodos hojas. Por ejemplo, un árbol binario completo de 15 nodos tiene un nodo raíz, dos nodos en el segundo nivel, cuatro en el tercer nivel, y ocho nodos en el cuarto nivel, y $\lfloor \lg 15 \rfloor + 1 = \lfloor 3.9 \rfloor + 1 = 4$. Observa que si se agrega un nuevo nodo, este iniciará un nuevo nivel, entonces $\lfloor \lg 16 \rfloor + 1 = \lfloor 4 \rfloor + 1 = 5$. El árbol binario más grande (de más niveles) que tiene n nodos tendrá n niveles si cada nodo tiene exactamente 1 hijo (que de hecho es una lista).

Si enumeramos los niveles del árbol, considerando que la raíz debe estar en el nivel 1, habrá 2^{k-1} nodos en el nivel k. Un árbol binario completo con j niveles (enumerados desde 1 hasta j) tiene todos los nodos hoja en el nivel j, y todos los demás nodos tienen exactamente dos hijos. Un árbol binario completo de j niveles tiene $2^j - 1$ nodos. Esta información será muy útil en diferentes análisis que haremos.

1.1.4. Probabilidades

Como analizaremos algoritmos en dependencia de sus entradas, en ocasiones necesitaremos considerar la **probabilidad** de que ocurra cierto conjunto de entradas. Esto significa que necesitaremos trabajar con la probabilidad de que una entrada se encontraá con alguna condición. La probabilidad de que algo ocurra es un número entre 0 y 1, donde 0 es la completa certeza de que no ocurriá, mientras que 1 es la completa certeza de que el evento ocurrirá. Si sabemos que hay exactamente 10 entradas, podemos decir que la probabilidad de cada una de esas entradas, que es un número entre 0 y 1 pero la suma de todas las probabilidades es 1, porque una de esas entradas debe ocurrir. Si cada entrada tiene la misma oportunidad de ocurrir, cada una de ellas tendrá 1/10 (0.10).

Para la mayoría de nuestros análisis, primero determinaremos cuántas posibles situaciones hay, y luego supondremos que todas ellas pueden ocurrir con la misma oportunidad. Si determinamos que hay n posibles situaciones, resulta que cada una de esas situaciones tiene una probabilidad de ocurrencia de $\frac{1}{n}$.

1.1.5. Sumas

Como lo experimentamos antes al hacer un conteo de las operaciones utilizadas, utlizaremos **sumas**. Digamos que tenemos un algoritmo que incluye un ciclo. Cuando el límite del ciclo es 5, el procedimiento dentro del ciclo se debe realizar 5 veces, y cuando es 20, se debe hacer 20 veces. En general cuando el límite del ciclo es m, el conjunto de ordenes

dentro del ciclo se hace m veces. Cuando el límite del ciclo está sujeto a una variable que toma valores desde 1 hasta n, el número de veces que se hace el procedimiento dentro del ciclo es la suma desde 1 hasta n que se puede expresar mediante

$$\sum_{\substack{i=1 \\ i=i+1}}^{n} i$$

En la expresión anterior, $i = 1$ indica que i es la variable que cambia de valor y que 1 es el primer valor que adquiere; la parte $i = i + 1$ indica que el valor de i aumenta de 1 en 1 y n es el último. Cuando el incremento de la variable es 1 usualmente no se escribe, solamente cuando se quiere hacer énfasis en el incremento que debe aplicarse.

Una vez que hemos expresado alguna solución en una notación de suma, lo que sigue es tratar de simplificar la expresión, de modo que podemos comparar la expresión con otras ya conocidas. Por ejemplo, debemos decidir si $\sum_{i=11}^{n} i^2$ es mayor que $\sum_{i=0}^{n}(i^{20} - 20i)$. Para poder ayudar a determinar eso hay una lista de expresiones que pueden utilizarse:

1. $\sum_{i=1}^{n} c * i = c \sum_{i=1}^{n} i$, con c una constante independiente de i.
2. $\sum_{i=c}^{n} i = \sum_{i=0}^{n-c}(i + c)$
3. $\sum_{i=c}^{n} i = \sum_{i=0}^{n} i - \sum_{i=0}^{c-1} i$
4. $\sum_{i=1}^{n}(a + b) = \sum_{i=1}^{n} a + \sum_{i=1}^{n} b$
5. $\sum_{i=0}^{n}(n - i) = \sum_{i=0}^{n} i$
6. $\sum_{i=1}^{n} 1 = n$
7. $\sum_{i=1}^{n} c = c * n$
8. $\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6} = \frac{2n^3 + 3n^2 + n}{6}$
9. $\sum_{i=0}^{n} 2^i = 2^{n+1} - 1$
10. $\sum_{i=0}^{n} a^i = \frac{a^{n+1} - 1}{a - 1}$, para algún número a.
11. $\sum_{i=1}^{n} i2^i = (n - 1)2^{n+1} + 2$
12. $\sum_{i=1}^{n} \frac{1}{i} = \ln n$

13. $\sum_{i=1}^{n} \log i \approx n \lg n - 1.5$

Cuando tratamos de simplificar una ecuación de suma, podemos aplicar alguna o algunas de las primeras expresiones anteriores para dividir la ecuación en expresiones más simples, y entonces aplicar otras para obtener una expresión equivalente sin sumas.

1.2. Qué contar y considerar

Para saber qué contar y qué considerar se requiere escoger las operaciones importantes y decidir cuáles de ellas son relevantes para el algoritmo, cuáles son de caracter general y cuáles son de conteo.

Hay dos tipos de operaciones que típicamente se eligen como operaciones importantes:

1. las operaciones lógicas [Comparaciones o funciones booleanas],
2. las operaciones aritméticas.

Los operadores lógicos se consideran equivalentes y se deben considerar en algoritmos de búsqueda y de ordenamiento. En estos tipos de algoritmos, la tarea importante que se hace es la comparción de dos valores para determinar cuándo se encuentra el valor deseado [si es de búsqueda] o cuándo un valor está fuera de orden [en el caso de un ordenamiento]. Los operadores lógicos son $\wedge$, $\vee$, $\neg$, $=$, $\neq$, $<$, $>$, $\leq$ y $\geq$, etc.

Los operadores aritméticos los vamos a agrupar en dos clases: los aditivos y los multiplicativos. Los operadores aditivos son $+$, $-$, `+1` [incremento] y `-1` [decremento]. Los operadores multiplicativos incluyen $*$, $/$, mód .

Estos dos grupos se cuentan por separado porque las multiplicaciones son operaciones más complejas que las sumas y tomas más tiempo en realizarse. De hecho, algunos algoritmos son considerados «mejores», si tienen un menor número de multiplicaciones, aún si eso significa aumentar el número de sumas. Hay otras operaciones que usualmente aparecen en los algoritmos, los logaritmos y las funciones geométricas, que consumen aún más tiempo.

Un caso particular es la multiplicación de enteros, o la división por potencias de 2. Estas operaciones se pueden reducir a una operación de corrimiento de bits, que son consideradas como operaciones de suma porque se pueden hacer muy rápidamente.

Para ilustrar lo anterior, consideremos el siguiente problema computacional:

```
PROBLEMA:Determinar el numero de caracteres diferentes en una lista
ENTRADA: lst # La lista de caracteres de entrada
SALIDA: ncar_dif # El número de caracteres diferentes
```

Supongamos un algoritmo que es descrito como sigue:

```
for ch en el rango de 0 a 255
  cont[ch] ←- 0
end
while not(empty(lst))
  ch ←- get(next_char(lst)) # obtener el siguiente caracter
  inc(cont[ch]) # incrementar el contador de este caracter en 1
end
return 256 - num_zeros(cont) # devuelve el numero de caracteres diferentes
```

Al observar este algoritmo notamos que se divide en dos partes: la primera se dedica a inicializar variables; mientras que la segunda parte realiza la cuenta de los caracteres.

En el primer ciclo se hacen actividades de inicialización y se realizan operaciones aritméricas, de asignación y de comparación.

Operación	Cantidad
Aritmética [+1]	256
Asignación [←-]	256
Lógicas	257

Se realizan 256 asignaciones para la variable `ch`; y se realizan 257 comparaciones (256 en el rango y una comparación extra para determinar que se ha terminado el ciclo); y re realizan otras 256 asignaciones de inicialización para cada contador.

La segunda parte depende de la longitud de la lista `lst`. Si $|\texttt{lst}| = n$, el ciclo `while` se realiza $n + 1$ veces. Se realizan n asignaciones para `ch`; y otras 256 asignaciones para incrementar el valor del contador respectivo.

Operación	Cantidad
Aritmética [`+1`]	n
Asignación [$\longleftarrow$]	$2n$; Cada contador y `ch`
Lógicas	$2n + 2$; hay dos operaciones lógicas

En total, considerando ambos ciclos tenemos:

1. Aritméticas : $256 + n$
2. Asignaciones : $256 + 2n$
3. Lógicas : $257 + 2n + 2$

Y todas las operaciones del algoritmo son $5n + 771$

La pregunta que surge ahora es ¿Qué operaciones son las que en verdad determinan la complejidad del algoritmo? Veamos un par de casos de estudio.

Supongamos en primer lugar que $|\texttt{lst}| = 50$, entonces en total se deberán hacer 1021, de las cuales 769 (75.3 %) se dedican a la inicialización de variables y el resto (24.7 %) son para la detección de caracteres repetidos.

Supongamos ahora que $|\texttt{lst}| = 50000$. Ahora el algoritmo realiza $250,771$ operaciones, de las cuales 769 (3 %) son las mismas actividades de inicialización, y el resto (97 %) se utilizan para la detección de caracteres repetidos en la lista.

Con este ejemplo podemos constatar que en los algoritmos hay actividades de importancia relativa [en este ejemplo son las actividades de inicialización] que cuando el tamaño de entrada es suficientemente grande, la importancia de esas operaciones es prácticamente nula.

Otra consideración es que analizar un algoritmo es igualmente válido si éste se encuentra escrito en un lenguaje lenguaje de programación como `C`, `C++`, `Java`, `Python` o cualquier otro lenguaje de alto nivel o incluso un pseudocódigo. Encontramos que es suficiente mientras se cuente con operaciones de ciclo como `for`, `while`, `repeat`; o bien con operaciones de selección como `case`, `elif` o `switch`.

2. Notación asintótica

Siempre hay que tener presente que el objetivo de analizar un algoritmo es tener una idea de los recursos computacionales que se ocuparán en la ejecución del algoritmo en forma de un proceso [programa en ejecución].

El término «recursos computacionales» se refiere principalmente a dos aspectos:

- La memoria que es empleada para la ejecución del programa, tanto memoria principal como memoria secundaria.
- El tiempo de ejecución.

Aunque la memoria de la computadora es limitada, la cantidad de memoria que se requiere para ejecutar un algoritmo generalmente es muy poca en comparación con las cantidad de memoria disponible en un equipo de cómputo moderno, esto además de que, a medida que pasa el tiempo el costo de tener más memoria se reduce, hace que el análisis de la memoria ocupada no sea tan importante como el análisis de requerimientos temporales, es por esto que el análisis te algoritmos se enfoque principalmente al tiempo de ejecución [análisis temporal] en lugar de la memoria requerida [análisis espacial].

Podemos analizar un algoritmo desde tres enfoques diferentes:

Análisis en el peor caso. Este enfoque proporciona las condiciones que garantizan que el algoritmo termine en el mayor tiempo posible. Para hacer este análisis es encesario porporcionar un ejemplo de entrada que permita que el algoritmo tarde

lo más posible. El análisis en el peor caso es el que se practica con mayor frecuencia porque proposciona un límite superior de la cantidad de tiempo requerido.

Análisis en el mejor caso. En el análisis en el mejor caso, se debe considerar las condiciones en las que un algoritmo termine en el menor tiempo posible. Se debe estudiar la naturaleza del problema y el algoritmo mismo para describir las condiciones en las que el algoritmo tarda lo menos posible. Este análisis proporciona un límite inferior al tiempo requerido.

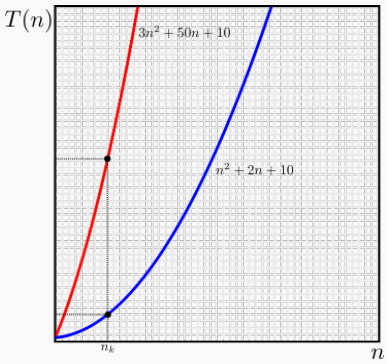
Análisis en el caso promedio. Algunas veces, tanto el caso promedio como el peor caso casi nunca ocurren en la práctica real, claro que esta aseveración debe ser demostrada al hacer este análisis en el caso promedio. Hacer un estudio probabilístico de los casos que se pueden presentar al ejecutar un algoritmo, permite establecer las condiciones promedio a las que se enfrenta el algoritmo. Se proporcina una predicción acerca del tiempo de ejecución. En la práctica, hacer este análisis es más difícil y tardado que hacer el análisis en el peor caso, y los resultados suelen no ser tan diferentes.

Si estableciéramos un orden en el los diferentes análisis de algoritmos, respecto del tiempo de ejecución, el ordenamiento de menor a mayor sería:

$$\begin{array}{ccccc} \text{Mejor caso} & \leq & \text{Caso promedio} & \leq & \text{Peor caso} \\ \text{límite inferior} & \leq & \text{tiempo promedio} & \leq & \text{límite superior} \end{array}$$

Ejemplo 1

Supongamos que un algoritmo tiene una razón de crecimiento en el peor caso de $\mathtt{f}(n) = 3n^2 + 50n + 10$ y en el mejor caso se comporta como una función $\mathtt{f}(n) = n^2 + 2n + 10$.

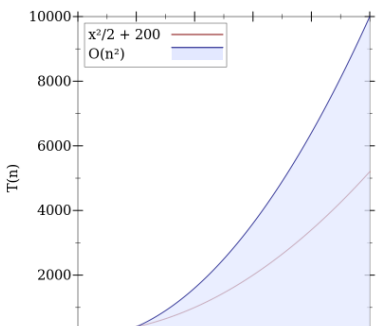


Una vez que se tienen las expresiones de las razones de crecimiento que delimitan el comportamiento del algoritmo, es necesaria una notación para representar los límites inferior y superior. Para estudiar esta notación, supondremos que $\mathtt{f}(n)$ y $\mathtt{g}(n)$ representan funciones que describen comportamientos algorítmicos.

2.1. Notación *O*

La notación ***O***-grande [*big-Oh*] se estableció por primera vez en 1894 por el matemático alemán Paul Bachmann, con el fin de establecer el orden de crecimiento de una función. $\boldsymbol{O}(\mathtt{g}(n))$ se utiliza para identificar una familia de funciones que tienen como base una función $\mathtt{g}(n)$ en particular.

La clase $\boldsymbol{O}(g(n))$ funciona como una cota superior, de modo que ninguna función en esta clase crece más rapidamente que $g(n)$. Formalmente se define como



Definición 1 (***O***-grande)**.**

$$\boldsymbol{O}(\mathtt{g}(n)) = \{\mathtt{f}(n) \mid \exists c_2 > 0, n_0 > 0 : [\forall n \geq n_0 : 0 \leq \mathtt{f}(n) \leq c_2 \mathtt{g}(n)]\}$$

Por ejemplo, si un algoritmo tiene una función de crecimiento $\frac{n^2}{2} + 200$, podemos decir que este algoritmo es de orden $\boldsymbol{O}\left(n^2\right)$, porque a partir de cierto punto [cercano a $n_0 = 20$], cualquier tamaño de entrada crece más lentamente que n^2.

Ejemplo 2

Determina la cota superior para $\mathtt{f}\,(n) = 3n + 8$.
Este ejercicio se resuelve al encontrar las constantes c_2 y n_0 de la clase $\boldsymbol{O}$.

1. Proponemos la función $4n$ como límite superior.
2. Hacemos $3n + 8 \leq 4n$.
3. Buscamos la constante n_0 a partir de la cual es cierta la desigualdad:
   $3n - 4n \leq -8$,
   $-n \leq -8$,
   $n \leq 8$.
4. Con las constantes $c = 4$ y $n_0 = 8$ se garantiza que la función $4n$ acota por arriba a la función $3n + 8$.

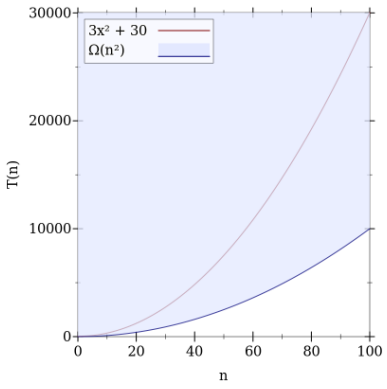
Por lo tanto $3n + 8$ es de orden $\boldsymbol{O}\,(n)$, con $c = 4$ y $n_0 = 8$.

n	$3n+8$	$4n$
1		
2		
3		
4		
5		
6		
7		
8		
9		

Es bueno mencionar que en general no hay una única solución para encontrar la clase $\boldsymbol{O}\,(\mathtt{f}\,(n))$ de una función $\mathtt{g}\,(n)$. Por ejemplo, supongamos que $n^2 + 3n + 1$ es la función que describe el crecimiento en el tiempo de ejecución de un algoritmo; algunos ordenes diferentes pueden ser n^2, n^3, n^4, etc. Se trata entonces de ofrecer la clase de funciones de menor grado que acoten por encima a la función dada.

2.2. Notación Ω

Utilizamos la notación $\boldsymbol{\Omega}\,(g(n))$, llamada *omega grande*, para representar la clase de funciones que crecen al menos tan rápido como lo hace la función $\mathtt{g}$. Esto significa que para todos los valores de n mayores que algún valor clave n_0, todas las funciones en $\boldsymbol{\Omega}\,(\mathtt{g}\,(n))$ tienen valores que son al menos tan grandes como $\mathtt{g}$. $\boldsymbol{\Omega}\,(\mathtt{g}\,(n))$ es como un límite inferior en una función, porque todas las funciones en esta clase van a crecer tan rápido como $\mathtt{g}$, o incluso más rápido. Formalmente:



Definición 2 ($\boldsymbol{\Omega}$-grande)**.**

$$\boldsymbol{\Omega}\,(\mathtt{g}\,(n)) = \{\mathtt{f}\,(n)\,|\exists c_1 > 0, n_0 > 0 : [\forall n \geq n_0 : 0 \leq c_!\mathtt{g}\,(n) \leq \mathtt{f}\,(n)]\}$$

Por ejemplo, si un algoritmo A es de clase $\boldsymbol{\Omega}\left(n^2\right)$, significa que la función que caracteriza el crecimiento del algoritmo A pertenece al conjunto $\boldsymbol{\Omega}\left(n^2\right)$. Con mayor precisión decimos que la razón de crecimiento del algoritmo A aumenta, al menos tan rápidamente como lo hace la función n^2.

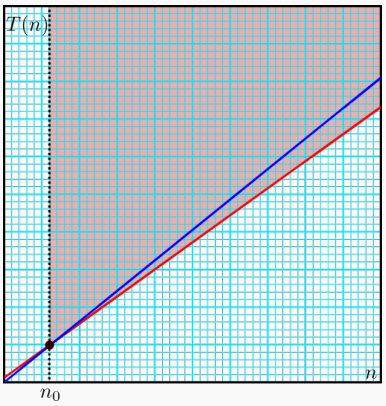
Suponiendo que se ha determinado que un algoritmo tiene un crecimiento que es expresado por la función $3n^2 + 30$. Eliminando los términos de menor crecimiento y las constantes, si decimos que este algoritmo es de orden $\boldsymbol{\Omega}\left(n^2\right)$ entonces significa que a partir de cierto valor de n [el tamaño de entrada] el algoritmo crece al menos tan rápido como n^2 o aún más rápido.

Ejemplo 3

Demuestra que $8n + 12$ pertenece a la clase $\mathbf{\Omega}(n)$.
Tenemos que demostrar que existen constantes $c > 0$ y $n_0 > 0$ tales que

$$c\mathrm{g}(n) \leq 8n + 12$$

$cn \leq 8n + 12$
$cn - 8n \leq 12$
$(c-8)n \leq 12$
$n \leq \frac{12}{c-8}$
Ahora, si hecemos $c = 9$ tenemos que $n_0 = 12$. Al dar las constantes queda demostrado que $8n + 12 \in \mathbf{\Omega}(n)$.



Ejemplo 4

Demuestra que $5n^2 + 2n - 6$ pertenece a la clase $\mathbf{\Omega}\left(n^2\right)$.
Tenemos que demostrar que existen constantes $c > 0$ y $n_0 > 0$ tales que

$$c\mathrm{g}(n) \leq 5n^2 + 2n - 6$$

Supongamos primero que tenemos que demostrar que

$$c\mathrm{g}(n) \leq 5n^2$$

$$cn^2 \leq 5n^2 \implies cn^2 - 5n^2 \leq 0 \implies (c-5)n^2 \leq 0$$

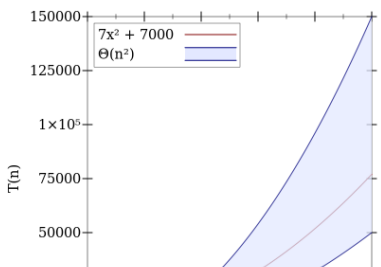
Ahora podemos suponer que $c = 5$, con lo que se cumple la desigualdad. Sin embargo, aún tenemos que garantizar que $2n - 6 \geq 0$ ya que de otro modo puede haber algún valor que impida el cumplimiento de la desigualdad.

$$2n - 6 \geq 0 \implies n \geq \frac{6}{2} \implies n \geq 3$$

Por lo que podemos proponer $c = 5$ y $n_0 = 3$, con lo que se demuestra que $5n^2 + 2n - 6$ pertenece a la clase $\mathbf{\Omega}\left(n^2\right)$.

2.3. Notación Θ

Utilizaremos la notación $\mathbf{\Theta}(g(n))$, llamada «Theta grande» para representar el conjunto de funciones que crecen más o menos al mismo ritmo que la función $g(n)$. El adjetivo «más o menos» significa que hay un límite superior y un límite inferior que acotan el crecimiento de la función. La definición formal es



Definición 3 ($\boldsymbol{\Theta}$-grande)**.**

$$\boldsymbol{\Theta}(\mathtt{g}(n)) = \{\mathtt{f}(n) \,|\exists c_1 > 0, c_2 > 0, n_0 > 0 : [\forall n \geq n_0 : 0 \leq c_1\mathtt{g}(n) \leq \mathtt{f}(n) \leq c_2\mathtt{g}(n)]\}$$

Un algoritmo que crece de acuerdo a una función $\mathtt{f}(n)$, pertenece a la clase $\boldsymbol{\Theta}(g(n))$ si hay constantes positivas c_1 y c_2 tales que puedan encerrar por encima y por debajo a la función $\mathtt{f}(n)$. Como $\boldsymbol{\Theta}(g(n))$ es un conjunto, podemos escribir $\mathtt{f}(n) \in \boldsymbol{\Theta}(g(n))$, pero usualmente escribimos $\mathtt{f}(n) = \boldsymbol{\Theta}(g(n))$ para expresar la misma noción.

En la figura se muestra una idea intuitiva de las funciones $\mathtt{f}(n)$ y $\mathtt{g}(n)$, donde $\mathtt{f}(n) = \boldsymbol{\Theta}(\mathtt{g}(n))$. Para todos los valores de n mayor o igual que n_0, el valor de $\mathtt{f}(n)$ está entre las funciones $c_1\mathtt{g}(n)$ y $c_2\mathtt{g}(n)$. Se puede probar que $\boldsymbol{\Theta}(\mathtt{g}(n)) = \boldsymbol{\Omega}(\mathtt{g}(n)) \cap \boldsymbol{O}(\mathtt{g}(n))$.

Ejemplo 5

Demuestra que $\frac{n^2}{2} - \frac{n}{2}$ pertenece a la clase $\boldsymbol{\Theta}(n^2)$ Tenemos que demostrar que

$$c_1\mathtt{g}(n) \leq \frac{n^2}{2} - \frac{n}{2} \leq c_2\mathtt{g}(n)$$

Para ciertas constantes positivas c_1, c_2 y a partir de cierto valor positivo n_0. Al hacer $c_1 = \frac{1}{2}$, $c_2 = 1$ y $n_0 = 1$ se verifica la proposición:

$$c_1 n^2 \leq \frac{n^2}{2} - \frac{n}{2} \leq c_2 n^2$$

Cuando analicemos algoritmos, estaremos interesados en encontrar algoritmos que sean más eficientes que un algoritmo determinado; de modo que encontrar un algoritmo que sea de complejidad $\boldsymbol{\Theta}(\mathtt{g}(n))$ [en otras palabras, de la misma complejidad] no es muy interesante. Muy pocas veces nos enfocaremos a esta clase de algoritmos.

Como hemos visto, si un algoritmo tiene un tiempo de ejecución que es descrito por una función $\mathtt{f}(n)$, en el análisis de algoritmos tratamos de encontrar otra función [posiblemente la misma] $\mathtt{g}(n)$ que se aproxime a $\mathtt{f}(n)$. Esto significa que $\mathtt{g}(n)$ también en una curva que se aproxima a $\mathtt{f}(n)$ para grandes valores de n. En otras palabras, $\mathtt{g}(n)$ es una curva asintótica para $\mathtt{g}(n)$. Por esta razón se llama *análisis asintótico*.

2.4. Consejos para el análisis asintótico

Aquí hay algunos consejos que pueden ser útiles a la hora de determinar el tiempo de ejecución de un algoritmo.

1. **Ciclos.** El tiempo de ejecucióon es, cuando mucho, el tiempo de ejecuci
   on de las sentencias dentro del ciclo [incluyendo las verificaciones] multiplicado por el número de iteraciones.

```
for i in range(0,n): # Este ciclo se ejecuta n veces
  print("Iteracion: ", i) # tiempo constante c
```

   Tiene un tiempo total de

$$\mathtt{T}(n) = c \times n = cn = \boldsymbol{O}(n).$$

2. **Ciclos anidados**: Se analizan desde el ciclo más interno hasta el más externo. El tiempo de ejecución total es el producto de los tamaños de todos los ciclos.

```
for i in range(0,n): # Este ciclo se ejecuta n veces
  print("Iter. externa:.... ", i)  # constante k1
  for j in range(0,n): # Este ciclo se ejecuta n veces
    print(" iter interna: ", j) # tiempo constante k2
  print()
```

   Tiene un tiempo de ejecución

$$\mathtt{T}(n) = (k_2 n)(k_1 n) = (k_1 \cdot k_2)n^2 = kn^2 = \boldsymbol{O}(n^2)$$

3. **Sentencias consecutivas**: Se suma el tiempo de ejecución de cada sentencia.

```
n = int(input()) # constante k1
m = int(input()) # constante k2

for i in range(0,n): #  n veces
  print("iteracion: ", i)  # constante k3

for i in range(0,n): # Este ciclo se ejecuta n veces
  print("Iter. externa:.... ", i)  # constante k4
  for j in range(0,n): # Este ciclo se ejecuta n veces
    print(" iter interna: ", j) # tiempo constante k5
  print() # constante k6
```

Tiene un tiempo de ejecución:

$$\mathtt{T}(n) = k_1 + k_2 + k_3 n + (k_4 * k_5) n^2 = c_1 + k_3 n + c_2 n^2 = \boldsymbol{O}\left(n^2\right)$$

4. **Sentencias condicionales**: El tiempo total se toma de acuerdo al análisis de cada caso, pero en general es el tiempo de la verificación mas el tiempo de las sentencias en el caso que sea verificado.

```
if n == 1:   # constante c1
  print("Valor equivocado: ", n) # constante c2
else:
  for i in range(0,n):  # se realiza n veces
    print("Iteracion : ", i)  # constante c3
```

El tiempo de ejecución para el caso alterno es

$$\mathtt{T}(n) = c_1 + c_3 n = \boldsymbol{O}(n)$$

5. **Complejidad logarítmica**: Un algoritmo tiene complejidad $\boldsymbol{O}(\mathtt{log}(n))$ si toma un tiempo constante para dividir el tamaño del problema por una fracción [generalmente a la mitad]. Como ejemplo considrea esta función:

```
def logaritmos(n):
  i = 1
  while (i <= n):
    i = i * 2
    print(i)

logaritmos(100)
```

Aquí hay que observar cuidadosamente que la variable `i` se duplica en cada iteración dentro del ciclo `while`, empieza con `i = 1`, luego `i = 2`, luego `i = 4`, y así en adelante. Supongamos que el ciclo se ejecuta k veces, por lo que el valor de `i` en el k-ésimo paso será 2^k y si suponemos que ahí termina el ciclo, entonces $2^k = n$. Aplicando el logaritmo en cada lado:

$$\begin{aligned} &\mathtt{log}\left(2^k\right) = \mathtt{log}(n) \\ &k\mathtt{log}(2) = \mathtt{log}(n) \\ &k = \mathtt{log}(n)\,;\ \textit{Suponiendo que la base es 2} \end{aligned}$$

Entonces el tiempo total es $\mathtt{T}(n) = \mathtt{log}(n)$.

Ahora, si en lugar de la variable `i` se duplique, hacemos que tome la mitad de su valor en cada iteración, esto es `i = i // 2`, también [por la misma razón]

$$\mathtt{T}(n) = \mathtt{log}(n)\,.$$

6. **Funciones recursivas**. Para encontrar la complejidad algorítmica de una función recursiva se sigue un procedimiento minucioso, en el que se considera el caso base y la parte recursiva.

   Los algoritmos recursivos pueden ser de diferentes naturalezas, los hay de recurrencias simples, que son una versión diferente de un ciclo simple, pero también los hay de recurrencias más complicadas.

   En el siguiente capítulo estudiaremos con mucho mayor detalle cómo resolver ecuaciones de recurrencia.

Ejercicios

Criterio de evaluación

Para que seas conciente de cómo se evalúan y califican estos ejercicios, considera el siguiente criterio que otorga cierto número de puntos por cada ejercicio:

- 3: Un ejercicio entregado a tiempo y sin alguna falla. Un ejercicio de programación corre a la perfección; sigue fielmente las instrucciones y su notación, y tiene su contrato correctamente escrito.
- 2: Un ejercicio mayormente correcto entregado a tiempo. Un ejercicio de programación o bien no tiene su contrato; o bien ha cambiado identificadores o es parcialmente correcto.
- 1: Un ejercicio entregado a destiempo o mayormente equivocado. Un ejercicio de programación no tiene su contrato y ha cambiado identificadores y es parcialmente correcto.
- 0: Un ejercicio completamente equivocado o no se entrega.

La calificación de la lista de ejercicios se obtiene al ponderar el número de puntos alzanzado en una escala de 0 a 10.

1. Los siguientes ejercicios se tratan de logaritmos:
   a) Escribe la forma logarítmica de $2^6 = 64$, menciona cuál es la base, la potencia y el exponente.
   b) Escribe la forma exponencial de $\log_6(36)$, menciona la base, el argumento [antilogaritmo] y el logaritmo.
   c) Aplica las leyes de los logaritmos para resolver $\log_3 \frac{x}{x+2}$
2. Calcula el valor de las siguientes sumas:
   a) $\sum_{j=1}^{n} 2k$
   b) $\sum_{k=1}^{n} (3k+1)$
   c) $\sum_{j=1}^{n} j(j+1)$
3. Mediante un programa en `Python`, grafica las funciones $4n$ y $3n + 8$ en una sola gráfica. Permite que n tome valores desde 0 a 100. Coloca etiquetas para cada función, y coloca un punto en las coordenadas $\langle 8, 32 \rangle$, que representa la constante n_0. *Consejo:* Puedes utilizar `matplotlib` para hacer esta gráfica, puedes guiarte de https://matplotlib.org/tutorials/introductory/pyplot.html
4. Demuestra que $\mathtt{f}(n) = 6n - 9$ pertenece a la clase $\boldsymbol{O}(n)$.
5. Demuestra que $\mathtt{f}(n) = 2n^3$ pertenece a la clase $\boldsymbol{\Theta}(n^3)$.
6. Demuestra que $\mathtt{f}(n) = \frac{1}{2}n^2 - 6n$ pertenece a la clase $\boldsymbol{\Theta}(n^2)$.
7. Considera la siguiente función en `Python`:

```
def busqueda_lineal(k, lst):
    j = len(lst) - 1
    encontrado = False
    while not(encontrado) and j > 0:
        if lst[j] == k:
```

```
            encontrado = True
        j = j - 1
    return encontrado
```

- ¿En qué condiciones se realizan más operaciones?
- ¿En qué condiciones se realizan menos operaciones?
- Si $|\texttt{lst}| = n$ y considerando el peor caso:
  - *a*) ¿Cuántas asignaciones se hacen?
  - *b*) ¿Cuántas operaciones aritméticas se hacen?
  - *c*) ¿Cuántas operaciones lógicas se hacen?

8. Escribe un programa en `Python` que calcule el número de años que pertenecen a cierta década dada, dentro de un rango de años dado. Es decir si se buscan los años que pertenecen a la década de los 40, dentro del rango que va desde 1847 hasta 1942, entonces la respuesta es 6, porque hay seis años que pertenecen a esa década los cuales son 1847, 1848, 1849, 1940, 1941 y 1942.

```
PROBLEMA: Determinar cuántos años de la misma década hay en un rango de años.
ENTRADA: Tres números.
a : la década solicitada
b : límite inferior del rango de años
c : límite superior del rango de años
SALIDA: El número de años que hay de la misma década en el rango dado.
```

Entrada	Salida
30 1932 2038	10

*Problema tomado de OmegaUp y ligeramente modificado https://omegaup.com/arena/problem/1979/