APUNTES PARA ALGORITMOS Y ESTRUCTURAS DE DATOS II

DANIEL FRIDLENDER

Por favor, reportar errores, omisiones y sugerencias a dfridlender@gmail.com

Observaciones Preliminares

En las materias **Introducción a los Algoritmos** y **Algoritmos y Estructuras de Datos I** el énfasis estaba puesto en **qué** hace un programa. Se especificaba detalladamente las pre- y pos- condiciones que el programa debía satisfacer y se derivaba un programa que satisfaciera dicha especificación.

En **Algoritmos y Estructuras de Datos II**, sin descuidar las especificaciones de pre- y pos- condiciones ni la formulación de los invariantes de los ciclos, el énfasis estará puesto en **cómo** resuelve el programa el problema especificado. Desarrollaremos instrumentos que nos permitan comparar diferentes programas que resuelven un mismo problema.

Uno de los aspectos que es importante considerar al comparar algoritmos es el referido a los recursos que el programa necesita para ejecutarse: tiempo de procesamiento, espacio de memoria, tiempo de utilización de un dispositivo. El estudio de la necesidad de recursos de un programa o algoritmo se llama **análisis del algoritmo** y lo que dicho análisis determina es la **eficiencia** del mismo.

Ejemplos motivadores. Considere las siguientes preguntas:

1. Un pintor demora 1 hora y media en pintar una pared de 3m de largo. ¿Cuánto demorará en pintar una de 5m de largo?
2. Un pintor demora 1 hora y media en pintar una pared cuadrada de 3m de lado. ¿Cuánto demorará en pintar una de 5m de lado?
3. Si lleva 5 horas inflar un globo aerostático esférico de 4m de diámetro, ¿cuánto llevará inflar uno de 8m de diámetro?
4. Un bibliotecario demora 1 día en ordenar alfabéticamente una biblioteca con 1000 expedientes. ¿Cuánto demorará en ordenar una con 2000 expedientes?

La respuesta a las primeras 3 preguntas puede darse en forma precisa porque conocemos la relación entre el dato (longitud del lado o del diámetro) y la magnitud de la tarea. Además, se asume que se conoce un método simple para pintar una pared o inflar un globo. La respuesta a la cuarta pregunta, en cambio, no parece fácil de responder. No conocemos cuánto más trabajo es ordenar 2000 expedientes que 1000. Existen numerosos métodos de ordenación que usamos en la práctica sin preguntarnos realmente cuál es el mejor. La respuesta a la pregunta **depende** del algoritmo de ordenación que esté utilizando el bibliotecario. Supongamos que está utilizando el algoritmo de **ordenación por selección** (selection sort):

proc ssort (a: **array**[1..n] **of T**) $\{pre : n \geq 0 \wedge a = A\}$
  var i, minp: **int**
  i:= 1 $\{inv : a[1, i)$ *está ordenado* $\wedge\ a$ *es permutación de* $A \wedge\}$
  $\{\wedge$ *los elementos de* $a[1, i)$ *son menores o iguales a los de* $a[i, n]\}$
  do i < n → minp:= select(a,i)
    swap(a,i,minp)
    i:= i+1
  od
end proc $\{pos : a$ *está ordenado y es permutación de* $A\}$

En la primera ejecución del ciclo, este algoritmo utiliza la función select para encontrar la posición minp donde se encuentra el mínimo de todo el arreglo y luego ubica el mínimo encontrado en la primera posición del arreglo utilizando el procedimiento swap. En la segunda ejecución del ciclo, vuelve a utilizar la función select para encontrar la posición del segundo mínimo y luego ubica el segundo mínimo en la segunda posición del arreglo. En general, una vez ubicado el i-1 ésimo mínimo del arreglo en la posición i-1, el algoritmo utiliza select para encontrar la posición del i-ésimo mínimo del arreglo y luego el procedimiento swap para colocarlo en la posición i. Toda modificación del arreglo se realiza a través de swap que, como vemos a continuación, produce una permutación del arreglo dado. Esto garantiza que ningún valor de a se pierde ni duplica.

proc swap (a: **array**[1..n] **of T**, i,j: **int**) $\{pre : a = A \wedge 1 \leq i, j \leq n\}$
  var tmp: **T**
  tmp:= a[i]
  a[i]:= a[j]
  a[j]:= tmp
end proc $\{pos : a[i] = A[j] \wedge a[j] = A[i] \wedge \forall k.k \notin \{i, j\} \Rightarrow a[k] = A[k]\}$

A continuación, detallamos el algoritmo utilizado para encontrar la posición del i-ésimo mínimo. Dado que el algoritmo ssort va ubicando los primeros i-1 mínimos en los primeros i-1 lugares del arreglo, para encontrar el i-ésimo mínimo de a basta con buscar el mínimo de a[i,n].

fun select (a: **array**[1..n] **of T**, i: **int**) **ret** minp: **int** $\{pre : 0 < i \leq n\}$
  var j: **int**
  minp:= i
  j:= i+1 $\{inv : a[minp]$ *es el mínimo de* $a[i, j)\}$
  do j ≤ n → **if** a[j] < a[minp] **then** minp:= j **fi**
    j:= j+1
  od
end fun $\{pos : a[minp]$ *es el mínimo de* $a[i, n]\}$

¿Por qué la función select no devuelve el mínimo sino su posición?
¿Por qué dice i < n en vez de i ≤ n en la condición del **do** del procedimiento ssort?

El comando for. En el algoritmo presentado los 2 ciclos **do** se utilizan para recorrer un arreglo desde una posición predeterminada hasta otra también predeterminada. Además, el índice utilizado para recorrer el arreglo (i en un caso y j en el otro) sólo son

modificados al final del cuerpo de cada ciclo, cuando se los incrementa. En estas situaciones, utilizaremos una notación más compacta. En primer lugar, omitiremos declarar el índice. Además, en vez de escribir

```
k:= a
do k ≤ b → c
            k:= k+1
od
```

escribiremos

for k:= a **to** b **do** c **od**

Para que esta notación tenga sentido es fundamental que k no sea modificado en el cuerpo c del ciclo. Observar que esta notación hace más evidente que c se ejecuta una vez para cada valor de k desde a hasta b.

Utilizando ciclos **for** el procedimiento ssort puede escribirse

```
proc ssort (a: array[1..n] of T)                    {pre : n ≥ 0 ∧ a = A}
    var minp: int
    for i:= 1 to n-1 do        {inv : a[1, i) está ordenado ∧ a es permutación de A ∧}
                          {∧ los elementos de a[1, i) son menores o iguales a los de a[i, n]}
        minp:= select(a,i)
        swap(a,i,minp)
    od
end proc                                  {pos : a está ordenado y es permutación de A}
```

Asimismo, la función select se puede escribir

```
fun select (a: array[1..n] of T, i: int) ret minp: int          {pre : 0 < i ≤ n}
    minp:= i
    for j:= i+1 to n do                          {inv : a[minp] es el mínimo de a[i, j)}
        if a[j] < a[minp] then minp:= j fi
    od
end fun                                          {pos : a[minp] es el mínimo de a[i, n]}
```

Número de operaciones de un comando. Frecuentemente vamos a querer contar o estimar el número de operaciones que se realizan al ejecutarse un comando determinado. Como no todas las operaciones son igualmente significativas para la performance de un programa dado frecuentemente se cuentan sólo operaciones de un cierto tipo. La cuenta que se realice dependerá de las operaciones que se pretenden contar y del comando que se está analizando. A continuación, una descripción intuitiva de cómo se cuentan las operaciones que se realizan durante la ejecución de un comando dado.

Si queremos contar el número de operaciones que se realizan al ejecutarse la secuencia de comandos c1;c2;...;cn, sumamos las operaciones que se realizan durante la ejecución de cada comando de la secuencia:

$$\text{ops(c1;c2;...;cn)} = \text{ops(c1)} + \text{ops(c2)} + \ldots + \text{ops(cn)}$$

En particular, como **skip** representa una secuencia vacía de comandos, ops(**skip**) = 0.

El ciclo **for** k:= a **to** b **do** c(k) **od** puede verse intuitivamente como una abreviatura de la secuencia de comandos c(a);c(a+1);...;c(b). Por ello, si queremos contar el número de

operaciones de un ciclo **for**, sumamos las operaciones que se realizan en cada ejecución del cuerpo del mismo. Podemos utilizar por ejemplo la fórmula:

$$\text{ops}(\textbf{for } \text{k:= a } \textbf{to } \text{b } \textbf{do } \text{c(k) } \textbf{od}) = \sum_{\text{k=a}}^{\text{b}} \text{ops(c(k))}$$

Esta fórmula no tiene en cuenta las operaciones necesarias para modificar k ni para compararlo con b. **Ejercicio:** Proponer una fórmula que sí las tenga en cuenta. La fórmula a aplicar en un caso concreto dependerá de qué operaciones uno desea contar.

Si queremos contar el número de operaciones que se realizan al ejecutarse el comando condicional **if** b **then** c **else** d **fi**, debemos considerar dos casos: b verdadero ó b falso:

$$\text{ops}(\textbf{if } \text{b } \textbf{then } \text{c } \textbf{else } \text{d } \textbf{fi}) = \begin{cases} \text{ops(b)+ops(c)} & \text{b verdadero} \\ \text{ops(b)+ops(d)} & \text{b falso} \end{cases}$$

Si queremos contar el número de operaciones que se realizan al ejecutarse la asignación x:= e, tenemos 2 fórmulas dependiendo de si queremos o no contar la asignación en sí como una operación. En el primer caso tenemos ops(x:=e) = ops(e)+1 mientras que en el segundo tenemos simplemente ops(x:=e) = ops(e).

En los casos del comando condicional y de la asignación, ops(b) y ops(e) representan los números de operaciones necesarios para evaluar las expresiones b y e.

Para contar el número de operaciones que se realizan al ejecutarse un ciclo **do** es necesario contar el número de veces que se ejecutará el cuerpo del ciclo. Una vez determinado este número, se suman las operaciones que se realizan en cada ejecución del cuerpo y en cada evaluación de la condición o guarda.

Número de comparaciones de la ordenación por selección. A modo de ejemplo, contemos el número de **comparaciones entre elementos del arreglo a** en el algoritmo de ordenación por selección. Observemos que sólo se realizan comparaciones entre elementos de a durante la ejecución de la función select. Para cada valor de i entre 1 y n esta función se ejecuta una vez. Cuando i=1, ésta realiza para cada j desde 2 hasta n la comparación a[j] < a[minp], totalizando n-1 comparaciones. Cuando i=2, las comparaciones que realiza la función select son n-2, y así sucesivamente. Esto da (n-1) + (n-2) + ... + 1 = $\frac{\text{n*(n-1)}}{2} = \frac{\text{n}^2}{2} - \frac{\text{n}}{2}$ comparaciones.

Utilizando las fórmulas propuestas anteriormente, enseguida llegamos al mismo resultado:

$$\begin{aligned} \text{ops(ssort(a))} &= \textstyle\sum_{\text{i=1}}^{\text{n-1}} \text{ ops(select(a,i))} \\ &= \textstyle\sum_{\text{i=1}}^{\text{n-1}} \sum_{\text{j=i+1}}^{\text{n}} \text{ops(a[j] < a[minp])} \\ &= \textstyle\sum_{\text{i=1}}^{\text{n-1}} \sum_{\text{j=i+1}}^{\text{n}} 1 \\ &= \textstyle\sum_{\text{i=1}}^{\text{n-1}} \text{(n-i)} \\ &= \textstyle\sum_{\text{i=1}}^{\text{n-1}} \text{i} \\ &= \frac{\text{n*(n-1)}}{2} \\ &= \frac{\text{n}^2}{2} - \frac{\text{n}}{2} \end{aligned}$$

Estas expresiones indican que el número de comparaciones de la ordenación por selección **es del orden de** n^2, terminología que haremos más precisa pronto. Intuitivamente, el número de comparaciones de la ordenación por selección es proporcional a n^2.

Número de intercambios (swaps) de la ordenación por selección. Observemos que sólo se realizan swaps durante la ejecución del procedimiento ssort y no durante la de la función select. Por cada valor de i se hace exactamente un intercambio (swap) entre a[i] y a[minp] al final del cuerpo del **for** de ssort. Como i toma n-1 valores, son n-1 intercambios.

Utilizando las fórmulas se obtiene lo mismo

$$\begin{aligned} \text{ops(ssort(a))} &= \sum_{\text{i=1}}^{\text{n-1}} \text{ops(swap(a,i,minp))} \\ &= \sum_{\text{i=1}}^{\text{n-1}} 1 \\ &= \text{n-1} \end{aligned}$$

Es decir que el número de intercambios (swaps) de la ordenación por selección **es del orden de** n. Intuitivamente, el número de intercambios de la ordenación por selección es proporcional a n.

Contestando la pregunta 4. Asumiendo que el bibliotecario utiliza el método de ordenación por selección, hace del orden de n^2 comparaciones. Le llevó un día hacer 1.000.000 de ellas, por lo que le llevará aproximadamente 4 días hacer 4.000.000 de ellas.

Existen sin embargo algoritmos de ordenación mejores que le permitirían ordenar 2000 expedientes en poco más del doble del tiempo que le lleva ordenar 1000.

Otro ejemplo: ordenación por inserción. No siempre es posible calcular el número exacto de operaciones, dado que puede depender de cada input. El siguiente algoritmo de ordenación por inserción es un ejemplo de ello:

proc isort (a: **array**[1..n] **of T**) $\{pre : n \geq 0 \wedge a = A\}$
  for i:= 1 **to** n-1 **do** $\{inv : a[1,i)$ *está ordenado* $\wedge\ a$ *es permutación de* $A\}$
    insert(a,i)
  od
end proc $\{pos : a$ *está ordenado y es permutación de* $A\}$

Para escribir el invariante del procedimiento insert denotamos por $a[1\hat{\ }j\hat{\ }i]$ la secuencia de celdas de a desde la posición 1 hasta la i salteando la j-ésima.

proc insert (a: **array**[1..n] **of T**, i: **int**) **ret** $\{pre : 0 < i \leq n \wedge a = A\}$
  j:= i $\{inv : a[1\hat{\ }j\hat{\ }i]$ *y* $a[j,i]$ *están ordenados* $\wedge\ a$ *es permutación de* $A\}$
  do $j > 1 \wedge a[j] < a[j-1] \rightarrow$ swap(a,j-1,j) **od**
end proc $\{pos : a[1,i]$ *está ordenado* $\wedge\ a$ *es permutación de* $A\}$

Número de comparaciones de la ordenación por inserción. Contamos comparaciones de la forma $a[j] < a[j-1]$, que sólo se realizan en el procedimiento insert. Intuitivamente, cuando i=1, no se realiza ninguna, cuando i=2 se realiza 1, cuando i=3 se realizan al menos 1 y a lo sumo 2, ..., cuando i=n se realizan al menos 1 y a lo sumo n-1. No podemos obtener el número exacto de comparaciones ya que éste depende del

input. Pero podemos afirmar que en el **mejor caso** serán 0 + 1 + 1 + ... + 1 = n-1 comparaciones (es decir, del orden de n comparaciones) y en el **peor caso**, 0 + 1 + 2 + ... + (n-1) = $\frac{n^2}{2} - \frac{n}{2}$ comparaciones (es decir, del orden de n^2 comparaciones).

Obtener el número de comparaciones en el peor caso es importante pues establece una **cota**, una **garantía** sobre el comportamiento del algoritmo.

Obtener el número de comparaciones en el mejor caso no tiene la misma importancia, sólo establece una **posibilidad** sobre el comportamiento del algoritmo.

Sí es importante establecer el número de comparaciones en el **caso promedio** o **caso medio** ya que éste determina el comportamiento del algoritmo en la **práctica**. Para calcularlo se necesitan conocimientos de probabilidades. El número de comparaciones de la ordenación por inserción en el caso promedio es del orden de n^2.

Análisis de Algoritmos

El ejemplo ilustra varios aspectos del análisis del comportamiento de un algoritmo:

casos: no siempre es posible contar exactamente. Se distingue entre mejor caso, peor caso y caso promedio. El estudio del peor caso permite establecer una cota superior, una garantía de comportamiento. El caso medio revela el comportamiento en la práctica, pero es más difícil de establecer.

operación elemental: se cuenta el número de operaciones elementales. Ésta debe:

- ser de duración constante, su duración no debe depender de n,
- ser representativa del comportamiento, toda otra operación debe repetirse a lo sumo en forma proporcional a la operación elemental elegida. En la ordenación por selección, el intercambio (swap) no era una operación representativa.

aproximación: se ignoran constantes multiplicativas y los términos que resultan despreciables cuando n crece. Esto puede justificarse de varias maneras:

- la duración de la operación elemental depende del hardware, mejor hardware modificaría esas constantes.
- uno cuenta operaciones, pero no hace precisa la unidad de tiempo, no se sabe si cuenta segundos, días, microsegundos, años, etc. No estando determinada la unidad, las constantes pierden sentido.
- uno puede querer comparar 2 algoritmos cuyos análisis fueron hechos eligiendo operaciones elementales distintas en uno y en otro, sin saber a priori si dichas operaciones elementales tienen igual o diferente duración.

Pero hay que tener presente que se está haciendo una aproximación. Las constantes y términos que acabamos de llamar "despreciables" pueden ser significativos para valores suficientemente bajos de n.

La Notación $\mathcal{O}$

Sean $c(n)$ el número de comparaciones de la ordenación por selección para arreglos de tamaño n, $s(n)$ el número de swaps del mismo algoritmo para arreglos del mismo tamaño. Por lo visto anteriormente, $c(n) = \frac{n^2}{2} - \frac{n}{2}$ y $s(n) = n - 1$. Observemos el crecimiento de estas funciones cuando n se duplica o triplica:

n	1.000	2.000	3.000
$c(n) = \frac{n^2}{2} - \frac{n}{2}$	499.500	1.999.000	4.498.500
n^2	1.000.000	4.000.000	9.000.000
$s(n) = n - 1$	999	1.999	2.999

¿Cuánto crece $c(n)$ cuando n se duplica o triplica? Puede observarse que los valores de $c(n)$ se cuadruplican o nonuplican. Lo mismo ocurre con n^2. Podemos decir que "a la larga, $c(n)$ crece al mismo ritmo que n^2" o que "a la larga, $c(n)$ es proporcional a n^2" o, como dijimos antes, que "$c(n)$ es del orden de n^2". De la misma manera, decimos que "a la larga, $s(n)$ crece al mismo ritmo que n" o que "a la larga, $s(n)$ es proporcional a n" o que "$s(n)$ es del orden de n".

Cuando lo que uno quiere expresar es el ritmo de crecimiento de una función $t(n)$ resulta conveniente utilizar funciones que tengan el mismo ritmo de crecimiento que $t(n)$ pero cuya expresión sea más simple. En el ejemplo anterior, si lo que se intenta comunicar es el ritmo de crecimiento, conviene decir que el número de comparaciones es proporcional a n^2 que decir que es exactamente igual a $\frac{n^2}{2} - \frac{n}{2}$. La primera expresión, más simple, me dice con la mayor sencillez posible que cuando n se duplica el número de comparaciones de la ordenación por selección se cuadruplica.

Pronto desarrollaremos una notación para expresar justamente que una cierta función crece a la larga al mismo ritmo que otra función. Antes de eso es conveniente introducir una notación para un concepto más sencillo de definir: "a la larga, $t(n)$ crece **a lo sumo** al ritmo de $f(n)$", que denotaremos $t(n) \in \mathcal{O}(f(n))$:

Definición: Dada $f : \mathbb{N} \to \mathbb{R}^{\geq 0}$, se define el conjunto

$$\mathcal{O}(f(n)) = \{t : \mathbb{N} \to \mathbb{R}^{\geq 0} | \exists c \in \mathbb{R}^+ . \forall^\infty n \in \mathbb{N} . t(n) \leq c f(n)\}$$

de todas las funciones que a la larga crecen a lo sumo al ritmo de $f(n)$.

La notación $\forall^\infty n \in \mathbb{N}.\mathcal{P}(n)$ expresa que $\mathcal{P}$ se cumple para casi todo $n \in \mathbb{N}$, es decir, se cumple salvo a lo sumo en una cantidad finita de números naturales. Otra forma de expresar lo mismo es escribiendo $\exists n_0 \in \mathbb{N}. \forall n \in \mathbb{N}. n \geq n_0 \Rightarrow \mathcal{P}(n)$. Preferimos la notación utilizada en la definición por ser más compacta y evitar mencionar n_0.

Otra observación es que sólo importa el comportamiento de las funciones para n suficientemente grande, por lo que uno podría debilitar la condición $t, f : \mathbb{N} \to \mathbb{R}^{\geq 0}$, requiriendo sólo que $\forall^\infty n \in \mathbb{N}.\ t(n), f(n) \in \mathbb{R}^{\geq 0}$, es decir, permitiendo que $t(n)$ y $f(n)$ devuelvan valores negativos para una cantidad finita de n's. Éste es el caso de la función $\log_a n$ más abajo.

Si $t(n) \in \mathcal{O}(f(n))$ decimos que $t(n)$ es (a lo sumo) **del orden de** $f(n)$.

En particular, para el caso de la ordenación por selección, el número de comparaciones $t(n) = \frac{n^2}{2} - \frac{n}{2}$ es del orden de n^2, dado que $t(n) \leq n^2$ para todo $n \in \mathbb{N}$.

Otro ejemplo interesante lo proporciona la función $t(n) = 5n^2 + 300n + 15$. A pesar de las constantes 5, 300 y 15, se puede comprobar fácilmente que $t(n)$ es del orden de n^2.

También podemos comprobar que la definición de $\mathcal{O}$ determina que las constantes multiplicativas sean ignoradas. Esta propiedad es esperada ya que si d_1 y d_2 son no nulas, $d_1 t(n)$ y $d_2 f(n)$ son proporcionales a $t(n)$ y $f(n)$ respectivamente:

Proposición: Si $t(n) \in \mathcal{O}(f(n))$, entonces $d_1 t(n) \in \mathcal{O}(d_2 f(n))$ para todo $d_1, d_2 \in \mathbb{R}^+$.

Demostración: Sea $c \in \mathbb{R}^+$ tal que $t(n) \leq cf(n)$ se cumple para casi todo $n \in \mathbb{N}$. Entonces, $d_1 t(n) \leq d_1 cf(n) = (\frac{d_1 c}{d_2}) d_2 f(n)$.

Proposición: La relación "es a lo sumo del orden de" entre funciones de $\mathbb{N}$ en $\mathbb{R}^{\geq 0}$ es reflexiva y transitiva.

Demostración: Reflexividad: como $\forall n \in \mathbb{N}, f(n) \leq f(n)$ trivialmente, tomando $c = 1$ tenemos $f(n) \in \mathcal{O}(f(n))$.

Transitividad: Sean $c_1, c_2 \in \mathbb{R}^+$ tales que $\forall^\infty n \in \mathbb{N}, f(n) \leq c_1 g(n)$ y $\forall^\infty n \in \mathbb{N}, g(n) \leq c_2 h(n)$. Los naturales que satisfacen ambas desigualdades siguen siendo todos salvo a lo sumo un número finito. Por lo tanto, $\forall^\infty n \in \mathbb{N}, f(n) \leq c_1 g(n) \leq (c_1 c_2) h(n)$.

Corolario: $g(n) \in \mathcal{O}(h(n))$ sii $\mathcal{O}(g(n)) \subseteq \mathcal{O}(h(n))$.

Demostración: El "sólo si" se cumple por transitividad, y el "si" por reflexividad.

Corolario: $f(n) \in \mathcal{O}(g(n)) \wedge g(n) \in \mathcal{O}(f(n))$ sii $\mathcal{O}(f(n)) = \mathcal{O}(g(n))$.

Observar que esto no significa que la relación "es a lo sumo del orden de" sea antisimétrica, de hecho no lo es. Antisimetría debería establecer que $f = g$.

Corolario: Para todo $a, b \in \mathbb{R}^+$ tales que $a, b \neq 1$, $\mathcal{O}(\log_a n) = \mathcal{O}(\log_b n)$.

Demostración: Para todo $n \in \mathbb{N}^+$, $\log_a n = \log_a b \ \log_b n$. Luego, $\log_a b$ es la constante que prueba que $\log_a n \in \mathcal{O}(\log_b n)$. Similarmente se prueba que $\log_b n \in \mathcal{O}(\log_a n)$.

Esto demuestra que en este contexto la base del logaritmo es irrelevante y puede ignorarse.

Regla del Límite

La siguiente regla es útil para demostrar cuándo una función es de un cierto orden, y cuándo no lo es.

Proposición (Regla de Límite): $\forall f, g \in \mathbb{N} \to \mathbb{R}^{\geq 0}$, si $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ existe los siguientes enunciados se cumplen:

1. $\lim_{n\to\infty} \dfrac{f(n)}{g(n)} \in \mathbb{R}^+ \Rightarrow \mathcal{O}(f(n)) = \mathcal{O}(g(n))$.
2. $\lim_{n\to\infty} \dfrac{f(n)}{g(n)} = 0 \Rightarrow \mathcal{O}(f(n)) \subset \mathcal{O}(g(n))$.
3. $\lim_{n\to\infty} \dfrac{f(n)}{g(n)} = +\infty \Rightarrow \mathcal{O}(g(n)) \subset \mathcal{O}(f(n))$.

Observar que en los dos últimos casos, la inclusión es estricta.

Demostración:

1. Alcanza con comprobar que $\mathcal{O}(f(n)) \subseteq \mathcal{O}(g(n))$, dado que si $\lim_{n\to\infty} \frac{f(n)}{g(n)} = l \in \mathbb{R}^+$ entonces $\lim_{n\to\infty} \frac{g(n)}{f(n)} = \frac{1}{l} \in \mathbb{R}^+$. Veamos entonces que $f(n) \in \mathcal{O}(g(n))$. Sea $\delta \in \mathcal{R}^+$, se tiene que $\forall^\infty n \in \mathbb{N}. \frac{f(n)}{g(n)} - l < \delta$, luego $f(n) < (\delta + l) g(n)$.
2. El razonamiento anterior funciona incluso si $l = 0$. Tenemos pues $\mathcal{O}(f(n)) \subseteq \mathcal{O}(g(n))$. Veamos que esta inclusión es estricta verificando que $g(n) \notin \mathcal{O}(f(n))$. Si $g(n) \in \mathcal{O}(f(n))$, entonces para una constante $c \in \mathbb{R}^+, \forall^\infty n \in \mathbb{N}, g(n) \leq cf(n)$. Entonces tendríamos $\frac{f(n)}{g(n)} \geq \frac{1}{c}$, con lo cual $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ no podría ser menor que $\frac{1}{c}$. Luego $\mathcal{O}(f(n)) \subset \mathcal{O}(g(n))$.
3. Como $\lim_{n\to\infty} \frac{f(n)}{g(n)} = +\infty$ implica que $\lim_{n\to\infty} \frac{g(n)}{f(n)} = 0$, vale por (2).

Corolario:

1. $x < y \Rightarrow \mathcal{O}(n^x) \subset \mathcal{O}(n^y)$,
2. $x \in \mathbb{R}^+ \Rightarrow \mathcal{O}(\log n) \subset \mathcal{O}(n^x)$,
3. $x \in \mathbb{R}^{\leq 0} \Rightarrow \mathcal{O}(n^x) \subset O(\log n)$,
4. $c > 1 \Rightarrow \mathcal{O}(n^x) \subset \mathcal{O}(c^n)$.
5. $0 \leq c < 1 \Rightarrow \mathcal{O}(c^n) \subset \mathcal{O}(n^x)$.
6. $c > d \geq 0 \Rightarrow \mathcal{O}(d^n) \subset \mathcal{O}(c^n)$.
7. $d \geq 0 \Rightarrow \mathcal{O}(d^n) \subset \mathcal{O}(n!)$.

Los incisos 3 y 5 son de poco interés ya que no es esperable tener programas cuyo número de comparaciones decrece a medida que n crece. Se enuncian para proporcionar una visión un poco más completa de la jerarquía de funciones determinada por la notación $\mathcal{O}$.

Demostración:

1. $\lim_{n\to\infty} \frac{n^x}{n^y} = \lim_{n\to\infty} \frac{1}{n^{y-x}} = 0$.
2. Como $\lim_{n\to\infty} \log n = +\infty = \lim_{n\to\infty} n^x$, por la Regla de L'Hôpital tenemos $\lim_{n\to\infty} \frac{\log n}{n^x} = \lim_{n\to\infty} \frac{\frac{1}{n}}{xn^{x-1}} = \lim_{n\to\infty} \frac{1}{xn^x} = 0$.
3. $\lim_{n\to\infty} \frac{\log n}{n^0} = \lim_{n\to\infty} \frac{\log n}{1} = \lim_{n\to\infty} \log n = +\infty$. Luego, $\mathcal{O}(n^0) \subset O(\log n)$. Para todo $x \in \mathbb{R}^-$, $\mathcal{O}(n^x) \subset \mathcal{O}(n^0) \subset O(\log n)$.
4. Demostraremos por inducción que para todo $k \in \mathbb{N}$, $\lim_{n\to\infty} \frac{n^k}{c^n} = 0$. Para $k = 0$ la prueba es trivial. Asumimos como hipótesis inductiva que $\lim_{n\to\infty} \frac{n^k}{c^n} = 0$. Por la Regla de L'Hôpital, $\lim_{n\to\infty} \frac{n^{k+1}}{c^n} = \lim_{n\to\infty} \frac{(k+1)n^k}{c^n \log_e c} = \frac{k+1}{\log_e c} \lim_{n\to\infty} \frac{n^k}{c^n} = 0$. Por lo tanto, para todo $k \in \mathbb{N}$, $\mathcal{O}(n^k) \subset \mathcal{O}(c^n)$. Sea $x \in \mathbb{R}$. Sea $k \in \mathbb{N}$ tal que $x \leq k$. Se obtiene $\mathcal{O}(n^x) \subseteq \mathcal{O}(n^k) \subset \mathcal{O}(c^n)$.
5. Similar al anterior, demostrando que para todo $k \in \mathbb{N}$, $\lim_{n\to\infty} \frac{n^k}{c^n} = +\infty$.
6. Sigue de $\lim_{n\to\infty} \frac{c^n}{d^n} = \lim_{n\to\infty} (\frac{c}{d})^n = +\infty$, que vale pues $\frac{c}{d} > 1$.
7. Sea $c > d$. Para todo $n \geq 2c^2$ se puede ver que $n! \geq n(n-1)\ldots(n - \lfloor \frac{n}{2} \rfloor) \geq \lceil \frac{n}{2} \rceil^{\lceil \frac{n}{2} \rceil} \geq c^{2\lceil \frac{n}{2} \rceil} \geq c^n$. Por lo tanto, $c^n \in \mathcal{O}(n!)$ que implica $\mathcal{O}(c^n) \subseteq \mathcal{O}(n!)$. Por el inciso anterior, tenemos $\mathcal{O}(d^n) \subset \mathcal{O}(n!)$.

Proposición: $g(n) \in \mathcal{O}(f(n)) \Rightarrow \mathcal{O}(f(n) + g(n)) = \mathcal{O}(f(n))$.

Demostración: Para $c \in \mathbb{R}^+$, $\forall^\infty n \in \mathbb{N}. g(n) \leq cf(n)$. Luego, $f(n)+g(n) \leq (1+c)f(n)$.

Corolario: $\mathcal{O}(a_k n^k + \ldots + a_1 n + a_0) = \mathcal{O}(n^k)$, si $a_k \neq 0$.

Demostración: Usando la proposición anterior y que $\lim_{n\to\infty} \frac{a_{k-1}n^{k-1}+\ldots+a_1 n+a_0}{a_k n^k} = 0$.

Proposición: $\mathcal{O}(f(n) + g(n)) = \mathcal{O}(\text{máx}(f(n), g(n)))$.

Demostración: Fácil pues $f(n)+g(n) \leq 2\,\text{máx}(f(n), g(n))$ y $\text{máx}(f(n), g(n)) \leq f(n)+g(n)$.

Proposición: Si $\forall^\infty n \in \mathbb{N}. f(n) > 0$, entonces $\mathcal{O}(g(n)) \subseteq \mathcal{O}(h(n))$ sii $\mathcal{O}(f(n)g(n)) \subseteq \mathcal{O}(f(n)h(n))$.

Demostración: Fácil pues $\forall^\infty n \in \mathbb{N}$, se verifica $g(n) \leq ch(n)$ sii $f(n)g(n) \leq cf(n)h(n)$.

Definición: Si $\mathcal{O}(t(n)) = \mathcal{O}(1)$, t se dice **constante**. Si $\mathcal{O}(t(n)) = \mathcal{O}(\log n)$, t se dice **logarítmico**. Si $\mathcal{O}(t(n)) = \mathcal{O}(n)$, t se dice **lineal**. Si $\mathcal{O}(t(n)) = \mathcal{O}(n^2)$, t se dice **cuadrática**. Si $\mathcal{O}(t(n)) = \mathcal{O}(n^3)$, t se dice **cúbica**. Si $\mathcal{O}(t(n)) = \mathcal{O}(n^k)$ para algún $k \in \mathbb{N}$, t se dice **polinomial**. Si $\mathcal{O}(t(n)) = \mathcal{O}(c^n)$, para algún $c > 1$, t se dice **exponencial**. Si

$t(n)$ expresa la eficiencia de un algoritmo, éste se dice, respectivamente, constante, logarítimico, lineal, cuadrático, cúbico, polinomial, exponencial. Si $t(n)$ expresa la eficiencia del algoritmo más eficiente posible para resolver un determinado problema, también se habla de problema constante, logarítmico, lineal, cuadrático, polinomial, exponencial.

Notación Complementaria. La notación $\mathcal{O}$ está destinada especialmente a establecer cotas superiores a la eficiencia de un programa, es decir, para afirmar que una cierta función a la larga crece a lo sumo al ritmo de otra. También suele ser útil establecer cotas inferiores, para expresar que una cierta función a la larga crece **por lo menos** al ritmo de otra. Y como anunciamos en la página 7, también pretendemos definir una notación que exprese que una cierta función a la larga crece al mismo ritmo que otra, es decir, que una cierta cota es ajustada.

Definición: Dada $f : \mathbb{N} \to \mathbb{R}^{\geq 0}$, se define el conjunto

$$\Omega(f(n)) = \{t : \mathbb{N} \to \mathbb{R}^{\geq 0} | \exists d \in \mathbb{R}^{+}. \forall^{\infty} n \in \mathbb{N}. t(n) \geq df(n)\}.$$

Así, $t(n) \in \Omega(f(n))$ dice que a la larga $t(n)$ crece como mínimo al ritmo de $f(n)$. Se puede comprobar inmediatamente que $g(n) \in \Omega(f(n))$ sii $f(n) \in \mathcal{O}(g(n))$.

Definición: Dada $f : \mathbb{N} \to \mathbb{R}^{\geq 0}$, se define el conjunto, $\Theta(f(n)) = \mathcal{O}(f(n)) \bigcap \Omega(f(n))$.

Así, $t(n) \in \Theta(f(n))$ dice que a la larga $t(n)$ crece al mismo ritmo que $f(n)$.

Proposición:

1. $f(n) \in \Omega(g(n))$ sii $g(n) \in \mathcal{O}(f(n))$ sii $\mathcal{O}(g(n)) \subseteq \mathcal{O}(f(n))$ sii $\Omega(f(n)) \subseteq \Omega(g(n))$
2. $f(n) \in \Theta(g(n))$ sii $g(n) \in \Theta(f(n))$ sii $\mathcal{O}(f(n)) = \mathcal{O}(g(n))$ sii $\Omega(f(n)) = \Omega(g(n))$