Capítulo 2

Notación asintótica

2.1. Competencias del tema

Las competencias a desarrollar en el tema son:

- **CTEC1** Capacidad para tener un conocimiento profundo de los principios fundamentales y modelos de la computación y saberlos aplicar para interpretar, seleccionar, valorar, modelar, y crear nuevos conceptos, teorías, usos y desarrollos tecnológicos relacionados con la informática.

- **CTEC3** Capacidad para evaluar la complejidad computacional de un problema, conocer estrategias algorítmicas que puedan conducir a su resolución y recomendar, desarrollar e implementar aquella que garantice el mejor rendimiento de acuerdo con los requisitos establecidos.

2.2. Introducción.

En el tema anterior se ha resaltado la importancia del estudio de la eficiencia y complejidad de los algoritmos ya que nos permitiría la elección del algoritmo más apropiado de entre varios candidatos. Para dicho estudio se ha de evaluar matemáticamente la cantidad de recursos que necesita el algoritmo en función del tamaño del problema. Como se vió en el tema anterior, es el cálculo de la complejidad temporal lo que nos interesa. Dicho cálculo se realizará en función del tamaño del problema. Si f es la función que indica cuanto tarda en ejecutarse un algoritmo, entonces $f(n)$ es el tiempo que requiere cuando el tamaño del problema es n. La eficiencia de los algoritmos se estudia por medio de $f(n)$, usando valores de n suficientemente grandes (valores en el límite del parámetro n), por eso se denomina **análisis asintótico**.

Para poder realizar dicho estudio se requiere de la utilización de las notaciones asintóticas, las cuales establecen unas cotas superiores o inferiores del tiempo de ejecución de un algoritmo:

- Notación orden de $f(n)$ ($O()$): da una cota superior.

- Notación $\Omega()$: da una cota inferior.

- Notación $\Theta()$: establece una cota inferior y superior.

2.3. Notación orden de $f(n)$ ($O()$).

Sea $f : N \to R^+ \cup \{0\}$. El conjunto de las funciones *del orden de* $f(n)$, denotado por $O(f(n))$, se define como sigue:

$$O(f(n)) = \{g : N \to R^+ \cup \{0\} \mid \exists c \in R^+, n_0 \in N, \forall n \geq n_0 \; g(n) \leq cf(n)\}$$

Se dice que g *es del orden de* $f(n)$ cuando $g \in O(f(n))$.

Resumiendo, se dice que g *es del orden de* $f(n)$ si $g(n)$ está acotada superiormente por un múltiplo real positivo de $f(n)$ para todo n suficientemente grande.

Generalizando, se tiene que si una función es negativa o indefinida para valores de $n < n_0$, pero cumple la definición anterior para valores de $n \geq n_0$, entonces también pertenece a $O(f(n))$.

Finalmente, se podría concluir que una función pertenece al conjunto $O(f(n))$ cuando está acotada superiormente por $f(n)$ para valores de n *suficientemente* grandes y haciendo abstracción de posibles constantes multiplicativas.

Ejemplo: La función $g(n) = 5n^3 + 3n^2 - 1 \in O(n^3)$.
Aplicando la definición

$$g(n) = 5n^3 + 3n^2 - 1$$
$$f(n) = n^3$$

y haciendo $n_0 = 3$ y $c = 6$ se tiene que

$$5n^3 + 3n^2 - 1 \leq 6n^3$$

La definición de $O(f(n))$ garantiza *el principio de invariancia* descrito en el tema anterior, es decir, que, si el tiempo de ejecución de una implementación concreta de un algoritmo está descrito por una función $g(n)$ *del orden de* $f(n)$, el tiempo $g'(n)$ empleado por cualquier otra implementación del mismo, que difiera de la anterior en el lenguaje, el compilador y/o la máquina empleada, también será *del orden de* $f(n)$, ya que solamente diferirá de $g(n)$ en una constante multiplicativa. En general, se dirá que el tiempo de ejecución de un algoritmo (y, por tanto, de todas sus implementaciones) *es del orden de* $f(n)$.

El conjunto $O(f(n))$ define *un orden de complejidad* y se escogerá como representante la función $f(n)$ más sencilla posible dentro del mismo. De esta forma se tiene que $O(n)$ representa el orden de complejidad *lineal*, $O(n^2)$ el de complejidad *cuadrática*, $O(1)$ el de las funciones *constantes*, etc.

2.4. Notación $\Omega()$.

Para demostrar la necesidad de esta notación se va a utilizar un ejemplo basado en los métodos de ordenación. Los métodos de ordenación no sofisticados o más simples (inserción, burbuja, selección, etc.) requieren un tiempo de ejecución de orden $O(n^2)$), mientras que los más sofisticados como el quicksort requieren un tiempo de orden $O(nlogn)$). Se puede demostrar que $nlogn \in O(n^2)$. Como consecuencia de ello se podría afirmar que el quicksort tiene un tiempo de $O(n^2)$, o incluso también sería correcto decir que tiene un tiempo de $O(n^3)$. Esto es consecuencia de que la notación $O(f(n))$ solo proporciona cotas superiores. Por lo tanto es necesaria una notación dual para las cotas inferiores y ésta es la notación $\Omega()$.

Sea $f : N \to R^+ \cup \{0\}$. El conjunto de las funciones $\Omega(f(n))$, leído *omega de* $f(n)$, se define como:

$$\Omega(f(n)) = \{g : N \to R^+ \cup \{0\} \mid \exists c \in R^+, n_0 \in N, \forall n \geq n_0 \; g(n) \geq cf(n)\}$$

La notación $\Omega(f(n))$ da una cota inferior respecto del tiempo de ejecución de un algoritmo.

Resumiendo, se dice que g *está en Omega de* $f(n)$ si $g(n)$ está acotada inferiormente por un múltiplo real positivo de $f(n)$ para todo n suficientemente grande.

Aunque la dualidad entre ambas notaciones parece evidente, hay que resaltar que esto no es cierto cuando se analiza la eficiencia en el caso peor, lo cual es bastantte frecuente. Si $t(n)$ representa el tiempo de ejecución de un algoritmo y $t(n) \in \Omega(f(n))$, entonces la medida $\Omega(f(n))$ no se refiere al mejor tiempo del mismo, sino a una cota inferior de $t(n)$. Si se realiza un análisis en el peor caso, $t(n)$ representa el tiempo del peor ejemplar de tamaño n, mientras que $f(n)$ es una cota inferior del mismo. Debido a este matiz en el caso del estudio del peor caso, hay una asimetría entre las medidas $O(f(n))$ y $\Omega(f(n))$:

- La medida $O(f(n))$ establece una cota superior al peor caso y, por tanto, a todos los ejemplares de tamaño n.
- La medida $\Omega(f(n))$ establece una cota inferior al caso peor, lo cual permite que puedan existir infinitos ejemplares de tamaño n con un tiempo mejor que $f(n)$.

Regla de la dualidad: $f(n) \in O(g(n)) \Leftrightarrow g(n) \in \Omega(f(n))$
Demostración:
$\boxed{\Rightarrow}$ Sea $f(n) \in O(g(n)$.

Por definición, $\exists c \in R^+, n_0 \in N, \forall n \geq n_0\ f(n) \leq cg(n)$

$\Rightarrow \exists c \in R^+, n_0 \in N, \forall n \geq n_0\ (1/c)f(n) \leq g(n)$

Si $c' = 1/c$, entonces $\exists c' \in R^+, n_0 \in N, \forall n \geq n_0\ g(n) \geq c'f(n)$

$\Rightarrow g(n) \in \Omega(f(n))$

$\boxed{\Leftarrow}$ Análoga a la anterior.

2.5. Notación $\Theta()$.

Mediante esta notación se pretende acotar el tiempo de ejecución de un algoritmo tanto por encima como por debajo mediante múltiplos reales positivos, posiblemente distintos, de una misma función.

Sea $f : N \to R^+ \cup \{0\}$. El conjunto de las funciones $\Theta(f(n))$, leído *del orden exacto de* $f(n)$, se define como:

$$\Theta(f(n)) = O(f(n)) \cap \Omega(f(n))$$

Esta definición es equivalente a

$$\Theta(f(n)) = \{g : N \to R^+ \cup \{0\} | \exists c, d \in R^+, n_0 \in N, \forall n \geq n_0 \\ df(n) \leq g(n) \leq cf(n)\}$$

La notación $\Theta()$ es más precisa que las notaciones $\mathcal{O}()$ y $\Omega()$, ya que $g(n)$ *está en el orden exacto de* $f(n)$ ($g(n) \in \Theta(f(n))$) si y sólo si $f(n)$ es a la vez una cota inferior y superior de $g(n)$.

Ejemplo: $\forall k \geq 0\ f(n) = \sum_{i=1}^{n} i^k \in \Theta(n^{k+1}) = O(n^{k+1}) \cap \Omega(n^{k+1})$

$\boxed{\Rightarrow} f(n) = \sum_{i=1}^{n} i^k \in O(n^{k+1})$

$\forall i \in \{1, ..., n\}\ i^k \leq n^k$

$\Rightarrow f(n) = \sum_{i=1}^{n} i^k \leq \sum_{i=1}^{n} n^k = n^{k+1}\ \forall n \geq 1$

$\Rightarrow \exists c = 1 \in R^+, n_0 = 1 \in N, \forall n \geq n_0\ f(n) \leq c\, n^{k+1}$

$\Rightarrow f(n) \in O(n^{k+1})$

$\boxed{\Leftarrow}\ f(n) = \sum_{i=1}^{n} i^k \in \Omega(n^{k+1})$

$\forall i \in \{\lceil n/2 \rceil, ..., n\}\ i^k \geq (n/2)^k$

$n - \lceil n/2 \rceil + 1 > n/2$

$f(n) = \sum_{i=1}^{n} i^k \geq \sum_{i=\lceil n/2 \rceil}^{n} i^k \geq \sum_{i=\lceil n/2 \rceil}^{n} (n/2)^k \geq \frac{n}{2} \times (\frac{n}{2})^k = (\frac{1}{2})^{k+1} \times n^{k+1}$

$\Rightarrow \exists c = (\frac{1}{2})^{k+1} \in R^+, n_0 = 1 \in N, \forall n \geq n_0\ f(n) \geq c\, n^{k+1}$

$\Rightarrow f(n) \in \Omega(n^{k+1})$

2.6. Notación asintótica con varios parámetros.

A veces puede ocurrir que el tiempo de ejecución de un algoritmo dependa de más de un parámetro del ejemplar o caso en cuestión. Por ejemplo el tiempo de ejecución de los algoritmos sobre grafos puede depender del número de nodos del grafo y del número de lados del mismo. En estos casos la notación asintótica se puede generalizar para admitir funciones de varias variables.

Sea $f : NxN \rightarrow R^+ \cup \{0\}$ una función $f(m, n)$ de parejas de números naturales en los reales no negativos.

Sea $g : NxN \rightarrow R^+ \cup \{0\}$ otra función del mismo tipo. Se dice que $g(m, n)$ es del orden de $f(m, n)$, lo cual se lee $g(m, n) \in O(f(m, n))$ si $g(m, n)$ está acotada superiormente por un múltiplo positivo de $f(m, n)$ siempre que tanto m como n sean suficientemenete grandes. De manera formal, $O(f(m, n))$ se define como:

$$O(f(m,n)) = \{g : NxN \rightarrow R^+ \cup \{0\} \mid \exists c \in R^+, m_0 \in N, \forall m \geq m_0, \forall n \geq m_0\ g(m,n) \leq cf(m,n)\}$$

No hay necesidad de definir dos umbrales para m y n, se toma el mayor de ambos. La generalización a más de un parámetro de las otras dos notaciones se hace de forma similar.

2.7. Propiedades de la notación $O()$.

Las propiedades que posee la notación $O()$ son:

1.- Propiedad reflexiva:

$\forall f : N \rightarrow R^+ \cup \{0\}$,

$f(n) \in O(f(n))$

La demostración es obvia, ya que sólo hay que tomar $n_0 = 1$ y $c = 1$ para comprobar que $\forall n \geq n_0\ f(n) \leq cf(n)$

2.- Propiedad transitiva:

$\forall f, g, h : N \to R^+ \cup \{0\}$,

si $f(n) \in O(g(n))$ y $g(n) \in O(h(n))$, entonces $f(n) \in O(h(n))$

Demostración:

(1) Si $f(n) \in O(g(n))$, entonces $\exists c_1 \in R^+, n_1 \in N, \forall n \geq n_1\ f(n) \leq c_1 g(n)$.

(2) Si $g(n) \in O(h(n))$, entonces $\exists c_2 \in R^+, n_2 \in N, \forall n \geq n_2\ g(n) \leq c_2 h(n)$.

Sea $n_0 = máximo(n_0, n_1)$ y $c = c_1 c_2$.

Se tiene por (1) y (2) que $\forall n \geq n_0\ f(n) \leq c_1 g(n) \leq c_1 c_2 h(n)$, lo cual implica que $f(n) \in O(h(n))$.

Las dos primeras propiedades definen una ordenación parcial en el conjunto de las funciones y, consiguientemente, en el conjunto de las eficiencias relativas de los diferentes algoritmos para resolver un problema dado. Sin embargo, el orden inducido no es total por cuanto existen funciones $f, g : N \to R^+ \cup \{0\}$ tales que $f(n) \notin O(g(n)$ y $g(n) \notin O(f(n)$. Considérese, por ejemplo, las siguientes funciones:

$$f(n) = \begin{cases} n^2 & si\ n\ es\ par \\ n^5 & si\ n\ es\ impar \end{cases} \quad \text{y} \quad g(n) = \begin{cases} n^5 & si\ n\ es\ par \\ n^2 & si\ n\ es\ impar \end{cases}$$

3.- Si $g(n) \in O(f(n))$ y $f(n) \in O(g(n))$, entonces $O(f(n)) = O(g(n))$.

Demostración:

$\boxed{\Rightarrow}$ $O(g(n)) \subset O(f(n))$

Si $h(n) \in O(g(n))$, como $g(n) \in O(f(n))$, entonces, gracias a la propiedad transitiva, se tiene que $h(n) \in O(f(n))$

$\boxed{\Leftarrow}$ $O(f(n)) \subset O(g(n))$

Trivial, ya que sólo hay que intercambiar $f(n)$ y $g(n)$.

4.- $\forall c \in R^+\ O(c\ f(n)) = O(f(n))$

La demostración de esta propiedad se deduce trivialmente de la definición de la notación de $\mathcal{O}()$.

Esta propiedad permite elegir como representante de un orden a aquella función que tiene como coeficiente a 1. Por ejemplo, $\forall k \in R^+\ O(kn^2) = O(n^2)$ y $O(k) = O(1)$

5.- Si $a, b > 1$, entonces $O(log_a n) = O(log_b n)$

Normalmente resulta innecesario especificar la base del logaritmo dentro de la notación asintótica, ya que[1] $log_a n = log_a b \times log_b n \quad \forall a, b$ y $n \in R^+$ tales que ni a ni b sean iguales a 1. La constante $log_a b$ es positiva si a y b son mayores que 1 y, por lo tanto, $log_a n$ y $log_b n$ difieren en una constante multiplicativa. A partir de esto, resulta elemental demostrar que $O(log_a n) = O(log_b n)$ lo cual se simplificará como $O(log n)$

6.- La regla del máximo (o de la suma): $O(f(n) + g(n)) = O(máximo(f(n), g(n)))$

Demostración:

$\boxed{\Rightarrow}$ $O(f(n) + g(n)) \subset O(máximo(f(n), g(n)))$

[1] $log_b n = x \Rightarrow n = b^x, log_a n = log_a b^x = x log_a b = log_b n \times log_a b$.

Se tiene que

$f(n) + g(n) = mínimo(f(n), g(n)) + máximo(f(n), g(n))$

y $0 \leq mínimo(f(n), g(n)) \leq máximo(f(n), g(n))$

deduciéndose que

(3) $máximo(f(n), g(n)) \leq f(n) + g(n) \leq 2\ máximo(f(n), g(n))$

(4) Si $h(n) \in O(f(n) + g(n))$,

entonces $\exists c \in R^+, n_0 \in N, \forall n \geq n_0\ h(n) \leq c(f(n) + g(n))$.

Por (3) y (4) se sigue que

$\forall n \geq n_0 \quad h(n) \leq c(f(n) + g(n)) \leq 2\ c\ máximo(f(n), g(n))$

y tomando $c' = 2c$ se tiene que $\forall n \geq n_0 \quad h(n) \leq c'\ máximo(f(n), g(n))$,

lo cual implica que $h(n) \in O(máximo(f(n), g(n)))$.

$\boxed{\Leftarrow} \quad O(máximo(f(n), g(n))) \subset O(f(n) + g(n))$

Si $h(n) \in O(máximo(f(n), g(n)))$,

entonces $\exists c \in R^+, n_0 \in N, \forall n \geq n_0\ h(n) \leq c\ máximo(f(n), g(n))$.

Por (3) se tiene que $h(n) \leq c\ máximo(f(n), g(n)) \leq c(f(n) + g(n))$

lo cual implica que $h(n) \in O(f(n) + g(n))$.

La regla del máximo se puede generalizar a un número finito de funciones no negativas.

La regla del máximo permite simplificar drásticamente el cálculo de la eficiencia de un algoritmo, ya que si una función $t(n)$ es muy complicada y $f(n)$ es el término más significativo de la misma, entonces $O(t(n)) = O(f(n))$.

Ejemplo: Sea la función $t(n) = 5n^3logn - 3n^2 + log^2n + 12$

$$\begin{aligned} O(t(n)) &= O(5n^3logn - 3n^2 + log^2n + 12) \\ &= O(máximo(4n^3logn, n^3logn - 3n^2, log^2n + 12)) \\ &= O(4n^3logn) \\ &= O(n^3logn) \end{aligned}$$

Hay que hacer dos consideraciones:

a) La función $n^3logn - 3n^2$ es positiva para valores de $n \geq 3$ y debido a que se realiza un análisis asintótico (para valores de n suficientemente grandes) no importa que sea negativa para valores pequeños de n.

b) No siempre se puede utilizar cualquier descomposición de la suma, ya que puede haber funciones que sean negativas siempre o para los valores de n suficientemente grandes. Por ejemplo, la siguiente descomposición no sería correcta, ya que la función $-3n^2$ siempre es negativa:

$$\begin{aligned} O(t(n)) &= O(5n^3logn - 3n^2 + log^2n + 12) \\ &= O(máximo(5n^3logn, -3n^2, log^2n + 12)) \\ &= O(5n^3logn) \\ &= O(n^3logn) \end{aligned}$$

7.- La regla del límite: $\forall f, g : N \to R^+ \cup \{0\}$

a) si $\lim_{n\to\infty} \frac{f(n)}{g(n)} \in R^+$ entonces $f(n) \in O(g(n))$ y $g(n) \in O(f(n))$

si $\lim_{n\to\infty} \frac{f(n)}{g(n)} = l \in R^+$, entonces, por definición de límite,

$\forall \epsilon > 0\ \exists n_0 \in N$ tal que $\forall n \geq n_0\ \mid \frac{f(n)}{g(n)} - l \mid < \epsilon \Leftrightarrow -\epsilon < \frac{f(n)}{g(n)} - l < \epsilon \Rightarrow \frac{f(n)}{g(n)} < l + \epsilon = c \Rightarrow f(n) < c\, g(n)$
lo cual implica que $f(n) \in O(g(n))$.
Para demostrar que $g(n) \in O(f(n))$ sólo hay que tener en cuenta que
$\lim_{n\to\infty} \frac{g(n)}{f(n)} = \frac{1}{l} \in R^+$, y, por tanto, se puede aplicar un razonamiento análogo al anterior.

b) si $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$ entonces $f(n) \in O(g(n))$ y $g(n) \notin O(f(n))$
si $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$, entonces, por definición de límite,
$\forall \epsilon > 0\ \exists n_0 \in N\ \forall n \geq n_0\ \mid \frac{f(n)}{g(n)} \mid < \epsilon \Leftrightarrow -\epsilon < \frac{f(n)}{g(n)} < \epsilon \Rightarrow \frac{f(n)}{g(n)} < \epsilon = c \Rightarrow f(n) < c\, g(n)$ lo cual implica que $f(n) \in O(g(n))$.
Para demostrar que $g(n) \notin O(f(n))$, sólo hay que razonar por reducción al absurdo: si se supusiera lo contrario, entonces
$\exists c \in R^+, n_0 \in N, \forall n \geq n_0\ g(n) \leq cf(n) \Rightarrow \frac{1}{c} \leq \frac{f(n)}{g(n)}\ \forall n \geq n_0 \Rightarrow \lim_{n\to\infty} \frac{f(n)}{g(n)} \geq \lim_{n\to\infty} \frac{1}{c} = \frac{1}{c} \neq 0\ \boxed{\rightarrow\leftarrow}$.
Por tanto $g(n) \notin O(f(n))$

c) si $\lim_{n\to\infty} \frac{f(n)}{g(n)} = +\infty$ entonces $f(n) \notin O(g(n))$ y $g(n) \in O(f(n))$
Si $\lim_{n\to\infty} \frac{f(n)}{g(n)} = +\infty$ entonces $\lim_{n\to\infty} \frac{g(n)}{f(n)} = 0$ y, por tanto, se puede aplicar el caso anterior intercambiando $f(n)$ y $g(n)$.

Ejemplo: Todo polinomio en n de grado m cuyo coeficiente líder sea positivo es *del orden de* n^m.

Sea $P_m(n) = a_m n^m + a_{m-1} n^{m-1} + ... + a_1 n + a_0$

$$\begin{aligned} \lim_{n\to\infty} P_m(n)/n^m &= \lim_{n\to\infty} (a_m n^m + a_{m-1} n^{m-1} + ... + a_1 n + a_0)/n^m \\ &= \lim_{n\to\infty} a_m n^m / n^m \\ &= a_m \in R^+ \end{aligned}$$

y por 7.a y 3 se tiene que $O(P_m(n)) = O(n^m)$

La inversa de la regla del límite no es necesariamente válida, ya que puede que $f(n) \in O(g(n))$ y $g(n) \in O(f(n))$ pero que $\lim_{n\to\infty} f(n)/(g(n)$ no exista.

Por ejemplo: $f(n) = \begin{cases} 2n^2 & si\ n\ es\ par \\ 3n^3 & si\ n\ es\ impar \end{cases}$ y $g(n) = \begin{cases} n^2 & si\ n\ es\ par \\ n^3 & si\ n\ es\ impar \end{cases}$

No existe $\lim_{n\to\infty} f(n)/(g(n)$, porque los elementos pares se aproximan a 2, mientras que los impares lo hacen a 3.

2.8. Jerarquía de los órdenes de complejidad.

Las funciones de complejidad algorítmica más usuales, ordenadas de mayor a menor eficiencia (o de menor a mayor complejidad), son:

- $O(1)$: *complejidad constante*. Es la complejidad más deseada, ya que indica que la ejecución del algoritmo no depende del tamaño de los datos del problema.
- $O(logn)$: *complejidad logarítmica*. Esta complejidad suele aparecer en determinados algoritmos de iteración o recursión no estructural[2] (por ejemplo, búsqueda binaria). En virtud de

[2] Se denomina recursividad bien fundada o no estructural a aquella que se aplica sobre datos que no son definidos de forma recursiva. Véase el libro Galve, J. y otros, .Algorítmica: diseño y análisis de algoritmos funcionales e imperativos, Editorial Ra-Ma, páginas 71 y 72.

la propiedad número 4, todos los logaritmos son del mismo orden, por eso no es necesario indicar la base.

- $O(n)$: *complejidad lineal*. Es, en general, una complejidad buena y bastante usual. Suele aparecer en la evaluación de un bucle simple cuando la complejidad de las operaciones interiores es constante o en algoritmos con recursión estructural.

- $O(nlogn)$: Aparece en algoritmos con recursión no estructural (método de ordenación *quicksort*) y se considera una complejidad buena.

- $O(n^2)$: *complejidad cuadrática*. Aparece en bucles doblemente anidados, como por ejemplo en la suma de matrices de orden n.

- $O(n^3)$: *complejidad cúbica*. Aparece en bucles triplemente anidados, como por ejemplo el producto de matrices de orden n. Para un valor grande de n, empieza a crecer en exceso.

- $O(n^k)$: *complejidad polinómica ($k > 3$)*. Si k crece, la complejidad es bastante mala. Por ejemplo la potencia n-ésima de una matriz de orden n es $O(n^4)$.

- $O(2^n)$: *complejidad exponencial*. Debe evitarse en la medida de lo posible. Puede aparecer en un subprograma recursivo que contenga dos o más llamadas internas. En problemas donde aparece esta complejidad suele hablarse de *explosión combinatoria*. Un caso típico es el de las tores de Hanoi o el de la sucesión de Fibinacci.

- $O(n!)$: *complejidad factorial*. El tiempo de ejecución de los algoritmos de este orden crece alarmantemente cuando n crece. El algoritmo que resuleve el problema del viajante de comercio de una formá óptima es de complejidad factorial.

Las dos últimas complejidades forman parte de *las complejidades superpolinómicas o intratables*, mientras que las restantes pertenecen al grupo de *complejidades polinómicas*.

Como ya se indicó al describir las propiedades reflexiva y transitiva, la notación asintótica $\mathcal{O}()$ induce un orden parcial sobre las funciones $f : N \rightarrow R^+ \cup 0$. En particular, los órdenes descritos poseen la siguiente relación de orden:

$$O(1) \subset O(logn) \subset O(n) \subset O(nlogn) \subset O(n^2) \subset O(n^3) \subset O(n^k) \subset O(2^n) \subset O(n!) \subset O(n^n)$$

2.9. Consideraciones importantes.

A la hora de evaluar la eficiencia de un algoritmo se han de tener en cuenta las siguientes consideraciones:

- El hecho de que un algoritmo tenga una eficiencia que sea *"buena"* en términos generales no implica que lo sea en términos particulares, ya que puede haber otro algoritmo que resuelva el mismo problema de una forma más eficiente.

  Por ejemplo: el que un problema *P* se resuelva mediante un algoritmo *A* con una eficiencia lineal no implica que dicho algoritmo sea necesariamente el mejor, ya que puede haber otro algoritmo *B* con una eficiencia logarítmica que también resuelva dicho problema.

- Si se comparan las eficiencias de dos algoritmos, se ha de tener en cuenta que el cálculo del orden de complejidad es una medida *"asintótica"*, es decir, se realiza un estudio teórico para valores de n *suficientemente grandes*. Puede ocurrir que un algoritmo tenga un eficiencia *asintótica* peor que la de otro, pero que tenga un mejor rendimiento para valores pequeños de n.

Además, si el tamaño de los datos del problema es reducido, entonces no se puede despreciar el valor de las constantes multiplicativas, ya que pueden afectar bastante a la eficiencia de los algoritmos.

Por ejemplo: supóngase que un problema se puede resolver mediante dos algoritmos diferentes cuyas funciones de complejidad son, respectivamente, $f(n) = n^3 - 3n^2 + 10$ y $g(n) = 100n^2 log n + 10$.

La primera función posee una complejidad *"asintótica"* peor que la segunda, pero posee un mejor rendimiento que la segunda para valores de n menores que 651. Por tanto, si el tamaño de los datos del problema es inferior a 651, entonces conviene utilizar el primer algoritmo aunque su comportamiento asintótico sea peor que el del segundo algoritmo.