Comparando algoritmos(I)

- Algoritmos para calcular b^n, con $n = 2^i$, i=0,1,...
- Método directo:

```
//n es potencia de 2
int potencia(int b, int n) {
int res = b;
for (int i = 2; i <= n; i++) {
        res = res * b;
}
return res;
}
```

- Enfoque alternativo:

```
//n es potencia de 2
int potencia2(int b, int n) {
int res = b;
if (n > 1) {
        int aux = potencia2(b, n/2);
        res = aux * aux;
}
return res;
}
```

- ¿Cuál es más eficiente?

Comparando Algoritmos (II)

- Para la instancia b = 2 y n = 16
  - Método directo:

    15 multiplicaciones

    $2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2$

  - Enfoque alternativo:

    4 multiplicaciones

    $2 \cdot 2 = 2^2$

    $2^2 \cdot 2^2 = 2^4$

    $2^4 \cdot 2^4 = 2^8$

    $2^8 \cdot 2^8 = 2^{16}$

- Para la instancia b = 2 y n = 32
  - Método directo:

    31 multiplicaciones

  - Enfoque alternativo:

    5 multiplicaciones

    $2 \cdot 2 = 2^2$

    $2^2 \cdot 2^2 = 2^4$

    $2^4 \cdot 2^4 = 2^8$

    $2^8 \cdot 2^8 = 2^{16}$

    $2^{16} \cdot 2^{16} = 2^{32}$

Comparando Algoritmos (III)

- Complejidad Método Directo:

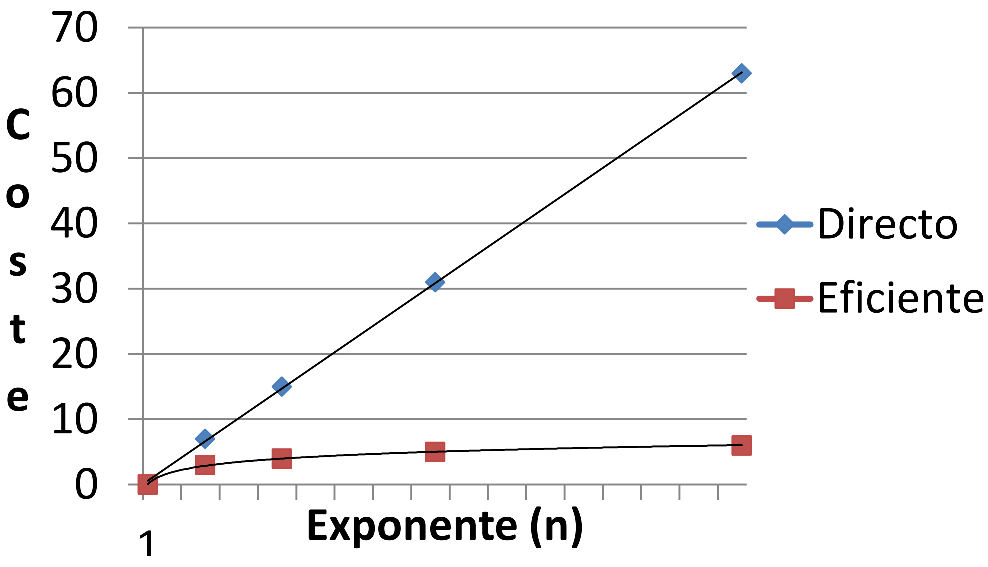
$$T(n) = \sum_{i=2}^{n} 1 = n - 1$$

- Complejidad Método Alternativo:

$$T(n) = \begin{cases} 0 & n = 1 \\ T\left(\frac{n}{2}\right) + 1 & n > 1 \end{cases}$$

Tras resolver la recurrencia

$$T(n) = \log(\text{n})$$



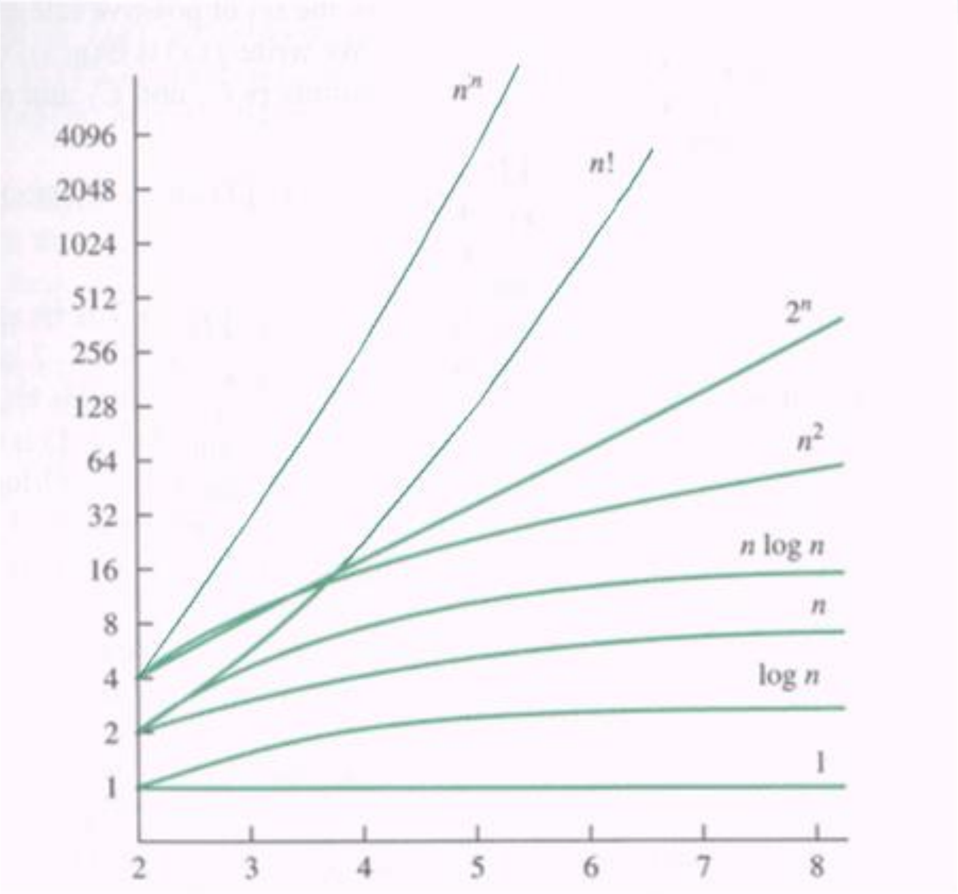
- Estudio asintótico: al crecer el tamaño del problema el coste computacional del primer algoritmo crece mucho más que el del segundo.
- La complejidad del primer algoritmo parece mayor que la del segundo.

Órdenes de crecimiento (I)

- Sabemos estimar el coste computacional *T(n)* de un algoritmo en función del tamaño de los datos de entrada.

- El orden de crecimiento nos indica cómo varía el coste computacional cuando aumenta el tamaño de la entrada, e.g., logarítmicamente, linealmente, cuadráticamente, exponencialmente, etc.

- Dados dos algoritmos, consideraremos más eficiente (menos complejo) a aquél que tenga un menor orden de crecimiento del coste computacional.



Órdenes de crecimiento(II)

	Logarítmico	Lineal	Casi lineal	Cuadrático	Cúbico	Exponencial	Factorial
Entrada	$\log_2(n)$	n	$n \log_2(n)$	n^2	n^3	2^n	$n!$
10^1	3.3	10^1	$3.3 \cdot 10^1$	10^2	10^3	$1.0 \cdot 10^3$	$3.6 \cdot 10^6$
10^2	6.6	10^2	$6.6 \cdot 10^2$	10^4	10^6	$1.3 \cdot 10^{30}$	$9.3 \cdot 10^{157}$
10^3	10	10^3	$1.0 \cdot 10^4$	10^6	10^9	$1.1 \cdot 10^{301}$	$4.0 \cdot 10^{2567}$
10^4	13	10^4	$1.3 \cdot 10^5$	10^8	10^{12}	$2.0 \cdot 10^{3010}$	$2.8 \cdot 10^{35659}$
10^5	17	10^5	$1.7 \cdot 10^6$	10^{10}	10^{15}	$1.0 \cdot 10^{30103}$	$2.8 \cdot 10^{456573}$
10^6	20	10^6	$2.0 \cdot 10^7$	10^{12}	10^{18}	$9.9 \cdot 10^{301029}$	$8.3 \cdot 10^{5565708}$

Algoritmos Eficientes

Algoritmos Tratables

Algoritmos Intratables

Sólo son prácticos para resolver problemas de pequeño tamaño

Notación Asintótica

- La notación asintótica nos permite comparar los órdenes de crecimiento de diferentes algoritmos para un tamaño de la entrada lo suficientemente grande.

- Cada notación permite acotar de diferente manera el orden de crecimiento de un algoritmo.

- Estudiaremos:
  - Cota superior: O(·) (O grande)
  - Cota inferior: Ω(·) (Omega grande)
  - Orden exacto: Θ(·) (Zeta)

Notación Asintótica *O(g(n)) (I)*

- Definición:

$$O\big(g(n)\big) = \{f: \mathbb{N} \rightarrow \mathbb{N} \mid \exists k \geq 0, c > 0: (\forall n \geq k: f(n) \leq cg(n))\}$$

- Informalmente, *O(g(n))* es el conjunto de funciones acotadas superiormente por un múltiplo de *g*.

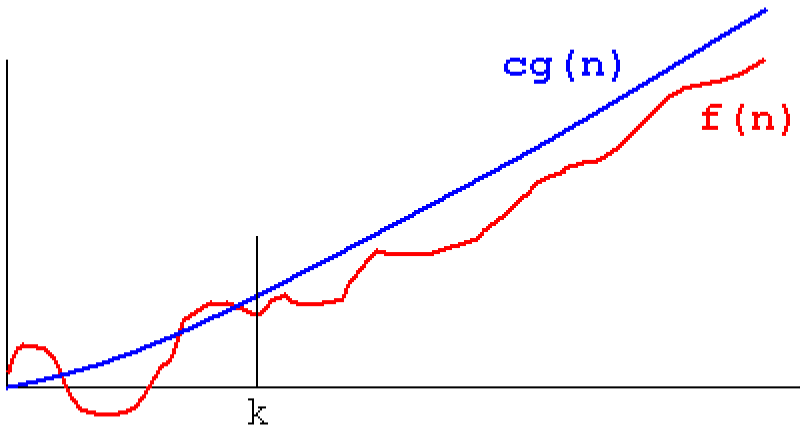
- Se utiliza para probar que la complejidad de un algoritmo como muy mal se va a comportar como la función *g*, que se toma como referencia.

- Observa que los valores iniciales de ambas funciones no importan, lo que es relevante es que a partir de un cierto tamaño de entrada *k*, *f* se comporte mejor o igual que un múltiplo de *g*.

Notación Asintótica *O(g(n)) (II)*

- Definición:

$$O\big(g(n)\big) = \{f: \mathbb{N} \to \mathbb{N} \mid \exists k \geq 0, c > 0: (\forall n \geq k: f(n) \leq cg(n))\}$$



- Ejemplos:

$$n \in O(n^2) \qquad 100n + 5 \in O(n^2) \qquad 0.5n\,(n-1) \in O(n^2)$$

$$n^3 \notin O(n^2) \qquad 0.00001n^3 \notin O(n^2) \qquad n^4 + n + 1 \notin O(n^2)$$

Notación Asintótica $\Omega(g(n))$ (I)

- Definición:

$$\Omega\big(g(n)\big) = \{f: \mathbb{N} \to \mathbb{N} \mid \exists k \geq 0, c > 0: (\forall n \geq k: f(n) \geq cg(n))\}$$

- Informalmente, $\Omega(g(n))$ es el conjunto de funciones acotadas inferiormente por un múltiplo de *g*.

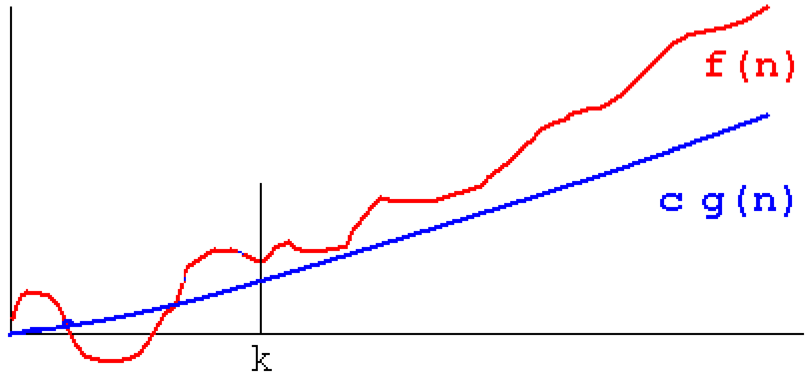
- Se utiliza para probar que la complejidad de un algoritmo como muy bien se va a comportar como la función *g*, que se toma como referencia.

- Como en el caso de *O*, los valores iniciales de ambas funciones no importan, lo que es relevante es que a partir de una cierto tamaño de entrada *k*, *f* se comporte peor o igual que un múltiplo de *g*.

Notación Asintótica $\Omega(g(n))$ (II)

- Definición:

$$\Omega(g(n)) = \{f: \mathbb{N} \rightarrow \mathbb{N} \mid \exists k \geq 0, c > 0: (\forall n \geq k: f(n) \geq cg(n))\}$$



- Ejemplos:

$$n^3 \in \Omega(n^2) \quad 100n + 5 \in \Omega(n) \quad 0.5n\,(n-1) \in \Omega(n^2)$$

$$n \notin \Omega(n^2) \quad 0.01n^2 \in \Omega(n) \quad n^2 + n + 1 \notin O(n^3)$$

Notación Asintótica *Θ(g(n)) (I)*

- Definición:

$$\Theta(g(n)) = \{f: \mathbb{N} \rightarrow \mathbb{N} \mid \exists k \geq 0, c_1 > 0, c_2 > 0: (\forall n \geq k: c_1 g(n) \leq f(n) \leq c_2 g(n))\}$$

- Informalmente, *Θ(g(n))* es el conjunto de funciones con el mismo orden de crecimiento que g.

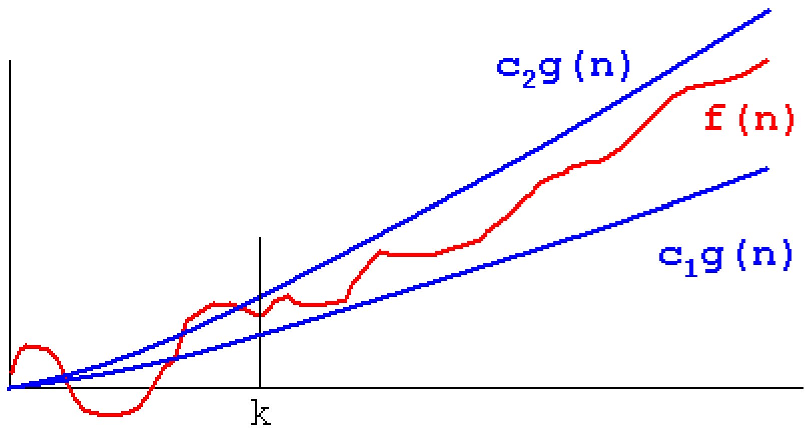
- Se utiliza para probar que la complejidad de un algoritmo es igual asintóticamente a la función *g*, que se toma como referencia.

- Como en los dos casos anteriores, los valores iniciales de ambas funciones no importan, lo que es relevante es que a partir de una cierto tamaño de entrada *k*, *f* crezca de forma *similar* a *g*.

Notación Asintótica $\Theta(g(n))$ (II)

- Definición:

$$\Theta\big(g(n)\big) = \{f\colon \mathbb{N} \to \mathbb{N} \mid \exists k \geq 0, c_1 > 0, c_2 > 0\colon (\forall n \geq k\colon c_1 g(n) \leq f(n) \leq c_2 g(n))\}$$



- Ejemplos:

$$100n^2 \in \Theta(n^2) \quad 100n + 5 \in \Theta(n) \quad 0.5n\,(n-1) \in \Theta(n^2)$$

$$n \notin \Theta(n^2) \quad 0.01n \notin \Theta(n^2) \quad n^2 + n + 1 \notin \Theta(n^3)$$

Principio de Invarianza

- La complejidad de dos implementaciones de un mismo algoritmo no diferirán más que en una constante multiplicativa.

- Si $T_1(n)$ y $T_2(n)$ determinan el coste temporal de dos implementaciones de un mismo algoritmo, se verifica que:

$$T_1(n) \in \Theta(T_2(n))$$

$$T_2(n) \in \Theta(T_1(n))$$

- Para resolver un problema de una forma más eficiente hay que encontrar un algoritmo mejor, no implementar el mismo algoritmo de otra forma o con otro lenguaje de programación.

Notación Asintótica: propiedades (I)

- Transitiva (también para $\Omega(\cdot)$ y $O(\cdot)$):

  $f(n) \in \Theta(g(n))$ y $g(n) \in \Theta(h(n)) \Rightarrow f(n) \in \Theta(h(n))$.

- Reflexiva (también para $\Omega(\cdot)$ y $O(\cdot)$): $g(n) \in \Theta(g(n))$

- Simétrica: $h(n) \in \Theta(g(n)) \Leftrightarrow g(n) \in \Theta(h(n))$

- Simétrica transpuesta: $h(n) \in \Omega(g(n)) \Leftrightarrow g(n) \in O(h(n))$.

- $f(n) \in O(g(n))$ y $f(n) \in \Omega(g(n)) \Leftrightarrow f(n) \in \Theta(g(n))$.

Notación Asintótica: propiedades (II)

- Suma de funciones:

$$\text{Si } f_1(n) \in O(g_1(n)) \quad \text{y} \quad f_2(n) \in O(g_2(n)), \text{ entonces}$$

$$f_1(n) + f_2(n) \in O(max\{g_1(n), g_2(n)\})$$

- Ídem para $\Omega(\cdot)$ y $\Theta(\cdot)$.

- Esta propiedad es útil cuando tenemos algoritmos compuestos de partes diferenciadas que se ejecutan secuencialmente y para las que conocemos la complejidad.

- Ejemplo: Buscar duplicados en un array de *n* elementos desordenados.

  Para ello:
  - Ordenamos el array usando un método de complejidad $O(n^2)$.
  - Comparamos elementos consecutivos, que tiene una complejidad de orden $O(n)$.
  - La complejidad del algoritmo, $T(n)$, está dominada por la primera parte, por lo que, $T(n) \in O(n^2) = O(\max\{n^2, n\})$.

Notación Asintótica: propiedades (III)

- Multiplicación de funciones:

Si $f_1(n) \in O(g_1(n))$ y $f_2(n) \in O(g_2(n))$, entonces

$$f_1(n) \cdot f_2(n) \in O(g_1(n) \cdot g_2(n)\})$$

- Ídem para $\Omega(\cdot)$ y $\Theta(\cdot)$.

- Esta propiedad es útil cuando tenemos conocemos que una parte del algoritmo tiene complejidad $O(g_1(n))$ y es invocada $O(g_2(n))$ veces.

- Ejemplo: Ordenación por selección. Para ello buscamos sucesivamente el i-ésimo menor elemento y lo colocamos en la posición i, $(0 <= i < n - 1)$.
  - El recorrido de la lista es de orden O(n).
  - En cada paso, buscar el menor de n-i elementos es $O(n)$.
  - La complejidad del algoritmo es $O(n \cdot n) = O(n^2)$.

Notación Asintótica: propiedades (IV)

- Comparación de órdenes de crecimiento:

$$\lim_{n\to\infty}\frac{f(n)}{g(n)}=\begin{cases}0 & f(n)\in O(g(n))\\ c>0 & f(n)\in\Theta(g(n))\\ \infty & f(n)\in\Omega(g(n))\end{cases}$$

- Ejemplos:

$$\lim_{n\to\infty}\frac{1}{n^2}=0\Longrightarrow 1\in O(n^2)$$

$$\lim_{n\to\infty}\frac{2n}{n}=2>0\Longrightarrow 2n\in\Theta(n)$$

$$\lim_{n\to\infty}\frac{2n^2}{n}=\infty\Longrightarrow 2n^2\in\Omega(n)$$

$$\lim_{n\to\infty}\frac{\frac{1}{2}n(n-1)}{n^2}=\frac{1}{2}\lim_{n\to\infty}\frac{n^2-n}{n^2}=\frac{1}{2}\lim_{n\to\infty}(1-\frac{1}{n})=\frac{1}{2}\Longrightarrow\frac{1}{2}n(n-1)\in O(n^2)$$

Notación Asintótica: propiedades (V)

- Regla de L'Hôpital:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{f'(n)}{g'(n)}$$

donde f'(n) = d f(n) /dn

- Ejemplos:

$$\lim_{n\to\infty} \frac{\log(n)}{\sqrt{n}} = \frac{\infty}{\infty}$$

$$\lim_{n\to\infty} \frac{\log(n)}{n^{1/2}} = \lim_{n\to\infty} \frac{1/n}{\frac{1}{2}n^{-1/2}} = \lim_{n\to\infty} \frac{1}{\frac{1}{2}\sqrt{n}} = \frac{1}{\infty} = 0$$

Por tanto, $\log(n) \in O(\sqrt{n})$

Notación Asintótica: propiedades (VI)

$O(\log_a n) = O(\log_b n)$

Ya que, por la propiedad de cambio de base de los logaritmos

$$\log_a n = \frac{\log_b n}{\log_b a} = \frac{1}{\log_b a} \log_b n = Cte \cdot \log_b n$$

Por esta razón no es necesario especificar la base del logaritmo: O(log n).

El Teorema Maestro (I)

- El Teorema Maestro nos da el orden de complejidad del algoritmo sin llegar a conocer la expresión exacta de la función complejidad.

- Se puede aplicar cuando al analizar un algoritmo se obtiene la expresión

$$T(n) = \begin{cases} aT(^n/_b) + f(n) & n > 1 \\ \Theta(1) & n = 1 \end{cases}$$ con a≥1, b>1

$$T(n) \in \begin{cases} \Theta\left(n^{log_b a}\right) & \exists \varepsilon > 0 : f(n) \in O\left(n^{log_b a - \varepsilon}\right) \\ \Theta\left(n^{log_b a} \cdot log^{k+1} n\right) & f(n) \in \Theta(n^{log_b a} \cdot log^k n) \\ \Theta\left(f(n)\right) & \begin{array}{l} \exists \varepsilon > 0 : f(n) \in \Omega\left(n^{log_b a + \varepsilon}\right) \ y \\ \exists e < 1, n_0 \geq 0 \ : \ (\forall n \geq n_0 : a \cdot f(^n/_b) \leq e \cdot f(n)) \end{array} \end{cases}$$

El Teorema Maestro (II)

$$T(n) \in \begin{cases} \Theta\left(n^{\log_b a}\right) & \exists \varepsilon > 0 : f(n) \in O\left(n^{\log_b a - \varepsilon}\right) \\ \Theta\left(n^{\log_b a} \cdot \log^{k+1} n\right) & f(n) \in \Theta(n^{\log_b a} \cdot \log^k n) \\ \Theta\left(f(n)\right) & \begin{array}{l} \exists \varepsilon > 0 : f(n) \in \Omega\left(n^{\log_b a + \varepsilon}\right) \ y \\ \exists e < 1, n_0 \geq 0 \ : \ (\forall n \geq n_0 : a \cdot f(n/b) \leq e \cdot f(n)) \end{array} \end{cases}$$

- Intuitivamente el teorema maestro está comparando *f(n)* con $n^{\log_b a}$ y aplicando la propiedad de la suma de funciones.
  - Caso 1: Si *f(n)* es estrictamente mejor que $n^{\log_b a}$, entonces $T(n) \in \Theta\left(n^{\log_b a}\right)$.
  - Caso 2: Si *f(n)* es del mismo orden que $n^{\log_b a}$, salvo alguna constante logarítmica $\log^k n$, entonces la complejidad es la de ambas funciones aumentando un grado la potencia del factor logarítmico *k*. $T(n) \in \Theta\left(n^{\log_b a} \log^{k+1} n\right)$.
  - Caso 3: Si *f(n)* es estrictamente peor que $n^{\log_b a}$entonces $T(n) \in \Theta(f(n))$. Sólo se puede aplicar si *f(n)* satisface la condición de regularidad $af(n/b) \leq ef(n)$ para valores de *n* lo suficientemente grandes y una constante positiva *e < 1*.

Teorema Maestro. Ejemplos (I)

- $T(n) = 9T(n/3) + n$

  Identificamos *f(n) = n* y $n^{log_b a} = n^{log_3 9} = n^2$. Suponemos un *ε > 0* cercano a *0.*

$$\lim_{n\to\infty} \frac{n}{n^{2-\varepsilon}} = \lim_{n\to\infty} \frac{1}{n^{2-\varepsilon-1}} = \lim_{n\to\infty} \frac{1}{n^{1-\varepsilon}} = 0 \Longrightarrow f(n) = n \in O(n^{2-\varepsilon}) = O(n^{log_b a-\varepsilon})$$

  Por tanto, $\mathrm{T(n)} \in \Theta(n^2)$

- $T(n) = T(2n/3) + 1$

  Identificamos *f(n) = 1* y $n^{log_b a} = n^{log_{3/2} 1} = n^0 = 1.$

  En este caso las dos funciones son del mismo orden, salvo la constante logarítmica $log^0 n$.

  Por tanto, $\mathrm{T}(n) \in \Theta(1 \cdot log^{0+1} n) = \Theta(logn)$

Teorema Maestro. Ejemplos (II)

- $T(n) = 3T(n/4) + n \log n$

  Identificamos *f(n) = n log n* y $n^{log_b a} = n^{log_4 3}$. Como $log_4 3 < 1$, parece que estamos en el caso 3. Suponemos un *ε > 0* cercano a *0.*

$$\lim_{n\to\infty} \frac{n \log n}{n^{log_4 3+\varepsilon}} = \lim_{n\to\infty} \frac{\log n}{n^{log_4 3+\varepsilon-1}} = \lim_{n\to\infty} \log n \cdot n^{1-(log_4 3+\varepsilon)} = \infty$$

$$\Longrightarrow f(n) = nlog\ n \in \Omega\left(n^{log_4 3+\varepsilon}\right)$$

  Por otro lado, hay que encontrar un valor 0 < e < 1 que satisfaga $af(n/b) \leq ef(n)$

$$3f(n/4) \leq ef(n); 3\frac{n}{4}\log\left({}^{n}\!/_{4}\right) \leq e\ n\ log\ n;$$

$$\frac{3}{4}n\log n - \frac{3}{4}n\log 4 \leq e\ n\ log\ n; \frac{3}{4} - \frac{3\log 4}{4\log n} \leq e$$

e = 3/4 < 1 satisface la inecuación.

  Por tanto, $\mathrm{T(n)} \in \Theta(n \log n)$

- $T(n) = 2T(n/2) + n \log n$

  Identificamos *f(n) =* $n \log n$ y $n^{log_b a} = n^{log_2 2} = n$.

  En este caso las dos funciones son del mismo orden, salvo la constante logarítmica $log^1 n$.

  Por tanto, $\mathrm{T}(n) \in \Theta(n \cdot log^{1+1} n) = \Theta(n \cdot log^2 n)$

Versión reducida del Teorema Maestro

- Caso especial que se puede aplicar cuando *f(n)* es un polinomio. Es decir, si la función de complejidad tiene la forma $T(n) = aT(n/_b) + f(n)$ y $f(n) \in \Theta(n^d)$ con *d≥0*.

$$T(n) \in \begin{cases} \Theta\left(n^{log_b a}\right) & a > b^d \\ \Theta(n^d \log n) & a = b^d \\ \Theta\left(n^d\right) & a < b^d \end{cases}$$

```
/**
 * @param l lista con n > 0 elementosr
 * @param min y max, índices máximo y mínimo a suma
 * @return Suma de los elementos de la lista
 * Implmentación recursiva
 */
public static double suma(List<Double> l,int min,int max){
      if (max < min) return 0;
      if (max == min) return l.get(min);
      else {
         double s1 = suma(l, min,(min+max)/2);
         double s2 = suma(l, (min+max)/2+1,max);
         return s1+s2;
      }
}
```

- $T(n) = 2T(n/_2) + 1$ y $f(n) \in \Theta(n^0)$.
- Identificamos *a = 2*, *b = 2* y *d = 0*.
- Como *a = 2 > 2⁰ = b^d* podemos afirmar por el teorema maestro reducido que $T(n) \in \Theta(n) = \Theta\left(n^{log_2 2}\right)$