Estructuras de datos y de la información

Las estructuras de datos se utilizan en la implementación de sistemas informáticos.

Para la correcta implementación de estos sistemas es necesario conocer las siguientes técnicas:

- Análisis de la eficiencia de los algoritmos.
- Especificación de algoritmos.
- Técnicas de programación estructurada y de programación recursiva.
- Abstracción de datos y encapsulamiento.

El análisis de la eficiencia de un algoritmo estudia el tiempo que tarda un algoritmo en ejecutarse y la memoria que requiere.

El gasto en tiempo y en espacio son normalmente factores contrapuestos. Muchas veces se puede mejorar el tiempo de ejecución a costa de incrementar el espacio ocupado por el algoritmo.

Tiempo de ejecución de un algoritmo

El tiempo de ejecución se mide contando el número de *pasos* de programa.

1. Los comentarios y las instrucciones de declaración no son pasos de programa.
2. Las expresiones y las instrucciones de asignación son un paso de programa.
   - Si las expresiones contienen llamadas a funciones al número de pasos se le suma el número de pasos de la función. Si la llamada a la función tiene paso de parámatros por valor hay que tener en cuenta las asignaciones a estos parámetros. Si la función es recursiva deben considerarse también las variables locales, ya que deben ser almacenadas en la pila del sistema.
   - Si las expresiones manejan tipos compuestos (por ejemplo vectores) el número de pasos depende del tamaño de los datos.
3. Las instrucciones de iteración.
   - La ejecución de la parte de control cuenta como el número de pasos de la expresión que debe ser evaluada.
   - El número de pasos necesarios para ejecutar el cuerpo del bucle debe multiplicarse por el número de veces que se ejecuta el bucle.

Tiempo de ejecución de un algoritmo

4. Instrucciones de selección. La ejecución de la parte de control cuenta como el número de pasos de la expresión que debe ser evaluada. Se considerarán las diversas alternativas al contar el número de pasos total del programa.
5. La instrucción de retorno *return* cuenta como el número de pasos de la expresión que debe ser evaluada.

Ejemplos:

```
float sum (float *a, const int n)  {
1   float s = 0;
2   int i = 0;
3   while (i < n) {
4     s +=a[i];
5     i++; }
6   return s;
    }
```

línea	contador	frecuencia	total pasos
1	1	1	1
2	1	1	1
3	1	$n+1$	$n+1$
4	1	n	n
5	1	n	n
6	1	1	1
			$3n+4$

Tiempo de ejecución de un algoritmo

```
float rsum (float *a, const int n) {
1    if (n <= 0)
2       return 0;
3    else return (rsum(a, n-1) + a[n-1]);
  }
```

línea	contador	frecuencia		total pasos	
		$n=0$	$n>0$	$n=0$	$n>0$
1	1	1	1	1	1
2	1	1	0	1	0
3	$1+t_{rsum}(n-1)$	0	1	0	$1+t_{rsum}(n-1)$
				2	$2+t_{rsum}(n-1)$

Factores de los que depende el tiempo de ejecución de un algoritmo

- tamaño de los datos de entrada
- *contenido* de los datos de entrada
- el código generado por el compilador y el computador concreto

Ejemplo: Algoritmo de ordenación por selección

```
1 for (int i = 1; i < n; i++)
  {  // pmin es la posición del minimo de a[i..n]
2   int pmin = i;
3   for (int j = i+1; j < n; j++)
4     if (a[j] < a[pmin]) pmin = j;
5   intercambiar(a[i],a[pmin]);
}
```

Ejercicio: Contar el número de pasos de programa del algoritmo de ordenación por selección

Factor 1: tamaño de los datos de entrada n

En el algoritmo de ordenación por selección:

- Para un vector de 10 posiciones se realizan *como mucho* 10^2 comparaciones, $10^2 + 10$ asignaciones y 10 intercambios.
- Para un vector de 100 posiciones se realizan *como mucho* 100^2 comparaciones, $100^2 + 100$ asignaciones y 100 intercambios.

Normalmente si la entrada son vectores o ficheros n es el tamaño de los mismos, si son matrices cuadradas, su dimensión. Para algoritmos numéricos n suele ser el valor de la entrada en lugar de su longitud.

El tamaño de los datos de entrada puede depender de más de un parámetro, por ejemplo en matrices no cuadradas o vectores de longitudes no relacionadas

Factor 2: *contenido* de los datos de entrada

En el algoritmo de ordenación por selección:

- Si el vector está ordenado se realizan n asignaciones y n^2 comparaciones,
- Si el vector está completamente desordenado (ordenado en orden descendente), se realizan n asignaciones, n^2 comparaciones y asignaciones y n intercambios

Analizar la eficiencia del algoritmo en el *caso peor*: fijado un tamaño de problema, la eficiencia de aquellos ejemplares de dicho tamaño en los que el algoritmo emplea más tiempo. Obtiene una cota superior del tiempo de ejecución para cualquier ejemplar.

Analizar la eficiencia del algoritmo en el *caso promedio*: es necesario conocer el tiempo de ejecución de cada ejemplar y la frecuencia con que se presenta cada uno de ellos (su distribución de probabilidades). Es difícil conocer la distribución de probabilidades y el análisis matemático para realizar el cálculo.

Factor 3: el compilador y el computador concreto

Se utiliza el *criterio asintótico*, esto es ignorar el efecto del computador, ya que:

- Se pretende analizar la eficiencia del algoritmo de un modo totalmente independiente de las máquinas y lenguajes existentes
- Diferentes implementaciones de un algoritmo diferirán en sus tiempos de ejecución a lo sumo en una constante multiplicativa, para tamaño de los datos de entrada suficientemente grandes

Un factor constante en los tiempos de ejecución no se considera en general importante frente a una dependencia del tamaño n de los datos de entrada.

Coste		$n = 10^3$	Tiempo	$n = 10^6$	Tiempo
Logarítmico	$log_2(n)$	10	10 segundos	20	20 segundos
Lineal	n	10^3	16 minutos	10^6	11 días
Cuadrático	n^2	10^6	11 días	10^{12}	30.000 años

Medidas asintóticas

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

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

Una función g es del orden de $f(n)$ cuando $g \in \mathcal{O}(f(n))$.

El conjunto $\mathcal{O}(f(n))$ define un *orden de complejidad*. Se escoge como representante del orden $\mathcal{O}(f(n))$ la función $f(n)$ más sencilla posible dentro del mismo.

Complejidad lineal	$\mathcal{O}(n)$
Complejidad cuadrática	$\mathcal{O}(n^2)$
Complejidad constante	$\mathcal{O}(1)$

Funciones del orden de $f(n)$

Proposiciones:

1. $\mathcal{O}(af) = \mathcal{O}(f)$ con $a \in \mathcal{R}^+$
   - $\supseteq$ $g \in \mathcal{O}(f)$ entonces $\exists n_0 \in \mathcal{N}, \exists c \in \mathcal{R}^+$ tal que $\forall n \geq n_0, g(n) \leq cf(n)$.
     Sea $c' = \frac{c}{a}$ entonces $c' \in \mathcal{R}^+$ y $\forall n \geq n_0, g(n) \leq c'af(n)$.
     Por lo tanto $g \in \mathcal{O}(af)$.
   - $\subseteq$ $g \in \mathcal{O}(af)$ entonces $\exists n_0 \in \mathcal{N}, \exists c \in \mathcal{R}^+$ tal que $\forall n \geq n_0 g(n) \leq caf(n)$.
     Sea $c' = ca$ entonces $c' \in \mathcal{R}^+$ y $\forall n \geq n_0, g(n) \leq c'f(n)$.
     Por lo tanto $g \in \mathcal{O}(f)$.
2. $\mathcal{O}(n^p) \subset \mathcal{O}(n^{p+1})$.
   (a) $g \in \mathcal{O}(n^p)$ entonces $\exists n_0 \in \mathcal{N}, \exists c \in \mathcal{R}^+$ tal que $\forall n \geq n_0, g(n) \leq cn^p$.
   (b) Tenemos $\forall n \geq n_0, g(n) \leq cn^{p+1}$.
   (c) Por lo tanto $g \in \mathcal{O}(n^{p+1})$.
   La inclusión es estricta porque $n^{p+1} \notin \mathcal{O}(n^p)$.

Funciones del orden de $f(n)$

3. Si $g_1(n), g_2(n) \in \mathcal{O}(f(n))$ entonces $g_1(n) + g_2(n) \in \mathcal{O}(f(n))$
   (a) $g_1 \in \mathcal{O}(f(n))$ entonces $\exists n_1 \in \mathcal{N}, \exists c_1 \in \mathcal{R}^+$ tal que $\forall n \geq n_1, g(n) \leq c_1 f(n)$.
   (b) $g_2 \in \mathcal{O}(f(n))$ entonces $\exists n_2 \in \mathcal{N}, \exists c_2 \in \mathcal{R}^+$ tal que $\forall n \geq n_2, g(n) \leq c_2 f(n)$.
   (c) Sea $n_0 = max(n_1, n_2)$ y $c = c_1 + c_2$, entonces $\forall n \geq n_0, g_1(n) + g_2(n) \leq cf(n)$.
   (d) Por lo tanto $g_1 + g_2 \in \mathcal{O}(f)$.

4. Todo polinomio en n de grado m cuyo coeficiente correspondiente al mayor grado sea positivo es del orden de n^m.
   (a) Eliminar del polinomio los términos con coeficiente negativo, ya que si $P(n, m) \in \mathcal{O}(f(n))$, entonces $P(n, m) - P'(n, m') \in \mathcal{O}(f(n))$ para cualquier polinomio $P'(n, m')$ con $m' < m$.
   (b) Todos los términos del polinomio resultante están en el orden $\mathcal{O}(n^m)$.
   (c) Como los órdenes son cerrados bajo la suma, la suma está en $\mathcal{O}(n^m)$.

Funciones del orden de $f(n)$

Teorema.

1. $\lim_{n\to\infty} \frac{f(n)}{g(n)} = m \in \mathcal{R}^+ \Rightarrow f \in \mathcal{O}(g)$ y $g \in \mathcal{O}(f)$.

   $f(n)$ y $g(n)$ tienen la misma forma de crecimiento

   Demostración:
   (a) $\forall \varepsilon > 0, \exists n_0 \in \mathcal{N}$ tal que $\forall n \geq n_0$ tenemos $\left| \frac{f(n)}{g(n)} - m \right| < \varepsilon$.
   (b) Si $\frac{f(n)}{g(n)} - m > 0$ entonces, fijado ε y tomando $c = \varepsilon + m$, tenemos $\forall n \geq n_0, f(n) < cg(n)$.
   (c) Si $\frac{f(n)}{g(n)} - m < 0$ entonces, fijado ε y tomando $c = m - \varepsilon$, tenemos $\forall n \geq n_0, f(n) < cg(n)$. Por lo tanto $f \in \mathcal{O}(g)$.
   (d) Por otra parte, si $\lim_{n\to\infty} \frac{f(n)}{g(n)} = m$ entonces $\lim_{n\to\infty} \frac{g(n)}{f(n)} = \frac{1}{m}$. Como $m \in \mathcal{R}^+$ tenemos $\frac{1}{m} \in \mathcal{R}^+$, y razonando como en el caso anterior tenemos $g \in \mathcal{O}(f)$.

Funciones del orden de $f(n)$

2. $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \Rightarrow f \in \mathcal{O}(g)$ y $g \notin \mathcal{O}(f)$.

   $f(n)$ crece mucho menos que $g(n)$

   (a) $\forall \varepsilon > 0, \exists n_0 \in \mathcal{N}$ tal que $\forall n \geq n_0$ tenemos $\left|\frac{f(n)}{g(n)}\right| < \varepsilon$.

   (b) Fijando ε y tomando $c = \varepsilon$, tenemos que $\forall n \geq n_0, f(n) < cg(n)$ y por lo tanto $f \in \mathcal{O}(g)$.

   (c) Demostramos que $g \notin \mathcal{O}(f)$ por reducción al absurdo.

   (d) Supongamos $g \in \mathcal{O}(f)$, entonces $\exists n_0 \in \mathcal{N}$ y $\exists c \in \mathcal{R}^+$ tal que $\forall n \geq n_0, g(n) < cf(n)$, por lo tanto $\forall n \geq n_0, \frac{1}{c} < \frac{f(n)}{g(n)}$

   (e) $\frac{1}{c} = \lim_{n\to\infty} \frac{1}{c} \leq \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$

   (f) Por lo tanto $\frac{1}{c} \leq 0$, lo que es un contradicción ya que $c \in \mathcal{R}^+$.

Funciones del orden de $f(n)$

3. $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty \Rightarrow f \notin \mathcal{O}(g)$ y $g \in \mathcal{O}(f)$.

   $f(n)$ crece mucho más que $g(n)$

   Demostración:

   (a) $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty \Leftrightarrow \lim_{n\to\infty} \frac{g(n)}{f(n)} = 0$,

   (b) Por el teorema anterior llegamos a $g \in \mathcal{O}(f)$ y $f \notin \mathcal{O}(g)$.

Se deduce por lo tanto la siguiente cadena de inclusiones:

$$\mathcal{O}(1) \subset \mathcal{O}(log(n)) \subset \mathcal{O}(n) \subset \mathcal{O}(n^2) \subset \ldots \subset \mathcal{O}(n^a) \subset \ldots \subset \mathcal{O}(2^n) \subset \mathcal{O}(n!).$$

Funciones del orden de $f(n)$

Problemas razonables hasta $\mathcal{O}(n^3)$

Problemas tratables los de complejidad polinómica.

Problemas intratables los de complejidad mayor que polinómica.

Matizaciones:

- En algunos casos la constante multiplicativa es tan grande que en la práctica son preferibles algoritmos teóricamente más ineficientes:

  Ejemplo: Tenemos un algoritmo cuyo tiempo de ejecución es $T_1 = 3n^3$ y otro cuyo tiempo de ejecución es $T_2 = 600n^2$. El primero es más rápido hasta $n > 200$

- Si un programa se va a utilizar pocas veces puede resultar preferible un algoritmo más ineficiente pero más rápido de implementar, ya que el coste de un programa incluye también su desarrollo y su mantenimiento.

Medidas asintóticas. $\Omega(f(n))$

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

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

La complejidad del algoritmo no es mejor que la representada por la función f. Si estamos analizando la eficiencia en el caso peor, se garantiza que el caso peor no es mejor que el representado por la función, pero puede haber casos que si sean mejores.

Proposición: (principio de dualidad)

$$g \in \Omega(f) \Leftrightarrow f \in \mathcal{O}(g)$$

Demostración:

- $f \in \mathcal{O}(g) \Leftrightarrow$
- $\exists n_0 \in \mathcal{N}, \exists c \in \mathcal{R}^+$ tal que $\forall n \geq n_0, f(n) \leq cg(n) \Leftrightarrow$
- $\exists n_0 \in \mathcal{N}, \exists \frac{1}{c} \in \mathcal{R}^+$ tal que $\forall n \geq n_0, g(n) \geq \frac{1}{c} f(n) \Leftrightarrow$
- $g \in \Omega(f)$.

Medidas asintóticas. $\Omega(f(n))$

Teorema (para Ω)

- $\lim_{n\to\infty} \frac{f(n)}{g(n)} = m \in \mathcal{R}+ \Rightarrow g \in \Omega(f)$ y $f \in \Omega(g)$.
- $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \in \mathcal{R}+ \Rightarrow g \in \Omega(f)$ y $f \notin \Omega(g)$.
- $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty \in \mathcal{R}+ \Rightarrow g \notin \Omega(f)$ y $f \in \Omega(g)$.

Proposición:

$$\Omega(f+g) = \Omega(max(f,g))$$

Demostración:

- $h(n) \in \Omega(f(n)+g(n)) \Leftrightarrow$
- $\exists n_0 \in \mathcal{N}, \exists c \in \mathcal{R}^+$ tal que $\forall n \geq n_0, h(n) \geq c(f(n)+g(n)) \Leftrightarrow$
- $\exists n_0 \in \mathcal{N}, \exists c \in \mathcal{R}^+$ tal que $\forall n \geq n_0, h(n) \geq c(max(f(n),g(n))) \Leftrightarrow$
- $h(n) \in \Omega(max(f(n),g(n)))$.

Medidas asintóticas. $\Omega(f(n))$

- $h(n) \in \Omega(max(f(n),g(n))) \Leftrightarrow$
- $\exists n_0 \in \mathcal{N}, \exists c \in \mathcal{R}^+$ tal que $\forall n \geq n_0, h(n) \geq c(max(f(n),g(n))) \Leftrightarrow$
- $\exists n_0 \in \mathcal{N}, \exists c \in \mathcal{R}^+$ tal que $\forall n \geq n_0, h(n) \geq \frac{c}{2}(f(n)+g(n)) \Leftrightarrow$
- $\exists n_0 \in \mathcal{N}, \exists c' = \frac{c}{2} \in \mathcal{R}^+$ tal que $\forall n \geq n_0, h(n) \geq c'(f(n)+g(n)) \Leftrightarrow$
- $h(n) \in \Omega(f(n)+g(n))$.

Medidas asintóticas

El conjunto de funciones $\Theta(f(n))$, leido *del orden exacto de f(n)*, se define como:

$$\Theta(f(n)) = \mathcal{O}(f(n)) \cap \Omega(f(n))$$

Teorema (para Θ)

- $\lim_{n\to\infty} \frac{f(n)}{g(n)} = m \in \mathcal{R}+ \Rightarrow g \in \Theta(f)$ y $f \in \Theta(g)$.
- $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \in \mathcal{R}+ \Rightarrow g \in \Theta(f)$ y $f \notin \Theta(g)$.
- $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty \in \mathcal{R}+ \Rightarrow g \notin \Theta(f)$ y $f \in \Theta(g)$.

Proposición:

$$\Theta(f+g) = \Theta(max(f,g)).$$

Análisis de algoritmos. (caso peor)

- Operaciones básicas: Asignación, operaciones booleanas, operaciones aritméticas.
  Coste: $\Theta(1)$.
- Composición secuencial: $S_1; S_2$ con t_1 coste de S_1 y t_2 coste de S_2.
  Coste: $\Theta(t_1+t_2) = \Theta(max(t_1,t_2))$.
- Casos: `if` B $\{S_1\}$ `else` $\{S_2\}$ con b coste de B, y t_1, t_2 costes de S_1 y S_2.
  Coste: $\mathcal{O}(max(b,t_1,t_2))$.
  $\Omega(max(b,min(t_1,t_2)))$.
- Bucles: `for`$(i = C_1; i <= C_2; i++)$ $S(i)$ con c_1,c_2 coste de C_1 y C_2 y $S(i) \in \Theta(t(i))$
  Coste: $\Theta(\max(c_1, c_2, \sum_{i=C_1}^{C_2}(t(i))))$

  En los bucles *while* se calcula el número de vueltas dependiendo del análisis que se esté haciendo (p.e. caso peor)

Análisis de algoritmos

Ejemplo: Algoritmo de ordenación por selección, complejidad en el caso peor

```
1 for (int i = 1; i < n; i++) {
2    int pmin = 1;
3    for (int j = i + 1; j <= n; j++)
4       if (a[j] < a[pmin]) pmin = j;
5    intercambiar(a[i],a[pmin]) }
```

Coste del bucle 3:
$\Sigma_{j=i+1}^{n}(c_p) = (n-i) * c_p \in \Theta(n)$.
Siendo c_p el coste de la instrucción 4.

Coste del bucle 1:
$\Sigma_{i=1}^{n-1}((n-i) * c_p + b) = (n-1) * (n * c_p + b) - c_p * \Sigma_{i=1}^{n-1}(i) =$
$(n-1) * (n * c_p + b) - c_p * \frac{n*(n+1)}{2} \in \Theta(n^2)$
Siendo b la suma del coste de las instrucciones 2 y 4.

Análisis de algoritmos

Ejemplo: Algoritmo de ordenación por inserción

```
1  for (int i = 2; i <= n; i++) {
2     elem x = T[i];
3     int j = i-1;
4     while ((j > 0) && (x < T[j])) {
5        T[j+1] = T[j];
6        j--;
      }
7     T[j+1] = x;
   }
```

Complejidad en el caso peor:
(vector ordenado de mayor a menor)
Coste del bucle 4:
$\Sigma_{j=1}^{i-1}(c) = i * c$
Coste del bucle 1: $\Sigma_{i=2}^{n}(i * c + b) =$
$c * (\frac{n*(n+1)}{2} - 1) + (n-1) * b \in \Theta(n^2)$
Complejidad en el caso mejor:
(vector ordenado de menor a mayor)
Coste del bucle 4: c
Coste del bucle 1:
$\Sigma_{i=2}^{n}(c + b) = (n-1) * (c + b) \in \Theta(n)$

Al evaluar la condición del *while* C++ no evalua la segunda condición si la primera es falsa.

Análisis de algoritmos

- LLamadas recursivas.

	Coste	Coste total
$T(\overline{P})$ `if`(B) `return` X `else` $T = T(\overline{P'})$	$T(\overline{P}) \in \mathcal{O}(t(p))$	$t(p)$ si caso base $t(p) = t(p') + d$ si caso recursivo

Ejemplo:

```
1 if (n == 0)
2    return 1;
3 else return fact(n-1) * n
```

Coste para $n = 0$: $t(n) = k_1$
Coste para $n > 0$: $t(n) = t(n-1) + k_2$
donde k_1 es el coste de devolver un 1 y
k_2 el coste de realizar una multiplicación

Hay que calcular $t(n)$ en función de $t(n-1) \Rightarrow$ *resolución de recurrencias*

Resolución de recurrencias. Ejemplos

Cálculo del factorial:

$$T(n) = \begin{cases} k_1 & n = 0 \\ T(n-1) + k_2 & n > 0 \end{cases}$$

Desplegamos la expresión $T(n)$
$T(n) = T(n-1) + k_2 = T(n-2) + k_2 + k_2 = \ldots = k_1 + n * k_2 \in \Theta(n)$

Búsqueda dicotómica:

```
if (c > f) return false
else { m = (c+f) / 2
   if (x < a[m]) busca(a,x,c,m-1)
   else if (x = a[m]) return true
   else busca(a,x,m+1,f) }
```

$$T(n) = \begin{cases} k_1 & n = 1 (\texttt{c} = \texttt{f}) \\ T(n/2) + k_2 & n > 1 \end{cases}$$

$T(n) = T(n/2) + k_2 = \ldots =$
$k_1 + \log_2(n) * k_2 \in \Theta(\log_2(n))$

Resolución de recurrencias

Teorema de la resta:

Dada una recurrencia del tipo :

$$T(n) = \begin{cases} c * n^k & 0 \leq n < b \\ a * T(n-b) + c * n^k & n \geq b \end{cases}$$

donde $a, c \in \mathcal{R}^+$, $k \in R^+ \cup \{0\}$, $n, b \in \mathcal{N}$

entonces

$$T(n) \in \begin{cases} \Theta(n^k) & a < 1 \\ \Theta(n^{k+1}) & a = 1 \\ \Theta(a^{n \ div \ b}) & a > 1 \end{cases}$$

- a representa el número de subproblemas que se generan
- b representa el factor de reducción del problema
- k representa el coste de las tareas que no son llamadas recursivas

Resolución de recurrencias

Teorema de la división:

Dada una recurrencia del tipo :

$$T(n) = \begin{cases} c * n^k & 0 \leq n < b \\ a * T(n/b) + c * n^k & n \geq b \end{cases}$$

donde $a, c \in \mathcal{R}^+$, $k \in R^+ \cup \{0\}$, $n, b \in \mathcal{N}$ y $b > 1$

entonces

$$T(n) \in \begin{cases} \Theta(n^k) & a < b^k \\ \Theta(n^k * \log_b(n)) & a = b^k \\ \Theta(n^{\log_b(a)}) & a > b^k \end{cases}$$