

Estructuras de Datos y Algoritmos

Tema 2. Análisis de la eficiencia

Prof. Dr. P. Javier Herrera

Contenido

- Introducción a la eficiencia computacional
- Cálculo básico de complejidades

Una leyenda ajedrecística

- Mucho tiempo atrás, el espabilado visir Sissa ben Dahir inventó el juego del ajedrez para el rey Shirham de la India. El rey ofreció a Sissa la posibilidad de elegir su propia recompensa. Sissa le dijo al rey que podía recompensarle en trigo con una cantidad equivalente a la cosecha de trigo en su reino de dos años, o bien con una cantidad de trigo que se calcularía de la siguiente forma:
 - un grano de trigo en la primera casilla de un tablero de ajedrez,
 - mas dos granos de trigo en la segunda casilla,
 - mas cuatro granos de trigo en la tercera casilla,
 - y así sucesivamente, duplicando el número de granos en cada casilla, hasta llegar a la última casilla.
- El rey pensó que la primera posibilidad era demasiado cara mientras que la segunda, medida además en simples granos de trigo, daba la impresión de serle claramente favorable.
- Así que sin pensárselo dos veces pidió que trajeran un saco de trigo para hacer la cuenta sobre el tablero de ajedrez y recompensar inmediatamente al visir.

¿Es una buena elección?

- El número de granos en la primera fila resultó ser:

$$2^0 + 2^1 + 2^2 + 2^3 + 2^4 + 2^5 + 2^6 + 2^7 = 255$$

- La cantidad de granos en las dos primeras filas es:

$$\sum_{i=0}^{15} 2^i = 2^{16} - 1 = 65\,535$$

- Al llegar a la tercera fila el rey empezó a pensar que su elección no había sido acertada, pues para llenar las tres filas necesitaba

$$\sum_{i=0}^{23} 2^i = 2^{24} - 1 = 16\,777\,216$$

granos, que pesan alrededor de 600 kilos ...

Endeudado hasta las cejas

- En efecto, para rellenar las 64 casillas del tablero hacen falta

$$\sum_{i=0}^{63} 2^i = 2^{64} - 1 = 18\,446\,744\,073\,709\,551\,615 \approx 1,84 * 10^{19}$$

granos, cantidad equivalente a las cosechas mundiales actuales de 1000 años.

- La función $2^n - 1$ (exponencial) representa el número de granos adeudados en función del número n de casillas a rellenar. Toma valores desmesurados aunque el número de casillas sea pequeño.
- El coste en tiempo de algunos algoritmos expresado en función del tamaño de los datos de entrada es también exponencial. Por ello **es importante estudiar el coste de los algoritmos y ser capaces de comparar los costes de algoritmos que resuelven un mismo problema.**

Motivación

- Entendemos por **eficiencia** el **rendimiento de una actividad en relación con el consumo de un cierto recurso**. Es diferente de la efectividad.
- Ejemplo para ver la importancia de que el coste del algoritmo sea pequeño:

n	$\log_{10} n$	n	$n \log_{10} n$	n^2	n^3	2^n
10	1 ms	10 ms	10 ms	0,1 s	1 s	1,02 s
10^2	2 ms	0,1 s	0,2 s	10 s	16,67 m	$4,02 * 10^{20}$ sig
10^3	3 ms	1 s	3 s	16,67 m	11,57 d	$3,4 * 10^{291}$ sig
10^4	4 ms	10 s	40 s	1,16 d	31,71 a	$6,3 * 10^{3000}$ sig
10^5	5 ms	1,67 m	8,33 m	115,74 d	317,1 sig	$3,16 * 10^{30093}$ sig
10^6	6 ms	16,67 m	1,67 h	31,71 a	317 097,9 sig	$3,1 * 10^{301020}$ sig

- Es un error pensar que basta esperar algunos años para que algoritmos tan costosos se puedan ejecutar con un coste en tiempo razonable.

¿Qué medimos y cómo?

- La eficiencia es mayor cuanto menor es la complejidad o el coste (consumo de recursos).
- Necesitamos determinar **cómo se ha de medir el coste de un algoritmo**, de forma que sea posible **compararlo con otros** que resuelven el mismo problema y **decidir cuál de todos es el más eficiente**.
- Una posibilidad para medir el coste de un algoritmo es contar cuántas instrucciones de cada tipo se ejecutan, multiplicar este número por el tiempo que emplea la instrucción en ejecutarse, y realizar la suma para los diferentes tipos.

t_a = tiempo de asignación

t_c = tiempo de comparación

t_i = tiempo de incremento

t_v = tiempo de acceso a un vector

Ejemplo

- Ordenación por selección del vector $V[1..n]$

para $i := 1$ **hasta** $n - 1$ **hacer**

$pmin := i$;

para $j = i + 1$ **hasta** n **hacer**

si $V[j] < V[pmin]$ **entonces** $pmin := j$ **fsi**

fpara ;

intercambiar($V[i], V[pmin]$)

fpara

- Control del primer bucle: $t_a + (n - 1)t_i + nt_c$
- Primera asignación: $(n - 1)t_a$
- Control del bucle interno, para cada i : $t_a + (n - i)t_i + (n - i + 1)t_c$
- instrucción **si**, para cada i , tiempo mínimo: $(n - i)(2t_v + t_c)$
tiempo máximo: $(n - i)(2t_v + t_c) + (n - i)t_a$
- Intercambiar: $(n - 1)(2t_v + 3t_a)$

Ejemplo

- El bucle interno en total en el caso peor, el más desfavorable,

$$\sum_{i=1}^{n-1} (t_a + t_c + (n-i)(t_i + 2t_c + 2t_v + t_a))$$

- Concluimos que

$$T_{\min} = An^2 - Bn + C$$

$$T_{\max} = A'n^2 - B'n + C'$$

Factores

- El tiempo de ejecución de un algoritmo depende en general de tres factores:
 1. El **tamaño** de los datos de entrada. Por ejemplo:
 - Para un vector: su longitud.
 - Para un número natural: su valor o el número de dígitos.
 - Para un grafo: el número de vértices y/o el número de aristas.
 2. El **contenido** de esos datos.
 3. El código generado por el **compilador** y el **ordenador** concreto utilizados.

Factores

- Si el tiempo que tarda un algoritmo A en procesar una entrada concreta $\bar{x}$ lo denotamos por $t_A(\bar{x})$, definimos la complejidad de A en el **caso peor** como

$$T_A(n) = \max \{ t_A(\bar{x}) \mid \bar{x} \text{ de tamaño } n \}$$

- Otra posibilidad es realizar un análisis de la eficiencia en el **caso promedio**. Para ello necesitamos conocer el tiempo de ejecución de cada posible ejemplar y la frecuencia con que se presenta, es decir, su distribución de probabilidades.

Definimos la complejidad de un algoritmo A en el caso promedio como

$$TM_A(n) = \sum_{\bar{x} \text{ de tamaño } n} p(\bar{x}) t_A(\bar{x})$$

siendo $p(\bar{x}) \in [0..1]$ la probabilidad de que la entrada sea $\bar{x}$

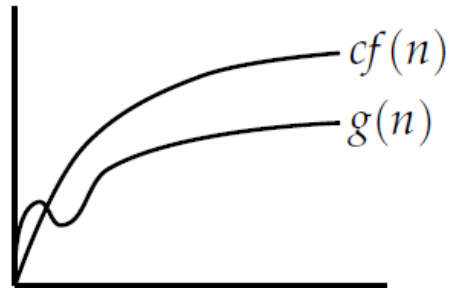
Medidas asintóticas

- El único factor del que van a depender las funciones que representan el coste de un algoritmo es el **tamaño de los datos de entrada**. Por eso, trabajamos con funciones $f : \mathbb{N} \longrightarrow \mathbb{R}_0^+$. **No nos importa tanto las funciones concretas, sino la forma en la que crecen.**
- Definición:** El conjunto de las funciones **del orden de $f(n)$** , denotado $O(f(n))$, se define como

$$O(f(n)) = \{g : \mathbb{N} \longrightarrow \mathbb{R}_0^+ \mid \exists c \in \mathbb{R}^+, n_0 \in \mathbb{N}. \forall n \geq n_0. g(n) \leq cf(n)\}$$

Asimismo, diremos que una función g es del orden de $f(n)$ cuando $g \in O(f(n))$.

- Decimos que el conjunto $O(f(n))$ define un **orden de complejidad**.



- Si el tiempo de un algoritmo está descrito por una función $T(n) \in O(f(n))$ diremos que el tiempo de ejecución del algoritmo es **del orden de $f(n)$** .

Propiedades

- $O(a \cdot f(n)) = O(f(n))$ con $a \in \mathbb{R}^+$
- La base del logaritmo no importa: $O(\log_a n) = O(\log_b n)$, con $a, b > 1$.

$$\log_b n = \frac{\log_a n}{\log_a b}$$

- Si $f \in O(g)$ y $g \in O(h)$, entonces $f \in O(h)$.
- Regla de la suma: $O(f + g) = O(\max(f, g))$.
- Regla del producto: Si $g_1 \in O(f_1)$ y $g_2 \in O(f_2)$, entonces $g_1 \cdot g_2 \in O(f_1 \cdot f_2)$.

Teorema del límite

- $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = m \in \mathbb{R}^+ \Rightarrow f \in O(g) \text{ y } g \in O(f) \Leftrightarrow O(f) = O(g)$
- $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0 \Rightarrow f \in O(g) \text{ y } g \notin O(f) \Leftrightarrow O(f) \subset O(g)$
- $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = +\infty \Rightarrow f \notin O(g) \text{ y } g \in O(f) \Leftrightarrow O(f) \supset O(g)$

Ejemplos

- $\log n \in O(n)$

$$\lim_{n \rightarrow \infty} \frac{n}{\log n} = \left[\frac{\infty}{\infty} \right] \stackrel{L'Hopital}{=} \lim_{n \rightarrow \infty} \frac{n \ln 2}{\ln n} = \lim_{n \rightarrow \infty} \frac{\ln 2}{1/n} = \lim_{n \rightarrow \infty} n \ln 2 = \infty$$

- $P(x) = a_k x^k + a_{k-1} x^{k-1} + \dots + a_1 x + a_0$, con $a_k \in \mathbb{R}^+$, $P(x) \in O(x^k)$

$$\lim_{n \rightarrow \infty} \frac{P(x)}{x^k} = a_k > 0$$

- $O(n^k) \subset O(2^n)$

$$\begin{aligned} \lim_{n \rightarrow \infty} \frac{2^n}{n^k} &= \left[\frac{\infty}{\infty} \right] \stackrel{L'Hopital}{=} \\ &= \lim_{n \rightarrow \infty} \frac{2^n \ln 2}{k n^{k-1}} = \lim_{n \rightarrow \infty} \frac{2^n (\ln 2)^2}{k(k-1) n^{k-2}} = (k \text{ veces}) = \\ &= \lim_{n \rightarrow \infty} \frac{2^n (\ln 2)^k}{k! n^0} = \frac{(\ln 2)^k}{k!} \lim_{n \rightarrow \infty} 2^n = \infty \end{aligned}$$

- $O(2^n) \subset O(n!)$

$$\lim_{n \rightarrow \infty} \frac{n!}{2^n} = \lim_{n \rightarrow \infty} \frac{n}{2} \frac{n-1}{2} \dots \frac{4}{2} \frac{3}{2} \frac{2}{2} \frac{1}{2} \geq \lim_{n \rightarrow \infty} \frac{4}{2} \frac{4}{2} \dots \frac{4}{2} \frac{3}{2} \frac{2}{2} \frac{1}{2} = \frac{3}{4} \lim_{n \rightarrow \infty} 2^{n-3} = \infty$$

Jerarquía de órdenes de complejidad

$$\underbrace{O(1) \subset O(\log n) \subset O(n) \subset O(n \log n) \subset O(n^2) \subset \dots \subset O(n^k) \subset \dots \subset O(2^n) \subset O(n!)}_{\substack{\text{razonables en la práctica} \\ \text{tratables}}} \quad \underbrace{\hspace{10em}}_{\text{intratables}}$$

La notación $O(f)$ nos da una *cota superior* del tiempo de ejecución $T(n)$ de un algoritmo.

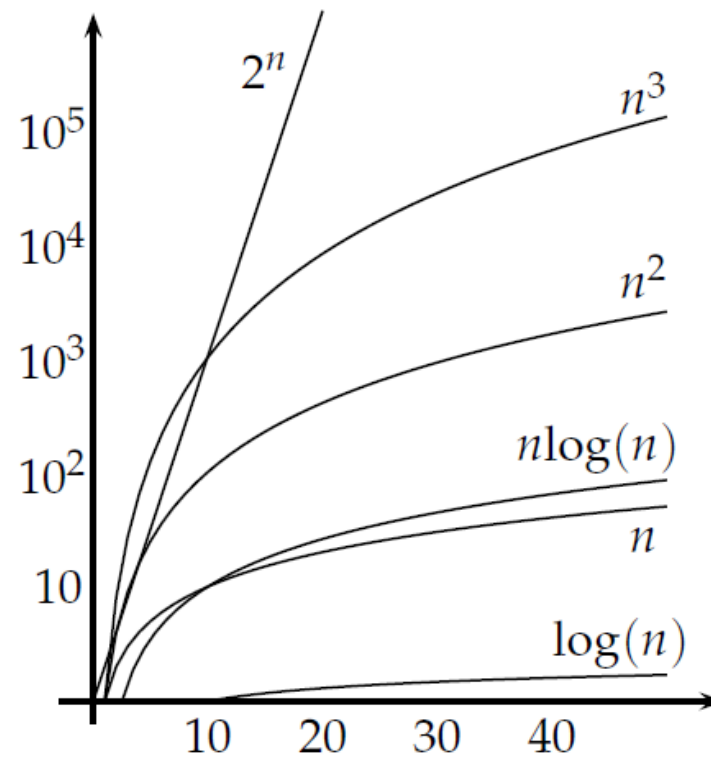
Normalmente estaremos interesados en la **menor** función $f(n)$ tal que

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

Ejemplos: cotas superiores hay muchas, pero algunas son poco informativas

$$\begin{aligned} 3n &\in O(n) \\ 3n &\in O(n^2) \\ 3n &\in O(2^n) \end{aligned}$$

Órdenes de complejidad



Órdenes de complejidad

Supongamos que tenemos 6 algoritmos diferentes tales que su menor cota superior está en $O(\log n)$, $O(n)$, $O(n \log n)$, $O(n^2)$, $O(n^3)$ y $O(2^n)$.

Supongamos que para un tamaño $n = 100$ todos tardan 1 hora en ejecutarse.

¿Qué ocurre si duplicamos el tamaño de los datos?

$T(n)$	$n = 100$	$n = 200$
$k_1 \cdot \log n$	1h.	1,15h.
$k_2 \cdot n$	1h.	2h.
$k_3 \cdot n \log n$	1h.	2,3h.
$k_4 \cdot n^2$	1h.	4h.
$k_5 \cdot n^3$	1h.	8h.
$k_6 \cdot 2^n$	1h.	$1,27 \cdot 10^{30}h.$

Órdenes de complejidad

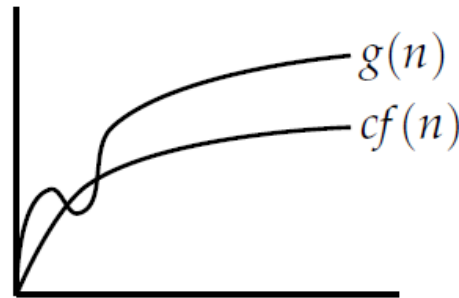
¿Qué ocurre si duplicamos la velocidad del computador? O lo que es lo mismo, ¿qué ocurre si duplicamos el tiempo disponible?

$T(n)$	$t = 1h.$	$t = 2h.$
$k_1 \cdot \log n$	$n = 100$	$n = 10000$
$k_2 \cdot n$	$n = 100$	$n = 200$
$k_3 \cdot n \log n$	$n = 100$	$n = 178$
$k_4 \cdot n^2$	$n = 100$	$n = 141$
$k_5 \cdot n^3$	$n = 100$	$n = 126$
$k_6 \cdot 2^n$	$n = 100$	$n = 101$

Cotas inferiores

Definición Sea $f : \mathbb{N} \longrightarrow \mathbb{R}_0^+$. El conjunto $\Omega(f(n))$, leído **omega de $f(n)$** , se define como

$$\Omega(f(n)) = \{g : \mathbb{N} \longrightarrow \mathbb{R}_0^+ \mid \exists c \in \mathbb{R}^+, n_0 \in \mathbb{N}. \forall n \geq n_0. g(n) \geq cf(n)\}$$



Cuando decimos que el coste de un algoritmo está en $\Omega(f(n))$ lo que estamos diciendo es que la complejidad del algoritmo *no es mejor* que la representada por la función f .

Principio de dualidad $g \in \Omega(f) \Leftrightarrow f \in O(g)$

Teorema del límite

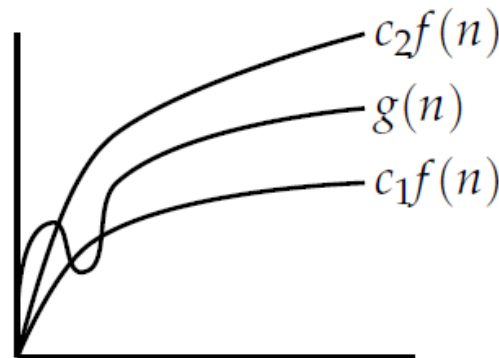
- $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = m \in \mathbb{R}^+ \Rightarrow g \in \Omega(f) \text{ y } f \in \Omega(g)$
- $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0 \Rightarrow g \in \Omega(f) \text{ y } f \notin \Omega(g)$
- $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = +\infty \Rightarrow g \notin \Omega(f) \text{ y } f \in \Omega(g)$

Orden exacto

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

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

$$\Theta(f(n)) = \{g : \mathbb{N} \longrightarrow \mathbb{R}_0^+ \mid \exists c_1, c_2 \in \mathbb{R}^+, n_0 \in \mathbb{N}. \forall n \geq n_0. c_1 f(n) \leq g(n) \leq c_2 f(n)\}$$



Teorema del límite

- $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = m \in \mathbb{R}^+ \Rightarrow g \in \Theta(f) \text{ y } f \in \Theta(g)$
- $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0 \Rightarrow f \in O(g) \text{ pero } f \notin \Theta(g)$
- $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = +\infty \Rightarrow g \in O(f) \text{ pero } g \notin \Theta(f)$

Ejemplo:

$P(x) = a_k x^k + a_{k-1} x^{k-1} + \dots + a_1 x + a_0$, con $a_k > 0$,

$Q(x) = b_l x^l + b_{l-1} x^{l-1} + \dots + b_1 x + b_0$, con $b_l > 0$, entonces

$$\lim_{n \rightarrow \infty} \frac{P(x)}{Q(x)} = 0 \text{ si } k < l \Rightarrow P(x) \in O(Q(x))$$

$$\lim_{n \rightarrow \infty} \frac{P(x)}{Q(x)} = \frac{a_k}{b_l} \text{ si } k = l \Rightarrow P(x) \in \Theta(Q(x))$$

$$\lim_{n \rightarrow \infty} \frac{P(x)}{Q(x)} = \infty \text{ si } k > l \Rightarrow P(x) \in \Omega(Q(x))$$

Análisis de algoritmos

Podemos simplificar el cálculo del coste (en el caso peor) de un algoritmo gracias a la teoría presentada.

- 1 Las instrucciones de asignación, entrada/salida, o expresiones aritméticas tienen un coste en $\Theta(1)$.
- 2 Para calcular el coste de una composición secuencial se utiliza la **regla de la suma**. Si el coste de S_1 está en $\Theta(f_1(n))$ y el coste de S_2 está en $\Theta(f_2(n))$, entonces el coste de $S_1 ; S_2$ está en $\Theta(\max(f_1(n), f_2(n)))$.
- 3 El coste de una instrucción condicional **si B entonces S_1 si no S_2 fsi** está en $\Theta(\max(f_B(n), f_1(n), f_2(n)))$.
- 4 Para calcular el coste de una instrucción iterativa

mientras B hacer S fmientras

utilizamos la **regla del producto**. Si el número de iteraciones está en $\Theta(f_{iter}(n))$, el coste total del bucle está en $\Theta(f_{B,S}(n) \cdot f_{iter}(n))$.

Si el coste de cada iteración es distinto, realizamos una suma desde 1 hasta $f_{iter}(n)$ de los costes individuales.

Ejemplos

Producto de matrices cuadradas.

```
fun producto( $A[1..n, 1..n], B[1..n, 1..n]$  de  $ent$ )  
    dev  $C[1..n, 1..n]$  de  $ent$   
var  $i, j, k : nat, s : ent$   
    para  $i = 1$  hasta  $n$  hacer  
        para  $j = 1$  hasta  $n$  hacer  
             $s := 0 ;$   
            para  $k = 1$  hasta  $n$  hacer  
                 $s := s + A[i, k] * B[k, j]$   
            fpara ;  
             $C[i, j] := s$   
        fpara  
    fpara  
ffun
```

$$T(n) \in \Theta(n^3)$$

Ejemplos

Ordenación por selección.

```
proc ord-selección( $V[1..n]$  de  $ent$ )  
  para  $i = 1$  hasta  $n - 1$  hacer  
     $pmin := i$  ;  
    para  $j = i + 1$  hasta  $n$  hacer  
      si  $V[j] < V[pmin]$  entonces  $pmin := j$  fsi  
    fpara ;  
    intercambiar( $V[i], V[pmin]$ )  
  fpara  
fproc
```

$$T(n) \in \Theta\left(\sum_{i=1}^{n-1} (n - i)\right) = \Theta(n^2)$$

Ejemplos

Determinar si una matriz cuadrada es simétrica.

```
fun simétrica?(V[1..n,1..n] de ent) dev b : bool
var i, j : nat
    b := cierto ; i := 1 ;
    mientras i ≤ n ∧ b hacer
        j := i + 1 ;
        mientras j ≤ n ∧ b hacer
            b := (V[i, j] = V[j, i]) ;
            j := j + 1
        fmientras ;
        i := i + 1
    fmientras
ffun
```

$$T_{\min}(n) \in \Theta(1) \qquad T_{\max}(n) \in \Theta(n^2)$$

Instrucción crítica

Se puede simplificar más el cálculo si hacemos uso del concepto de **instrucción crítica**: instrucción que más veces se ejecuta.

Calcular el número de veces que se ejecuta la instrucción crítica.

```
proc ord-selección( $V[1..n]$  de  $ent$ )  
  para  $i = 1$  hasta  $n - 1$  hacer  
     $pmin := i$ ;  
    para  $j = i + 1$  hasta  $n$  hacer  
      si  $V[j] < V[pmin]$  entonces  $pmin := j$  fsi  
    fpara ;  
    intercambiar( $V[i], V[pmin]$ )  
  fpara  
fproc
```

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

Ejemplo

```
proc ord-inserción( $V[1..n]$  de  $ent$ )  
  para  $i = 2$  hasta  $n$  hacer  
     $elem := V[i]$  ;  
     $j := i - 1$  ;  
    mientras  $j > 0 \wedge_c elem < V[j]$  hacer  
       $V[j + 1] := V[j]$  ;  
       $j := j - 1$   
    fmientras ;  
     $V[j + 1] := elem$   
fpara  
fproc
```

caso peor: $T_{\text{máx}}(n) = \sum_{i=2}^n i = \frac{(n+2)(n-1)}{2} \in \Theta(n^2)$

caso mejor: $T_{\text{mín}}(n) = \sum_{i=2}^n 1 = n - 1 \in \Theta(n)$

caso promedio: $\Theta(n^2)$

Limitaciones prácticas

Si tenemos dos algoritmos con costes $T_1(n) = 3n^3$ y $T_2(n) = 600n^2$,
¿cuál es mejor?

En principio es mejor el segundo, ya que $O(n^2) \subset O(n^3)$.

Pero esto es “para valores de n suficientemente grandes”.

Aquí $n \geq 200$.

Formulas útiles para el análisis de algoritmos

Sumatorios

$$\sum_{i=1}^n i = \frac{n(n+1)}{2} \approx \frac{1}{2}n^2$$

$$\sum_{i=1}^n i^2 = \frac{n(n+1)(2n+1)}{6} \approx \frac{1}{3}n^3$$

$$\sum_{i=1}^n i^k \approx \frac{1}{k+1}n^{k+1}$$

$$\sum_{i=0}^n a^i = \frac{a^{n+1} - 1}{a - 1} \quad \text{si } a \neq 1$$

$$\sum_{i=1}^n i2^i = (n-1)2^{n+1} + 2$$

$$\sum_{i=1}^n \log i \approx n \log n$$

Formulas útiles para el análisis de algoritmos

Propiedades de los logaritmos, $a, b > 1$

$$\log_a 1 = 0$$

$$\log_a a = 1$$

$$\log_a x^y = y \log_a x$$

$$\log_a xy = \log_a x + \log_a y$$

$$\log_a \frac{x}{y} = \log_a x - \log_a y$$

$$a^{\log_b x} = x^{\log_b a}$$

$$\log_a x = \frac{\log_b x}{\log_b a}$$

Ejercicio propuesto

- Calcular el coste (en el caso caso peor) de los algoritmos que aparecen en las páginas 8, 25, 26, 27. Realizarlo de la forma explicada en la página 7, y de la forma simplificada explicada en la página 24.

Bibliografía

- Peña, R.; Diseño de programas. *Formalismo y abstracción*. Tercera edición. Prentice Hall, 2005. [Capítulo 1](#)
- Martí, N., Palomino, M., Verdejo, J. A. *Introducción a la computación*. Colección Base Universitaria, Anaya, 2006. [Capítulo 4](#)
- Martí, N., Segura, C. M., Verdejo, J. A. *Especificación, derivación y análisis de algoritmos*. Colección Prentice Practica, Pearson, Prentice Hall, 2006. [Capítulo 3](#)

(Estas transparencias se han realizado a partir de aquéllas desarrolladas por los profesores Alberto Verdejo y Rafael del Vado de la UCM, y basadas en la bibliografía anterior)