



Departamento de Ingeniería Informática
Grado en Ingeniería Informática

Elisa Guerrero Vázquez

Esther L. Silva Ramírez

Metodología de la Programación

Tema 4 – Teoría

ANÁLISIS DE ALGORITMOS

1. Introducción

- Problema=Epecificación de la tarea a automatizar
- Algoritmo=Procedimiento de resolución de un problema que debe cumplir:
 - Acciones bien definidas.
 - Secuencia finita de operaciones en un orden definido.
 - Debe acabar en un tiempo finito.
- Programa=Algoritmo implementado en un lenguaje de programación.
- Pueden existir múltiples algoritmos que resuelvan un mismo problema. Por ejemplo: Búsqueda de un elemento en un vector ordenado.

1. Introducción

Problema: **Búsqueda de un elemento en un Vector Ordenado**


2 métodos distintos para resolver un mismo problema:

- Algoritmo 1: Búsqueda Secuencial
- Algoritmo 2: Búsqueda Dicotómica



Para resolver el problema ¿Cuál se elige?

Conceptos

- Determinar dominio de definición cuando se especifica un problema.
- Dominio D de definición de un problema – conjunto de ejemplares que deben considerarse para dicho problema. 
- Entrada – ejemplar o caso del problema.
- Cálculo – ejecución del algoritmo.
- Salida – solución del ejemplar.
- Tamaño n del ejemplar d – cualquier entero que mida de alguna manera el nº de componentes del ejemplar
 $n = \|d\| \geq 0$



2. Eficiencia de un algoritmo

- Vg. Dos algoritmos de ordenación (quicksort y burbuja) en dos ordenadores distintos.
- ¿Cómo seleccionar de entre todos los algoritmos de que disponemos para resolver un problema el mejor?
 - Si se tiene que resolver pocos casos pequeños seleccionar el más fácil de programar.
 - Si es un problema difícil y es necesario resolver muchos casos hay que llevar a cabo una selección más precisa.

2. Eficiencia de un algoritmo

¿Qué algoritmo es mejor?

- **Tiempo** que tarda en resolver el problema.
- Recursos que se necesitan para implementar el algoritmo: memoria principal y memoria secundaria.

Única dependencia importante en la mayoría de los casos para medir la eficiencia de un algoritmo es la entrada, pero a veces conviene medirlo en función del tamaño n de la entrada.

Casos a considerar:

- Mejor: menor tiempo de ejecución

$$t_{\min}(n) = \min\{t(d) \mid d \in D \wedge \|d\| = n\}$$

- **Peor : mayor tiempo de ejecución**

$$t_{\max}(n) = \max\{t(d) \mid d \in D \wedge \|d\| = n\}$$

- Promedio: Se debe conocer el tiempo de ejecución de cada caso y la frecuencia con que se presentan (distribución de probabilidades)

$$t_{\text{promedio}}(n) = \sum_{d \in D} Pr(d) \cdot t(d) = \sum_{k \geq 0} k \cdot Pr(t(d) = k)$$

$$t_{\text{mejor}}(n) \leq t_{\text{promedio}}(n) \leq t_{\text{peor}}(n), d \in D$$

2. Eficiencia de un algoritmo

Enfoques para seleccionar un algoritmo:

- **Enfoque empírico** (a posteriori).
- **Enfoque teórico** (a priori).
- **Enfoque híbrido** (mezcla de los anteriores).

Ventajas de la aproximación teórica:

- Independiente del ordenador, lenguaje de programación y habilidades del programador.
- Ahorro tiempo: no se implementa hasta el final. Con el empírico se deben implementar todos.
- Permite realizar un estudio para casos de todos los tamaños. Con el empírico sólo pueden probarse un nº pequeño de ejemplares.

3. Criterio Asintótico

A. **INDEPENDENCIA DEL HARDWARE**

Programa: implementación concreta de un determinado algoritmo, sujeta a una serie de limitaciones, tales como el tamaño de la memoria, de los datos que representa, ... etc.

B. **PRINCIPIO DE INVARIANZA**

2 implementaciones distintas de un mismo algoritmo no diferirán en su eficiencia en más de alguna constante multiplicativa.

3. Criterio Asintótico

INDEPENDENCIA DEL HARDWARE

Pueden existir múltiples algoritmos que resuelvan un mismo problema.

Programa: implementación concreta de un determinado algoritmo, sujeta a una serie de limitaciones, tales como el tamaño de la memoria, de los datos que representa, ... etc.

Única dependencia importante en la mayoría de los casos para medir la eficiencia de un algoritmo es la entrada, pero a veces conviene medirlo en función del tamaño n de la entrada.

Problema: **Búsqueda de un elemento en un Vector Ordenado**

- Algoritmo 1: Búsqueda Secuencial
 - Versión 1: usando menor número de variables
 - Versión 2: usando variable Encontrado

- Algoritmo 2: Búsqueda Dicotómica
 - Implementación Dicotómica Iterativa
 - Implementación Dicotómica Recursiva

3. Criterio Asintótico

PRINCIPIO DE INVARIANZA

2 implementaciones distintas de un mismo algoritmo no diferirán en su eficiencia en más de alguna constante multiplicativa.

- Búsqueda Secuencial A1:

$$t_{A1}(n) = 2n \text{ Unidades de tiempo}$$

- Búsqueda Secuencial A2:

$$t_{A2}(n) = 4n \text{ Unidades de tiempo}$$

Por tanto: $t_{A1}(n) \cong m * t_{A2}(n)$

NO OLVIDAR LAS CONSTANTES OCULTAS – tamaño n grande

4. ¿Por qué hay que buscar la eficiencia?

Algoritmo con tiempo $t_1(n) = 10^{-4} \cdot 2^n$

- Invertir en maquinaria (máquina 100 veces más rápida):

$$t'_1(n) = 10^{-6} \cdot 2^n$$

Calcular para tamaños $n=10$ y $n=20$.

¿Tamaño se conseguiría resolver en un año?



- Invertir en algoritmia:

$$t_2(n) = 10^{-2} \cdot n^3$$

Calcular para tamaños $n=10$ y $n=20$.

¿Tamaño se conseguiría resolver en un año?

- Invertir en ambos:

$$t'_2(n) = 10^{-4} \cdot n^3$$

Calcular para tamaños $n=10$ y $n=20$.

¿Tamaño se conseguiría resolver en un año?

5. Análisis de la Complejidad

Búsqueda Secuencial

- 1) Empieza en la primera posición del vector
- 2) **Mientras**
 - no se sobrepase la longitud del vector
 - los elementos del vector sean menores que el buscado
 - Avanzar una posición más en el vector
- 3) **Si** el elemento de la posición actual coincide con el elemento buscado,
 - Entonces** devolver dicha posición
 - Si no** devolver longitud+1

5. Análisis de la Complejidad

const

N=10; // por ejemplo

tipo

vector[N] de entero: Vec

entero **funcion** busqueda(E entero: elem, E Vec: v, E entero: n)

var

entero: i

inicio

(1) $i \leftarrow 1$

(2) **mientras** $(i \leq n) \wedge (elem \neq v[i])$

(3) $i \leftarrow i + 1$

fin_mientras

(4) **devolver** i

fin_funcion

5. Análisis de la Complejidad

¿Cómo medir el tiempo $t(n)$ en un algoritmo?

1. Contar cuántas instrucciones de cada tipo
2. Multiplicar por el tiempo que tarda en ejecutarse
3. Realizar la suma

■ **Por ejemplo:**

- t_a : tiempo en ejecutar una asignación
- t_c : tiempo en ejecutar una comparación
- t_o : tiempo en ejecutar una operación aritmético-lógica
- t_v : tiempo de acceso a un vector
- t_d : tiempo en ejecutar la instrucción **devolver**

5. Análisis de la Complejidad

const

N=10; // por ejemplo

tipo

vector[N] de entero: Vec

entero **funcion** busqueda(E entero: elem, E Vec: v, E entero: n)

var

Entero: i

inicio

(1) $i \leftarrow 1$ t_a

(2) **mientras** $(i \leq n) \wedge (elem \neq v[i])$ $t_c + t_o + t_v + t_c$

(3) $i \leftarrow i+1$ $t_a + t_o$

fin_mientras

(4) **devolver** i t_d

fin_funcion

} **Caso Peor:** se repite n veces
 } **Caso Mejor:** sólo la comparación
 del bucle pero no itera

5. Análisis de la Complejidad

- $t_{\min}(n) = t_a + 2t_c + t_o + t_v + t_d$
- $t_{\max}(n) = t_a + (2t_c + t_o + t_v) * (n+1) + n * (t_a + t_o) + t_d$

Si consideramos que cada operación elemental
 $t_x = \text{Coste Unitario}$ $t_x \in \Theta(1)$

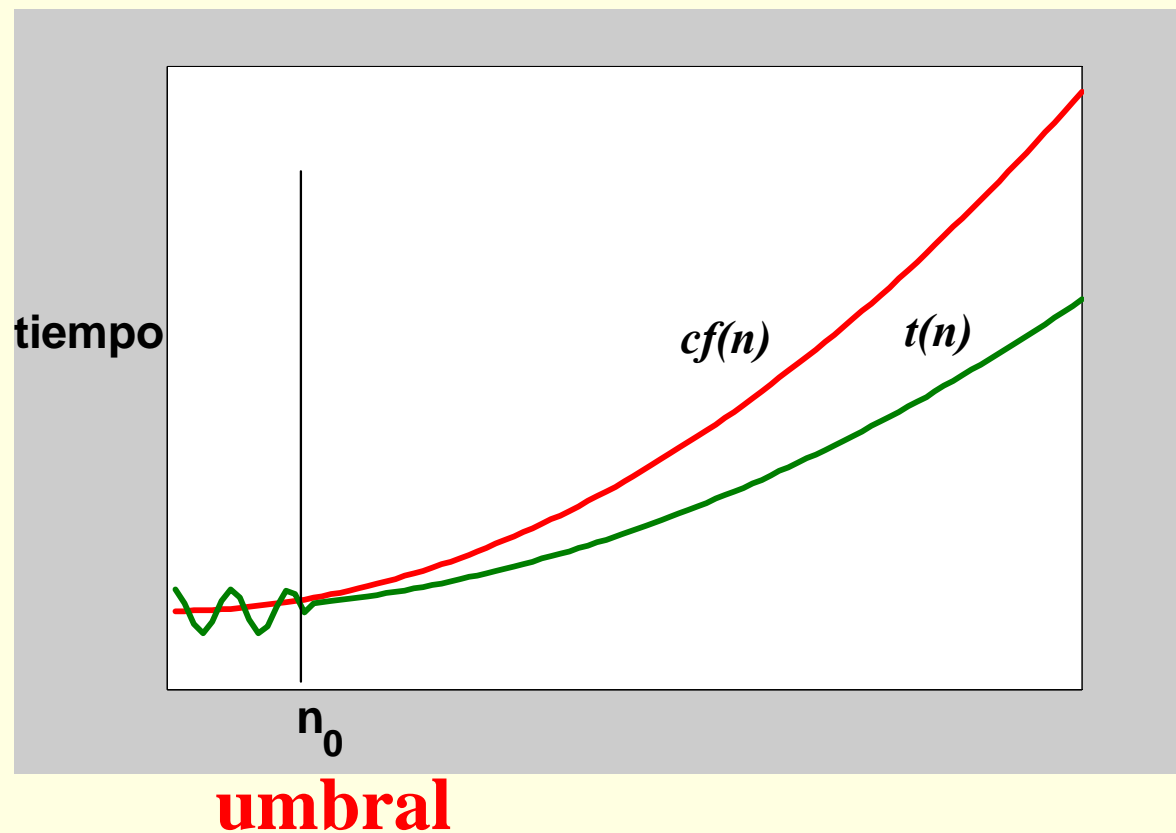
Caso Peor: $t_{\max}(n) = 1 + (n+1) * 4 + 2 * n + 1 = 6 * n + 6$

6. Estimación Objetiva

- Se denomina **Notación Asintótica** porque trata acerca del comportamiento de funciones en el límite, es decir, para valores suficientemente grandes de su parámetro.
- En Análisis de Algoritmos se estudia cómo crece el tiempo de proceso a medida que crece n .

6. Orden de Complejidad: Cota Superior

$$O(f(n)) = \left\{ t : \mathbb{N} \rightarrow \mathbb{R}^{\geq 0} \mid \exists c \in \mathbb{R}^+, \exists n_0 \in \mathbb{N} \mid \forall n \geq n_0 : t(n) \leq c \cdot f(n) \right\}$$



6.- Orden de Complejidad: Cota Superior

$$O(f(n)) = \left\{ t : \mathbb{N} \rightarrow \mathbb{R}^{\geq 0} \mid \exists c \in \mathbb{R}^+, \exists n_0 \in \mathbb{N} \mid \forall n \geq n_0 : t(n) \leq c \cdot f(n) \right\}$$



Una función $t(n)$ está en el orden de $f(n)$ sii $t(n) \in O(f(n))$

Significa que $t(n)$ está acotada superiormente por $f(n)$ para valores de n suficientemente grandes y haciendo abstracción de posibles constantes multiplicativas

6. Orden de Complejidad: Cota Superior

$$O(f(n)) = \left\{ t : \mathbb{N} \rightarrow \mathbb{R}^{\geq 0} \mid \exists c \in \mathbb{R}^+, \exists n_0 \in \mathbb{N} \mid \forall n \geq n_0 : t(n) \leq c \cdot f(n) \right\}$$

Ejemplo:

¿ $t(n)=n+1$ pertenece a $O(n)$?

Demostración:

Se cumple si existe n_0 y c tal que:

$$t(n) \leq cf(n)$$

$$n+1 \leq cn \quad \text{por ejemplo para } n_0=1 \text{ y } c=2$$

$$\text{Por tanto: } \forall n \geq 1 \quad n+1 \leq 2n$$

6. Orden de Complejidad: Cota Superior

$$O(f(n)) = \left\{ t : \mathbb{N} \rightarrow \mathbb{R}^{\geq 0} \mid \exists c \in \mathbb{R}^+, \exists n_0 \in \mathbb{N} \mid \forall n \geq n_0 : t(n) \leq c \cdot f(n) \right\}$$

Ejemplo: ¿A qué orden pertenece la función de búsqueda Secuencial?

¿ $t(n)=6n+6$ pertenece a $O(n)$?

Demostración:

Se cumple si existe n_0 y c tal que:

$$t(n) \leq cf(n)$$

$$6n+6 \leq cn \quad \text{por ejemplo para } c=7 \text{ y } n_0=6$$

$$\text{ó } n_0=1 \text{ y } c=12$$



Por tanto: $\forall n \geq 2 \quad 6n+6 \leq 7n$

6. Orden de Complejidad

$$O(f(n)) = \left\{ t : \mathbb{N} \rightarrow \mathbb{R}^{\geq 0} \mid \exists c \in \mathbb{R}^+, \exists n_0 \in \mathbb{N} \mid \forall n \geq n_0 : t(n) \leq c \cdot f(n) \right\}$$

Demostrar la veracidad o falsedad de las siguientes afirmaciones:

- $t(n)=10n^2+9n+1 \in O(n^2)$
- $t(n)=10n^2+9n+1 \in O(n)$
- $t(n)=6 \cdot 2^n + n^2 \in O(2^n)$
- $t(n) = 3 \in O(1)$

6. Pertenencia y contención para O

$$f \in O(g) \Leftrightarrow O(f) \subseteq O(g)$$

$$f \in O(g) \wedge g \notin O(f) \Leftrightarrow O(f) \subset O(g)$$

$$f \in O(g) \wedge g \in O(f) \Leftrightarrow O(f) = O(g)$$

6. Regla del Límite para O

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = k \in \mathbb{R}^{\geq 0}$$

a) Si $k = 0$ entonces $f \in O(g)$ pero $g \notin O(f)$, $O(f) \subset O(g)$

b) Si $k = +\infty$ entonces $f \notin O(g)$ pero $g \in O(f)$, $O(g) \subset O(f)$

c) Si $k \neq 0$ y $k < \infty$ entonces $O(f) = O(g)$

Esta Regla es muy importante para poder demostrar las pertenencias de cada función a los órdenes de magnitud

6. Regla del Límite

$t(n)=6n+6$ pertenece a $O(n)$

Demostración:

$$\lim_{n \rightarrow \infty} \frac{6n + 6}{n} = \lim_{n \rightarrow \infty} \left(6 + \frac{6}{n} \right) = 6$$

Por lo que se cumple $6n + 6 \in O(n)$

6. Regla del Límite

$$n^2 \in O(n^3)$$

$$\text{Sí, porque } \lim_{n \rightarrow \infty} \frac{n^2}{n^3} = \lim_{n \rightarrow \infty} \frac{1}{n} = 0$$

$$\text{y además } n^3 \notin O(n^2) \quad \lim_{n \rightarrow \infty} \frac{n^3}{n^2} = \infty$$

6. Operaciones entre Órdenes

■ Regla de la Suma:

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

$$\text{Si } f_1 \in O(g) \wedge f_2 \in O(h) \Rightarrow f_1 + f_2 \in O(\max(g,h))$$

■ Regla del Producto:

$$O(f) \cdot O(g) = O(f \cdot g)$$

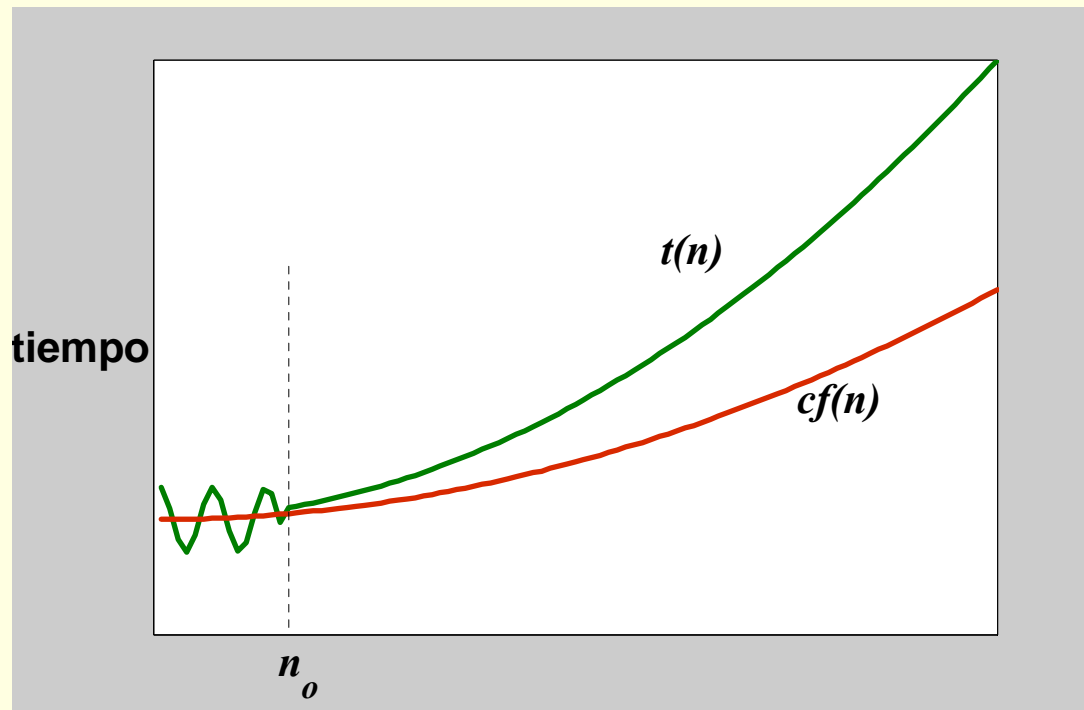
$$\text{Si } f_1 \in O(g) \wedge f_2 \in O(h) \Rightarrow f_1 \cdot f_2 \in O(g \cdot h)$$

■ Propiedad Transitiva

$$\text{Si } f \in O(g) \wedge g \in O(h) \Rightarrow f \in O(h)$$

6. Omega: Cota inferior

$$\Omega(f(n)) = \left\{ t : \mathbb{N} \rightarrow \mathbb{R}^{\geq 0} \mid \exists c \in \mathbb{R}^+, \exists n_0 \in \mathbb{N} \mid \forall n \geq n_0 : t(n) \geq c \cdot f(n) \right\}$$



umbral

6. Omega: Cota inferior

$$\Omega(f(n)) = \left\{ t : \mathbb{N} \rightarrow \mathbb{R}^{\geq 0} \mid \exists c \in \mathbb{R}^+, \exists n_0 \in \mathbb{N} \mid \forall n \geq n_0 : t(n) \geq c \cdot f(n) \right\}$$

Una función $t(n)$ está en omega de $f(n)$ sii $t(n) \in \Omega(f(n))$

Significa que $t(n)$ está acotada inferiormente por $f(n)$ para valores de n suficientemente grandes y haciendo abstracción de posibles constantes multiplicativas

6. Omega: Cota Inferior

$$\Omega(f(n)) = \left\{ t : \mathbb{N} \rightarrow \mathbb{R}^{\geq 0} \mid \exists c \in \mathbb{R}^+, \exists n_0 \in \mathbb{N} \mid \forall n \geq n_0 : t(n) \geq c \cdot f(n) \right\}$$

Asintóticamente podemos acotar Inferiormente $t(n)$ con una función proporcional a $f(n)$: $\forall n \geq n_0 \ t(n) \geq cf(n)$

Ejemplo:

$$\text{¿ } t(n)=10n^2+9n+1 \in \Omega(n^2) \text{ ?}$$

Demostración:

Se cumple si existe n_0 y c tal que:

$$t(n) \geq cf(n)$$

$$10n^2+9n+1 \geq cn^2 \quad \text{por ejemplo para } n_0=1 \text{ y } c=10$$

$$\text{Por tanto: } \forall n \geq 1 \quad 10n^2+9n+1 \geq 10n^2$$

6. Omega: Cota Inferior

- **Dualidad:**

$$g \in \Omega(f) \Leftrightarrow f \in O(g)$$

Permite traspasar las propiedades de O a Ω y viceversa.

- **Relación entre O y Ω**

$$O(f) = O(g) \Leftrightarrow \Omega(f) = \Omega(g)$$

$$3n^3 + 7n^2 \in \Omega(n)$$

- Por la regla del máximo: $\Omega(f+g) = \Omega(\max(f,g))$

$$\Omega(3n^3 + 7n^2) = \Omega(\max(3n^3, 7n^2)) = \Omega(3n^3)$$

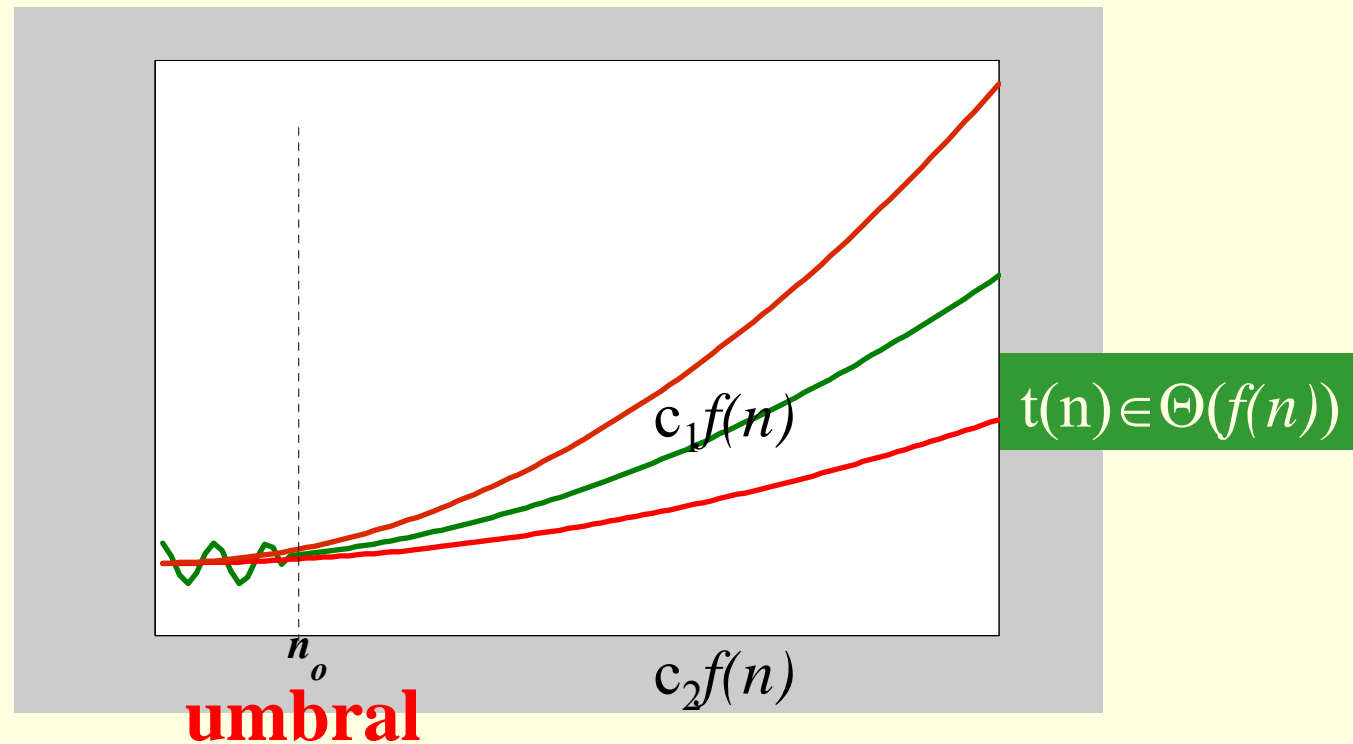
- Por tanto hay que demostrar que: $3n^3 \in \Omega(n)$
- Se cumple si existe n_0 y c tal que: $3n^3 \geq cn$

Y en efecto, para $n_0=1$ y $c=2$ se verifica que:

$$3n^3 \geq 2n$$

6. Zeta: Orden Exacto

$$\Theta(f(n)) = \left\{ t : \mathbb{N} \rightarrow \mathbb{R}^{\geq 0} \mid \exists c, d \in \mathbb{R}^+, \exists n_0 \in \mathbb{N} \mid \forall n \geq n_0 : \right. \\ \left. d \cdot f(n) \leq t(n) \leq c \cdot f(n) \right\}$$



6. Zeta: Exacto

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

Ejemplo:

$$\text{¿ } t(n)=3n+1 \in \Theta(n) \text{ ?}$$

Demostración

$3n+1 \in O(n)$ Se cumple si existe n_0 y c_1 tal que:

$$t(n) \leq c_1 f(n) \quad \forall n \geq n_0$$

➡ Para $n_0=1$ y $c_1=4$ se cumple que: $3n+1 \leq 4n$

$3n+1 \in \Omega(n)$ Se cumple si existe n_0 y c_2 tal que:

$$t(n) \geq c_2 f(n) \quad \forall n \geq n_0$$

➡ Para $n_0=1$ y $c_2=2$ se cumple que: $3n+1 \geq 2n$

Por tanto: $\forall n \geq 1 \quad 3n+1 \in \Theta(n)$

6. Zeta: Exacto

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

Considerando de nuevo la búsqueda secuencial:

$$¿ t(n)=6n + 6 \in \Theta(n) ?$$

Demostración

$6n+6 \in O(n)$ Se cumple si existe n_0 y c_1 tal que:

$$t(n) \leq c_1 f(n) \quad \forall n \geq n_0$$

➡ Para $n_0=1$ y $c_1=12$ se cumple que: $6n+6 \leq 12n \quad \forall n \geq n_0$

$6n+6 \in \Omega(n)$ Se cumple si existe n_0 y c_2 tal que:

$$t(n) \geq c_2 f(n) \quad \forall n \geq n_0$$

➡ Para $n_0=1$ y $c_2=4$ se cumple que: $6n+6 \geq 4n \quad \forall n \geq n_0$

Por tanto: $\forall n \geq 1 \quad 6n+6 \in \Theta(n)$

6. Regla del Límite para el Orden Exacto

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = k \in \mathbb{R}^{\geq 0}$$

a) Si $k = 0$ entonces $f \in O(g)$ pero $f \notin \Theta(g)$

b) Si $k = +\infty$ entonces $f \in \Omega(g)$ pero $f \notin \Theta(g)$

c) Si $k \neq 0$ y $k < \infty$ entonces $f \in \Theta(g)$

6. Relación entre O , Ω y Θ

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

$$O(f) = O(g) \Leftrightarrow \Theta(f) = \Theta(g)$$

$$\Omega(f) = \Omega(g) \Leftrightarrow \Theta(f) = \Theta(g)$$

6. Propiedades de los Órdenes

Dado $\Xi \in \{O, \Omega, \Theta\}$ y $f, g : \mathbb{N} \rightarrow \mathbb{R}^{\geq 0}$ se definen las propiedades:

* Propiedad reflexiva. Para cualquier función f se cumple $f \in \Xi(f)$

* Invariancia aditiva. Para toda constante $c \in \mathbb{R}^+$, $g \in \Xi(f) \Leftrightarrow c + g \in \Xi(f)$

* Invariancia multiplicativa. Para toda constante $c \in \mathbb{R}^+$, $g \in \Xi(f) \Leftrightarrow c \cdot g \in \Xi(f)$

* Regla de la Suma.

Si $f_1 \in \Xi(g) \wedge f_2 \in \Xi(h) \Rightarrow f + g \in \Xi(g + h) \Rightarrow f + g \in \Xi(\max(g, h))$

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

* Regla del Producto. Si $f_1 \in \Xi(g) \wedge f_2 \in \Xi(h) \Rightarrow f \cdot g \in \Xi(g \cdot h)$

$\Xi(f) \cdot \Xi(g) = \Xi(f \cdot g)$

* Propiedad Transitiva. Si $f \in \Xi(g) \wedge g \in \Xi(h) \Rightarrow f \in \Xi(h)$

6. Regla del Límite para O , Ω , Θ

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = k \in \mathbb{R}^{\geq 0}$$

- a) Si $k = 0$ entonces $f \in O(g)$ $g \notin O(f)$ $g \notin \Theta(f)$ $g \in \Omega(f)$ $O(f) \subset O(g)$
- b) Si $k = +\infty$ entonces $f \notin O(g)$ $f \notin \Theta(g)$ $g \in O(f)$ $f \in \Omega(g)$ $O(g) \subset O(f)$
- c) Si $k \neq 0$ y $k < \infty$ entonces $O(f) = O(g)$ $f \in \Theta(g)$ $\Omega(f) = \Omega(g)$ $\Theta(g) = \Theta(f)$

$$f \in O(g) \Leftrightarrow g \in \Omega(f)$$

$$O(f) = O(g) \Leftrightarrow \Omega(f) = \Omega(g)$$

$$f \in \Theta(g) \Leftrightarrow g \in \Theta(f)$$

Ejemplo:

$$2^n \in O(50n^2)$$

$$\lim_{n \rightarrow \infty} \frac{2^n}{50n^2} = +\infty \quad \text{por lo que } 2^n \notin O(50n^2)$$

$$\text{y además } 50n^2 \in O(2^n)$$

$$3n^3 * 7n^2 \in \Theta(n^5)$$

- Por la regla del Producto: $f * g \in \Theta(f * g)$

$$\Theta(3n^3 * 7n^2) = \Theta(21 * n^5)$$

Por tanto, se verifica que pertenece al Orden Exacto de n^5

- Aplicando la definición: $21 * n^5 \in \Theta(n^5)$

- Por la Regla del Límite $\lim_{n \rightarrow \infty} \frac{21 * n^5}{n^5} = 21$

Notación asintótica

- *Está en el orden de (Cota Superior):*

$$O(f(n)) = \left\{ t : \mathbb{N} \rightarrow \mathbb{R}^{\geq 0} \mid \exists c \in \mathbb{R}^+, \exists n_0 \in \mathbb{N} \mid \forall n \geq n_0 : t(n) \leq c \cdot f(n) \right\}$$

- *Está en el orden de (Cota Superior):*

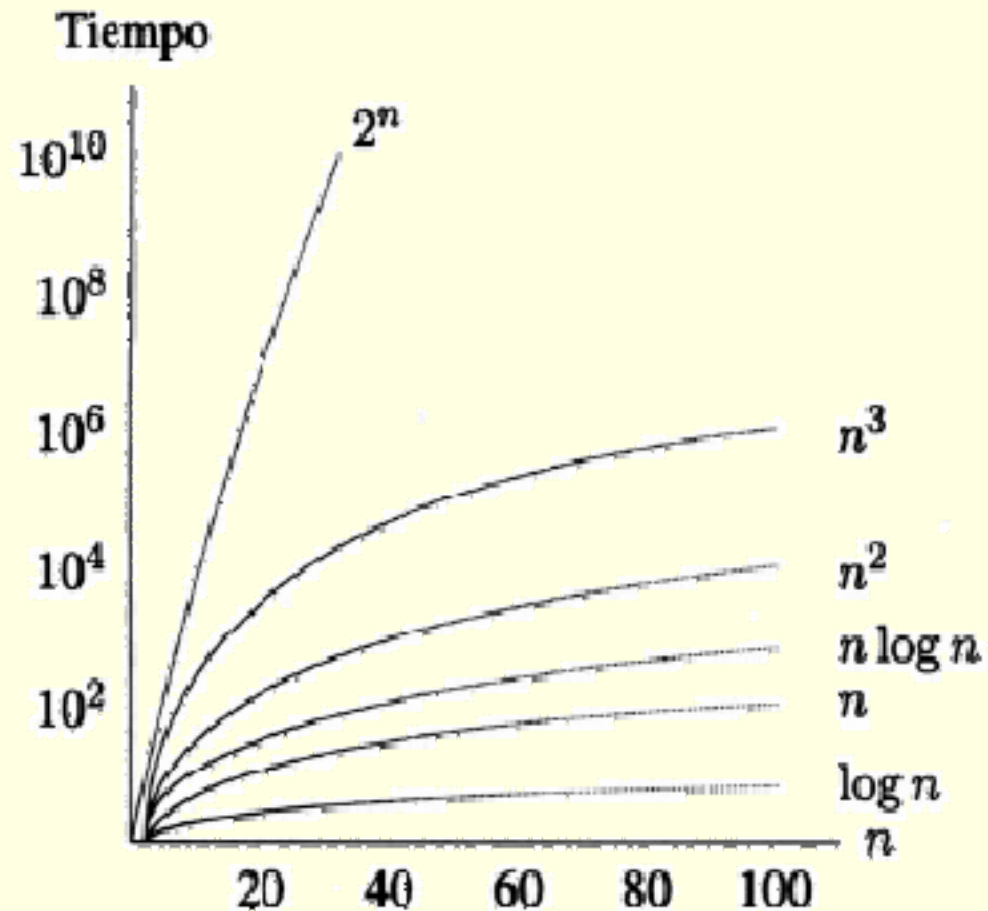
$$\Omega(f(n)) = \left\{ t : \mathbb{N} \rightarrow \mathbb{R}^{\geq 0} \mid \exists c \in \mathbb{R}^+, \exists n_0 \in \mathbb{N} \mid \forall n \geq n_0 : t(n) \geq c \cdot f(n) \right\}$$

- *Está en Theta de (Orden de magnitud exacto)*

$$\Theta(f(n)) = \left\{ t : \mathbb{N} \rightarrow \mathbb{R}^{\geq 0} \mid \exists c, d \in \mathbb{R}^+, \exists n_0 \in \mathbb{N} \mid \forall n \geq n_0 : \right. \\ \left. d \cdot f(n) \leq t(n) \leq c \cdot f(n) \right\}$$

7. Jerarquía de Órdenes

$O(1) \subset O(\log n) \subset O(\sqrt{n}) \subset O(n) \subset O(n \cdot \log n) \subset O(n^2) \subset \dots \subset$
 $O(n^k) \subset O(2^n) \subset O(n!) \subset O(n^n)$



7. Jerarquía de Órdenes

$$O(1) \subset O(\log n) \subset O(\sqrt[n]{n}) \subset O(n) \subset O(n \cdot \log n) \subset O(n^2) \subset \dots \subset O(n^k) \subset O(2n) \subset O(n!) \subset O(n^n)$$

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

Efecto de duplicar el tamaño del problema

7. Jerarquía de Órdenes

$$O(1) \subset O(\log n) \subset O(\sqrt[n]{n}) \subset O(n) \subset O(n \cdot \log n) \subset O(n^2) \subset \dots \subset O(n^k) \subset O(2n) \subset O(n!) \subset O(n^n)$$

$t(n)$	Tiempo=1h.	Tiempo=2h.
$k_1 \log n$	$n=100$	$n=10.000$
$k_2 n$	$n=100$	$n=200$
$k_3 n \log n$	$n=100$	$n=178$
$k_4 n^2$	$n=100$	$n=141$
$k_5 n^3$	$n=100$	$n=126$
$k_6 2^n$	$n=100$	$n=101$

Efecto de duplicar el tiempo disponible

8. Reglas prácticas - cálculo de eficiencia

1. Instrucciones elementales $\Theta(1)$

2. Composición:

s_1 $t_{s1}(n)$

s_2 $t_{s2}(n)$

$$t(n) = t_{s1}(n) + t_{s2}(n)$$

3. Condicional simple:

si B entonces $t_B(n)$

s_1 $t_s(n)$

$$t(n) = t_B(n) + t_s(n)$$

4. Condicional:

si B entonces $t_B(n)$

s_1 $t_{s1}(n)$

si_no

s_2 $t_{s2}(n)$

$$t(n) = t_B(n) + \max\{t_{s1}(n), t_{s2}(n)\}$$

8. Reglas prácticas - cálculo de eficiencia

5. Estructura repetitiva:

Sea i el número de iteración

Sea $p(n)$ el número total de iteraciones

mientras B hacer $t_B(n)$ no varía en cada iteración

S $t_s(n)$ no varía en cada iteración

$$t(n) = t_B(n) \cdot (p(n) + 1) + t_s(n) \cdot p(n)$$

mientras B hacer $t_B(n)$ no varía en cada iteración

S $t_s(i,n)$ varía en cada iteración

$$t(n) = t_B(n) \cdot (p(n) + 1) + \sum_{i=1}^{p(n)} t_s(i,n)$$

8. Reglas prácticas - cálculo de eficiencia

5. Estructura repetitiva:

Sea i el número de iteración

Sea $p(n)$ el número total de iteraciones

mientras B hacer $t_B(i,n)$ varía en cada iteración
 $t_s(i,n)$ varía en cada iteración

$$t(n) = \sum_{i=1}^{p(n)+1} t_B(i,n) + \sum_{i=1}^{p(n)} t_s(i,n)$$

6. Si se producen llamadas a un subalgoritmo se calcula el tiempo que tarda en ejecutarse éste.

El coste total del algoritmo es la suma de todos los costes de las instrucciones que lo forman.

Notación asintótica. Operación crítica

- Una **operación** es **elemental** si se ejecuta en un tiempo acotado por una constante.
- Una **operación** es **crítica** si no hay ninguna que se ejecute más que ella. Se ejecuta al menos tantas veces como cualquier otra.

Dado el siguiente procedimiento, calcule su orden de complejidad:

procedimiento proc(E entero: n)

var

entero: z, d, x

inicio

z \leftarrow 0

d \leftarrow 1

mientras d \leq n **hacer**

x \leftarrow d

mientras x \geq 0 **hacer**

z \leftarrow z + x

x \leftarrow x - 1

fin_mientras

d \leftarrow d + 1

fin_mientras

fin_procedimiento

entero **función** busq_binaria_rec(E/S Vect: v, E entero: n, E entero: elem, E entero: izqda, E entero: drcha)

var

entero: central, pos **inicio**

inicio

izqda \leftarrow 1

drcha \leftarrow n

central \leftarrow (izqda + drcha) **div** 2

mientras v[central] \neq elem \wedge izqda < drcha **hacer**

si elem > v[central] **entonces**

 izqda \leftarrow central+1

si_no

 drcha \leftarrow central -1

fin_si

 central \leftarrow (izqda + drcha) **div** 2

fin mientras

si v[central]=elem **entonces** pos=central

si_no pos=n+1

devolver pos

fin_función

Búsqueda Dicotómica

entero **función** busq_binaria_rec(E/S Vect: v, E entero: n, E entero: elem, E entero: izqda, E entero: drcha)

var

entero: central, pos **inicio**

inicio

izqda \leftarrow 1

drcha \leftarrow n

central \leftarrow (izqda + drcha) **div** 2

mientras v[central] \neq elem \wedge izqda < drcha **hacer**

si elem > v[central] **entonces**

 izqda \leftarrow central+1

si_no

 drcha \leftarrow central -1

fin_si

 central \leftarrow (izqda + drcha) **div** 2

fin mientras

si v[central]=elem

entonces pos \leftarrow central

si_no pos \leftarrow n+1 **fin_si**

devolver pos

fin_función

Búsqueda Dicotómica

Caso Mejor: el elemento buscado es el central

Caso Peor: buscar un elemento que no está en el vector

Búsqueda Dicotómica

- Peor Caso: buscar un elemento que no se encuentra en el vector.
- Cada iteración del bucle reduce el vector a la mitad:
 - Inicialmente: n
 - Tras la 1ª iteración: $\frac{n}{2}$
 - Tras la 2ª iteración: $\frac{n}{4}$
 - Tras la 3ª iteración: $\frac{n}{8}$

Búsqueda Dicotómica

- Peor Caso: buscar un elemento que no se encuentra en el vector.
- Cada iteración del bucle reduce el vector a la mitad:
 - Inicialmente: n
 - Tras la 1ª iteración: $\frac{n}{2} = \frac{n}{2^1}$
 - Tras la 2ª iteración: $\frac{n}{4} = \frac{n}{2^2}$
 - Tras la 3ª iteración: $\frac{n}{8} = \frac{n}{2^3}$
 - De forma general: $\frac{n}{2^{nveces}}$

Búsqueda Dicotómica

- Última iteración cuando sólo quede un elemento en el vector por tanto:

Cuando $\frac{n}{2^{nveces}} = 1$

$$\frac{n}{2^{nveces}} = 1 \quad \text{aplicando logaritmos}$$

$$\log_2 \frac{n}{2^{nveces}} = \log_2 1 \quad \text{y como } \log_2 1 = 0 \quad \text{entonces}$$

$$\log_2 \frac{n}{2^{nveces}} = \log_2 n - \log_2 2^{nveces}$$

$$\log_2 n = \log_2 2^{nveces}$$

$$\boxed{\log_2 n = nveces}$$

El número de iteraciones está acotado por el logaritmo en base 2 de n

Búsqueda Dicotómica

Todas las operaciones elementales son de $O(1)$, por tanto:

Peor Caso:

$$t(n) = 5 + 4 + \sum_{\alpha=1}^{nveces} (4 + 7) + 5$$

$$\text{nveces} = \log_2 n$$

$$t(n) = 5 + 4 + \sum_{\alpha=1}^{\log_2 n} (4 + 7) + 5 = 14 + 11 \cdot \log_2 n$$

Búsqueda Dicotómica $\in \Theta(\log n)$

Aplicando conjuntamente las propiedades:

Invariancia multiplicativa: $c \in \mathbb{R}^+, g \in \Theta(f) \Leftrightarrow c \cdot g \in \Theta(f)$

Y regla de la Suma: $\Theta(f + g) = \Theta(\max(f, g))$

Peor Caso: $t(n) = 14 + 11 \cdot \log_2 n$

Por tanto, $t(n) \in \Theta(14 + 11\log_2 n) = \Theta(11\log_2 n) = \Theta(\log_2 n)$

Búsqueda Dicotómica – Operación crítica

Este cálculo del coste total del algoritmo se puede simplificar aún más, teniendo en cuenta la *operación crítica*.

Como operación crítica seleccionamos la comparación entre elementos:

$$v[\text{central}] \neq \text{elem}$$

Esta operación se repite con tanta frecuencia como cualquier otra y refleja la esencia del algoritmo, ya que comprueba si el elemento buscado es igual a un determinado elemento del vector. Se podría considerar cualquier otra operación, por ejemplo suma o decremento, siempre y cuando el número de veces que se ejecute sea el mismo orden.

Búsqueda Dicotómica $\in \Theta(\log n)$

Así, se puede considerar el coste en el peor caso como el número de veces que se repite esta operación.

Peor Caso: $t(n) = \log_2 n$

Por lo que calculando dicho número de veces, se obtiene directamente el orden de complejidad.

Por tanto, $t(n) \in \Theta(\log_2 n)$

Ejercicios



entero **función** func(E entero: n)

var

entero: prod

inicio

prod \leftarrow 1

mientras n>0 **hacer**

prod \leftarrow prod*n

n \leftarrow n-1

fin_mientras

devolver prod

fin_función

Calcular el orden de complejidad del procedimiento en función del valor de n.

Ejercicios

entero **funcion** Ejemplo(E entero: n)

var

entero: a, b, i, t

inicio

a ← 1

b ← 1

desde i ← 0 **hasta** n-1 **hacer**

t ← b

b ← suma(i)

a ← t

fin_desde

devolver a

fin_función

Calcular el orden de complejidad del algoritmo en función del valor de n:

a) Suponiendo que el tiempo de la función *suma* es constante.

b) Suponiendo que el tiempo de la función *suma* estará en $\Theta(n)$.

Ejercicios

```
entero funcion suma-cuadrados(E entero: n)
var
inicio
    s ← 0
    desde i ← 1 hasta n hacer
        p ← producto(i,i)
        s ← s+p
    fin_desde
    devolver s
fin_función
```

Calcular el orden de complejidad del algoritmo en función del valor de n:

- Suponiendo que el tiempo de la función *producto* es constante.
- Suponiendo que el tiempo de la función *producto* estará en $\Theta(n)$.

Ejercicios

procedimiento proc(E entero: n)

var

entero: x

inicio

$x \leftarrow 0$

mientras $(x+1)^*(x+1) \leq n$ **hacer**

$x \leftarrow x+1$

fin_mientras

fin_procedimiento

Calcular el orden de complejidad del procedimiento en función del valor de n.

Ejercicios

```

entero funcion func(E entero: n)
var
    entero: prod, acum, a
inicio
    acum  $\leftarrow$  0
    prod  $\leftarrow$  n
    mientras prod  $\geq$  1 hacer
        a  $\leftarrow$  0
        desde i $\leftarrow$ 1 hasta n hacer
            si i  $\leq$  prod entonces
                a  $\leftarrow$  a + cuadrado(i)
            si_no
                a  $\leftarrow$  a – cuadrado(i)
        fin_desde
        acum  $\leftarrow$  acum + a
        prod  $\leftarrow$  prod/2
    fin_mientras
    devolver prod
fin_función

```

Calcular el orden de complejidad del algoritmo en función del valor de n:

- Suponiendo que el tiempo de la función *cuadrado* es constante.
- Suponiendo que el tiempo de la función *cuadrado* estará en $\Theta(n)$.

Ejercicios

```
entero función func(E entero: n)
var
    entero: a, b, res
inicio
    res  $\leftarrow$  0
    b  $\leftarrow$  1
    mientras b  $\leq$  n hacer
        a  $\leftarrow$  b
        mientras a > 0 hacer
            res  $\leftarrow$  res + 2·a

            a  $\leftarrow$   $\left\lfloor \frac{a}{2} \right\rfloor$ 
        fin_mientras
        b  $\leftarrow$  b + 1
    fin_mientras
    devolver res
fin_función
```

Calcular el orden de complejidad en función del valor de n.

Ejercicios

entero **función** func(E entero: n)

var

entero: z, d, x

inicio

z \leftarrow 0

d \leftarrow 1

mientras $d*d \leq n$ **hacer**

 x \leftarrow d

mientras $x \geq 0$ **hacer**

 z \leftarrow z + x

 x \leftarrow x - 1

fin_mientras

 d \leftarrow d + 1

fin_mientras

devolver z

fin_función

Calcular el orden de complejidad en función del valor de n.

Ejercicios

entero **función** fun(E entero: n)

var

entero: i,j,k,sum

inicio

$i \leftarrow \left\lfloor \frac{n}{2} \right\rfloor$

sum \leftarrow 0

mientras i > 0 **hacer**

j \leftarrow 1

mientras j <= n **hacer**

sum \leftarrow opera(sum)

j \leftarrow j*2

fin_mientras

i \leftarrow i-2

fin_mientras

devolver sum

fin_función

Calcular el orden de complejidad en función del valor de n, considerando la función *opera* $\in O(n)$.

Ejercicios

entero **función** fun(E entero: n)

var

entero: i, j, suma

inicio

i ← 1

suma ← 0

mientras i ≤ n **hacer**

desde i ← i **hasta** n **hacer**

 suma ← suma + i + j

fin_desde

 i ← i + 1

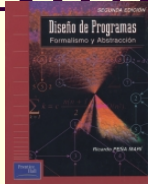
fin_mientras

devolver suma

fin_función

Calcular el orden de complejidad en función del valor de n.

Bibliografía



- Peña Marí, Ricardo; (1998) Diseño de Programas. Formalismo y Abstracción. Prentice Hall.



- Bálcazar José Luis (2001). Programación Metódica. McGraw-Hill.



- Castro Rabal, Jorge; Cucker Farkas, Felipe (1993). Curso de programación. McGraw-Hill / Interamericana de España, S.A.



- Martí Oliet Narciso (2006), Segura Díaz Clara M., Verdejo López José A. Especificación, derivación y análisis de algoritmos : ejercicios resueltos