Tema 3
ALGORITMIA

Parte II. *Notación Asintótica* .

Irene Martínez Masegosa
Depto. de Informática
UNIVERSIDAD DE ALMERÍA

3.2-1

Contents

3.2-2

Lo que ya sabemos de la Notación Asintótica

- Nos interesa estimar el tiempo de ejecución de un algoritmo en base al **número máximo de instrucciones** que ejecute con un método que no considere aspectos dependientes de la implementación y del hardware.
- Para ello definiremos una **función** $f(n)$ que modelice el tiempo empleado por el algoritmo en función del tamaño de la entrada n.
- Nos interesa el comportamiento de esa función para **valores grandes** de n.
- La notación **asintótica** representa el comportamiento de la función cuando el tamaño de la **entrada tiende a infinito**.
- Son funciones de $\mathbb{N}$ en $\mathbb{R}^+$ ya que el tiempo de ejecución no puede ser negativo

3.2-3

Objetivos de esta lección

1. Estudiar cómo calcular **cotas superiores** e **inferiores** del tiempo de ejecución de un algoritmo
2. Presentar técnicas básicas para el análisis de algoritmos
3. Destacar la **importancia** y las **limitaciones** de la Notación Asintótica

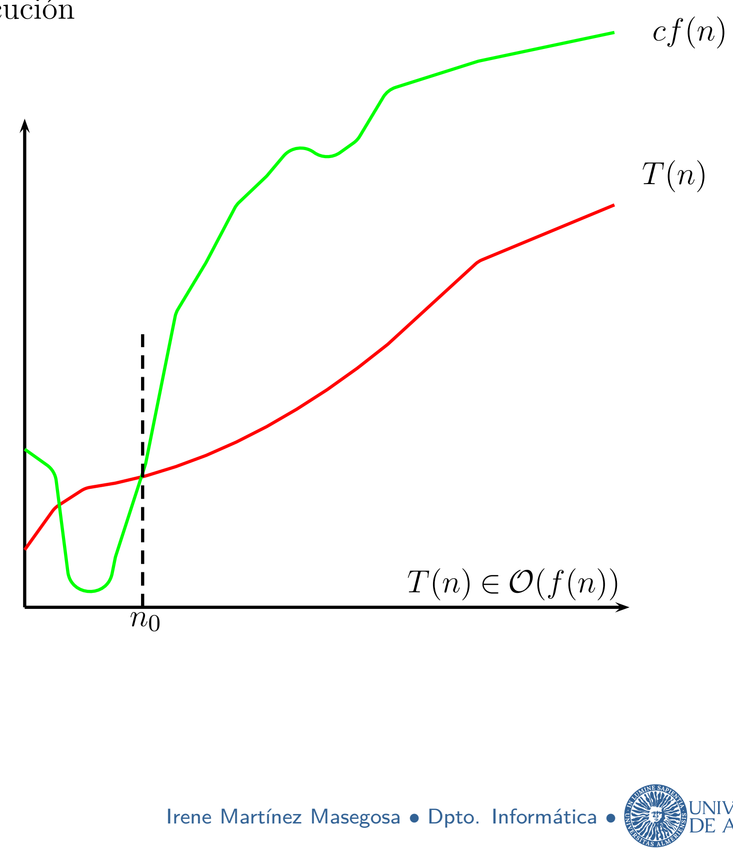
3.2-4

1 Estudio de las Notaciones Asintóticas

1.1 Notación $\mathcal{O}$ Grande *(Big Oh)*

Definición. Notación $\mathcal{O}$

- Proporciona una ***cota superior*** de la forma en que crece el tiempo de ejecución



3.2-5

Definición formal: $\mathcal{O}$ conjunto de funciones

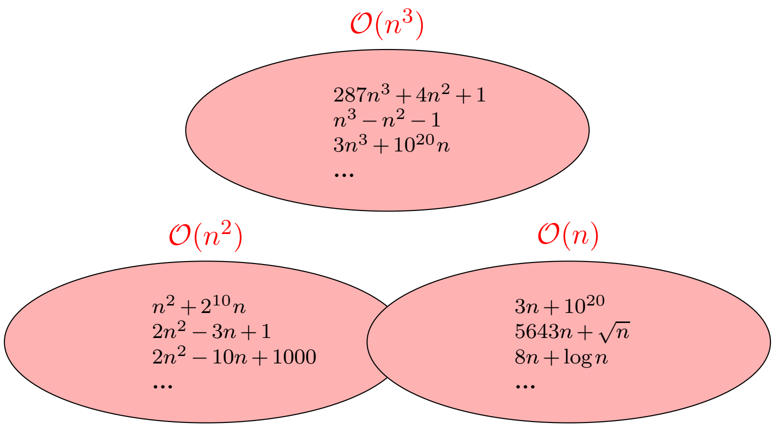
- $\mathcal{O}(f)$ denota el conjunto de funciones t que crecen **a lo sumo tan rápido como f**

Dada una función $f : \mathbb{N} \to \mathbb{R}^+$, llamamos **orden de f** al conjunto de todas las funciones de $\mathbb{N}$ en $\mathbb{R}^+$ acotadas **superiormente** por un múltiplo real positivo de f, para valores de n suficientemente grandes:

$$\mathcal{O}(f) = \{\mathbf{t} \mid \exists c \in \mathbb{R}^+, \exists n_0 \in \mathbb{N}, \forall n \geq n_0 : \quad \mathbf{t(n) \leq cf(n)}\}$$

- Se nota $t \in \mathcal{O}(f)$ o también como $t = \mathcal{O}(f)$
- Se utiliza en el análisis del caso peor 3.2-6

Ejemplos. Conjuntos de funciones



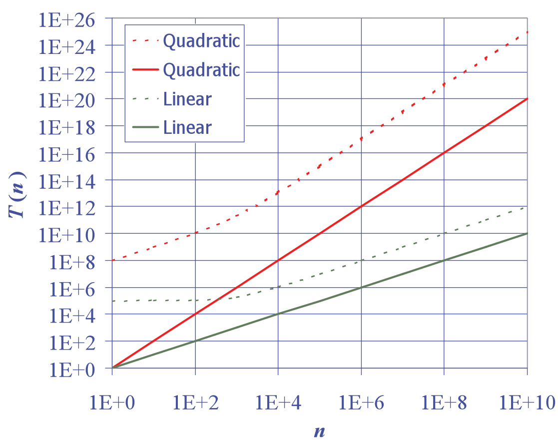
3.2-7

Ejemplos

- Recordemos que la notación asintótica **no se ve afectada por**:
  - **factores constantes**
  - **términos de menor orden**

En la siguiente gráfica tenemos estas funciones:

- $T(n) = 10^2 n + 10^5$ es una función lineal , $T(n) \in \mathcal{O}(n)$
- $T(n) = 10^5 n^2 + 10^8 n$ es una función cuadrática , $T(n) \in \mathcal{O}(n^2)$



Y del mismo modo, se cumple que:

- $T(n) = 50n\log n$ está en $\mathcal{O}(n\log n)$
- $T(n) = 3n^2\log n + 5n^2$ está en $\mathcal{O}(n^2\log n)$

3.2-8

Ejemplos:

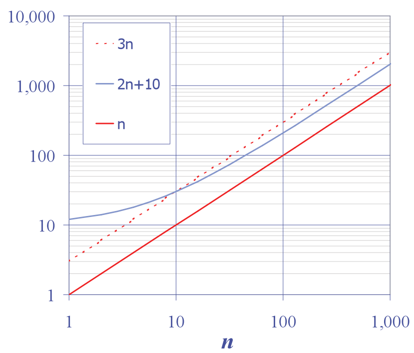
- En los ejemplos de la primera parte vimos que:
  - $T(n) = 1+n$, entonces $f(n) = n$ y por tanto $T(n) \in \mathcal{O}(n)$.
  - $T(n) = 5n^2+27n+1005$, $f(n) = n^2$ y por tanto $T(n) \in \mathcal{O}(n^2)$
- Para demostrarlo basta encontrar dos valores n_0 y c para los que se cumpla que: $T(n) \leq cf(n)$.
  - $T(n) = 1+n \in \mathcal{O}(n)$
    entonces se debe cumplir que: $1+n \leq cn$ a partir de un $n \geq n_0$.
    Cierto para n_0=1 y $c = 10$
  - $T(n) = 5n^2+27n+1005 \in \mathcal{O}(n^2)$
    entonces: $5n^2+27n+1005 \leq cn^2$ a partir de un valor $n \geq n_0$:
    cierto para n_0=20 y $c = 10$

3.2-9

Notación $\mathcal{O}$. Ejemplos

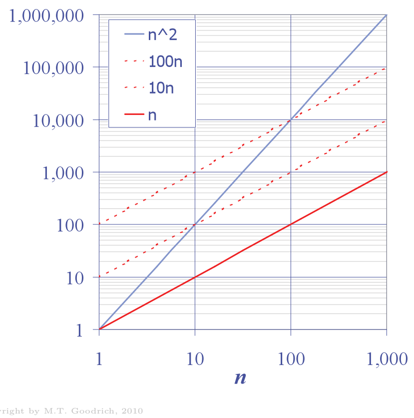
Ejemplo: $T(n) = 2n+10$
$T(n) = 2n+10 \in \mathcal{O}(n)$

- Se debe cumplir que $2n+10 \leq c.n$
  $cn-2n \geq 10$
  $(c-2)n \geq 10$
  $n \geq 10/(c-2)$
- Cierto para $c = 3$ y $n_0 = 10$
  Se puede ver en la gráfica:



Ejemplo: $T(n) = n^2$
La función n^2 *NO está en* $\mathcal{O}(n)$.

- Debería cumplirse que $n^2 \leq c.n$, luego:
  $n \leq c$
- La desigualdad anterior no puede satisfacerse porque c debe ser una constante.



Ejemplo: $T(n) = 7n - 2$
Para demostrar que $T(n) = 7n - 2$ está en $\mathcal{O}(n)$

- Necesitamos: $c > 0$ y $n_0 \geq 1$ tales que: $7n - 2 \leq c.n$
- Cierto para: $c = 7 \quad n_0 = 1$

Ejemplo: $T(n) = 3n^3 + 20n^2 + 5$
Para demostrar que $T(n) = 3n^3 + 20n^2 + 5$ está en $\mathcal{O}(n^3)$

- Necesitamos $c > 0$ y $n_0 \geq 1$ tales que: $3n^3 + 20n^2 + 5 \leq c.n^3$
- Cierto para: $c = 4 \quad n_0 = 21$

3.2-10

Propiedades. Resumen

Sean f, g y h funciones de $\mathbb{N}$ en $\mathbb{R}^+$, y $c, d \in \mathbb{R}^+$ constantes:

- *La tasa de crecimiento no se ve afectada por suma o producto de constantes*
  - $g \in \mathcal{O}(f) \Leftrightarrow c.g \in \mathcal{O}(f)$
  - $g \in \mathcal{O}(f) \Leftrightarrow c + g \in \mathcal{O}(f)$
- *p y q polinomios con coeficiente principal positivo*
  - $\mathcal{O}(p) = \mathcal{O}(q)$, si grado de p = grado de q
  - $\mathcal{O}(p) \subset \mathcal{O}(q)$, si grado de p < grado de q
  - $\mathcal{O}(p) \subset \mathcal{O}(2^n)$
- Reglas básicas en el análisis de algoritmos:
  si $g_1 \in \mathcal{O}(f_1), g_2 \in \mathcal{O}(f_2)$
  - ***Regla de la suma***: $\mathcal{O}(f(n) + g(n)) = \mathcal{O}(max(f(n), g(n)))$
    ➜ $g_1 + g_2 \in \mathcal{O}(max(f_1, f_2))$
  - ***Regla del producto***: $\mathcal{O}(f(n)).\mathcal{O}(g(n)) = \mathcal{O}(f(n).g(n))$
    ➜ $g_1.g_2 \in \mathcal{O}(f_1.f_2)$

3.2-11

Propiedades. Ejemplos

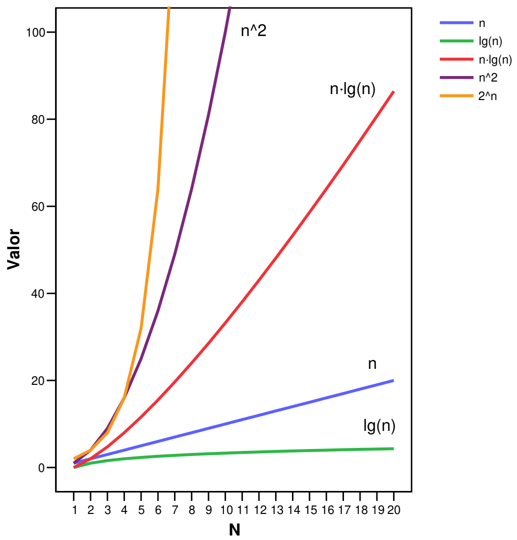
Ejemplos:

- Estos órdenes son iguales $\mathcal{O}(287n^3+4n^2+1)$ y $\mathcal{O}(n^3-n^2-1)$ ya que están incluidos en $\mathcal{O}(n^3)$.
- $\mathcal{O}(2n^2-3n+1) \subset \mathcal{O}(n^3)$
- $\mathcal{O}(5643n) = \mathcal{O}(n)$
- $\mathcal{O}(20n).\mathcal{O}(n) = \mathcal{O}(20n^2) = \mathcal{O}(n^2)$ (regla producto)
- $\mathcal{O}(n^2)+\mathcal{O}(2^{10}n) = \mathcal{O}(n^2)+\mathcal{O}(n) = \mathcal{O}(n^2+n) = \mathcal{O}(max(n^2,n)) = \mathcal{O}(n^2)$ (regla máximo)

3.2-12

Cotas de Complejidad Frecuentes

$\mathcal{O}(1) \subset \mathcal{O}(\log n) \subset \mathcal{O}(n^{\frac{1}{2}}) \subset \mathcal{O}(n) \subset \mathcal{O}(n\log n) \subset \mathcal{O}(n\log n\log n) \subset \mathcal{O}(n^2) \subset \mathcal{O}(n^3) \subset \mathcal{O}(2^n) \subset \mathcal{O}(n!) \subset \mathcal{O}(n^n)$



- $\log n \in \mathcal{O}(n^k)$ para cualquier $k > 0$
  La función *log* crece más lento que cualquier potencia positiva de n (incluidas las potencias fraccionales)
- $n^k \in \mathcal{O}(2^n)$ para cualquier $k > 0$
  Las potencias de n crecen más lentamente que la exponencial 2^n.
- *Los **logaritmos** son del mismo orden*, independien. de la base:

$$\forall B > 1 : \quad \log_B N \in \mathcal{O}(\log N)$$

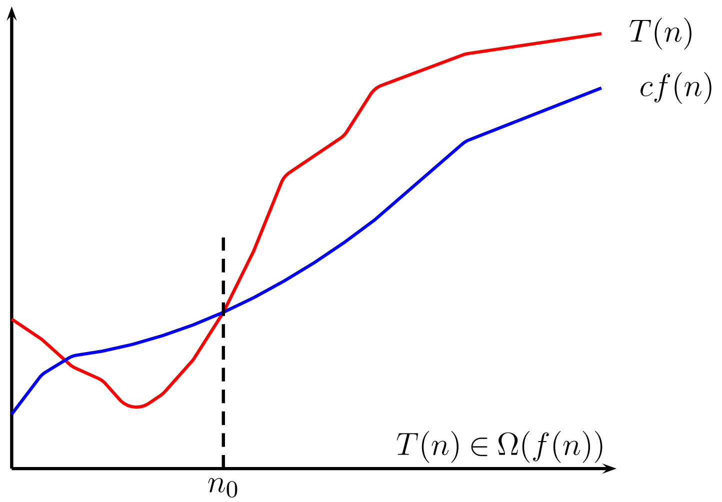
3.2-13

1.2 Notación Omega Ω *(Big Omega)*

- ¿Podemos definir una **cota inferior** del tiempo de ejecución?

3.2-14

Cota Inferior. Notación Ω



3.2-15

Notación Ω
Significa que a partir de un valor de la entrada $n \geq n_0$, o umbral, existe una constante positiva c tal que:

- el tiempo de ejecución $T(n) \geq cf(n)$
- y decimos que $T(n) \in \Omega(f(n))$
- es decir $T(n)$ está **acotado inferiormente** por $f(n)$

3.2-16

Definición formal

- $\Omega(f)$ denota el conjunto de funciones t que crecen **al menos tan rápido como** f a partir de un cierto n

Dada una función $f : \mathbb{N} \rightarrow \mathbb{R}^+$, llamamos **omega de** f al conjunto de todas las funciones de $\mathbb{N}$ en $\mathbb{R}^+$ acotadas **inferiormente** por un múltiplo real positivo de f, para valores de n suficientemente grandes:

$$\Omega(f) = \{\mathbf{t} \mid \exists c \in \mathbb{R}^+, \exists n_0 \in \mathbb{N}, \forall n \geq n_0 : \quad \mathbf{t}(\mathbf{n}) \geq \mathbf{cf}(\mathbf{n})\}$$

- Se nota $t \in \Omega(f)$ o también como $t = \Omega(f)$
- Se utiliza para describir tiempos de ejecución en el caso mejor o cotas inferiores de problemas algorítmicos.

3.2-17

Ejemplos. Notación Ω

Ejemplo: $5n^2 \in \Omega(n^2)$

- Necesitamos: $c > 0$ y $n_0 \geq 1$ t.q. $5n^2 \geq c.n^2$ para $n \geq n_0$
- Cierto para: $c = 5$ $n_0 = 1$

Ejemplo: $5n^2 \in \Omega(n)$

- Necesitamos: $c > 0$ y $n_0 \geq 1$ t.q. $5n^2 \geq c.n$ para $n \geq n_0$
- Cierto para: $c = 1$ $n_0 = 1$

Ejemplo: $10n^2 + 4n + 2 \in \Omega(n^2)$

- Necesitamos: $c > 0$ y $n_0 \geq 1$ t.q. $10n^2 + 4n + 2 \geq c.n^2$ para $n \geq n_0$
- Cierto para: $c = 11$ $n_0 = 5$

3.2-18

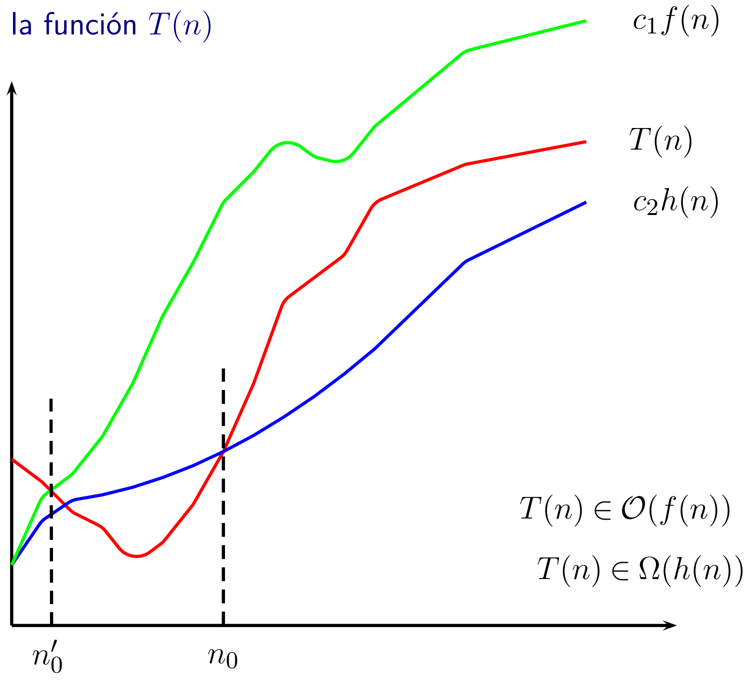
Propiedades. Resumen

Sean f, g y h funciones de $\mathbb{N}$ en $\mathbb{R}^+$, y $c, d \in \mathbb{R}^+$ constantes:

- *No se ve afectada por suma o producto de constantes*
  - $g \in \Omega(f) \Leftrightarrow c.g \in \Omega(f)$
  - $g \in \Omega(f) \Leftrightarrow c + g \in \Omega(f)$
- *p y q polinomios con coeficiente principal positivo*
  - $\Omega(p) = \Omega(q)$, si grado de p = grado de q
  - $\Omega(p) \supset \Omega(q)$, si grado de p < grado de q
  - $\Omega(p) \supset \Omega(2^n)$
- **Regla de la suma:** Si $g_1 \in \Omega(f_1)$ y $g_2 \in \Omega(f_2)$: $g_1 + g_2 \in \Omega(max(f_1, f_2))$
- **Regla de dualidad:**
  Si $g \in \Omega(f) \Leftrightarrow f \in \mathcal{O}(g)$

3.2-19

Cotas para la función $T(n)$



3.2-20

Ejemplo: algoritmo burbuja mejorado, caso peor

- Ordenar de forma ascendente los n elementos de un array
  - variable flag "cambiado" vale "true" si ha habido intercambio de elementos en el bucle interno
  - caso peor: el array está ordenado en orden opuesto $\{9,8,7,6,5,4,3,2,1\}$

```java
public static void burbujaMejora(int[] array) {
    boolean cambiado = true;
    int i = 1;

    while (cambiado && (i < array.length)) {
        cambiado = false;
        for (int j = 0; j < array.length - i; j++) {
            if (array[j] > (array[j + 1])) {
                // Intercambio
                int aux = array[j];
                array[j] = array[j + 1];
                array[j + 1] = aux;
                cambiado = true;
            }
        }
        i++;
    }
}
```

- Este método está en $\mathcal{O}(n^2)$

3.2-21

Ejemplo: algoritmo burbuja mejorado, caso mejor

- Ordenar de forma ascendente los n elementos de un array
  - caso mejor: el array ya está ordenado $\{1,2,3,4,5,6,7,8,9\}$
  - La primera vez que entra en el bucle `while` ejecuta el bucle `for` que recorre el array completo y NO hace intercambios. Y termina.

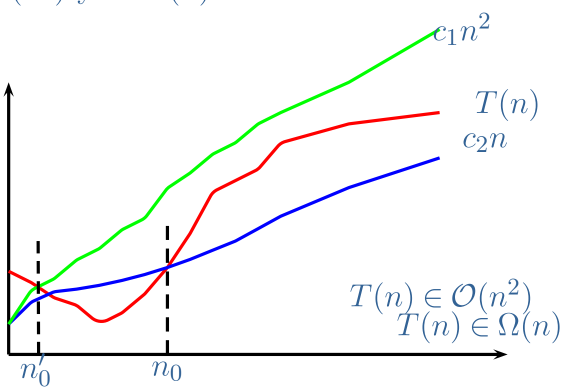
```java
public static void burbujaMejora(int[] array) {
    boolean cambiado = true;
    int i = 1;

    while (cambiado && (i < array.length)) {
        cambiado = false;
        for (int j = 0; j < array.length - i; j++) {
            if (array[j] > (array[j + 1])) {
                // Intercambio
                int aux = array[j];
                array[j] = array[j + 1];
                array[j + 1] = aux;
                cambiado = true;
            }
        }
        i++;
    }
}
```

- Este método está en $\Omega(n)$

3.2-22

Ejemplo: algoritmo burbuja mejorado. Cotas para $T(n)$

- $T(n)$ está en $\mathcal{O}(n^2)$ y en $\Omega(n)$

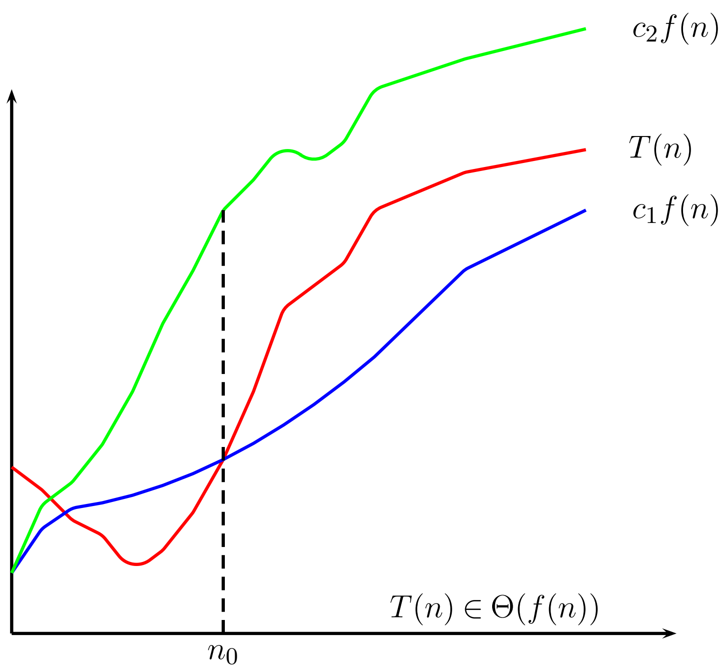


3.2-23

1.3 Notación Theta Θ

- Podemos definir una notación asintótica más:
  para el caso en el que **las cotas inferior y superior coinciden**
- En este caso **el tiempo de ejecución $T(n)$ está acotado superior e inferiormente por la misma función**, a diferencia de las constantes

3.2-24



3.2-25

Orden exacto. Notación Θ

Notación Θ

Significa que a partir de un valor de la entrada $n \geq n_0$, o umbral, existe unas constantes positivas c_1 y c_2 tales que:

$$c_1 f(n) \leq T(n) \leq c_2 f(n)$$

- el tiempo de ejecución: $T(n) \in \Theta(f(n))$
- y decimos que $T(n)$ está en el **orden exacto** de $f(n)$

3.2-26

Definición. Notación Theta Θ

- Denominada también ***orden exacto*** u ***orden de magnitud***
- $\Theta(f)$ denota el conjunto de funciones t con la **misma tasa de crecimiento que** f
- $\Theta(f) = \mathcal{O}(f) \cap \Omega(f)$

Dada una función $f : \mathbb{N} \to \mathbb{R}^+$, llamamos **orden de magnitud de** f al conjunto de todas las funciones de $\mathbb{N}$ en $\mathbb{R}^+$ acotadas **superior e inferiormente** por múltiplos reales positivos de f, para valores de n suficientemente grandes:

$$\Theta(f) = \{\mathbf{t} \mid \exists \mathbf{c}, \mathbf{d} \in \mathbb{R}^+, \exists n_0 \in \mathbb{N}, \forall n \geq n_0 : \mathbf{cf}(\mathbf{n}) \leq \mathbf{t}(\mathbf{n}) \leq \mathbf{df}(\mathbf{n})\}$$

3.2-27

Ejemplo: Notación Θ

Demostrar que $T(n) = 3n+2$ *está en* $\Theta(n)$

Tenemos que comprobar que $T(n)$ es $\mathcal{O}(n)$ y también $T(n)$ es $\Omega(n)$

a) Necesitamos: $c_1 > 0$ y $n_0 \geq 1$ t.q. $3n+2 \leq c_1.n$ para $n \geq n_0$

   Cierto para: $c_1 = 4$ $n_0 = 2$

b) Necesitamos: $c_2 > 0$ y $n_0 \geq 1$ t.q. $3n+2 \geq c_2.n$ para $n \geq n_0$

   Cierto para: $c_2 = 2$ $n_0 = 2$

3.2-28

Ejemplo: notación Θ

- Calcular la media de todos los elementos de una matriz cuadrada:
  - n es el número de filas y de columnas
  - la matriz tiene $n \times n = n^2$ elementos
  - $n =$ `matriz.length`

```java
public double getMedia() {
    int sum = 0;
    int numberOfElements = 0;
    for (int i = 0; i < matriz.length; i++) {
        for (int j = 0; j < matriz[i].length; j++) {
            sum += matriz[i][j];
            numberOfElements++;
        }
    }
    return (double)sum / (double)numberOfElements;
}
```

- Este método está en $\mathcal{O}(n^2)$ y también es $\Omega(n^2)$
- Es del orden exacto $\Theta(n^2)$ 3.2-29

Propiedades

Sean f, g y h funciones de $\mathbb{N}$ en $\mathbb{R}^+$, y $c,d \in \mathbb{R}^+$:

- *No se ve afectada por suma o producto de constantes*
  - $g \in \Theta(f) \Leftrightarrow c.g \in \Theta(f)$
  - $g \in \Theta(f) \Leftrightarrow c+g \in \Theta(f)$
- *Simetría:* $g \in \Theta(f) \Leftrightarrow f \in \Theta(g) \Leftrightarrow \Theta(f) = \Theta(g)$
- **Regla de la suma:** Si $g_1 \in \Theta(f_1)$ y $g_2 \in \Theta(f_2)$: $g_1 + g_2 \in \Theta(max(f_1, f_2))$
- **Regla del producto:** Si $g_1 \in \Theta(f_1)$ y $g_2 \in \Theta(f_2)$: $g_1.g_2 \in \Theta(f_1.f_2)$ 3.2-30

Relaciones entre las Notaciones Asintóticas

- *Notación Big-Oh:* $\mathcal{O}$ $g(n) \in \mathcal{O}(f(n))$ si
  $g(n)$ es asintóticamente **menor o igual** que $f(n)$
- *Notación Big Omega:* Ω $g(n) \in \Omega(f(n))$ si
  $g(n)$ es asintóticamente **mayor o igual** que $f(n)$
- *Notación Big Theta:* Θ $g(n) \in \Theta(f(n))$ si
  $g(n)$ es asintóticamente **igual** que $f(n)$ 3.2-31

1.4 Notación con Varios Parámetros

Notación Asintótica con dos Parámetros

- El tiempo de ejecución puede depender de **más de un parámetro**
  Por ejemplo, en algoritmos sobre matrices:
  - n: número de filas
  - m: número de columnas
- En el caso de dos parámetros:

Dada una función $f : \mathbb{N}x\mathbb{N} \to \mathbb{R}^+$, llamamos **orden de** $f(m,n)$ al conjunto de todas las funciones de $\mathbb{N}x\mathbb{N}$ en $\mathbb{R}^+$ acotadas superiormente por $f(m,n)$, para valores de m y n suficientemente grandes:

$$\mathcal{O}(f(m,n)) = \{t \mid \exists c \in \mathbb{R}^+, \exists \mathbf{m_0}, \mathbf{n_0} \in \mathbb{N}, \quad \forall \mathbf{m} \geq m_0, \mathbf{n} \geq n_0 : \\ t(m,n) \leq cf(m,n)\}$$

3.2-32

Ejemplo:

- En el ejemplo anterior, si la matriz NO es cuadrada:
  - n es el número de filas y m el de columnas
  - la matriz tiene $n \times m$ elementos
  - $n =$ `matriz.length` ,
  - $m =$ `matriz[i].length` :

```
public double getMedia() {
    int sum = 0;
    int numberOfElements = 0;
    for (int i = 0; i < matriz.length; i++) {
        for (int j = 0; j < matriz[i].length; j++) {
            sum += matriz[i][j];
            numberOfElements++;
        }
    }
    return (double)sum / (double)numberOfElements;
}
```

- Este método está en $\mathcal{O}(n \times m)$ y también es $\Omega(n \times m)$
- Es del orden exacto $\Theta(n \times m)$

3.2-33

Notación Asintótica con Varios Parámetros

- Generalización para funciones de varias variables:

Dada una función $f : \mathbb{N}^k \rightarrow \mathbb{R}^+$, llamamos
orden de f al conjunto de todas las funciones de $\mathbb{N}^k$ en $\mathbb{R}^+$ acotadas superiormente por un múltiplo de f, para valores de $n_1, n_2, ..., n_k$ suficientemente grandes:

$$\mathcal{O}(f) = \{t : \quad \exists c \in \mathbb{R}^+, \quad \exists n_1, n_2, ..., n_k \in \mathbb{N},$$

$$\forall m_1 \geq n_1, m_2 \geq n_2, ..., m_k \geq n_k :$$

$$t(m_1, m_2, ..., m_k) \leq c f(m_1, m_2, ..., m_k)\}$$

- Podemos extender los conceptos de $\Omega(f)$ y $\Theta(f)$ y las propiedades se cumplen

3.2-34

1.5 Cálculo de Relación Asintótica usando Límites

Cálculo de Límites

Relación entre $f, g : \mathbb{N} \rightarrow \mathbb{R}^+$ conociendo el valor, **si existe**, de:

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

- Los posibles valores de L determinan:

- Si $L = 0$ $\quad f(n)$ crece más rápido que $g(n)$
  - entonces: $\quad g \in \mathcal{O}(f) \qquad g \in o(f) \quad$ y $g \notin \Theta(f)$
- Si $L = \infty$ $\quad g(n)$ crece más rápido que $f(n)$
  - entonces: $\quad g \in \Omega(f) \qquad$ pero $g \notin \Theta(f)$
- Si $L \neq 0 \in \mathbb{R}^+$ $\quad f(n)$ y $g(n)$: misma tasa de crecimiento
  - entonces: $\quad g \in \Theta(f) \quad$ y $\quad f \in \Theta(g)$
  - $g \in \mathcal{O}(f) \quad , \quad g \in \Omega(f)$
  - $f \in \mathcal{O}(g) \quad , \quad f \in \Omega(g)$

Si no existe límite no podemos usar esta técnica para determinar la relación asintótica entre f y g. 3.2-35

2 Cómo Análizar Algoritmos

Análisis del Algoritmo en función de n

- Habíamos visto que:
  En el análisis de un algoritmo se estima el **número de operaciones**

```
Algoritmo Ejemplo(int n)
    A ← n
    for i ← 1 to n do
        instruccion1
        instruccion2
    ...
```

ANÁLISIS ALG.

‖

¿Cuántas Operaciones? ←

‖

☞ $T(n) \in \mathcal{O}(f(n))$

3.2-36

Análisis de un algoritmo

- A partir del estudio de las operaciones elementales del algoritmo se estima el $T(n)$ en función del tamaño de la entrada.
- Se obtiene su orden de complejidad, que puede ser logarítmico, lineal, cuadrático, exponencial,...

¿Cómo hacemos el análisis del algoritmo?

- El número de operaciones elementales hay que obtenerlo estudiando las estructuras de control del algoritmo: bucles, sentencias condicionales, etc.
- Este número se combina según las estructuras de control
- Mediante este análisis modelizamos el comportamiento del algoritmo para **valores muy grandes de** n (Notación asintótica).

3.2-37

Dos posibilidades:

Podemos contar el número de instrucciones elementales: *complicado en algoritmos largos*

funcion MaxArray(A,n)	Operaciones
$maxActual \leftarrow A[0]$	2
FOR $i \leftarrow 1$ hasta $n-1$ hacer	$n-1$
IF $A[i] > maxActual$ THEN	$2(n-1)$
$maxActual \leftarrow A[i]$	$2(n-1)$
Incrementar i	$2(n-1)$
devolver $maxActual$	1
	$7(n-1)+3 \in \mathcal{O}(n)$

O podemos estudiar directamente las estructuras de control del algoritmo:

```
funcion MaxArray(A,n)
  maxActual ← A[0]
  FOR i ← 1 hasta n−1 hacer
    IF A[i] > maxActual THEN
       maxActual ← A[i]
    Incrementar i
  devolver maxActual
```

Análisis del Algoritmo

estructura de control: bucle FOR

$\Downarrow$

$\mathcal{O}(n)$

3.2-38

2.1 Análisis de las estructuras de control

Secuencias de Instrucciones

- Sean $P1$ y $P2$ dos fragmentos **independientes** de un algoritmo, y
- t_1 y t_2 sus tiempos de ejecución:

```
{
        P1;
}
;
{
        P2;
}
```

$\Rightarrow$ El tiempo requerido para calcular $P1;P2$ es $T = t_1 + t_2$

Aplicando *regla de la suma*,
el tiempo para una **secuencia** $P1;P2$ está en

$$\mathcal{O}(max(t_1,t_2))$$

3.2-39

Sentencia Condicional

¿Cuál es el tiempo asociado a una sentencia *IF-ELSE* o *SWITCH*?

IF $(expresion == true)$
{
$P1;$
}
ELSE
{
$P2;$
}

- Calculamos el tiempo de las instrucciones del bloque $P1 = t_1$
- Calculamos el tiempo de las instrucciones del bloque `ELSE` $= t_2$
- El tiempo del bloque `IF-ELSE` será $T = \mathcal{O}(max(t_1, t_2))$ es decir, **de la parte que más tarda**
- Y le sumamos el tiempo de evaluar la *expresión*

3.2-40

Ejemplo: sentencia condicional

- En este método la parte `ELSE` es la que más tarda:

```
1   public static boolean esPrimo(int n) {
2       boolean primo = true;
3       int divisor;
4       if ((n % 2 == 0) && (n != 2))
5           primo = false;
6       else {
7           divisor = 3;
8           while (primo && divisor <= (int) Math.sqrt(n)) {
9               if (n % divisor == 0)
10                  primo = false;
11              divisor = divisor + 2;
12          }
13      }
14      if (primo)
15          return true;
16      else
17          return false;
18  }
```

- La parte `IF` (líneas 4 y 5) tarda un tiempo constante y la parte `ELSE` (6-13) es la que más tarda (contiene un bucle)
- Las líneas 14-17 también tienen tiempo constante
- El orden de este método está en $\mathcal{O}(tWhile(n))$ y también es $\Omega(1)$

3.2-41

Bucles FOR

FOR $i \leftarrow 1$ **to** m **do**
{
$P(i);$
}

- El algoritmo trabaja con ejemplares de tamaño n

- Podemos calcular el tiempo de una sentencia ***FOR*** **sumando** los **tiempos invertidos en cada pasada** del bucle:

$$\boxed{T = \sum_{i=1}^{m} t(i)}$$

  - $t(i)$: tiempo invertido en la iteración i
  - m número de veces que se ejecuta el bucle

- **Si en todas las iteraciones se invierte el mismo tiempo** t, entonces el tiempo del bucle será igual al producto de $m \times t$, y por tanto T está en $T \in \mathcal{O}(m.t)$

3.2-42

Ejemplo: Factorial int

- Cálculo del factorial de un número entero n.
- Todas las iteraciones tardan igual:

```
public static long factorialIterativo(int n){
    if (n == 0)
        return 1;
    long fact = 1;
    for(int i=1; i<=n; i++){
        fact = fact * i;
    }
    return fact;
}
```

- Bucle:
. Instrucciones **dentro del bucle son elementales, requieren un tiempo constante**: tiempo del bucle acotado superiormente por c, en todas las iteraciones
. Como el bucle se ejecuta n veces:

$$T_{bucle} \leq \sum_{i=1}^{n} c \quad = n.c \quad \Rightarrow T_{bucle} \in \mathcal{O}(n)$$

- Este método está en $\mathcal{O}(n)$ y también es $\Omega(1)$.

3.2-43

Ejemplo: Factorial BigInteger

- Cálculo del factorial de un número entero largo que se representa como un array de N dígitos.
- La operación producto ya NO es primitiva, hay que multiplicar arrays de N dígitos:
  - La versión sencilla es similar a la multiplicación clásica de orden N^2,
  - aunque hay algoritmos para multiplicar enteros largos de menor orden, como el Algoritmo de Karatsuba de $\mathcal{O}(N^{log3}) = \mathcal{O}(N^{1,585})$.

```
1   public static BigInteger factorialBigInt(int n) {
2      BigInteger resultado = BigInteger.ONE;
3      for (int i = 1; i <= n; i++)
4         resultado = resultado.multiply(new BigInteger(i + ""));
5      return resultado;
6   }
```

. El producto de la línea 4 depende de su **tamaño** y es $\mathcal{O}(N^2)$

. Entonces el coste de las operaciones dentro del bucle es $\mathcal{O}(N^2)$:

$$T_{bucle} \leq \sum_{i=1}^{n} \mathcal{O}(N^2) \quad = n \times \mathcal{O}(N^2) \in \mathcal{O}(n.N^2)$$

- Este método `factorialBigInteger(n,N)` $\in \mathcal{O}(n.N^2)$. 3.2-44

Bucles FOR anidados

¿Cuál es el tiempo asociado a un secuencia de bucles *FOR* **anidados**?

FOR $i \leftarrow 1$ **to** n **do**
  P(i) ;
  FOR $j \leftarrow 1$ **to** n **do**
    P(j) ;
    FOR $k \leftarrow 1$ **to** n **do**
      P(k)...

- **Se analizan los bucles de dentro hacia afuera**
  - Calculamos el tiempo empleado en las instrucciones de cada bucle, desde el más interno hasta el más externo
  - Suponiendo que se ejecuta un bloque de instrucciones $P(i)$ en cada bucle i, calculamos la suma del tiempo en cada uno:

$$T_{bucle} = \sum_{i=1}^{n} \left(P(i) + \sum_{j=1}^{n} \left(P(j) + \sum_{k=1}^{n} P(k) \right) \right)$$

3.2-45

Bucles FOR anidados. Ejemplo

Ejemplo:
Este fragmento de código es $\mathcal{O}(n^2)$:

FOR $i \leftarrow 1$ **to** n **do**
  FOR $j \leftarrow 1$ **to** n **do**
    $k \leftarrow k+1;$

$$T_{bucle} \leq \sum_{i=1}^{n} \left(\sum_{j=1}^{n} c \right) \quad = \quad \sum_{i=1}^{n} \overbrace{\sum_{j=1}^{n} c}^{n.c}$$

sustituyendo:

$$T_{bucle} \leq \sum_{i=1}^{n} n.c \quad = n.n.c \quad \in \mathcal{O}(n^2)$$

3.2-46

Ejemplo: Ordenación por Búrbuja

- Compara cada par de elementos consecutivos del array de tamaño n y los intercambia si no están ordenados.

```
funcion OrdenarBurbuja(A[n])
1. FOR i ← 1 to n do
2.     FOR j ← 1 to n − i do
3.         IF A[j] > A[j+1] THEN
               aux ← A[j];
               A[j] ← A[j+1];
               A[j+1] ← aux;
4.     fin;
```

$$T_{bucle} \leq \sum_{i=1}^{n} \left(\sum_{j=1}^{n-i} c \right)$$

- Resolvemos y obtenemos el orden del algoritmo:

$$T_{bucle} \leq \sum_{i=1}^{n} \overbrace{\left(\sum_{j=1}^{n-i} c \right)}^{(n-i).c} = \sum_{i=1}^{n} (n-i).c = \quad c \sum_{i=1}^{n} (n-i) =$$

Desdoblamos en dos sumatorios:

$$= c \left(\sum_{i=1}^{n} n - \sum_{i=1}^{n} i \right) = \quad c \left(n.n - \sum_{i=1}^{n} i \right) =$$

teniendo en cuenta que

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

, entonces:

$$= c \left(n^2 - \frac{n(n+1)}{2} \right) \quad \in \mathcal{O}(n^2)$$

- El resultado es $\mathbf{T_{Burbuja}} \in \mathcal{O}(n^2)$

3.2-47

Ejemplo: Subsecuencia de suma máxima

- Algoritmo que realiza una búsqueda exhaustiva: prueba todas las posibles subsecuencias y escoge la de suma con valor máximo

```
funcion SubsecFuerzaBruta(A[1..n])
1. maxActual ← −1;
2. FOR i ← 1 to n do
3.    FOR j ← i to n do
4.       suma ← 0;
5.       FOR   k ← i to j do
6.          suma ← suma + A[i];
7.          IF suma > maxActual THEN
               maxActual ← suma;
               secIni ← i;
               secFin ← j;
```

$$T_{bucle} \leq \sum_{i=1}^{n} \sum_{j=i}^{n} \left(c + \sum_{k=i}^{j} d \right)$$

- El resultado es $\binom{n}{3}$
- y por tanto el orden de complejidad $T_{SubsecFuerzaBruta} \in \mathcal{O}(n^3)$

➜ *Consultar demostración en libro de Weiss apartado 5.3*

3.2-48

Ejemplo: Ordenación por Selección

```
funcion OrdenacionSeleccion( T[1..n])
  1. FOR i ← 1 to n − 1 do
     (a) minj ← i;
     (b) minx ← T[i]
     (c) FOR   j ← i + 1 to n do
            · IF T[j] < minx
              minj ← j ;
              minx ← T[j]
     (d) T[minj] ← T[i]
     (e) T[i] ← minx
```

$$T(n) \leq \sum_{i=1}^{n-1} \left(b + \sum_{j=i+1}^{n} c \right)$$

↪ bucle interno (línea c):

- Se ejecuta $(n-i)$ veces ya que: $n-(i+1)+1 = n-i$
- y el tiempo de cada pasada está acotado por c:

$$\sum_{j=i+1}^{n} c = [n-(i+1)+1]c = (n-i)c$$

· Entonces, el tiempo del bucle interno lo acotamos por: $(n-i).c$

↪ bucle externo (línea 1), sustituimos:

· tiempo requerido por la i-ésima pasada acotado por $\leq b+(n-i).c$

· b, cte que acota operaciones elementales:

$$T(n) \leq \sum_{i=1}^{n-1} \left(b + \sum_{j=i+1}^{n} c \right) = \sum_{i=1}^{n-1} (b + (n-i).c)$$

- Resolviendo:

$$T(n) \leq \sum_{i=1}^{n-1} b + (n-i).c = \sum_{i=1}^{n-1} (b + cn - ci) =$$

Desdoblamos en dos sumatorios:

$$= \sum_{i=1}^{n-1} (b+cn) - c\sum_{i=1}^{n-1} i =$$

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

- Entonces, el tiempo del bucle externo está en $\mathcal{O}(n^2)$ y por tanto:
- Tiempo del algoritmo $T_{OrdSelec} \in \mathcal{O}(n^2)$

y también $T_{OrdSelec} \in \Theta(n^2)$

3.2-49

Bucles WHILE

$$\boxed{\begin{array}{l} \textbf{WHILE } (expresion = true) \quad \textbf{do} \\ \{ \\ \qquad\qquad P(i); \\ \} \end{array}}$$

- **No se conoce *a priori* el número de veces que se ejecuta el bucle**

- Dos soluciones:

  a) Necesitamos conocer el número de veces que se ejecuta el bucle, para poder plantear su coste como en los bucles FOR:

  → Hallar una función de las variables implicadas en la condición de parada del bucle

  → Comprobar que la función se **decrementa** en cada pasada y el bucle termina

  b) Tratar el bucle como un algoritmo recursivo

3.2-50

Ejemplo. Búsqueda binaria

funcion BusquedaBinaria($T[1..n], x$): int

1. $i \leftarrow 1; j \leftarrow n$
2. WHILE $i < j$ do
   . $k \leftarrow (i+j)/2$
   . IF $x < T[k] : j \leftarrow k-1$
   . IF $x > T[k] : i \leftarrow k+1$
   . IF $x = T[k] : i, j \leftarrow k$
3. *devolver* i

a) Hallar una función de las variables implicadas:

- Sea $d = j - i + 1$: **núm. de elementos** de T que **quedan por examinar**

  $\Rightarrow$ inicialmente $d = n$

  $\Rightarrow$ el bucle termina cuando $d \leq 1$ (peor caso)

- Tenemos que calcular el **número de veces** que se ejecuta el bucle y el **tiempo** que tarda en cada pasada

- Como el tiempo invertido en las líneas 1 y 3 es constante, $\mathcal{O}(1)$, el tiempo total del algoritmo vendrá determinado por el tiempo del bucle en el paso 2.

3.2-51

Consideramos, entonces: d': valor de d **después** del bucle

- Tres posibilidades:
  - si $x < T[k]$: sólo cambia j : $\quad d' \leq \frac{d}{2}$
  - si $x > T[k]$: sólo cambia i : $\quad d' \leq \frac{d}{2}$
  - si $x = T[k]$: $i = j$: $\quad d' = 1$
- Entonces:
  - En cada pasada dividimos el número de elementos por 2. Esto puede hacerse como máximo un número logarítmico de veces, luego:
  - **máximo número de veces** que se ejecuta el bucle: $\lceil \log_2 n \rceil$
  - cada pasada requiere un **tiempo constante** $\leq c$
  - $T_{Bucle} \leq c.\lceil \log_2 n \rceil \quad \Leftrightarrow \quad T_{Bucle} \in \mathcal{O}(\log n)$
  - $\mathbf{T_{BBin}} \in \mathcal{O}(\log \mathbf{n})$

3.2-52

Ejemplo. Ordenación por Inserción

funcion OrdenacionInsercion($T[1..n]$)

1. **FOR** $i \leftarrow 2$ **to** n **do:**
   (a) $x \leftarrow T[i]$
   (b) $j \leftarrow i-1;$
   (c) WHILE $\quad j > 0 \quad$ and $\quad x < T[j] \quad$ DO
       . $T[j+1] \leftarrow T[j]$
       . $j \leftarrow j-1$
   (d) $T[j+1] \leftarrow x$

- El tiempo depende del orden original de los elementos del vector
- Analizamos caso peor: inicialmente vector **ordenado decreciente**, hay que mover todos los elementos
- ¿Cuántas veces se ejecuta el bucle interno WHILE? Nos fijamos en j:
  - ANTES del bucle se inicializa $j = i-1$ y DENTRO se va decrementando si encontramos un elemento menor.
  - En el caso peor, la condición $x < T[j]$ va a ser siempre verdadera, y por tanto j llegará siempre hasta 1.
  - El bucle WHILE se ejecuta, en el peor caso, desde $j = i-1$ hasta $j = 1$, es decir $(i-1)$ veces.
  - Como tiene una coste constante cada pasada, podemos acotar el tiempo del bucle WHILE:

$$T_{Bucle} \leq (i-1).c$$

- Y por tanto, considerando los dos bucles, obtenemos un coste similar a la O. por Selección:

$$T(n) \leq \sum_{i=2}^{n} (b + (i-1).c) \qquad \text{está en} \quad \mathcal{O}(n^2)$$

- Tiempo del algoritmo: $T_{OrdInser} \in \mathcal{O}(n^2)$

3.2-53

3 Ejemplos

3.1 Algoritmos con Tiempo Logarítmico y Lineal

- $\mathcal{O}(\log n)$
- $\mathcal{O}(n)$
- $\mathcal{O}(n \log n)$

3.2-54

Algoritmos $\mathcal{O}(\log n)$

Se resuelve un problema transformándolo en una serie de problemas más pequeños, dividiendo el tamaño del problema por una fracción constante en cada paso

- Caso 1: **Realizar sucesivas divisiones por la mitad**
  - *Si empezamos con $X = n$, si n se divide forma repetida por la mitad, ¿Cuántas veces hay que dividir para hacer n menor o igual que 1?*
  - Solución: Sólo podemos dividir por la mitad un número dado un número logarítmico de veces

- Una algoritmo es $\mathcal{O}(\log n)$ si tarda un tiempo constante ($\mathcal{O}(1)$) en **dividir el tamaño del problema** por un fracción constante (generalmente 1/2)

3.2-55

Algoritmos $\mathcal{O}(\log n)$

- Caso 2: **Bits en un número binario**
  - *¿Cuántos bits son necesarios para representar n enteros consecutivos?*
  - Solución: El número de bits necesarios para representar números es logarítmico

    Son suficientes B bits para representar N enteros diferentes:
    * $2^B \geq N$
    * luego $B \geq \log N$
    * y el número mínimo de bits es $\lceil \log N \rceil$

3.2-56

Algoritmos Lineales $\mathcal{O}(n)$

Tiempo de ejecución es como mucho el producto de un factor constante por el tamaño de la entrada

- Una forma de conseguir este orden es realizando una **única pasada** sobre los datos de entrada y empleando una **cantidad de tiempo constante** en procesar cada uno de ellos
- Caso 1: ***Calcular el máximo de n números***

Máximo(T, n)

INPUT: $\mathbf{T} = \{a_1, \ldots, a_n\}$, lista de elementos
OUTPUT: Máximo de la lista $\mathbf{T}$

1. $max \leftarrow a_1$
2. FOR $i \leftarrow 1$ to n

   . IF $a_i > max$ THEN $max \leftarrow a_i$
3. Devolver max

3.2-57

- Caso 2: ***Mezclar dos listas ordenadas***

Mezclar(A,B, n)

INPUT: $\mathbf{A} = \{a_1, \ldots, a_n\}$, $\mathbf{B} = \{b_1, \ldots, b_n\}$, listas ordenadas
OUTPUT: Lista ordenada $\mathbf{L}$ de tamaño $2n$

1. $primA \leftarrow A[1]$; $primB \leftarrow B[1]$; *// primero de cada lista*
2. $i, j, k \leftarrow 1$;
3. WHILE ($i \leq n$ AND $j \leq n$) *// las listas no estén vacías*

   . $minimo \leftarrow \min(primA, primB)$

   . $L[k] \leftarrow minimo$;

   . $k \leftarrow k + 1$; *// avanzamos índice de elementos de L*

   . IF $(minimo == primA)$
      $i \leftarrow i + 1$; *// avanzamos índice de lista A*
      $primA \leftarrow A[i]$;

   . ELSE
      $j \leftarrow j + 1$; *// avanzamos índice de lista B*
      $primB \leftarrow B[j]$;
4. Añadir resto de la lista (A o B) al final de L
5. Devolver L

3.2-58

Algoritmos $\mathcal{O}(n \log n)$

Es el tiempo de un algoritmo que:

- **divide la entrada** en **dos** partes del **mismo tamaño**
- **resuelve** cada parte recursivamente
- y **combina las soluciones** en un **tiempo lineal**

- ***MergeSort***
  - Divide el conjunto de entrada en **dos subconjuntos** del mismo tamaño
  - Ordena **recursivamente** cada subconjunto
  - Mezcla las dos mitades ordenadas en una lista de salida

3.2-59

3.2 Algoritmos de Tiempo Polinómico

- $\mathcal{O}(n^2)$
- $\mathcal{O}(n^3)$
- $\mathcal{O}(n^k)$

3.2-60

Algoritmos $\mathcal{O}(n^2)$

Realizar una búsqueda sobre **todos los pares** de elementos de la entrada y emplear un **tiempo constante en cada par**

- Caso 1:
  Tenemos n puntos en un plano de coordenadas (x, y) y buscamos los dos más próximos
  - Solución básica:
    Enumerar los pares de puntos, calcular las distancias entre cada par y seleccionar el par cuya distancia sea mínima
  - Tiempo ?
    * Número de pares de puntos: $\binom{n}{2} = \frac{n(n-1)}{2}$ está en $\mathcal{O}(n^2)$
    * Distancia entre puntos (x_i, y_i) y (x_j, y_j):
      $\sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$ en tiempo constante

3.2-61

Algoritmo: consta de **dos bucles anidados**:
un bucle con $\mathcal{O}(n)$ iteraciones y para cada una de ellas se realiza otro bucle interno que emplea $\mathcal{O}(n)$

ParMinimo(L)

INPUT: $\mathbf{L} = \{(x_i, y_i)\}$, lista de n puntos
OUTPUT: Pareja (x_i, y_i) , (x_j, y_j) : puntos más cercanos

1. *minimaDis* $\leftarrow$ MAX_VALUE *// inicializamos con un valor muy alto*
2. FOR cada punto (x_i, y_i) de $\mathbf{L}$
   . FOR cada punto (x_j, y_j) de $\mathbf{L} - \{(x_i, y_i)\}$
      i. Calcular distancia $d = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$
      ii. IF $d <$ *minimaDis* THEN
         . *minimaDis* $\leftarrow d$
         . Actualizar *Pareja*
3. Devolver *Pareja*

- Caso 2: Algoritmos que recorren una matriz: para imprimirla, buscar un elemento, etc.

3.2-62

Algoritmos $\mathcal{O}(n^3)$

Tres bucles anidados:

el algoritmo consta de un bucle con $\mathcal{O}(n)$ iteraciones
y para cada una de ellas se realizan $\mathcal{O}(n)$ iteraciones
sobre otro bucle interno que emplea $\mathcal{O}(n)$

- Caso 1: Procesar los elementos de un cubo (array $nxnxn$)
- Caso 2: Realizar una búsqueda exhaustiva sobre todas las tripletas (subconjuntos de 3 elementos) de un conjunto. De aplicación en geometría computacional.

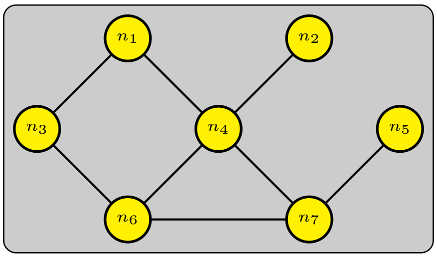
```
public static int count(int[] a)
{
   int N = a.length;
   int count = 0;
   for (int i = 0; i < N; i++)
      for (int j = i+1; j < N; j++)
         for (int k = j+1; k < N; k++)
            if (a[i] + a[j] + a[k] == 0)
               count++;
   return count;
}
```

3.2-63

Algoritmos $\mathcal{O}(n^k)$

Considerar todos los subconjuntos de k elementos obtenidos a partir de un conjunto inicial de n elementos

- *Dado un grafo de n nodos buscar un subconjunto de nodos de tamaño k que cumplan una determinada propiedad*



  - Solución básica:
    Enumerar los subconjuntos de k nodos, y para cada subconjunto S comprobar si se cumple la propiedad
  - ¿Tiempo?
    - Número subconjuntos de tamaño k:
      $$\binom{n}{k} = \frac{n(n-1)(n-2)..(n-k+1)}{k(k-1)(k-2)..2.1} \quad \leq \quad \frac{n^k}{k!} \quad \text{está en } \mathcal{O}(n^k)$$
    - Comprobar propiedad para cada S en tiempo constante
  - El bucle más externo del algoritmo ejecuta $\mathcal{O}(n^k)$ iteraciones y comprueba todos los posibles subconjuntos de k nodos del grafo

3.2-64

3.3 Algoritmos de Tiempo Super-Polinomial

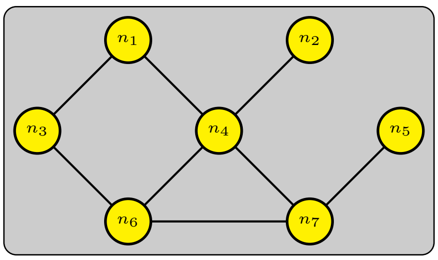
- $\mathcal{O}(2^n)$
- $\mathcal{O}(n!)$

3.2-65

Algoritmos $\mathcal{O}(2^n)$

Considerar todos los subconjuntos que puedan obtenerse a partir de un conjunto de n elementos: 2^n

- *Se tiene una grafo de n nodos y se quiere encontrar el* ***subconjunto de nodos de TAMAÑO MÁXIMO*** *que cumpla una determinada propiedad*



El bucle más externo del algoritmo ejecuta $\mathcal{O}(2^n)$ iteraciones y comprueba la propiedad en *todos los subconjuntos posibles*.

3.2-66

Algoritmos $\mathcal{O}(n!)$

> Se obtiene en problemas en los que el espacio de búsqueda consiste en todas las formas de ordenar n elementos

- **TSP: Travelman Sales Problem**

  Problema del *viajante de comercio* para n ciudades

  – Sean n ciudades de un territorio.

  Objetivo: encontrar una **ruta** que, comenzando y terminando en una ciudad concreta, pase **una sola vez** por cada una de las ciudades y **minimice la distancia recorrida** por el viajante.

  Se trata de un problema de mucho interés práctico:

  – ¿qué ruta debe seguir el camión de la basura para pasar por todos los puntos de recogida en el menor tiempo posible?

  – ¿cómo interconectamos varios ordenadores en una red en anillo con el menor consumo de cable?,

  – ¿cómo organiza su ruta un viajante de comercio que debe visitar una serie de establecimientos repartidos por el país?

Solución fuerza bruta:

- Calculamos **todas las rutas posibles** y elegimos **la mejor**:
  – Tenemos n posibilidades para la primera ciudad, $n-1$ para la segunda, $n-2$ para la tercera...
  – Se calculan $n(n-1)(n-2)... = (n-1)!$ rutas posibles
- Ejemplo: *para N=12 ciudades* , más de 479 millones de recorridos diferentes

http://www.math.uwaterloo.ca/tsp/
https://developers.google.com/optimization/routing/tsp
Aplicaciones:
https://dev.routific.com/use-cases/travelling-salesman-problem
http://www.math.uwaterloo.ca/tsp/apps/index.html

3.2-67

4 Limitaciones de la Notación Asintótica

¿Qué limitaciones tiene el análisis $\mathcal{O}$?

- No es apropiado para pequeñas cantidades de datos. Con pequeñas entradas es mejor el algoritmo más simple.
- La **constante implícita en la notación** $\mathcal{O}$ puede resultar demasiado grande en la práctica.
  - Por ejemplo, un algoritmo $2nlog(n)$ puede ser mejor que otro $1000n$, aunque su tasa de crecimiento sea mayor
- Las constantes grandes pueden entrar en juego cuando el algoritmo es muy complejo
- En nuestro análisis no tenemos en cuenta las constantes, ni el tiempo de operaciones como:
  - Accesos a memoria (que no son costosos)
  - Accesos a disco (muchos miles de veces más costosos)
  - Falta de memoria
- Hay algoritmos para los que la cota en el **caso peor** es una sobreestimación
- Las cotas para el **caso medio** son difíciles de obtener, incluso hay problemas para los que aún no se ha podido calcular (ShellSort)

3.2-68

Conclusiones

1. El **tiempo de ejecución** de la mayoría de los algoritmos **depende de sus datos de entrada**.
2. En el análisis de algoritmos buscamos **eliminar** de algún modo esa **dependencia**, porque generalmente no conocemos cuáles serán los datos de entrada cada vez que se ejecute el programa.
3. Hemos estudiado la **eficiencia de los algoritmos en el caso peor**, y podemos decir que el número de veces que se van a ejecutar ciertas operaciones **es menor que una determinada función del tamaño de la entrada**, no importa cuál sea dicha entrada.
4. Se garantiza que el tiempo de ejecución del algoritmo será **menor que una cierta cota**.
5. Cuando hacemos el análisis de un algoritmo con la notación $\mathcal{O}$ no consideramos las **características** particulares de la **máquina** en la que se implementa.
6. La notación $\mathcal{O}$ es una forma de **categorizar** algoritmos.

3.2-69

En la próxima lección...

1. Estudiaremos **cómo analizar algoritmos recursivos**.
2. Diferentes métodos para estimar **el tiempo de ejecución** en algoritmos recursivos.

3.2-70

References

[1] [Básica] Weiss M.A. *Estructuras de datos en Java.* Capítulo 5: 5.4 - 5.8 Pearson

[2] [Básica] Brassard G., Bratley P. *Fundamentos de Algoritmia.* (capítulo 4: 4.1 - 4.4) Prentice Hall, 1997 /2004