

# TEMA 2

## La eficiencia de los algoritmos

PROGRAMACIÓN Y ESTRUCTURAS DE DATOS

## La eficiencia de los algoritmos

- 1. Noción de complejidad
  - Complejidad temporal, tamaño del problema y paso
- 2. Cotas de complejidad
  - Cota superior, inferior y promedio
- 3. Notación asintótica
  - $O$ ,  $\Omega$ ,  $\Theta$
- 4. Obtención de cotas de complejidad

# 1. Noción de complejidad

## DEFINICIÓN

- Cálculo de complejidad: determinación de dos parámetros o funciones de coste:
  - Complejidad espacial : Cantidad de recursos espaciales ( de almacén) que un algoritmo consume o necesita para su ejecución
  - Complejidad temporal : Cantidad de tiempo que un algoritmo necesita para su ejecución
- Posibilidad de hacer
  - Valoraciones
    - el algoritmo es: “bueno”, “el mejor”, “prohibitivo”
  - Comparaciones
    - el algoritmo A es mejor que el B

3

# 1. Noción de complejidad

## COMPLEJIDAD TEMPORAL

- Factores de complejidad temporal:
  - Externos
    - La máquina en la que se va a ejecutar
    - El compilador: variables y modelo de memoria
    - La experiencia del programador
  - Internos
    - El número de instrucciones asociadas al algoritmo
- Complejidad temporal :  $Tiempo(A) = C + f(T)$ 
  - $C$  es la contribución de los factores externos (constante)
  - $f(T)$  es una función que depende de  $T$  (talla o tamaño del problema)

4

# 1. Noción de complejidad

## COMPLEJIDAD TEMPORAL

- Talla o tamaño de un problema:
  - Valor o conjunto de valores asociados a la **entrada** del problema que representa una medida de su tamaño respecto de otras entradas posibles
- Paso de programa:
  - Secuencia de operaciones con contenido semántico cuyo coste es **independiente** de la talla del problema
  - Unidad de medida de la complejidad de un algoritmo
- Expresión de la complejidad temporal:
  - Función que expresa el número de pasos de programa que un algoritmo necesita ejecutar para cualquier entrada posible (para cualquier talla posible)
  - No se tienen en cuenta los factores externos

5

# 1. Noción de complejidad

## COMPLEJIDAD TEMPORAL. Ejemplos

```
int ejemplo1 (int n)
{
    n+ = n;
    return n;
}
```

 $f(\text{ejemplo1}) = 1 \text{ pasos}$ 

```
int ejemplo2 (int n)
{
    int i;
    for (i=0; i ≤ 2000; i++)
        n+ = n;
    return n;
}
```

 $f(\text{ejemplo2}) = 1 \text{ pasos}$ 

6

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

# 1. Noción de complejidad

## COMPLEJIDAD TEMPORAL. Ejemplos

```
int ejemplo3 (int n)
{
  int i, j;
  j = 2;
  for (i=0; i ≤ 2000; i++)
    j=j*j;
  for (i=0; i ≤ n; i++)
  {
    j = j + j;
    j = j - 2;
  }
  return j;
}
```

$$f(\text{ejemplo3}) = 1 + 1 \cdot (n + 1) \text{ pasos}$$

7

# 1. Noción de complejidad

## COMPLEJIDAD TEMPORAL. Ejemplos

```
int ejemplo4 (int n)
{
  int i, j, k;
  k = 1;
  for (i=0; i ≤ n; i++)
    for (j=1; j ≤ n; j++)
      k = k + k;
  return k;
}
```

$$f(\text{ejemplo4}) = 1 + 1 \cdot n \cdot (n + 1) \text{ pasos}$$

```
int ejemplo5 (int n)
{
  int i, j, k;
  k = 1;
  for (i=0; i ≤ n; i++)
    for (j=i; j ≤ n; j++)
      k = k + k;
  return k;
}
```

$$f(\text{ejemplo5}) = 1 + \sum_{i=0..n} (\sum_{j=i..n} 1) \text{ pasos}$$

8

$$\sum_{i=0}^n \left( \sum_{j=0}^n 1 \right) = \sum_{i=0}^n (n-1+1) = (n+1) + n + (n-1) = \dots + 1 = \sum_{i=1}^{n+1} i = \frac{(n+1)(n+2)}{2}$$

$$\textcircled{1} \sum_{i=0}^{n-1} 1 = (n-1) - 0 + 1 = n \quad \textcircled{2} \sum_{i=0}^{n-1} \left( \sum_{j=1}^i 1 + 1 \right) = \sum_{i=0}^{n-1} (i+1) = \sum_{i=1}^n i = \frac{n(n+1)}{2} \quad \textcircled{3} \sum_{i=4}^{n-1} \left( \sum_{j=i-3}^i 1 + 1 \right) = \sum_{i=4}^{n-1} (4+1) = 5(n-1-4+1)$$

$\underbrace{i - (i-3) + 1}_{4}$

# 1. Noción de complejidad

## COMPLEJIDAD TEMPORAL. Ejercicios

- ① `for(i = sum = 0; i < n; i++) sum += a[i];`
- ② `for(i = 0; i < n; i++) {  
  for(j = 1, sum = a[0]; j <= i; j++) sum += a[j];  
  cout << "La suma del subarray " << i << " es " << sum << endl; }`
- ③ `for(i = 4; i < n; i++) {  
  for(j = i-3, sum = a[i-4]; j <= i; j++) sum += a[j];  
  cout << "La suma del subarray " << i-4 << " es " << sum << endl; }`
- ④ `for(i = 0, length = 1; i < n-1; i++) {  
  for(i1 = i2 = k = i; k < n-1 && a[k] < a[k+1]; k++, i2++);  
  if(length < i2 - i1 + 1) length = i2 - i1 + 1; }`

9

# 1. Noción de complejidad

## CONCLUSIONES

- Sólo nos ocuparemos de la complejidad temporal
- Normalmente son objetivos contrapuestos  
(complejidad temporal <--> complejidad espacial)
- Cálculo de la complejidad temporal:
  - *a priori*: contando pasos
  - *a posteriori*: generando instancias para distintos valores y cronometrando el tiempo
- Se trata de obtener la función. Las unidades de medida (paso, sg, msg, ...) no son relevantes (todo se traduce a un cambio de escala)
- El nº de pasos que se ejecutan siempre es función del tamaño (o talla) del problema

10

## 2. Cotas de complejidad

### INTRODUCCIÓN

- Dado un vector  $X$  de  $n$  números naturales y dado un número natural  $z$ :
  - encontrar el índice  $i$ :  $X_i = z$
  - Calcular el número de pasos que realiza

```

funcion BUSCAR (var X:vector[N]; z: N): devuelve N
var i:natural   fvar;
comienzo
  i:=1;
  mientras (i ≤ |X|) ∧ (Xi≠z) hacer
    i:=i+1;
  fmientras
  si i= |X|+1 entonces devuelve 0    (*No encontrado*)
    si_no devuelve i
fin

```

11

## 2. Cotas de complejidad

### LA SOLUCIÓN: cotas de complejidad

- Cuando aparecen diferentes casos para una misma talla genérica  $n$ , se introducen las cotas de complejidad:
  - Caso peor: cota superior del algoritmo  $\rightarrow C_s(n)$
  - Caso mejor: cota inferior del algoritmo  $\rightarrow C_i(n)$
  - Término medio: cota promedio  $\rightarrow C_m(n)$
- Todas son funciones del tamaño del problema ( $n$ )
- La cota promedio es difícil de evaluar *a priori*
  - Es necesario conocer la distribución de la probabilidad de entrada
  - No es la media de la inferior y de la superior (ni están todas ni tienen la misma proporción)

12

## 2. Cotas de complejidad

EJERCICIO: cotas superior e inferior

```

funcion BUSCAR (var X:vector[N]; z: N): devuelve N
var i:natural fvar;
comienzo
  i:=1;
  mientras (i ≤ |X|) ∧ (Xi≠z) hacer
    i:=i+1;
  fmientras
  si i= |X|+1 entonces devuelve 0    (*No encontrado*)
  si_no devuelve i
fin
  
```

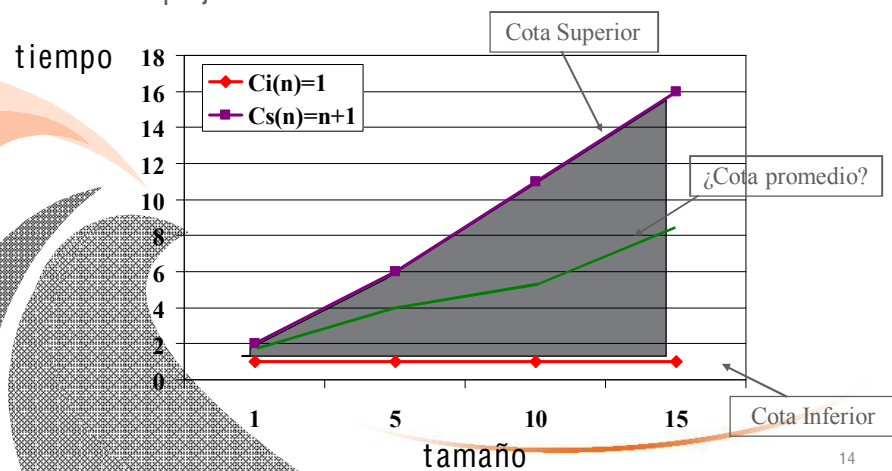
- Tamaño del problema: n° de elementos de X:  $n$
- ¿Existe caso mejor y caso peor?
  - Caso mejor: el elemento está el primero:  $X_1=z \rightarrow c_i(n) = 1$
  - Caso peor: el elemento no está:  $\forall i \ 1 \leq i \leq |X|, X_i \neq z \rightarrow c_s(n) = n+1$

13

## 2. Cotas de complejidad

EJERCICIO: cotas superior e inferior

- Complejidad función Buscar



14

## 2. Cotas de complejidad

### CONCLUSIONES

- La cota promedio no la calcularemos. Sólo se hablará de complejidad por término medio cuando la cota superior y la inferior coinciden
- El estudio de la complejidad se hace para tamaños grandes del problema por varios motivos:
  - Los resultados para tamaños pequeños o no son fiables o proporcionan poca información sobre el algoritmo
  - Es lógico invertir tiempo en el desarrollo de un buen algoritmo sólo si se prevé que éste realizará un gran volumen de operaciones

A la complejidad que resulta de tamaños grandes de problema se le denomina complejidad asintótica y la notación utilizada es la notación asintótica

15

## 3. Notación asintótica

### INTRODUCCIÓN

- Notación matemática utilizada para representar la complejidad espacial y temporal cuando  $n \rightarrow \infty$
- Se definen tres tipos de notación:
  - Notación  $O$  (big-omicron)  $\Rightarrow$  caso peor
  - Notación  $\Omega$  (omega)  $\Rightarrow$  caso mejor
  - Notación  $\Theta$  (big-theta)  $\Rightarrow$  caso promedio

16



### 3. Notación asintótica

Teorema de la escala de complejidad

$$O(1) \subset O(\lg \lg n) \subset O(\lg n) \subset O(\lg^{a>1} n) \subset O(\sqrt{n}) \subset O(n) \subset O(n \lg n) \subset O(n^2) \subset \dots \subset O(n^{a>2}) \subset O(2^n) \subset O(n!) \subset O(n^n)$$

□  $f(n) + g(n) + t(n) \in O(\text{Max}(f(n), g(n), t(n)))$

□ Ejemplos:

- $n + 1$  pertenece a  $O(n)$
- $n^2 + \log n$  pertenece a  $O(n^2)$
- $n^3 + 2^n + n \log n$  pertenece a  $O(2^n)$

□ Válido para Notación  $\Omega$  y Notación  $\Theta$

17

$O(1) = \text{CONSTANTE} = \text{TAILLA}$   
 $O(n) = \text{LINEAL} = \text{FOR}(i=1; i \leq n; i++)$   
 $O(n^2) = \text{CUADRÁTICA} = \text{FOR FOR}$   
 $O(\log[n]) = \text{LOGARÍTMICA} = \text{TAILLA} / 2$   
 $O(2^n) = \text{EXPONENCIAL} = \text{TAILLA} * 2$

### 3. Notación asintótica

NOTACIÓN O: escala de complejidad

Complejidad	n = 32	n = 64
$n^3$	3 seg.	26 seg.
$2^n$	5 días	58.10 <sup>6</sup> años

- Tiempos de respuesta para dos valores de la talla y complejidades  $n^3$  y  $2^n$ .  
(paso = 0'1 mseg.)

– Queda clara la necesidad del cálculo de complejidad

```

función POT_2 (n: natural): natural
opción
  n = 1: devuelve 2
  n > 1: devuelve 2 * POT_2(n-1)
fopción
ffunción
  
```

Coste lineal  
1 seg.

```

función POT_2 (n: natural): natural
opción
  n = 1: devuelve 2
  n > 1: devuelve POT_2(n-1)+POT_2(n-1)
fopción
ffunción
  
```

Coste exponencial  
miles de años

18

## 4. Obtención de cotas de complejidad

### INTRODUCCIÓN

- Etapas para obtener las cotas de complejidad:
  1. Determinación de la talla o tamaño (de la instancia) del problema
  2. Determinación del caso mejor y peor: instancias para las que el algoritmo tarda más o menos
    - No siempre existe mejor y peor caso ya que existen algoritmos que se comportan de igual forma para cualquier instancia del mismo tamaño
  3. Obtención de las cotas para cada caso. Métodos:
    - Conteo de pasos
    - Relaciones de recurrencia (funciones recursivas)

19

## 4. Obtención de cotas de complejidad

### INTRODUCCIÓN

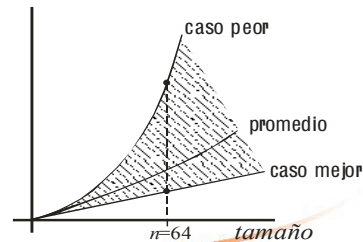
**función** FACTORIAL (n:natural): natural

- La talla es  $n$  y no existe caso mejor ni peor

**función** BUSCA (v: vector[natural]; x:natural)

- La talla es  $n=|v|$
- caso mejor: instancias donde  $x$  está en  $v[1]$
- caso peor: instancias donde  $x$  no está en  $v$

- Se trata de delimitar con una región el tiempo que tarda un algoritmo en ejecutarse



20

## 4. Obtención de cotas de complejidad

### Ejemplos

#### Cálculo del máximo de un vector

```

funcion MÁXIMO (var v : vector[n]; n:entero) : entero
var i, max : entero fvar
comienzo
  max:=v[1]
  para i:=2 hasta n hacer
    si v[i]>max entonces max:=v[i] fsi
  fpara
  devuelve max
fin

```

• determinar la talla del problema:  $n = \text{tamaño del vector}$

$$\left. \begin{array}{l}
 \text{• mejor caso} \quad c_i = 1 + \sum_{i=2}^n 1 = 1 + (n - 2 + 1) = n \in \Omega(n) \\
 \text{• peor caso} \quad c_s = 1 + \sum_{i=2}^n 2 = 1 + (n - 2 + 1) \cdot 2 = 2n - 1 \in O(n)
 \end{array} \right\} \in \Theta(n)$$

21

## 4. Obtención de cotas de complejidad

### Ejemplos

#### Búsqueda de un elemento en un vector ordenado (Búsqueda binaria)

```

funcion BUSCA (var v:vector[N]; x,pri,ult: natural): natural
var m: natural fvar
comienzo
  repetir
    m:= (pri+ult)/2
    si v[m]>x entonces ult:= m-1
    sino pri:= m+1
  fsi
  hasta (pri>ult) v v[m]=x
  si v[m]=x entonces devuelve m
  sino devuelve 0
  fsi
fin

```

22

## 4. Obtención de cotas de complejidad

### Ejemplos

- Determinar la talla del problema:  $n = \text{tamaño del vector}$
- *Mejor caso*:  $x$  está en la mitad del vector
- *Peor caso*:  $x$  no está en el vector
- *Complejidades*
  - *mejor caso*:  $1+1=2 \in \Omega(1)$
  - *peor caso*
    - $1+k \cdot 1$ , donde  $k$  es el  $n^\circ$  de veces que se ejecuta el bucle
    - 1ª iteración: Talla= $n$
    - 2ª iteración: Talla= $n/2$
    - 3ª iteración: Talla= $n/4$
    - .....
    - $k$ -ésima iteración: Talla= $n/2^{(k-1)}$
    - .....
    - última iteración: Talla = 1 ( $n/2^{(\text{última}-1)} = 1$ )
- Es decir, en la última iteración sólo nos queda 1 elemento
- Despejando última:
  - $n = 2^{\text{última}-1}; \log_2 n + 1 = \text{última};$
  - $1 + \text{última} - 1 = 1 + (\log_2 n + 1) \in O(\log_2 n)$

23

## 4. Obtención de cotas de complejidad

### Algoritmos de ordenación

- Directos
  - Inserción directa
  - Inserción binaria
  - Selección directa
  - Intercambio directo (burbuja)

24

## 4. Obtención de cotas de complejidad

### Algoritmos de ordenación

#### INSERCIÓN DIRECTA

- Divide lógicamente el vector en dos partes: origen y destino
- Comienzo:
  - *destino* tiene el primer elemento del vector
  - *origen* tiene los  $n-1$  elementos restantes
- Se va tomando el primer elemento de *origen* y se inserta en *destino* en el lugar adecuado, de forma que *destino* siempre está ordenado
- El algoritmo finaliza cuando no quedan elementos en *origen*

#### Características

- caso mejor: vector ordenado ascendentemente
- caso peor: vector ordenado inversamente

25

## 4. Obtención de cotas de complejidad

### Algoritmos de ordenación

#### INSERCIÓN DIRECTA

```

funcion INSERCIÓN_DIRECTA (var a:vector[natural]; n: natural)
var i,j: entero; x:natural fvar
comienzo
  para i:=2 hasta n hacer LINEAL
    x:=a[i]; j:=i-1
    mientras (j>0) ^ (a[j]>x) hacer CASO MEJOR: O(N) DE N : LINEAL
      a[j+1]:=a[j] CASO PEOR: CUADRÁTICA.
      j:=j-1
    fmientras
    a[j+1]:=x
  fpara
fin
  
```

26

## 4. Obtención de cotas de complejidad

### Algoritmos de ordenación

#### INSERCIÓN BINARIA

```

funcion INSERCIÓN_BINARIA (var a:vector[natural]; n: natural)
var i, j, iz, de, m: entero; x:natural fvar
comienzo
  para i:=2 hasta n hacer
    x:=a[i]; iz:=1; de:=i-1
    mientras (iz<de) hacer
      m:= (iz+de)/2
      si a[m]>x entonces de:= m-1
      sino iz:=m+1
    fsi
    fmientras
    para j:=i-1 hasta iz hacer (*decreciente*)
      a[j+1]:=a[j]
    fpara
    a[iz]:=x
  fpara
fin

```

27

## 4. Obtención de cotas de complejidad

### Algoritmos de ordenación

#### SELECCIÓN DIRECTA

```

funcion SELECCIÓN_DIRECTA (var a:vector[natural]; n:natural)
var i, j, posmin: entero; min:natural fvar
comienzo
  para i:=1 hasta n-1 hacer
    min:=a[i]; posmin:=i
    para j:=i+1 hasta n hacer
      si a[j]<min entonces
        min:=a[j]; posmin:=j
    fsi
    fpara
    a[posmin]:=a[i]; a[i]:=min
  fpara
fin

```

28

## 4. Obtención de cotas de complejidad

### Algoritmos de ordenación

#### INTERCAMBIO DIRECTO (Burbuja)

```
funcion INTERCAMBIO_DIRECTO (var a:vector[natural]; n:natural )  
var i,j:entero fvar  
comienzo  
  para i:=2 hasta n hacer  
    para j:=n hasta i hacer  
      si a[j]<a[j-1] entonces  
        SWAP(a[j],a[j-1])  
      fsi  
    fpara  
  fpara  
fin
```