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, Ω, Θ
- 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

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)

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

1. Noción de complejidad

COMPLEJIDAD TEMPORAL. Ejemplos

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

f (ejemplo1) = 1 pasos

Estas instrucciones se ejecutan siempre el mismo número de veces. Sea cual sea el valor de n. (1 paso)

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

f (ejemplo2) = 1 pasos

1. Noción de complejidad

COMPLEJIDAD TEMPORAL. Ejemplos

1 3 2

$f\,(ejemplo3) = 1 + 1 \cdot (n + 1)$ *pasos*

```
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;
}
```

1

Estas instrucciones se ejecutan siempre el mismo número de veces. Sea cual sea el valor de n. (1 paso)

2

El bucle for se repetirá n+1 veces.

3

Estas dos instrucciones dentro del bucle for siempre se van a repetir el mismo número de veces, por lo que se consideran 1 paso

1. Noción de complejidad

COMPLEJIDAD TEMPORAL. Ejemplos

1 4 3 2

$f\,(ejemplo4) = 1 + 1 \cdot n \cdot (n + 1)$ *pasos*

```
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;
}
```

1

Estas instrucciones se ejecutan siempre el mismo número de veces. Sea cual sea el valor de n. (1 paso)

2

El bucle for i=0; i<=n; i++) se repite n+1 veces

3

El bucle for (j=1; j<=n; j++) se repite n veces

4

La instrucción k=k+k es 1 paso

```
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;
}
```

$f\,(ejemplo5) = 1 + \Sigma_{i=0..n}(\Sigma_{j=i..n}\ 1)$ *pasos*

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;
}
```

$f\,(ejemplo4) = 1 + 1 \cdot n \cdot (n+1)$ *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;
}
```

$f\,(ejemplo5) = 1 + \Sigma_{i=0..n}(\Sigma_{j=i..n}\,1)$ *pasos*

1. Estas instrucciones se ejecutan siempre el mismo número de veces. Sea cual sea el valor de n. (1 paso)
2. El bucle for i=0; i<=n; i++) se repite n+1 veces
3. El bucle for (j=i; j<=n; j++) se repite n-i +1 veces
4. La instrucción k=k+k es 1 paso

1. Noción de complejidad

COMPLEJIDAD TEMPORAL. Ejemplos

```
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;
}
```

1 3 2

$f\,(ejemplo5) = 1 + \Sigma_{i=0..n}(\Sigma_{j=i..n}\,1)$ *pasos*

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

3 $\sum_{i=0}^{n} 1 \cdot (n-i+1) =$

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

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

Resolución de sumatorios:

$$\sum_{i=m}^{n} C = C \cdot (n-m+1)$$

$$\sum_{i=1}^{n} i = \frac{(a_1 + a_n)n}{2}\,(S.P.A)$$

S.P.A
- a_1=n+1
- a_n=1
- nº términos=n+1

1. Noción de complejidad

COMPLEJIDAD TEMPORAL. Ejercicios

```
for(i = sum = 0; i < n; i++) sum += a[i];
```

```
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; }
```

```
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; }
```

```
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; }
```


1. Noción de complejidad

COMPLEJIDAD TEMPORAL. Solución (I)

```
for(i = sum = 0; i < n; i++) sum += a[i];
```

$\Sigma_{i=0..n-1}$ *1 pasos* = *n* · *1 pasos* = ***O(n)***

$$\sum_{i=0}^{n-1} 1 = 1 \cdot (n-1-0+1) = n$$

Límite superior: $n-1$

Límite inferior: 0

```
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; }
```

El bucle exterior se repite *n* veces.El bucle interior se repite *i* veces, con *i* desde 1 hasta *n-1*. Por tanto:

Nº pasos: $\Sigma_{i=0..n-1}(1$ *[del cout]* $+ \Sigma_{j=1..i}(1)) = \Sigma_{i=0..n-1}(1 + i) =$

$(1 + n) * n / 2 = (n^2 + n) / 2$

Límite superior: i
Límite inferior: 1

Sumatorio=Límite superior – Límite inferior +1 =i-1+1 = i

Complejidad: *O(n^2)*

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

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

El número de veces que se ejecuta el bucle "mientras" dependerá del tamaño del vector y de la distribución interna de los elementos

1 paso

```
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
```

2. Cotas de complejidad

EL PROBLEMA

- No podemos contar el número de pasos porque depende:
  - del tamaño (TALLA) del problema |X|
  - de la **instancia** del problema que se pretende resolver (posible valor que puedan tomar las variables de entrada)
- Ejemplo:

Vector X	Elemento z	Nº PASOS
(0, 1)	1	
(1, 2, 3)	1	
(2)	3	
(1, 0, 2, 4)	3	
(1, 0, 2, 4)	0	
(1, 0, 2, 4)	1	

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)

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
```

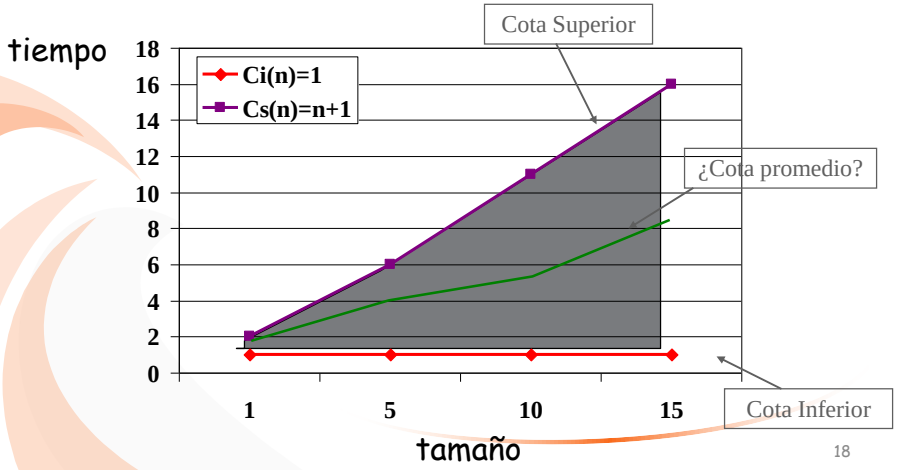
- Talla del problema: nº de elementos de X: *n*
- ¿Existe caso mejor y caso peor?
  - Caso mejor: el elemento está el primero: $X_1=z \rightarrow c_i(n) = 1$
  - Caso peor: el elemento no está: $\forall i\ 1 \leq i \leq |X|, Xi \neq z \rightarrow c_s(n) = n+1$

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

2. Cotas de complejidad

EJERCICIO: cotas superior e inferior

- Complejidad función "BUSCAR"



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

3. Notación asintótica

INTRODUCCIÓN

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

3. Notación asintótica

Teorema de la escala de complejidad

polilogarítmicas — logarítmicas — sublineal — lineal

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

$$\subset O(n\lg n) \subset O(n^2) \subset \cdots \subset O(n^{a>2}) \subset O(2^n) \subset O(n!) \subset O(n^n)$$

polinómicas — exponenciales

- f(n) + g(n) + t(n) ∈ O(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 Ω y Notación Θ

3. Notación asintótica

NOTACIÓN O: escala de complejidad

Complejidad	*n = 32*	*n = 64*
n^3	3 seg.	26 seg.
2^n	5 días	$58 \cdot 10^6$ 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

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 (**INSTANCIAS** de la TALLA)
     - 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:
     - **cuenta de pasos**
     - relaciones de recurrencia (**funciones recursivas**)

4. Obtención de cotas de complejidad

INTRODUCCIÓN

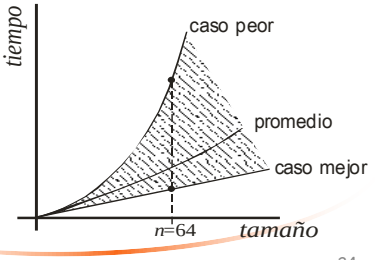
```
función FACTORIAL (n:natural): natural
```

- 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



4. Obtención de cotas de complejidad

Ejemplos: MÁXIMO de vector

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
```

1 paso

- determinar la **TALLA** del problema: ***n**=tamaño del vector*
- **mejor caso** $c_i = 1 + \sum_{i=2}^{n} 1 = 1 + (n-2+1) = n \in \Omega(n)$

La condición v[i]>max NUNCA se cumple

- **peor caso** $c_s = 1 + \sum_{i=2}^{n} 2 = 1 + (n-2+1)\cdot 2 = 2n-1 \in O(n)$

$\in \Theta(n)$

La condición v[i]>max SIEMPRE se cumple

4. Obtención de cotas de complejidad

Ejemplos : BUSQUEDA BINARIA en vector

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
```

4. Obtención de cotas de complejidad

Ejemplos : BUSQUEDA BINARIA en vector

- Determinar la **TALLA** del problema: ***n***=*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 nº de veces que se ejecuta el bucle*
      - *1ª iteración: Talla* **= ult-pri+1** $= n$
      - *2ª iteración: Talla* **= ult-pri+1** $= n/2$
      - *3ª iteración: Talla* **= ult-pri+1** $= n/4$
      - . . . . . .
      - *k-ésima interación: Talla* **= ult-pri+1** $= n/2^{(k-1)}$
      - . . . . . .
      - *última iteración: Talla* **= ult-pri+1** $= n/2^{(\text{última}-1)} = 1$
    - *Es decir, en la última iteración sólo nos queda 1 elemento*
      Despejando última:
      $n=2^{\text{ultima}-1} \rightarrow \log_2 n = \text{última} - 1 \rightarrow \log_2 n+1 = \textbf{última}$
      $1+ \text{última} \cdot 1 = 1+(\log_2 n+1) \in O(\log_2 n)$

4. Obtención de cotas de complejidad

Algoritmos de ordenación

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

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

4. Obtención de cotas de complejidad

Algoritmos de ordenación

INSERCIÓN DIRECTA

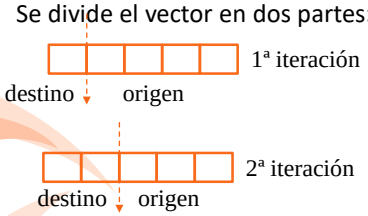
```
funcion  INSERCION_DIRECTA  (var a:vector[natural];  n: natural)
var i,j: entero;  x:natural  fvar
comienzo
   para  i:=2  hasta  n  hacer
      x:=a[i];  j:=i-1
      mientras   (j>0) ∧ (a[j]>x)   hacer
         a[j+1]:=a[j]
         j:=j-1
      fmientras
      a[j+1]:=x
   fpara
fin
```

4. Obtención de cotas de complejidad

Algoritmos de ordenación

INSERCIÓN DIRECTA

- Se divide el vector en dos partes: origen y destino



- En cada iteración se coloca el elemento a[i] del subvector "origen" en su posición correcta del subvector "destino" a[1..i-1]

4. Obtención de cotas de complejidad

Algoritmos de ordenación

INSERCIÓN BINARIA

EL BUCLE mientras BUSCA EL SITIO DEL PRIMER ELEMENTO DE **ORIGEN** EN **DESTINO**.

EL COSTE DE ESTE BUCLE ES **logn** (YA QUE LO HACE CON UNA BÚSQUEDA BINARIA).

```
funcion  INSERCION_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
```

4. Obtención de cotas de complejidad

Algoritmos de ordenación

INSERCIÓN BINARIA

- Es una mejora del algoritmo de inserción directa
- Cambia en un punto:
  - Cuando busca la posición donde debe ir el elemento en el subvector "destino" lo hace de forma dicotómica. Es decir, dividiendo el vector "destino" en dos mitades sucesivamente hasta encontrar la posición correcta.
  - Cuando encuentra la posición mueve los restantes elementos hacia la derecha

4. Obtención de cotas de complejidad

Algoritmos de ordenación

SELECCIÓN DIRECTA

EN CADA ITERACIÓN, SELECCIONA EL MÍNIMO DE **ORIGEN** Y LO INTERCAMBIA POR EL ÚLTIMO DE **DESTINO**.

```
funcion  SELECCION_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
```

Este algoritmo busca el menor elemento del subvector a[i..n-1] y lo intercambia con el elemento que está en la posición i

4. Obtención de cotas de complejidad

Algoritmos de ordenación

INTERCAMBIO DIRECTO (Burbuja)

Ir recorriendo el vector de derecha a izquierda **n-i** veces e ir intercambiando cada elemento con el anterior si es que es menor.

```
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
```