EFICIENCIA DE ALGORITMOS

Aránzazu Jurío

ALGORITMIA

2017/2018

Índice

- Eficiencia y complejidad
- Cotas
  - Cota superior. Notación O
  - Cota inferior. Notación Ω
  - Orden exacto. Notación Θ
- Ecuaciones en recurrencia

Índice

- Eficiencia y complejidad
- Cotas
  - Cota superior. Notación O
  - Cota inferior. Notación Ω
  - Orden exacto. Notación Θ
- Ecuaciones en recurrencia

Eficiencia y complejidad

- Para resolver un problema → diseño de algoritmo
- Varios algoritmos pueden resolver el mismo problema. ¿Cuál es mejor?
  - Calcular el máximo común divisor

- funcion1(a,b)
  - m=minimo(a,b)
  - para i=1 hasta m
    - si a%i=0 y b%i=0
      - maximo=i
    - fin si
  - fin para
- fin funcion

- funcion2(a,b)
  - m=minimo(a,b)
  - i=m
  - mientras (a%i≠0 o b%i≠0) y i≥1 hacer
    - i=i-1
  - fin mientras
  - maximo=i
- fin funcion

Eficiencia y complejidad

- Criterios para medir el rendimiento
  - Simplicidad
    - Facilita la verificación
    - Facilita el estudio de la eficiencia
    - Facilita el mantenimiento
  - Uso eficiente de recursos
    - Espacio: memoria que utiliza
    - Tiempo: tiempo que tarda en ejecutarse
    - Sirve además para comparar varios algoritmos entre sí
  - Muchas veces, es mejor buscar sencillez y legibilidad en el código que soluciones más crípticas y eficientes

Eficiencia y complejidad

- Tiempo de ejecución
  - Medida real (empírica) a posteriori
    - Escribir un programa que implemente un algoritmo, en un lenguaje dado
    - Seleccionar un conjunto de datos de entrada
    - Ejecutar el programa en un ordenador fijado, con esos datos de entrada
    - Medir el tiempo que tarda en ejecutarse
  - Medida teórica a priori
    - Utilizar una descripción de alto nivel del algoritmo (pseudocódigo)
    - Determinar matemáticamente la cantidad de recursos necesarios para ejecutar el algoritmo
    - Obtener una función genérica que permita hacer predicciones sobre la utilización de recursos, en función del tamaño de la entrada

Eficiencia y complejidad

- Tiempo de ejecución
  - El estudio real requiere la implementación del algoritmo. La medida teórica sólo requiere de una descripción en pseudocódigo
  - En el estudio real no tenemos la seguridad de los recursos que realmente se van a consumir si cambian las entradas
  - En el estudio real los resultados solo son válidos para unas determinadas condiciones de ejecución. Es difícil extrapolar los resultados si se producen cambios en el hardware, sistema operativo, lenguaje utilizado… El estudio teórico es independiente de las condiciones de ejecución
  - El estudio real permite hacer una evaluación experimental de los recursos consumidos que no es posible con el estudio teórico

Eficiencia y complejidad

- Medida teórica: Obtener una función que acote (por arriba o por abajo) el tiempo de ejecución del algoritmo para unos valores de entrada dados → Notación asintótica
  - Tamaño de entrada: número de componentes sobre los que se va a ejecutar el algoritmo
  - Unidad de tiempo: no hay un ordenador estándar, así que trabajamos con número de instrucciones ejecutadas en un ordenador idealizado

- Denotaremos por $T(n)$ el tiempo de ejecución de un algoritmo para una entrada de tamaño n

Notación asintótica

- Principio de Invarianza

Dado un algoritmo y dos implementaciones suyas I_1 e I_2, que tardan $T_1(n)$ y $T_2(n)$ segundos respectivamente, existe una constante real $c > 0$ y un número natural n_0 tales que para todo $n \geq n_0$ se verifica que $T_1(n) \leq cT_2(n)$.

  - El tiempo de ejecución de dos implementaciones distintas de un algoritmo no va a diferir más que en una constante multiplicativa, si el tamaño de entrada es suficientemente grande

Notación asintótica

- Principio de Invarianza
  - Hay que tener en cuenta la c y la n_0 específicas de cada caso
    - En principio, un algoritmo de orden cuadrático es mejor que uno de orden cúbico
    - Ejemplo
      - Algoritmo 1: $T_1(n) = 10^6 n^2$
      - Algoritmo 2: $T_2(n) = 5n^3$
      - Algoritmo 1 sólo es mejor que Algoritmo 2 si el tamaño de entrada es superior a 200.000

Notación asintótica

- Medimos el tiempo en función del número de operaciones elementales (OE):
  - Operaciones aritméticas básicas
  - Asignaciones de variables de tipos predefinidos
  - Llamadas a funciones
  - Retorno de funciones
  - Comparaciones lógicas
  - Acceso a vectores o matrices

Notación asintótica

- Contar el número de operaciones elementales

```
funcion cambiar(t: array[1..n] de enteros; x,y: entero)
    var j:entero
    begin
        z:=t[x];
        t[x]:=t[y];
        t[y]:=z;
    end
```

Notación asintótica

- Contar el número de operaciones elementales

```
funcion cambiar(t: array[1..n] de enteros; x,y: entero)
    var j:entero
    begin
        z:=t[x];                    (2 OE)
        t[x]:=t[y];                 (3 OE)
        t[y]:=z;                    (2 OE)
    end
```

Notación asintótica

- El comportamiento de un algoritmo no sólo depende del tamaño de entrada, sino también de los valores de entrada en sí
- Una vez fijado el tamaño de la entrada, distinguimos tres estudios:
  - Caso mejor: traza (secuencia de sentencias) del algoritmo que realiza menos instrucciones
  - Caso peor: traza del algoritmo que realiza más instrucciones
  - Caso medio: traza promedio del algoritmo. Tiene en cuenta las probabilidades de que ocurran cada una de las trazas

Notación asintótica

- Definición: Sea D_n el conjunto de datos de entrada de tamaño n para un algoritmo, y sea $I \in D_n$ un elemento cualquiera. Sea $t(I)$ la cantidad de trabajo realizado por el algoritmo para procesar la entrada I. Se definen entonces las siguientes funciones:
  - Complejidad del peor caso: $W(n) = \max\{t(I): I \in D_n\}$
  - Complejidad del mejor caso: $B(n) = \min\{t(I): I \in D_n\}$
  - Complejidad del caso promedio: $A(n) = \sum_{i \in D_n} \Pr(I)\, t(I)$, donde $\Pr(I)$ es la probabilidad de que ocurra la entrada I

Notación asintótica

- Identificamos el número de Operaciones Elementales

```
funcion Buscar (a: array[1..n] de enteros; c: entero)
    var j:entero
    begin
        j:=1;
        mientras (a[j] < c) y (j < n) hacer
            j:=j+1;
        fin mientras
        si a[j] = c entonces
            devolver j
        si no
            devolver 0
        fin si
    fin buscar
```

Notación asintótica

- Identificamos el número de Operaciones Elementales

```
funcion Buscar (a: array[1..n] de enteros; c: entero)
    var j:entero
    begin
        j:=1;                                          (1 OE)
        mientras (a[j] < c) y (j < n) hacer            (4 OE)
            j:=j+1;                                    (2 OE)
        fin mientras
        si a[j] = c entonces                           (2 OE)
            devolver j                                 (1 OE)
        si no
            devolver 0                                 (1 OE)
        fin si
    fin buscar
```

Notación asintótica

- Estudiamos el caso mejor

```
funcion Buscar (a: array[1..n] de enteros; c: entero)
    var j:entero
    begin
        j:=1;
        mientras (a[j] < c) y (j < n) hacer
            j:=j+1;
        fin mientras
        si a[j] = c entonces
            devolver j
        si no
            devolver 0
        fin si
    fin buscar
```

Notación asintótica

- Estudiamos el caso mejor

```
funcion Buscar (a: array[1..n] de enteros; c: entero)
    var j:entero
    begin
        j:=1;                                          (1 OE)
        mientras (a[j] < c) y (j < n) hacer            (2 OE)
            j:=j+1;
        fin mientras
        si a[j] = c entonces                           (2 OE)
            devolver j                                 (1 OE)
        si no
            devolver 0
        fin si
    fin buscar
```

Notación asintótica

- Estudiamos el caso mejor
  - $T(n) = 1 + 2 + 2 + 1 = 6$
  - Nota: en el bucle sólo se ejecuta la primera mitad, y como no se cumple, se continúa después del bucle

Notación asintótica

- Estudiamos el caso peor

```
funcion Buscar (a: array[1..n] de enteros; c: entero)
    var j:entero
    begin
        j:=1;
        mientras (a[j] < c) y (j < n) hacer
            j:=j+1;
        fin mientras
        si a[j] = c entonces
            devolver j
        si no
            devolver 0
        fin si
    fin buscar
```

Notación asintótica

- Estudiamos el caso peor

```
funcion Buscar (a: array[1..n] de enteros; c: entero)
    var j:entero
    begin
        j:=1;                                          (1 OE)
        mientras (a[j] < c) y (j < n) hacer            (4 OE)  n-1 veces + salir
            j:=j+1;                                    (2 OE)   n-1 veces
        fin mientras
        si a[j] = c entonces                           (2 OE)
            devolver j
        si no
            devolver 0                                 (1 OE)
        fin si
    fin buscar
```

Notación asintótica

- Estudiamos el caso peor
  - Bucle: $\sum_{i=1}^{n-1}(4+2)+4 = 6(n-1)+4$
  - Tiempo total

$$T(n) = 1 + 6(n-1) + 4 + 2 + 1 = 6n + 2$$

Notación asintótica

- Estudiamos el caso medio

```
funcion Buscar (a: array[1..n] de enteros; c: entero)
    var j:entero
    begin
        j:=1;
        mientras (a[j] < c) y (j < n) hacer
            j:=j+1;
        fin mientras
        si a[j] = c entonces
            devolver j
        si no
            devolver 0
        fin si
    fin buscar
```

Notación asintótica

- Estudiamos el caso medio (todos equiprobables)

```
funcion Buscar (a: array[1..n] de enteros; c: entero)
    var j:entero
    begin
        j:=1;                                          (1 OE)
        mientras (a[j] < c) y (j < n) hacer            (4 OE) (n-1)/2 veces + salir
            j:=j+1;                                    (2 OE)  (n-1)/2 veces
        fin mientras
        si a[j] = c entonces                           (2 OE)
            devolver j
        si no                                          } (1 OE)
            devolver 0
        fin si
    fin buscar
```

Notación asintótica

- Estudiamos el caso medio (todos equiprobables)
  - Bucle: $\sum_{i=1}^{(n-1)/2}(4+2)+4=\frac{6(n-1)}{2}+2=3(n-1)+4$
  - Tiempo total

$$T(n)=1+3(n-1)+2+2+1=3n+3$$

Nota: en este caso, para salir del bucle sólo se cuenta la primera parte: 2OE

Ejercicio

- Dado el siguiente algoritmo, calcula los tiempos de ejecución en el caso mejor, peor y caso medio.

```
algoritmo uno (ENT-SAL a:vector)
variables i,j,temp: entero
principio
        para i=1 a n-1 hacer
                para j=n a i+1 hacer
                        si a[j-1]>a[j] entonces
                                temp ← a[j-1]
                                a[j-1] ← a[j]
                                a[j] ← temp
                        fin si
                fin para
        fin para
fin
```

Índice

- Eficiencia y complejidad
- Cotas
  - Cota superior. Notación O
  - Cota inferior. Notación Ω
  - Orden exacto. Notación Θ
- Ecuaciones en recurrencia

Notación asintótica

- Ya sabemos cómo calcular el tiempo de ejecución de un algoritmo
- Ahora vamos a definir clases de equivalencia, correspondientes a funciones que “crecen de la misma forma”
- Así, vamos a acotar los algoritmos

Índice

- Eficiencia y complejidad
- Cotas
  - Cota superior. Notación O
  - Cota inferior. Notación Ω
  - Orden exacto. Notación Θ
- Ecuaciones en recurrencia

Cota superior. Notación O

- Dada una función f, queremos estudiar aquellas funciones g que a lo sumo crecen tan deprisa como f
- El conjunto de esas funciones se llaman cota superior de f y se denominan $O(f)$
- Podemos asegurar que, en ningún caso, el tiempo empleado será de un orden superior al de la cota

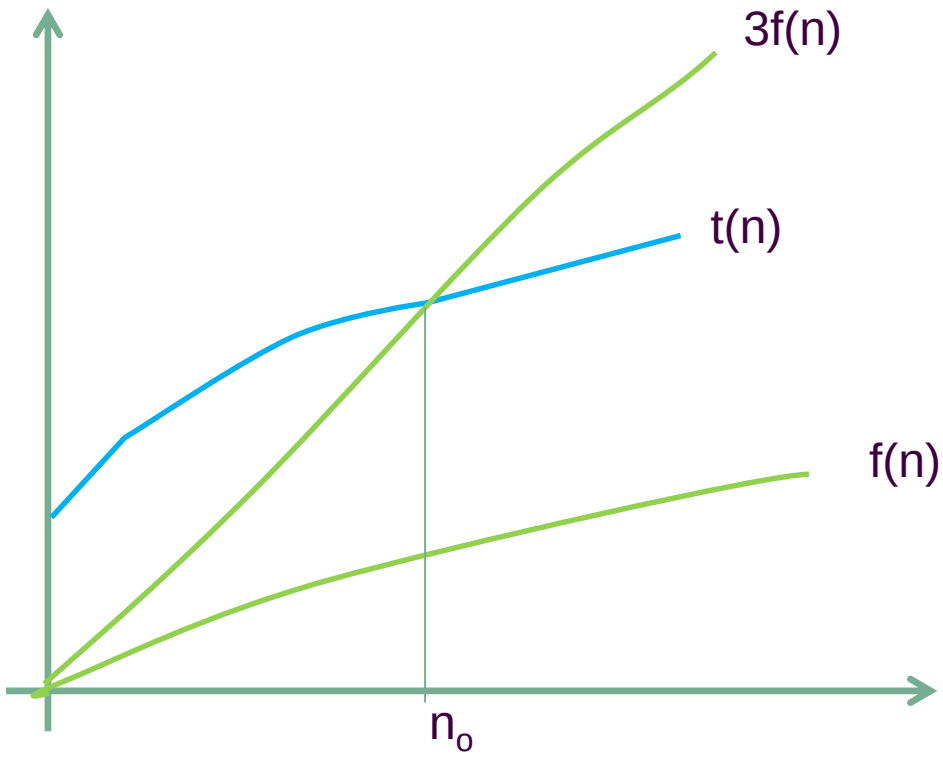
Cota superior. Notación O

Sea $f: \mathbb{N} \to [0, \infty)$. Se define el conjunto de funciones de orden O de f como:

$$O(f) = \{g: \mathbb{N} \to [0, \infty) | \exists c \in \mathbb{R}^+, \exists n_0 \in \mathbb{N}; \\ g(n) \leq cf(n) \; \forall n \geq n_0\}$$

- Diremos que una función $t: \mathbb{N} \to [0, \infty)$ es de orden O de f si $t \in O(f)$
- Intuitivamente, $t \in O(f)$ indica que t está acotada superiormente por algún múltiplo de f. Normalmente estaremos interesados en la menor función f tal que t pertenezca a $O(f)$

Cota superior. Notación O



Cota superior. Notación O

- ¿Cuáles son las cotas superiores del tiempo de ejecución en los casos mejor, peor y medio?

```
funcion Buscar (a: array[1..n] de enteros; c: entero)
    var j:entero
    begin
        j:=1;
        mientras (a[j] < c) y (j < n) hacer
            j:=j+1;
        fin mientras
        si a[j] = c entonces
            devolver j
        si no
            devolver 0
        fin si
    fin buscar
```

Cota superior. Notación O

- Mejor O(1)
- peor O(n)
- Medio O(n)

```
funcion Buscar (a: array[1..n] de enteros; c: entero)
    var j:entero
    begin
        j:=1;
        mientras (a[j] < c) y (j < n) hacer
            j:=j+1;
        fin mientras
        si a[j] = c entonces
            devolver j
        si no
            devolver 0
        fin si
    fin buscar
```

Cota superior. Notación O

- Propiedades de O
  - $f \in O(f)$
  - $O(f) = O(g)$ si y sólo si $f \in O(g)$ y $g \in O(f)$
  - Si $f \in O(g)$ y $g \in O(h)$ entonces $f \in O(h)$
  - Si $f_1 \in O(g)$ y $f_2 \in O(h)$ entonces $f_1 + f_2 \in O(\max(g, h))$
  - Si $f_1 \in O(g)$ y $f_2 \in O(h)$ entonces $f_1 \cdot f_2 \in O(g \cdot h)$
  - Regla del límite
    - Si $lim_{n\to\infty}\left(\frac{f(n)}{g(n)}\right) \in \mathbb{R}^+$, entonces $f(n) \in O(g(n))$ y $g(n) \in O(f(n))$
    - Si $lim_{n\to\infty}\left(\frac{f(n)}{g(n)}\right) = 0$, entonces $f(n) \in O(g(n))$ pero $g(n) \notin O(f(n))$
    - Si $lim_{n\to\infty}\left(\frac{f(n)}{g(n)}\right) = +\infty$, entonces $f(n) \notin O(g(n))$ pero $g(n) \in O(f(n))$

Índice

- Eficiencia y complejidad
- Cotas
  - Cota superior. Notación O
  - Cota inferior. Notación Ω
  - Orden exacto. Notación Θ
- Ecuaciones en recurrencia

Cota inferior. Notación Ω

- Dada una función f, queremos estudiar aquellas funciones g que a lo sumo crecen tan lentamente como f
- El conjunto de esas funciones se llaman cota inferior de f y se denominan $\Omega(f)$
- Podemos asegurar que, en ningún caso, el tiempo empleado será de un orden inferior al de la cota

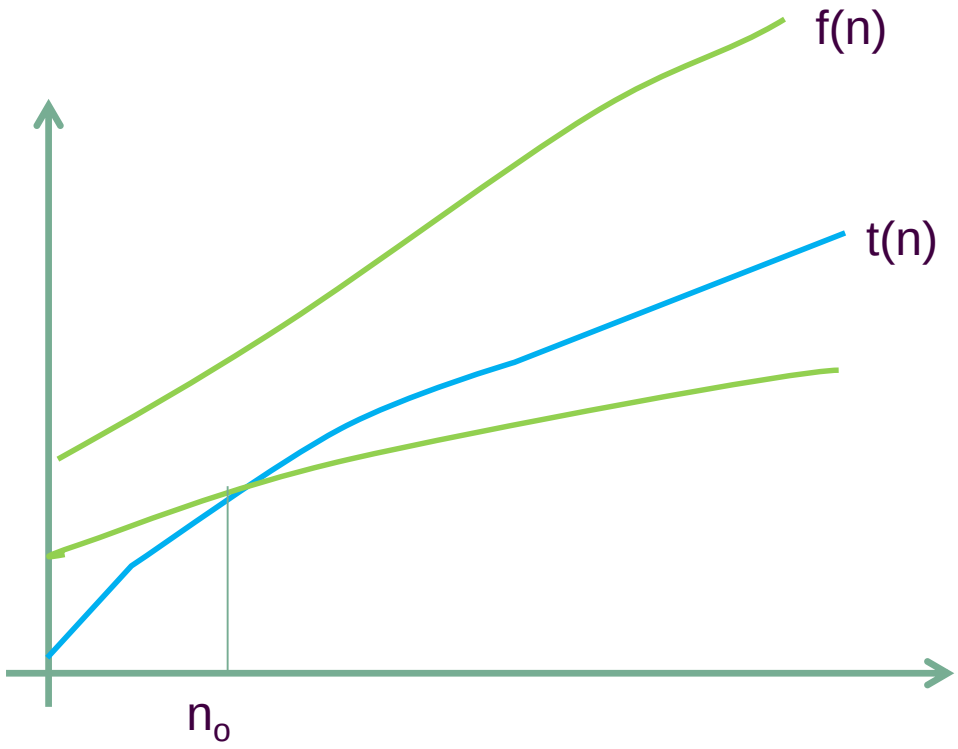
Cota inferior. Notación Ω

Sea $f: \mathbb{N} \to [0, \infty)$. Se define el conjunto de funciones de orden Ω de f como:

$$\Omega(f) = \{g: \mathbb{N} \to [0, \infty) | \exists c \in \mathbb{R}^+, \exists n_0 \in \mathbb{N}; \\ g(n) \geq cf(n) \; \forall n \geq n_0\}$$

- Diremos que una función $t: \mathbb{N} \to [0, \infty)$ es de orden Ω de f si $t \in \Omega(f)$
- Intuitivamente, $t \in \Omega(f)$ indica que t está acotada inferiormente por algún múltiplo de f. Normalmente estaremos interesados en la mayor función f tal que t pertenezca a $\Omega(f)$

Cota inferior. Notación Ω



Cota inferior. Notación Ω

- Obtener buenas cotas inferiores es, en general, muy difícil
- Existe siempre una cota trivial: tiempo de leer los datos y escribirlos
  - Ordenar n elementos → $\Omega(n)$
  - Multiplicar dos matrices de orden n → $\Omega(n^2)$

Cota inferior. Notación Ω

- Propiedades de Ω
  - $f \in \Omega(f)$
  - $\Omega(f) = \Omega(g)$ si y sólo si $f \in \Omega(g)$ y $g \in \Omega(f)$
  - Si $f \in \Omega(g)$ y $g \in \Omega(h)$ entonces $f \in \Omega(h)$
  - Si $f_1 \in \Omega(g)$ y $f_2 \in \Omega(h)$ entonces $f_1 + f_2 \in \Omega(g + h)$
  - Si $f_1 \in \Omega(g)$ y $f_2 \in \Omega(h)$ entonces $f_1 \cdot f_2 \in \mathrm{O}(g \cdot h)$
  - Regla del límite
    - Si $lim_{n\to\infty}\left(\frac{f(n)}{g(n)}\right) \in \mathbb{R}^+$, entonces $f(n) \in \Omega(g(n))$ y $g(n) \in \Omega(f(n))$
    - Si $lim_{n\to\infty}\left(\frac{f(n)}{g(n)}\right) = 0$, entonces $f(n) \notin \Omega(g(n))$ pero $g(n) \in \Omega(f(n))$
    - Si $lim_{n\to\infty}\left(\frac{f(n)}{g(n)}\right) = +\infty$, entonces $f(n) \in \Omega(g(n))$ pero $g(n) \notin \Omega(f(n))$

Cota inferior. Notación Ω

- Regla de dualidad

  $f \in O(g)$ si y sólo si $g \in \Omega(f)$

Índice

- Eficiencia y complejidad
- Cotas
  - Cota superior. Notación O
  - Cota inferior. Notación Ω
  - Orden exacto. Notación Θ
- Ecuaciones en recurrencia

Orden exacto. Notación Θ

- Conjunto de funciones que crecen asintóticamente de la misma forma

Sea $f: \mathbb{N} \to [0, \infty)$. Se define el conjunto de funciones de orden Θ de f como:

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

- Diremos que una función $t: \mathbb{N} \to [0, \infty)$ es de orden Θ de f si $t \in \Theta(f)$
- Intuitivamente, $t \in \Theta(f)$ indica que t está acotada tanto superior como inferiormente por múltiplos de f., es decir, que t y f crecen de la misma forma

Orden exacto. Notación Θ

- Propiedades de Θ
  - $f \in \Theta(f)$
  - $\Theta(f) = \Theta(g)$ si y sólo si $f \in \Theta(g)$ y $g \in \Theta(f)$
  - Si $f \in \Theta(g)$ y $g \in \Theta(h)$ entonces $f \in \Theta(h)$
  - Si $f_1 \in \Theta(g)$ y $f_2 \in \Theta(h)$ entonces $f_1 + f_2 \in \Theta(\max(g, h))$
  - Si $f_1 \in \Theta(g)$ y $f_2 \in \Theta(h)$ entonces $f_1 \cdot f_2 \in \Theta(g \cdot h)$
  - Regla del límite
    - Si $lim_{n\to\infty}\left(\frac{f(n)}{g(n)}\right) \in \mathbb{R}^+$, entonces $f(n) \in \Theta(g(n))$ y $g(n) \in \Theta(f(n))$
    - Si $lim_{n\to\infty}\left(\frac{f(n)}{g(n)}\right) = 0$, entonces $f(n) \notin \Omega(g(n))$ y $g(n) \notin \Omega(f(n))$
    - Si $lim_{n\to\infty}\left(\frac{f(n)}{g(n)}\right) = +\infty$, entonces $f(n) \notin \Omega(g(n))$ y $g(n) \notin \Omega(f(n))$

Ejercicio

- Dado un algoritmo cuyos tiempos de ejecución en el caso mejor, peor y medio son los siguientes, indica las cotas asintóticas O, Ω y Θ de dichas funciones
- Caso mejor
  - $T(n) = \frac{7n^2}{2} + \frac{5n}{2} - 3$
- Caso peor
  - $T(n) = 8n^2 - 2n - 3$
- Caso medio
  - $T(n) = \frac{23}{4}n^2 + \frac{1}{4}n - 3$

Ejercicio

- De las siguientes afirmaciones, indicar cuáles son ciertas y cuáles no:

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

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

c) $2^{n+1} \in O(2^n)$

d) $(n+1)! \in O(n!)$

e) $f(n) \in O(n) \Rightarrow 2^{f(n)} \in O(2^n)$

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

g) $\log n \in O(n^{1/2})$

h) $n^{1/2} \in O(\log n)$

i) $n^2 \in \Omega(n^3)$

j) $n^3 \in \Omega(n^2)$

k) $2^{n+1} \in \Omega(2^n)$

l) $(n+1)! \in \Omega(n!)$

m) $f(n) \in \Omega(n) \Rightarrow 2^{f(n)} \in \Omega(2^n)$

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

o) $\log n \in \Omega(n^{1/2})$

p) $n^{1/2} \in \Omega(\log n)$

Ejercicio

- Sea a una constante real, $0 \leq a \leq 1$. Usar las relaciones $\subset$ y = para ordenar los órdenes de complejidad de las siguientes funciones:
- $n \log n$
- $n^2 \log n$
- n^8
- n^{1+a}
- $(1+a)^n$
- $(n^2 + 8n + \log^3 n)^4$
- $n^2 / \log n$
- 2^n

Ejercicio

- Supongamos que tenemos el siguiente algoritmo. Calcular los tiempos de ejecución en el caso mejor y peor, y dar sus cotas asintóticas.

```
algoritmo simetria (ENT A:matriz[1..n][1..n] SAL booleano)
variables f,c: entero; es_traspuesta: booleano
principio
        es traspuesta=verdadero
        c=1
        mientras c<= n y es_traspuesta hacer
                f=n
                mientras f>c y es_traspuesta hacer
                        si A[f,c] ≠ A[c,f] entonces
                                es_traspuesta=falso
                        fin si
                        f=f-1
                fin mientras
                c=c+1
        fin mientras
        devolver es_traspuesta
fin
```

Ejercicio

- Supongamos que tenemos el siguiente algoritmo. Calcular los tiempos de ejecución en el caso mejor y peor, y dar sus cotas asintóticas.

```
algoritmo ecto (ENT a:vector[n], n:entero SAL entero)
variables i,j,x: entero; permuta: booleano
principio
        permuta=verdadero
        i=1
        mientras permuta hacer
                i=i+1
                permuta=falso
                para j=n hasta i hacer
                        si a[j] < a[j-1] entonces
                                x=a[j]
                                permuta=cierto
                                a[j]=a[j-1]
                                a[j-1]=x
                        fin si
                fin para
        fin mientras
        devolver i
fin
```

Ejercicio

- Ordena las siguientes funciones de acuerdo a su velocidad de crecimiento:
- n
- $\sqrt{n}$
- $\log n$
- $\log \log n$
- $\log^2 n$
- $n / \log n$
- $\sqrt{n} \log^2 n$
- $(1/3)^n$
- $(3/2)^n$
- 17
- n^2

Índice

- Eficiencia y complejidad
- Cotas
  - Cota superior. Notación O
  - Cota inferior. Notación Ω
  - Orden exacto. Notación Θ
- Ecuaciones en recurrencia

Ecuaciones en recurrencia

- Cuando en el algoritmo aparecen llamadas recursivas, ya no es suficiente con “contar” el número de Operaciones Elementales
- Comenzamos calculando la función recursiva del tiempo de ejecución, dejando como conocido el tiempo que tardan las llamadas recursivas a la función

Ejemplo

```
funcion uno(n,k: entero)
    var i,r:entero
    empezar
        si n<2 entonces
            devolver 1
        si no
            r = uno (n/2, k-1)
            r = r + uno (n/2, k+1)
            r = r * uno (n/2, k+2)
        fin si
    fin uno
```

Ejemplo

```
funcion uno(n,k: entero)
    var i,r:entero
    empezar
        si n<2 entonces                    (2 OE)
            devolver 1                     (1 OE)
        si no
            r = uno (n/2, k-1)             (4 OE) + T(n/2)
            r = r + uno (n/2, k+1)         (5 OE) + T(n/2)
            r = r * uno (n/2, k+2)         (5 OE) + T(n/2)
        fin si
    fin uno
```

Ejemplo

```
funcion uno(n,k: entero)
    var i,r:entero
    empezar
        si n<2 entonces                    (1 OE)
            devolver 1                     (1 OE)
        si no
            r = uno (n/2, k-1)             (4 OE) + T(n/2)
            r = r + uno (n/2, k+1)         (5 OE) + T(n/2)
            r = r * uno (n/2, k+2)         (5 OE) + T(n/2)
            devolver r                     (1 OE)
        fin si
    fin uno
```

$$T(n) = \begin{cases} 2 & si\ n < 2 \\ 16 + 3 * T(\frac{n}{2}) & si\ n \geq 2 \end{cases}$$

Ecuaciones en recurrencia

- Debemos encontrar una expresión no recursiva de la misma función
- Para ello utilizamos “suposiciones inteligentes” que nos permitan calcular dicha expresión
  - Estas suposiciones se refieren al valor (o tamaño de entrada)
  - Si cada vez divido en subproblemas de tamaño la mitad, puedo suponer que el tamaño original era potencia de 2
  - Si cada vez divido en subproblemas de tamaño un tercio, puedo suponer que el tamaño original era potencia de 3
  - …

Ejemplo

- $T(n) = \begin{cases} 2 & si\ n < 2 \\ 16 + 3 * T\left(\frac{n}{2}\right) & si\ n \geq 2 \end{cases}$
- ¿Cuánto vale $T\left(\frac{n}{2}\right)$ si $n > 4$?

Ejemplo

- $T(n) = \begin{cases} 2 & si\ n < 2 \\ 16 + 3 * T\left(\frac{n}{2}\right) & si\ n \geq 2 \end{cases}$
- ¿Cuánto vale $T\left(\frac{n}{2}\right)$ si $n > 4$?
- $T(n/2) = 16 + 3 * T\left(\frac{n}{4}\right)$

- Por tanto
- $T(n) = 16 + 3\left[16 + 3 * T\left(\frac{n}{4}\right)\right] = 16 + 3 * 16 + 3^2 * T\left(\frac{n}{2^2}\right)$

Ejemplo

- ¿Cuánto vale $T(\frac{n}{2^2})$?
- $T(n/2^2) = 16 + 3 * T(\frac{n}{2^3})$

- Por tanto
- $T(n) = 16 + 3 * 16 + 3^2 * [16 + 3 \ * T\left(\frac{n}{2^3}\right) =$

$$16 + 3 * 16 + 3^2 * 16 + 3^3 * T\left(\frac{n}{2^3}\right)$$

Ejemplo

- Asumimos que n es potencia de 2, por lo que $n = 2^k$
- Si continuamos resolviendo la recurrencia, llegamos a
- $T(n) = \sum_{i=0}^{k-1} 3^i * 16 + 3^k * T\left(\frac{n}{2^k}\right)$
- $\frac{n}{2^k} = 1,1 < 2$, luego hemos llegado al caso base

- Término general:
- $T(n) = \sum_{i=0}^{k-1} 3^i * 16 + 3^k * 2$

Ejemplo

- Eliminamos el sumatorio
- $T(n) = \sum_{i=0}^{k-1} 3^i * 16 + 3^k * 2$
- $T(n) = 16 * \frac{3^{k-1}-1}{3-1} + 3^k * 2 = 14 * 3^{k-1} - 8$
- Sustituimos k, para dejar todo en función de n
- $T(n) = \frac{14}{3} * 3^{\log_2 n} - 8$

- Hacemos el cambio de logaritmos
- $3^{\log_2 n} = n^{\log_2 3}$
- $T(n) = \frac{14}{3} * n^{\log_2 3} - 8$

Ejemplo

- ¿Cómo hemos eliminado el sumatorio? → Suma de una progresión geométrica
  - $S_n = a_1 + a_2 + a_3 + \cdots + a_n$

    - Multiplicamos a ambos lados por la razón
    - $r * S_n = r * (a_1 + a_2 + a_3 + \cdots + a_n)$
    - Por ser progresión geométrica, $a_i * r = a_{i+1}$
    - $r * S_n = a_2 + a_3 + \cdots + a_n + a_{n+1}$

    - Resto las dos sumas
    - $r * S_n - S_n = a_{n+1} - a_1$
    - Despejo S_n
    - $S_n = \frac{a_{n+1} - a_1}{r-1} = \frac{a_1 * r^n - a_1}{r-1} = a_1 \frac{r^n - 1}{r-1}$

Ejemplo

- ¿Y si fuese una progresión aritmética?
- $S_n = 1 + 2 + 3 + \cdots + n$

  - Sumo la misma progresión en orden inverso
  - $S_n = \quad 1 \quad + \quad 2 \quad + \quad 3 \quad + \ldots + \quad n$
  - $S_n = \quad n \quad + \; n-1 \; + \; n-2 \; + \ldots + \quad 1$
  - $2S_n = n+1 \; + \; n+1 \; + \; n+1 \; + \; \ldots + \; n+1 = n(n+1)$

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

Ejemplo

- ¿Cómo he hecho el cambio de logaritmos?
- $3^{\log_2 n} = n^{\log_2 3}$

  - $n = 2^{\log_2 n}$
  - $n^{\log_2 3} = \left(2^{\log_2 n}\right)^{\log_2 3} = 2^{(\log_2 n * \log_2 3)} = 2^{(\log_2 3 * \log_2 n)} =$
    $= \left(2^{\log_2 3}\right)^{\log_2 n} = 3^{\log_2 n}$

Ejemplo

- $T(n) = \frac{14}{3} * n^{\log_2 3} - 8$

- Damos sus cotas asintóticas
- $T(n) \in O(n^{\log_2 3})$
- $T(n) \in \Omega(n^{\log_2 3})$
- $T(n) \in \Theta(n^{\log_2 3})$

“Método maestro”

- Existe un método para saber las cotas asintóticas de nuestras funciones sin necesidad de realizar todos los cálculos matemáticos
- Es una especie de caja negra para solucionar recurrencias
- Asumimos que todos los subproblemas tienen el mismo tamaño
- A partir de la ecuación de recurrencia, podemos distinguir dos casos:
  - Reducción por división
  - Reducción por sustracción

“Método maestro”. Reducción por división

- Ecuación de recurrencia
- $$T(n) = \begin{cases} c * n^k & 1 \leq n < b \\ a * T\left(\frac{n}{b}\right) + c * n^k & n \geq b \end{cases}$$

- Cotas asintóticas
- $$T(n) \in \begin{cases} \Theta(n^k) & a < b^k \\ \Theta(n^k * \log n) & a = b^k \\ \Theta(n^{\log_b a}) & a > b^k \end{cases}$$

“Método maestro”. Reducción por sustracción

- Ecuación de recurrencia
- $$T(n) = \begin{cases} c * n^k & 1 \leq n < b \\ a * T(n-b) + c * n^k & n \geq b \end{cases}$$

- Cotas asintóticas
- $$T(n) \in \begin{cases} \Theta(n^k) & a < 1 \\ \Theta(n^{k+1}) & a = 1 \\ \Theta(a^{n\ div\ b}) & a > 1 \end{cases}$$

Ejemplo

- $T(n) = \begin{cases} 2 & si\ n < 2 \\ 16 + 3 * T\left(\frac{n}{2}\right) & si\ n \geq 2 \end{cases}$

- Reducción por división
  - a = 3
  - b = 2
  - c = 16
  - k = 0

- $\Theta\left(n^{\log_b a}\right) = \Theta(n^{\log_2 3})$

Ejercicio

- Dado el siguiente algoritmo, calcula su tiempo de ejecución y su orden de complejidad (a través de la ecuación de recurrencia).

```
algoritmo factorial (ENT n:entero SAL entero)
principio
        si n=1 entonces
                devolver 1
        si no
                devolver n*factorial(n-1)
        fin si
fin
```

Ejercicio

- Resuelve la siguiente ecuación y da su orden de complejidad:
- $T(n) = 4T\left(\frac{n}{2}\right) + n^2$,
- con n potencia de 2, $T(1) = 1$.

Ejercicio

- Considera el siguiente algoritmo:
-

```
algoritmo búsqueda_binaria (ENT a:tabla, prim, ult:entero, x:elemento)
variables mitad:entero
principio
        si prim>=ult entonces devolver a[ult]=x
        si no
                mitad ← (prim+ult) div 2
                si x=a[mitad] entonces devolver verdadero
                si no
                        si x<a[mitad] entonces
                                devolver búsqueda_binaria(a, prim, mitad-1,x)
                        si no
                                devolver búsqueda_binaria(a, mitad+1, ult,x)
                        fin si
                fin si
        fin si
fin
```

-
- Calcula sus tiempos de ejecución y sus órdenes de complejidad.
- Modifica el algoritmo eliminando la recursión.
- Calcula la complejidad del algoritmo modificado y justifica para qué casos es más conveniente usar uno u otro.

Ejercicio

- Dado el siguiente algoritmo, calcula su tiempo de ejecución y su orden de complejidad
-

```
algoritmo hanoi (ENT n,destino,origen,aux:entero)
principio
        si n>0 entonces
                hanoi(n-1,aux,origen,destino)
                hanoi(n-1,destino,aux,origen)
        fin si
fin
```

Ejercicio

- Dado el siguiente algoritmo, calcula su tiempo de ejecución y su orden de complejidad

```
algoritmo sumadigitos (ENT n:entero)
principio
        si n<10 entonces
                devolver n
        si no
                devolver (n mod 10) + sumadigitos(n div 10)

        fin si
fin
```

Ejercicio

- Resuelve la siguiente ecuación y da su orden de complejidad:
- $T(n) = \frac{1}{n}\left(\sum_{i=0}^{n-1} T(i)\right) + cn,$
- con $T(0) = 0.$