Metodología y Tecnología de la Programación

Ingeniería en Informática

Curso 2008-2009

Introducción al estudio de los algoritmos

Yolanda García Ruiz **D228** **ygarciar@fdi.ucm.es**
Jesús Correas **D228** **jcorreas@fdi.ucm.es**

Departamento de Sistemas Informáticos y Computación
Universidad Complutense de Madrid

(elaborado a partir de [BB97], [GV00] y notas de S. Estévez)

Bibliografía

- **Importante**: Estas transparencias son un material de apoyo a las clases presenciales y no sustituyen a la bibliografía básica ni a las propias clases presenciales para el estudio de la asignatura
- Bibliografía básica:
  - [BB97]: secciones 1.1 a 1.3 y capítulos 2, 3 y 4
  - [GV00]: capítulo 1
  - [NN98]: capítulo 1 y apéndice B
- Bibliografía complementaria:
  - [HSR97]: secciones 1.1 a 1.3
  - [PE04]: capítulo 1.

Introducción al estudio de los algoritmos

1. ¿Qué es un algoritmo?
2. Eficiencia de los algoritmos
3. Análisis de caso mejor, peor y medio
4. Cálculo del número de operaciones elementales
5. Notación asintótica. Órdenes de complejidad
6. Resolución de recurrencias

¿Qué es un algoritmo?

- Algoritmo (Diccionario R.A.E., 22ª edición):

  1. m. Conjunto ordenado y finito de operaciones que permite hallar la solución de un problema.

- Características que debe tener un algoritmo ([HSR97]):
  - Debe tener 0 o más **datos de entrada**
  - Debe producir al menos un **dato de salida**
  - Cada instrucción debe ser inequívoca y estar claramente **definida**
  - Debe **terminar** para todos los casos en un número finito de pasos
  - Cada instrucción debe ser **efectiva**: debe poder realizarse en un tiempo finito
- Admitiremos como algoritmos los procedimientos que realicen elecciones aleatorias (*algoritmos probabilistas*)

Problema, ejemplar (o instancia), algoritmo, programa

- Un **problema** es una pregunta para la cual se pretende buscar una respuesta.
  - ejemplo: "cuál es la lista resultado de ordenar una lista de enteros S de longitud n".
  - Una característica general de la definición de un problema es la presencia de *parámetros* (S, n)
- Un **ejemplar o instancia** de un problema es una asignación específica de valores a los parámetros del problema
  - Ejemplo: "cuál es la lista resultado de ordenar la lista de enteros $S = [5, 4, 3, 8, 1]$ de longitud 5".
- **algoritmo**
- Un algoritmo debe funcionar correctamente en todos los ejemplares o casos del problema que manifiesta resolver
- Un algoritmo se implementa mediante un **programa**

Algoritmia

- La algoritmia estudia técnicas de diseño de algoritmos y técnicas para medir la eficiencia de los algoritmos
- La **eficiencia** es una medida del coste en tiempo (tiempo de ejecución) o recursos (espacio de memoria utilizada, principalmente) que consume un algoritmo para encontrar una solución a un problema
- La medida de la eficiencia permite comparar distintos algoritmos que resuelven un determinado problema
- Por qué la eficiencia es importante
- Un ejemplo: algoritmos de búsqueda sobre una lista ordenada

Ejemplo: búsqueda en una lista ordenada

```
proc busqueda_sec(S[1..n],x,pos)
   pos ← 1
   mientras pos ≤ n Y S[pos] < x
   hacer
      pos ← pos + 1
   fin mientras
fin proc
```

```
proc busqueda_bin(S[1..n],x,pos)
   inf ← 1
   sup ← n
   pos ← 0
   mientras inf ≤ sup Y pos = 0 hacer
      mitad ← ⌊ (inf+sup) / 2 ⌋
      si x = S[mitad] entonces
         pos ← mitad
      si no si x < S[mitad] entonces
         sup ← mitad - 1
      si no
         inf ← mitad + 1
      fin si
   fin mientras
fin proc
```

Notación para describir algoritmos

- Utilizaremos pseudocódigo imperativo para describir los algoritmos
- Las instrucciones más habituales en este tipo de pseudocódigo son la asignación, las estructuras **si-entonces-si no**, **desde-hasta**, **mientras** y **repetir**
- Se pueden utilizar procedimientos y funciones, y los argumentos se pasan por referencia (aunque no utilizaremos notación para ello).
- En la especificación de algoritmos también permitiremos:
  - vectores de subíndices en cualquier rango (S[0..n-1], T[2..m]), multidimensionales (matrices) y de tamaño variable
  - operaciones sobre conjuntos (unión de conjuntos, diferencia, etc.), aunque el tiempo de ejecución de estas operaciones depende del tamaño de los conjuntos

Introducción al estudio de los algoritmos

1. ¿Qué es un algoritmo?
2. Eficiencia de los algoritmos
3. Análisis de caso mejor, peor y medio
4. Cálculo del número de operaciones elementales
5. Notación asintótica. Órdenes de complejidad
6. Resolución de recurrencias

Eficiencia de los algoritmos

- Eficiencia en tiempo vs. en espacio
- Formas de medir la eficiencia: teórica (*a priori*) o experimental (*a posteriori*)
- Estudiaremos la eficiencia teórica: obtendremos una función que acote el tiempo de ejecución para unos parámetros de entrada dados
- Ventajas de la eficiencia teórica:
  - Se ahorra tiempo de programación de distintos algoritmos
  - Se ahorra tiempo de pruebas de ejecución
  - Permite estudiar los algoritmos en casos de todos los tamaños, y para distintas clases de ejemplares

Eficiencia de los algoritmos (cont.)

- La medida de la eficiencia depende de:
  - el dominio de definición (por ejemplo, multiplicación de enteros grandes)
  - el tamaño de los datos de entrada
- La medida de eficiencia **no** debe depender de la velocidad de un ordenador específico, ni de una implementación particular, ni del lenguaje de programación
- Principio de invarianza:
  Dado un algoritmo y dos implementaciones del mismo 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 $T_1(n) \leq cT_2(n)$

Eficiencia de los algoritmos (cont.)

- La medida de la eficiencia depende del tamaño de los datos de entrada
- Definición formal de tamaño: *"número de bits necesarios para representar los datos de entrada"*
- Utilizaremos una representación del tamaño más sencilla:
  - Si los datos de entrada forman una lista, el tamaño es el número de elementos
  - Si los datos de entrada representan un grafo, el tamaño estará formado por el número de vértices, o el número de aristas, o ambos
  - Si el dato de entrada es un número entero, podremos utilizar en algunos casos el valor del número en lugar de su tamaño en número de bits necesarios para representarlo (por ejemplo, fibonacci)

Eficiencia de los algoritmos (cont.)

- La eficiencia teórica expresa el tiempo requerido por el algoritmo salvo una constante multiplicativa
- Sin embargo, no deben olvidarse las constantes ocultas: Dependiendo del ámbito de aplicación, un algoritmo más eficiente puede no ser el más adecuado:

  $$f(n) = 1000n \text{ frente a } g(n) = 0{,}5n^2$$

  El algoritmo de complejidad cuadrática es más eficiente que el de complejidad lineal para valores de $n \leq 2000$
- El esfuerzo de diseño, programación y mantenimiento también debe tenerse en cuenta
- Nosotros estudiaremos el comportamiento de un algoritmo al aumentar el tamaño de los datos; es decir, cómo aumenta su tiempo de ejecución.
- Esto se conoce como **eficiencia asintótica**, representada mediante **funciones de orden**. Esta notación facilita la comparación de eficiencia entre algoritmos diferentes

Introducción al estudio de los algoritmos

1. ¿Qué es un algoritmo?
2. Eficiencia de los algoritmos
3. Análisis de caso mejor, peor y medio
4. Cálculo del número de operaciones elementales
5. Notación asintótica. Órdenes de complejidad
6. Resolución de recurrencias

Análisis de caso mejor, peor y medio

- El comportamiento de un algoritmo cambia para diferentes datos de entrada (*instancias* o *ejemplares*), y el resultado del análisis puede variar según las suposiciones que se hagan para estos casos
  - **Caso mejor**: traza del algoritmo que realiza menos instrucciones para una entrada de tamaño n
  - **Caso peor**: traza del algoritmo que realiza más instrucciones para una entrada de tamaño n
  - **Caso medio**: traza del algoritmo para un número de instrucciones igual a la esperanza matemática de la variable aleatoria definida para todas las posibles trazas del algoritmo para una entrada de tamaño n
- En los tres casos se hace un análisis del algoritmo para una entrada de tamaño n
- El análisis de caso peor se utiliza para algoritmos críticos
- El análisis de caso medio se aplica a algoritmos que se usan muy frecuentemente

Ejemplo: Análisis de caso mejor y peor

```
proc insercion(T[1..n])
   desde i ← 2 hasta n hacer
      x ← T[i]
      j ← i-1
      mientras j > 0 Y x < T[j] hacer
         T[j+1] ← T[j]
         j ← j-1
      fin mientras
      T[j+1] ← x
   fin desde
fin proc
```

```
proc seleccion(T[1..n])
   desde i ← 1 hasta n-1 hacer
      minj ← i
      minx ← T[i]
      desde j ← i+1 hasta n hacer
         si T[j] < minx entonces
            minj ← j
            minx ← T[j]
         fin si
      fin desde
      T[minj] ← T[i]
      T[i] ← minx
   fin desde
fin proc
```

- Caso mejor: T ya está ordenada en orden creciente
- Caso peor: T está ordenada en orden decreciente
- Los algoritmos se comportan de forma diferente en el caso mejor
- El análisis de caso medio lo veremos más adelante

Introducción al estudio de los algoritmos

1. ¿Qué es un algoritmo?
2. Eficiencia de los algoritmos
3. Análisis de caso mejor, peor y medio
4. Cálculo del número de operaciones elementales
5. Notación asintótica. Órdenes de complejidad
6. Resolución de recurrencias

Número de operaciones elementales

- La eficiencia de un algoritmo se mide en función del número de operaciones elementales que se ejecutan para un tamaño de entrada n
- Las operaciones elementales son las que el ordenador realiza en un tiempo acotado por una constante:
  - operaciones aritméticas básicas
  - asignaciones a variables de tipo predefinido
  - saltos (llamadas a procedimientos, retorno de procedimientos, etc)
  - comparaciones lógicas
  - acceso a estructuras indexadas básicas (vectores y matrices)
- Cada operación elemental contabilizará como 1
- Observación: las operaciones elementales dependen del dominio de aplicación del algoritmo

Reglas para el cálculo del número de op. elementales

- Dependiendo del tipo de construcción del pseudocódigo utilizada, el cálculo del número de operaciones elementales es el siguiente:
  - **Secuencias** $S_1; \ldots; S_n$: El número de operaciones de la secuencia es la suma del número de operaciones de cada componente: $OE(S_1) + \cdots + OE(S_n)$
  - **si** C **entonces** S_1 **si no** S_2: número de operaciones de la evaluación de la condición más el máximo del número de operaciones de cada una de las alternativas: $OE(C) + max(OE(S_1), OE(S_2))$ **(caso peor)**
  - **mientras** C **hacer** S: $OE(C) + m(OE(C) + OE(S))$, donde m es el número de iteraciones del bucle
  - **repetir** S **hasta** C: $m(OE(C) + OE(S))$, donde m es el número de iteraciones del bucle
  - **desde** $v \leftarrow k$ **hasta** n **hacer** S: $1 + 1 + (n - k + 1)(2 + OE(S))$, donde k y n son constantes enteras y $k \leq n$ (¿cómo sería si k y n fueran expresiones?)
  - **Llamadas recursivas a un procedimiento** p: $1 + T_p(k)$, donde $T_p(k)$ es el número de operaciones de la llamada recursiva (si los argumentos son expresiones, hay que sumar el coste de su evaluación)

Ejemplo: Número de operaciones elementales

Análisis de caso peor del algoritmo de ordenación por inserción

```
 1: proc insercion(T[1..n])
 2:     desde i ← 2 hasta n hacer
 3:         x ← T[i]
 4:         j ← i-1
 5:         mientras j > 0 Y x < T[j] hacer
 6:             T[j+1] ← T[j]
 7:             j ← j-1
 8:         fin mientras
 9:         T[j+1] ← x
10:     fin desde
11: fin proc
```

línea	**Número de veces**
2:	asignación inicial: 1 vez
2:	condición del bucle: n veces
2:	incremento de i: $n-1$ veces
3,4,9:	$n-1$ veces
5:	condición del bucle: $\sum_{i=2}^{n} i$ veces
6,7:	$\sum_{i=2}^{n}(i-1)$ veces

Complejidad del algoritmo en el caso peor:

$$OE(n) = 1 + n + (n-1) + 7(n-1) + \sum_{i=2}^{n} 4i + \sum_{i=2}^{n} 6(i-1) =$$

$$9n - 7 + \sum_{i=2}^{n} 10i - 6n + 6 = 3n - 1 + 10\frac{n(n+1)}{2} - 10 = 5n^2 + 8n - 11$$

Ejercicio: Número de operaciones elementales

```
1:  proc seleccion(T[1..n])
2:      desde i ← 1 hasta n-1 hacer
3:          minj ← i
4:          minx ← T[i]
5:          desde j ← i+1 hasta n hacer
6:              si T[j] < minx entonces
7:                  minj ← j
8:                  minx ← T[j]
9:              fin si
10:         fin desde
11:         T[minj] ← T[i]
12:         T[i] ← minx
13:     fin desde
14: fin proc
```

- ¿Cuál es el número de operaciones elementales del algoritmo de ordenación por selección en el caso peor?
- ¿Y en el caso mejor?

El método de la instrucción característica

- Una forma alternativa de obtener la complejidad de un algoritmo es mediante el método de la instrucción característica
- Consiste en elegir una instrucción del algoritmo que se ejecute un número de veces proporcional a la complejidad del algoritmo
- Cuando un algoritmo contiene varios bucles anidados, en general se puede utilizar una instrucción del bucle más interno como instrucción característica
- En estos casos, hay que tener en cuenta el control del bucle. Si algún bucle se puede ejecutar 0 veces, no se pueden considerar las instrucciones dentro de este bucle como instrucciones características
- Este método no permite obtener todas las constantes de la complejidad del algoritmo, pero es suficiente para clasificarlo en una categoría de complejidad
- Algunos autores denominan esta técnica método de la instrucción básica, crítica, o método del barómetro

Ejemplo: Método de la instrucción característica

```
1:  proc insercion(T[1..n])
2:      desde i ← 2 hasta n hacer
3:          x ← T[i]
4:          j ← i-1
5:          mientras j > 0 Y x < T[j] hacer
6:              T[j+1] ← T[j]
7:              j ← j-1
8:          fin mientras
9:          T[j+1] ← x
10:     fin desde
11: fin proc
```

- Consideramos el caso peor
- Utilizamos la comparación del bucle **mientras** en la línea 5 como instrucción característica
- Esta instrucción se ejecuta i veces en cada iteración del bucle **desde**

- Por tanto, $IC(n) = \sum_{i=2}^{n} i = \frac{n(n+1)}{2} - 1 = \frac{n^2}{2} + \frac{n}{2} - 1$
- $IC(n)$ **no** es el número de operaciones elementales del algoritmo. Solamente nos permite decir que el algoritmo, en el caso peor, tiene un comportamiento cuadrático ($IC(n) \in \Theta(n^2)$, como veremos más adelante)

Análisis de caso medio

- Los análisis de caso peor y caso mejor vistos anteriormente son útiles, pero en algunos casos es más interesante conocer cómo se comporta el algoritmo en promedio
- Para ello, deben asignarse probabilidades a todos los casos posibles de entrada de tamaño n
- Es importante que la asignación de probabilidades esté basada en información disponible
- Un ejemplo sencillo: búsqueda secuencial sobre una lista no ordenada

Ejemplo: Análisis de caso medio

```
proc busqueda_sec2(S[1..n],x,pos)
   pos ← 1
   mientras pos ≤ n Y S[pos] ≠ x hacer
      pos ← pos + 1
   fin mientras
   si S[pos] ≠ x entonces
      pos ← 0
   fin si
fin proc
```

- Si suponemos que x puede estar en cualquier posición del vector con igual probabilidad, ésta será $1/n$ en cada una de las posiciones.
- si utilizamos el método de la instrucción característica (comparación del bucle **mientras**):

$$A(n) = \sum_{k=1}^{n}(k \cdot \frac{1}{n}) = \frac{1}{n}\sum_{k=1}^{n} k = \frac{1}{n} \cdot \frac{n(n+1)}{2} = \frac{n+1}{2}$$

- Observación: hemos supuesto que x siempre está en el vector

Ejemplo: Análisis de caso medio (cont.)

- Si ahora suponemos que x puede no estar en el vector, el resultado varía
- Si x no está en el vector, el algoritmo debe recorrer todo el vector para un número posiblemente grande de valores de x. Podemos suponer que esto ocurre con una probabilidad $1-p$
- Utilizando de nuevo el método de la instrucción característica sobre la comparación del bucle **mientras**:

$$A(n) = \sum_{k=1}^{n}(k \cdot \frac{p}{n}) + n(1-p) = \frac{p}{n} \cdot \frac{n(n+1)}{2} + n(1-p) = n(1-\frac{p}{2}) + \frac{p}{2}$$

Introducción al estudio de los algoritmos

1. ¿Qué es un algoritmo?
2. Eficiencia de los algoritmos
3. Análisis de caso mejor, peor y medio
4. Cálculo del número de operaciones elementales
5. Notación asintótica. Órdenes de complejidad
6. Resolución de recurrencias

Notación asintótica $\mathcal{O}$

- Una función de complejidad $g(n)$ es una función $g : \mathbb{N} \rightarrow \mathbb{R}^{\geq 0}$
- Hay diversas formas de clasificar el comportamiento asintótico de una función de complejidad. Nosotros veremos las notaciones $\mathcal{O}$, Ω y Θ
- Dada una función de complejidad $g(n)$, $\mathcal{O}(g(n))$ es el conjunto de funciones $f : \mathbb{N} \rightarrow \mathbb{R}^{\geq 0}$ para las que existen $c \in \mathbb{R}^{+}$, $n_0 \in \mathbb{N}, n_0 \geq 0$ tales que para todo $n \geq n_0$,

$$f(n) \leq cg(n)$$

- Cuando $f(n) \in \mathcal{O}(g(n))$ se dice que $f(n)$ *está en el orden de* $g(n)$.
- $\mathcal{O}(g(n))$ representa una **cota superior** de la complejidad de una función $f(n)$
- **Notación**: algunos autores utilizan $f(n) = \mathcal{O}(g(n))$ para decir que $f(n)$ está en el orden de $g(n)$. El símbolo '=' no debe entenderse en este caso como la igualdad en el sentido matemático

Ejemplo de notación $\mathcal{O}(f(n))$

proc busqueda_sec(S[1..n],x,pos)
  pos $\leftarrow$ 1
  mientras pos $\leq$ n Y S[pos] $<$ x **hacer**
    pos $\leftarrow$ pos + 1
  fin mientras
fin proc

- En este ejemplo, el número de operaciones elementales es $T(n) = 5 + 6n$
- $T(n) \in \mathcal{O}(n)$ ($c = 7$ y $n_0 = 5$)
- $T(n) \in \mathcal{O}(n^2)$ ($c = 1$ y $n_0 = 7$, o bien $c = 3$ y $n_0 = 8$)
- $T(n) \notin \mathcal{O}(lg\ n)$

Propiedades de $\mathcal{O}(f(n))$

- $\forall f : \mathbb{N} \to \mathbb{R}^{\geq 0}$, $f(n) \in \mathcal{O}(f(n))$
- $f(n) \in \mathcal{O}(g(n)) \implies \mathcal{O}(f(n)) \subset \mathcal{O}(g(n))$
- $f(n) \in \mathcal{O}(g(n))$ y $g(n) \in \mathcal{O}(h(n)) \implies f(n) \in \mathcal{O}(h(n))$
- $\mathcal{O}(f(n)) = \mathcal{O}(g(n))$ sii $f(n) \in \mathcal{O}(g(n))$ y $g(n) \in \mathcal{O}(f(n))$
- $f(n) \in \mathcal{O}(g(n))$ y $f(n) \in \mathcal{O}(h(n)) \implies f(n) \in \mathcal{O}(min(g(n), h(n)))$
- $f_1(n) \in \mathcal{O}(g(n))$ y $f_2(n) \in \mathcal{O}(h(n)) \implies f_1(n) \cdot f_2(n) \in \mathcal{O}(g(n) \cdot h(n))$
- Utilizaremos la notación $f(n) \in \mathcal{O}(g(n))$ también cuando $f(n)$ no esté definida o sea negativa en un número finito de valores de n. Por ejemplo, $n/lg\ n \in \mathcal{O}(n)$

Propiedades de $\mathcal{O}(f(n))$ (cont.)

- Regla del máximo:
  - $f_1(n) \in \mathcal{O}(g(n))$ y $f_2(n) \in \mathcal{O}(h(n)) \implies f_1(n) + f_2(n) \in \mathcal{O}(max(g(n), h(n)))$
  - esta regla es extensible a $f_1(n) + \cdots + f_k(n)$, siempre que k no dependa de n
- Regla del límite. Dadas las funciones arbitrarias $f, g : \mathbb{N} \to \mathbb{R}^{\geq 0}$,
  - $\lim_{n\to\infty} \frac{f(n)}{g(n)} \in \mathbb{R}^+ \implies f(n) \in \mathcal{O}(g(n))$ y $g(n) \in \mathcal{O}(f(n))$
  - $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \implies f(n) \in \mathcal{O}(g(n))$, pero $g(n) \notin \mathcal{O}(f(n))$
  - $\lim_{n\to\infty} \frac{f(n)}{g(n)} = +\infty \implies f(n) \notin \mathcal{O}(g(n))$, pero $g(n) \in \mathcal{O}(f(n))$

Propiedades de $\mathcal{O}(f(n))$ (cont.)

- Órdenes de complejidad:
  $\mathcal{O}(lg\ n) \subset \mathcal{O}(n) \subset \mathcal{O}(n\ lg\ n) \subset \mathcal{O}(n^2) \subset \mathcal{O}(n^j) \subset \mathcal{O}(n^k) \subset \mathcal{O}(a^n)$
  $\subset \mathcal{O}(b^n) \subset \mathcal{O}(n!) \subset \mathcal{O}(n^n)$, donde $k > j > 2$, $b > a > 1$
- Ejemplos. Demostrar las siguientes afirmaciones:
  - $\log n \in \mathcal{O}(\sqrt{n})$
  - $\mathcal{O}(n^2) = \mathcal{O}(n^2 - \log n)$
  - $2^{n+1} \in \mathcal{O}(2^n)$

Notación asintótica Ω

- Del mismo modo que en el caso anterior, se puede definir una **cota inferior** de la complejidad de una función
- Dada una función de complejidad $g(n)$, $\Omega(g(n))$ es el conjunto de funciones $f : \mathbb{N} \to \mathbb{R}^{\geq 0}$ para las que existen $c \in \mathbb{R}^{+}$, $n_0 \in \mathbb{N}, n_0 \geq 0$ tales que para todo $n \geq n_0$,

$$f(n) \geq cg(n)$$

- Regla de dualidad: $f(n) \in \Omega(g(n)) \iff g(n) \in \mathcal{O}(f(n))$
- Si se está haciendo un **análisis de caso peor** de un algoritmo con complejidad $T(n)$:
  - si se determina que $T(n) \in \mathcal{O}(g(n))$, esto quiere decir que $g(n)$ es una cota superior para todos los casos de ejecución del algoritmo
  - si se determina que $T(n) \in \Omega(h(n))$, esto quiere decir que $h(n)$ es una cota inferior para el caso peor solamente; otros casos podrían estar por debajo de esta cota inferior

Ejemplo de Ω

Análisis de caso peor del algoritmo de ordenación por inserción

```
proc insercion(T[1..n])
   desde i ← 2 hasta n hacer
      x ← T[i]
      j ← i-1
      mientras j > 0 Y x < T[j] hacer
         T[j+1] ← T[j]
         j ← j-1
      fin mientras
      T[j+1] ← x
   fin desde
fin proc
```

- Número de operaciones elementales en el caso peor: $T(n) = 5n^2 + 8n - 11$
- $T(n) \in \mathcal{O}(n^2)$ $(c = 6,\ n_0 = 7)$
- $T(n) \in \Omega(n^2)$ $(c = 1,\ n_0 = 1)$
- Sin embargo, n^2 no es una cota inferior de todos los casos del algoritmo
- $T(n) \in \Omega(n)$. ¿Esta es una cota inferior de todos los casos del algoritmo?

Propiedades de Ω

- $\forall f : \mathbb{N} \to \mathbb{R}^{\geq 0}$, $f(n) \in \Omega(f(n))$
- $f(n) \in \Omega(g(n)) \implies \Omega(f(n)) \subseteq \Omega(g(n))$
- $f(n) \in \Omega(g(n))$ y $g(n) \in \Omega(h(n)) \implies f(n) \in \Omega(h(n))$
- $\Omega(f(n)) = \Omega(g(n)) \iff f(n) \in \Omega(g(n))$ y $g(n) \in \Omega(f(n))$
- $f(n) \in \Omega(g(n))$ y $f(n) \in \Omega(h(n)) \implies f(n) \in \Omega(max(g(n), h(n)))$
- $f_1(n) \in \Omega(g(n))$ y $f_2(n) \in \Omega(h(n)) \implies f_1(n) \cdot f_2(n) \in \Omega(g(n) \cdot h(n))$

Propiedades de Ω (cont.)

- Regla del máximo: $f_1(n) \in \Omega(g(n))$ y $f_2(n) \in \Omega(h(n)) \implies f_1(n) + f_2(n) \in \Omega(max(g(n), h(n)))$
- Regla del límite. Dadas las funciones arbitrarias $f, g : \mathbb{N} \to \mathbb{R}^{\geq 0}$,
  - $\lim_{n\to\infty} \frac{f(n)}{g(n)} \in \mathbb{R}^+ \implies f(n) \in \Omega(g(n))$ y $g(n) \in \Omega(f(n))$
  - $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \implies g(n) \in \Omega(f(n))$, pero $f(n) \notin \Omega(g(n))$
  - $\lim_{n\to\infty} \frac{f(n)}{g(n)} = +\infty \implies f(n) \in \Omega(g(n))$, pero $g(n) \notin \Omega(f(n))$

Notación asintótica Θ

- Si una función $g(n)$ pertenece a $\mathcal{O}(f(n))$ y a $\Omega(f(n))$, entonces pertenece a $\Theta(f(n))$:
- Dada una función de complejidad $f(n)$, $\Theta(f(n))$ es el conjunto de funciones de complejidad $g(n)$ para las que existen $c > 0$, $d > 0$, $n_0 \geq 0$ tales que para todo $n \geq n_0$,

$$cf(n) \leq g(n) \leq df(n)$$

- $\Theta(f(n))$ representa la **complejidad exacta** de una función. En este caso se dice que $g(n)$ está en el orden exacto de $f(n)$
- $\Theta(f(n))$ indica que la función de complejidad está acotada tanto inferiormente como superiormente por una misma función (normalmente con constantes multiplicativas distintas)
- $\Theta(f(n)) = O(f(n)) \cap \Omega(f(n))$

Ejemplo de Θ

Análisis de caso peor del algoritmo de ordenación por inserción

```
proc insercion(T[1..n])
   desde i ← 2 hasta n hacer
      x ← T[i]
      j ← i-1
      mientras j > 0 Y x < T[j] hacer
         T[j+1] ← T[j]
         j ← j-1
      fin mientras
      T[j+1] ← x
   fin desde
fin proc
```

- Número de operaciones elementales en el caso peor: $T(n) = 5n^2 + 8n - 11$
- $T(n) \in \mathcal{O}(n^2)$ $(c = 6,\ n_0 = 7)$
- $T(n) \in \Omega(n^2)$ $(c = 1,\ n_0 = 1)$
- $T(n) \in \Theta(n^2)$

Propiedades de Θ

- $\forall f : \mathbb{N} \to \mathbb{R}^{\geq 0}$, $f(n) \in \Theta(f(n))$
- $g(n) \in \Theta(f(n))$ si y sólo si $f(n) \in \Theta(g(n))$
- $f(n) \in \Theta(g(n))$ y $g(n) \in \Theta(h(n)) \implies f(n) \in \Theta(h(n))$
- Si $c \geq 0, d > 0, g(n) \in \Theta(f(n))$, y $h(n) \in \Theta(f(n))$, entonces
$$c.g(n) + d.h(n) \in \Theta(f(n))$$
- Θ permite clasificar las funciones de complejidad en conjuntos disjuntos, **categorías de complejidad**
- Las categorías de complejidad más comunes se nombran mediante el componente más sencillo:
$\Theta(lg\ n), \Theta(n), \Theta(n\ lg\ n), \Theta(n^2), \Theta(n^k), \Theta(a^n), \Theta(n!), \Theta(n^n)$, donde $k > 2$, $a > 1$

Propiedades de Θ (cont.)

- Regla del máximo: $f_1(n) \in \Theta(g(n))$ y $f_2(n) \in \Theta(h(n)) \implies f_1(n) + f_2(n) \in \Theta(max(g(n), h(n))$
- $f_1(n) \in \Theta(g(n))$ y $f_2(n) \in \Theta(h(n)) \implies f_1(n) \cdot f_2(n) \in \Theta(g(n) \cdot h(n))$
- Regla del límite: Dadas las funciones arbitrarias $f, g : \mathbb{N} \to \mathbb{R}^{\geq 0}$,
  - $\lim_{n\to\infty} \frac{f(n)}{g(n)} \in \mathbb{R}^+ \implies f(n) \in \Theta(g(n))$
  - $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \implies f(n) \in \mathcal{O}(g(n))$, pero $f(n) \notin \Theta(g(n))$
  - $\lim_{n\to\infty} \frac{f(n)}{g(n)} = +\infty \implies f(n) \in \Omega(g(n))$, pero $f(n) \notin \Theta(g(n))$

Introducción al estudio de los algoritmos

1. ¿Qué es un algoritmo?
2. Eficiencia de los algoritmos
3. Análisis de caso mejor, peor y medio
4. Cálculo del número de operaciones elementales
5. Notación asintótica. Órdenes de complejidad
6. Resolución de recurrencias

Funciones de complejidad en algoritmos recursivos

- Al analizar un algoritmo recursivo habitualmente se obtiene la función de complejidad en forma de **recurrencia**
- Por ejemplo:

```
fun factorial(n)
   si n=1 entonces
      devolver 1
   si no
      devolver n*factorial(n-1)
   fin si
fin fun
```

- El número de operaciones elementales en este caso es:

$$T(n) = \begin{cases} 2 & si\ n = 1 \\ 5 + T(n-1) & si\ n > 1 \end{cases}$$

- Para clasificar el algoritmo en una categoría de complejidad, necesitamos resolver este tipo de recurrencias

Métodos para resolver recurrencias

- Veremos algunos métodos para resolver las recurrencias más comunes:
- Expansión (o desplegado) de recurrencias
- Método de la ecuación característica
- Cambio de variable
- Recurrencias típicas
  - Reducción por sustracción
  - Reducción por división

Expansión de recurrencias

- Este método consiste en ir sustituyendo las llamadas recurrentes por su definición, con el objetivo de encontrar una regla general.
- Ejemplo: las torres de Hanoi

```
proc hanoi(m,i,j)
   si m>0 entonces
      hanoi(m-1,i,6-i-j)
      escribir(i," → ",j)
      hanoi(m-1,6-i-j,j)
   fin si
fin proc
```

- Número de operaciones elementales:

$$T(n) = \begin{cases} 1 & \text{si } n = 0 \\ 2\,T(n-1) + 10 & \text{si } n > 0 \end{cases}$$

Expansión de recurrencias (cont.)

- Expandimos la recurrencia para encontrar una regla general:

$$
\begin{aligned}
T(n) &= 2T(n-1) + 10 \\
&= 2 \cdot 2T(n-2) + 2 \cdot 10 + 10 \\
&= 2 \cdot 2 \cdot 2T(n-3) + 2 \cdot 2 \cdot 10 + 2 \cdot 10 + 10 \\
&= \underbrace{2 \cdot \ldots \cdot 2 \cdot 2}_{n} T(0) + \underbrace{\underbrace{2 \cdot \ldots \cdot 2}_{n-1} \cdot 10 + \ldots + 2 \cdot 10}_{n-1} + 10
\end{aligned}
$$

$$
= 2^n + 10 \sum_{i=0}^{n-1} 2^i = 2^n + 10 \frac{2^n - 1}{2 - 1} = 11 \cdot 2^n - 10
$$

Método de la ecuación característica

- Este método es útil para recurrencias lineales, en las que el valor n de la recurrencia depende de los k anteriores
- Se pueden dar dos casos:
  - Recurrencias homogéneas, de la forma:
$$a_0\,T(n) + a_1\,T(n-1) + \ldots + a_k\,T(n-k) = 0$$
  - Recurrencias no homogéneas, de la forma:
$$a_0\,T(n) + a_1\,T(n-1) + \ldots + a_k\,T(n-k) = f(n)$$
- En ambos casos este método consiste en obtener una *ecuación característica*, cuyas raíces nos permitirán obtener la representación no recurrente de la función de complejidad

Recurrencias homogéneas

- Las recurrencias homogéneas son de la forma:

$$a_0 T(n) + a_1 T(n-1) + ... + a_k T(n-k) = 0 \qquad (1)$$

- Haciendo el cambio $T(n) = x^n$, obtenemos:

$$a_0 x^n + a_1 x^{n-1} + ... + a_k x^{n-k} = 0.$$

- $r = 0$ es una raíz trivial sin interés de multiplicidad $n - k$ de la expresión anterior. Las demás raíces son las de la siguiente ecuación:

$$a_0 x^k + a_1 x^{k-1} + ... + a_k = 0 \text{ (**ecuación característica**)}$$

- Sean $r_1, ..., r_k$ las raíces de la ecuación característica (de multiplicidad 1)
- Deshaciendo el cambio, r_i^n son soluciones de (1)
- Por tanto, la forma de la función de complejidad es:

$$\boxed{T(n) = c_1 r_1^n + c_2 r_2^n + ... + c_k r_k^n}$$

- Los coeficientes c_i se obtienen a partir de las condiciones iniciales
- Ejemplo: $T(n) = T(n-1) + T(n-2)$, $T(0) = 0$, $T(1) = 1$ (secuencia de Fibonacci)

Recurrencias homogéneas (cont.)

- Si la ecuación característica tiene raíces de multiplicidad mayor a 1, el método es una generalización del anterior
- Sea la ecuación característica $a_0x^k + a_1x^{k-1} + ... + a_k = 0$ con raíces $r_1, ..., r_k$ de multiplicidad $m_1, ..., m_k$
- Se puede demostrar que la forma de la función de complejidad es:

$$\boxed{T(n) = \sum_{h=0}^{m_1-1} c_{1h}n^h r_1^n + ... + \sum_{h=0}^{m_k-1} c_{kh}n^h r_k^n}$$

- Los coeficientes c_{ij} se obtienen a partir de las condiciones iniciales
- Ejemplo: $T(n) = 5T(n-1) - 7T(n-2) + 3t(n-3)$, $T(0) = 1$, $T(1) = 2$, $T(2) = 3$
  (raíces del pol. característico: $r_1 = 3$ y $r_2 = 1$ con mult. 2)

Recurrencias no homogéneas

- Las recurrencias no homogéneas son de la forma:

$$a_0 T(n) + a_1 T(n-1) + \ldots + a_k T(n-k) = f(n)$$

donde $f(n)$ es distinta de la función nula ($f(n) \neq 0$ para algún n)

- No se conoce una solución general para cualquier $f(n)$, pero sí para las de la forma:

$$a_0 T(n) + a_1 T(n-1) + \ldots + a_k T(n-k) = b^n p(n)$$

donde $p(n)$ es un polinomio de grado d en n

- La ecuación característica en este caso es:

$$(a_0 x^k + a_1 x^{k-1} + \ldots + a_k)(x-b)^{d+1} = 0$$

- Se procede igual que en las recurrencias homogéneas
- Ejemplo: $T(n) = 3T(n-1) + 4^n(2n+1),\ T(0) = 0$
- Otro ejemplo: Hanoi: $T(n) = 2T(n-1) + 10,\ T(0) = 1$
- El caso en el que $f(n)$ tenga la forma $b_1^n p_1(n) + \cdots + b_s^n p_s(n)$, cada término contribuye a la ecuación característica.

Cambio de variable

- Consiste en hacer un cambio de variable para transformar la recurrencia original $T(n)$ en otra recurrencia t_k para la que se pueden aplicar las técnicas anteriores.
- Lo veremos con un ejemplo: $T(n) = 3T(n/2) + n,\ T(1) = 1$
- Supongamos que n es potencia de 2 y hagamos el cambio de variable $n = 2^i$ (y por tanto $i = lg\ n$):

$$T(2^i) = 3T(\tfrac{2^i}{2}) + 2^i \quad T(2^i) = 3T(2^{i-1}) + 2^i$$

- Si llamamos $t_i = T(2^i)$, tenemos

$$t_i = 3t_{i-1} + 2^i$$

- Esta es una recurrencia no homogénea, con solución

$$t_i = c_1 3^i + c_2 2^i$$

- Deshacemos el cambio de variable:
  $T(n) = c_1 3^{lg\ n} + c_2 2^{lg\ n} = c_1 n^{lg\ 3} + c_2 n$
- Aplicando las condiciones iniciales, obtenemos $T(n) = 3n^{lg\ 3} - 2n$

Cambio de variable (cont.)

- Hemos supuesto que n es una potencia exacta de 2. El resultado obtenido lo podemos extrapolar a cualquier valor de n bajo ciertas condiciones:
  - Definición: Una función $f(n)$ es **no decreciente** si para todo $n_1 > n_2$ se cumple que $f(n_1) \geq f(n_2)$
  - Definición: Una función $f(n)$ es **eventualmente no decreciente** si es no decreciente a partir de cierto valor $N > 0$
  - Definición: Una función es **suave** (smooth) si es eventualmente no decreciente y se cumple que $f(2n) \in \Theta(f(n))$
  - Ejemplos: $lg\ n, n, n\ lg\ n$ y n^k $(k \geq 0)$ son suaves. 2^n no lo es.
  - Se puede demostrar lo siguiente:
    Sea $b \in \mathbb{Z}, b \geq 2$, sea $f(n)$ una función suave, y sea $T(n)$ una función *eventualmente* no decreciente. Si

    $$T(n) \in \Theta(f(n)), \text{ para } n \text{ potencia de } b,$$

    entonces,

    $$T(n) \in \Theta(f(n)).$$

    Este resultado también es aplicable para $\mathcal{O}$ y Ω

Ecuaciones típicas

- **Reducción por sustracción**: Si una recurrencia tiene la forma:

$$T(n) = \begin{cases} c & si\ 0 \leq n < b \\ aT(n-b) + p(n) & si\ n \geq b \end{cases}$$

donde $a, c \in \mathbb{R}^+$, $p(n)$ es un polinomio de grado k, y $b \in \mathbb{N}$

entonces se tiene que: $T(n) \in \begin{cases} \Theta(n^k) & \text{si } a < 1 \\ \Theta(n^{k+1}) & \text{si } a = 1 \\ \Theta(a^{n\ div\ b}) & \text{si } a > 1 \end{cases}$

- **Reducción por división**: Si una recurrencia tiene la forma:

$$T(n) = \begin{cases} c & si\ 1 \leq n < b \\ aT(n/b) + f(n) & si\ n \geq b \end{cases}$$

donde $a, c \in \mathbb{R}^+$, $f(n) \in \Theta(n^k)$, $b \in \mathbb{N}$, $b > 1$

entonces se tiene que: $T(n) \in \begin{cases} \Theta(n^k) & \text{si } a < b^k \\ \Theta(n^k log\ n) & \text{si } a = b^k \\ \Theta(n^{log_b\ a}) & \text{si } a > b^k \end{cases}$