Parte de Algoritmos
de la asignatura de Programación
Master de Bioinformática

Análisis de Algoritmos

Web asignatura: http://dis.um.es/~domingo/algbio.html

E-mail profesor: domingo@um.es

Transparencias preparadas a partir de las del curso de
Algoritmos y Estructuras de Datos II, del Grado de Ingeniería Informática

BIBLIOGRAFÍA

- Texto guía de Algoritmos y Estructuras de Datos, volumen 2

- Brassard, Bratley: Fundamentos de algoritmos.

- Cormen, Leiserson, Rivest: Introduction to Algorithms. The MIT Press. 1990.

ALGORÍTMICA o ALGORITMIA

- Estudia:

  › Diseño de algoritmos: esquemas para resolver problemas. Divide y vencerás, avance rápido, programación dinámica, Backtracking, Branch & Bound...

  › Análisis de algoritmos: recursos necesarios para resolver el problema con el algoritmo elegido. Ocupación de memoria, tiempo de ejecución.
  De manera que se pueda decidir qué algoritmo es mejor para nuestro problema y entrada.

ALGORITMO

- Conjunto de reglas para resolver un problema. Propiedades:

  - **Definibilidad**. El conjunto debe estar bien definido, sin dejar dudas en su interpretación.
  - **Finitud**. Tener número finito de pasos, que se ejecute en un tiempo finito.
  - **Eficiencia.** Debemos diseñar algoritmos eficientes: que el tiempo para acabar sea "razonable" (**análisis de algoritmos**).
  - **Determinista** si para la misma entrada produce siempre la misma salida. No determinista en caso contrario (redondeos, probabilistas, paralelos…)

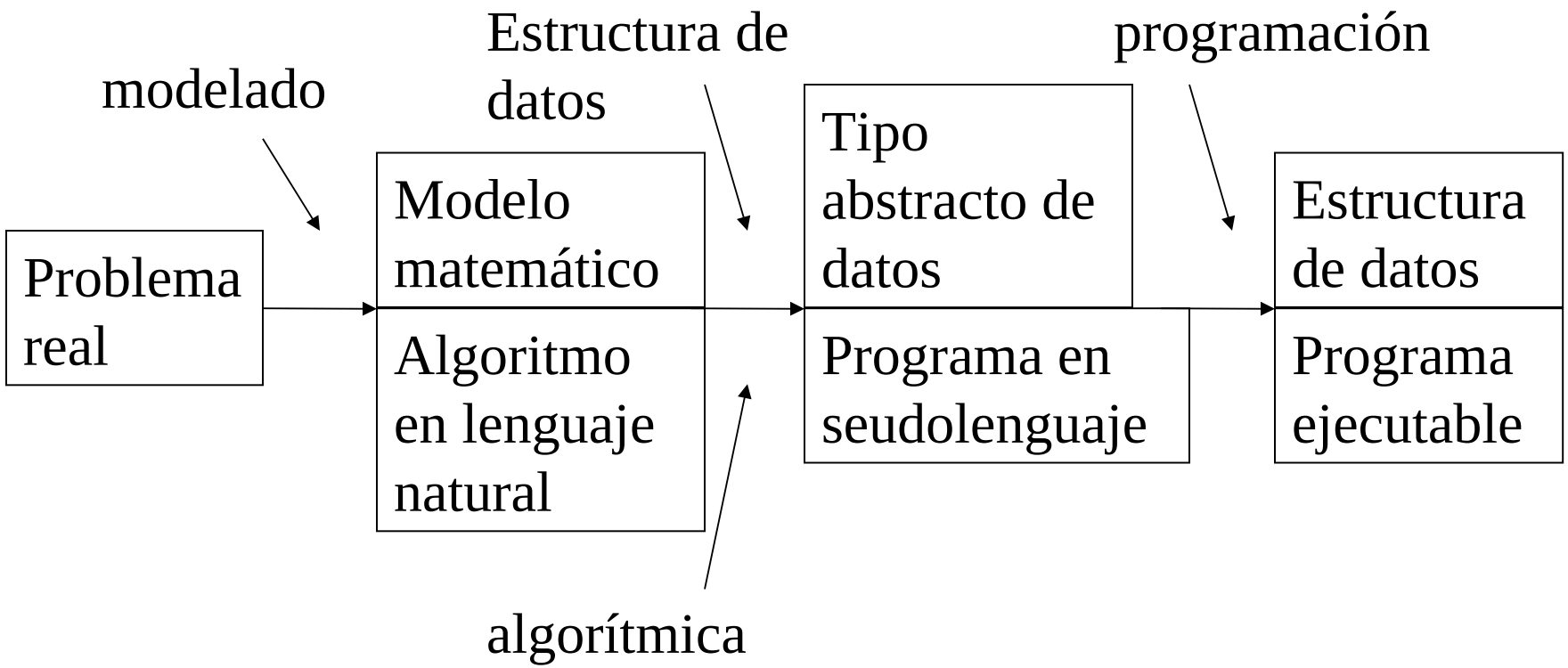
PROGRAMACIÓN

- Los algoritmos no son los únicos componentes en la resolución de un problema:

Algoritmos + Estructuras de Datos = Programas

› **Estructura de datos**: parte estática, almacenada
› **Algoritmo**: parte dinámica, manipulador de los datos

PROCESO DE PROGRAMACIÓN



Estudio de algoritmos

- Se trata de diseñar algoritmos eficientes: que usan pocos recursos:

  - memoria principal
  - tiempo de ejecución
  - otros (memoria secundaria, entradas/salidas, tiempo de programación, comunicaciones, procesadores…)

- Para decidir:

  - qué algoritmo programar
  - qué algoritmo utilizar en resolver un problema para una cierta entrada
  - detectar fallo en programa que no funciona bien

Tiempo de ejecución

- Se intenta estimar el tiempo de ejecución para un cierto tamaño de entrada, t(n)
- Se pueden considerar tamaños de entrada distintos:
  - Cálculo del factorial de un número n. Aunque es un único número, el tamaño es el valor n
  - Ordenación de n datos. El tamaño puede ser el número de datos n, aunque el valor que tienen también influye en el tiempo
  - Multiplicación de matrices rectangulares nxm y mxr. El tamaño viene dado por tres valores: t(n,m,r)
  - Multiplicación de matrices con programa paralelo usando p procesadores. Influye el tamaño del problema (n,m,r) y el del sistema (p): t(m,n,r,p)

Tiempo de ejecución

- Influyen factores externos al algoritmo:
  - Compilador y opciones de compilación
  - Máquina
  - Programador (uso de registros, gestión de accesos a memoria…)
- por lo que normalmente se estudia la forma en que crece t(n), y se utilizan **notaciones asintóticas**:
  - Ω, cota inferior de la forma en que crece
  - O, cota superior de la forma en que crece
  - θ, forma exacta en que crece
  - o, forma exacta en que crece afectada de la constante del término de mayor orden (para comparar algoritmos cuyo tiempo crece de la misma forma)

Tiempo de ejecución

- Pueden estudiarse:

Caso más favorable $t_m(n)$, tiempo para la entrada de datos que produce el menor tiempo

Caso más desfavorable $t_M(n)$, para la que da el mayor tiempo

Tiempo promedio $t_p(n)$, media de tiempos de todas las entradas posibles

Para cada uno de ellos se pueden usar las notaciones asintóticas

Lo mismo para el estudio del uso de memoria

Tiempo de ejecución

- **Conteo de instrucciones**: Se cuenta el número de operaciones, o las de un cierto tipo, o cada operación afectada de un coste diferente

- Todo lo dicho para tiempo de ejecución es válido para **ocupación de memoria**: $m_m(n)$, $m_M(n)$, $m_p(n)$ y notaciones asintóticas
  - Se cuenta la máxima cantidad de memoria ocupada durante la ejecución del programa: memoria mínima que necesitaremos para poder ejecutarlo

Conteo de instrucciones

Reglas básicas:

Número de instrucciones t(n) → sumar 1 por cada instrucción o línea de código de ejecución constante.

Tiempo de ejecución t(n) → sumar una constante (c_1, c_2...) por cada tipo de instrucción o grupo de instrucciones.

Bucles FOR: Se pueden expresar como un sumatorio, con los límites del FOR como límites del sumatorio.

$$\sum_{i=1}^{n} k = kn \qquad \sum_{i=a}^{b} k = k(b-a+1) \qquad \sum_{i=1}^{n} i = n(n+1)/2$$

$$\sum_{i=a}^{b} r^i = \frac{r^{b+1} - r^a}{r - 1} \qquad \sum_{i=1}^{n} i^2 \approx \int_{0}^{n} i^2 \, di = (i^3)/3 \Big]_{0}^{n} = (n^3)/3$$

Ej: Cálculo del factorial

```
fact(n):
    si n=1
        devolver 1
    en otro caso
        devolver (fact(n-1)*n)
```

- Tamaño de la entrada: n
- Tiempo:

$t(n)=t(n-1)+a=t(n-2)+2*a= ... =t(1)+(n-1)*a$

- Memoria:

$m(n)=m(n-1)+c=m(n-2)+2*c= ... =m(1)+(n-1)*c$

Usamos **expansión de recurrencia**

$\rightarrow$ Ejecutar el programa de cálculo de factorial. Ver las funciones de toma de tiempo, buscar si hay otras alternativas para la toma de tiempos.

Ej. Números de Fibonacci

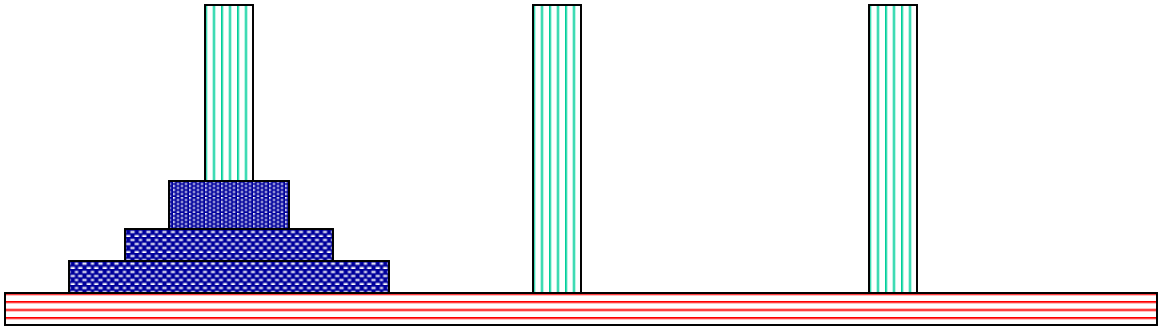
```
Funcion Fib(N: int): int;
if N<0 then
   error('No válido')
case N of
   0, 1: return N
else
   fnm2= 0
   fnm1= 1
   for i= 2 to N
      fn= fnm1 + fnm2
      fnm2= fnm1
      fnm1= fn
   end
   return fn
end
```

```
Funcion Fib(N: int): int;
if N<0 then
   error('No válido')
case N of
   0, 1: return N
else
  return Fib(N-1)+Fib(N-2)
End
```

→ Estudiar teóricamente el coste de tiempo y memoria de las dos versiones

→ Programarlas y comparar su tiempo experimentalmente

Ej: Torres de Hanoi



```
Hanoi(origen,destino,pivote,discos):
    si discos=1
        moveruno(origen,destino)
    en otro caso
        Hanoi(origen,pivote,destino,discos-1)
        moveruno(origen,destino)
        Hanoi(pivote,destino,origen,discos-1)
```

Ej: Torres de Hanoi

Tamaño del problema: número de discos

Tiempo a estudiar: número de movimientos de discos

t(1)=1 caso base de la recurrencia

t(n)=2t(n-1)+1, si n>1 ecuación de recurrencia

Expandiendo la recurrencia:

$t(n) = 2\ t(n-1) + 1 =$

$2\,(2t(n-2)+1) + 1 = 2^2\ t(n-2) + 1+2 =$

$2^2\,(2t(n-3)+1) + 1+2 = 2^3\, t(n-3)+1+2+2^2$

....

$2^{n-1}\, t(1)+1+2+\ldots+ 2^{n-2} = \mathbf{2^n-1}$

¿ocupación de memoria?

Normalmente la ocupación de memoria es más simple de estudiar y de menor orden de complejidad

NOTACIONES ASINTÓTICAS

- Se utilizan para estudiar tiempos de ejecución u ocupaciones de memoria.

- Representan la forma en que crece una función.

- En general no consideran constantes, que no son valores propios de los algoritmos.

- Son asintóticas pues representan el comportamiento cuando el tamaño de la entrada tiende a infinito, pues para valores grandes es cuando puede haber problemas de tiempo o memoria.

NOTACIÓN *O*

- Da una **cota superior** de la forma en que crece el tiempo de ejecución. Pero es aplicable a cualquier función, no solo de tiempos de ejecución

- DEFINICIÓN:

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

Se denota *O(f)*, y será:

$$O(f)=\{t: N \rightarrow R^+ / \exists c \in R^+, \exists n_0 \in N, \forall n >= n_0 : t(n) <= cf(n)\}$$

Es un **conjunto de funciones**, no una función

NOTACIÓN Ω

- Da una **cota inferior** de la forma en que crece el tiempo de ejecución.

- DEFINICIÓN:

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

Se denota $\Omega(f)$, y será:

$$\Omega(f)=\{t: N \rightarrow R^+ / \exists c \in R^+, \exists n_0 \in N, \forall n >= n_0 : t(n) >= cf(n)\}$$

NOTACIÓN θ (orden exacto)

- Da la forma en que crece el tiempo de ejecución.

- Son las funciones a las que *f* acota superior e inferiormente:

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

Es equivalente a:

$\theta(f) = \{t: N \to R^+ / \exists c,d \in R^+, \exists n_0 \in N, \forall n >= n_0 : cf(n) <= t(n) <= df(n)\}$

- No hay relaciones de inclusión.

- Si $t \in O(f)$ y $t \in \Omega(f)$ entonces $t \in \theta(f)$

NOTACIÓN *o* (o pequeña)

- Se utiliza para comparar tiempos con el mismo orden. Considera la constante que afecta al término de mayor orden.

- Aparecen constantes, por lo que es necesario hacer suposiciones sobre: coste de las operaciones, conteo de instrucciones, …

- Se define: $o(f)=\{t: N \rightarrow R^{+} / lim_{n \rightarrow \infty} t(n)/f(n)=1\}$

- No hay relaciones de inclusión.

COMPARACIÓN DE ÓRDENES

- $lim_{n \to \infty} f(n)/g(n) \in R^{+} \Rightarrow O(f)=O(g),\ \Omega(f)=\Omega(g),\ \theta(f)=\theta(g)$
- $lim_{n \to \infty} f(n)/g(n)=0 \Rightarrow O(f) \subseteq O(g),\ \Omega(g) \subseteq \Omega(f)$
- $lim_{n \to \infty} f(n)/g(n)=+\infty \Rightarrow O(g) \subseteq O(f),\ \Omega(f) \subseteq \Omega(g)$

COMPARACIÓN DE ÓRDENES

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

$$O((\log n)^p) \subset O(n^{1/q})$$

- Con Ω cambia el sentido de las inclusiones

Ecuaciones de recurrencia

- Es normal que un algoritmo se base en procedimientos auxiliares, haga llamadas recursivas para tamaños menores o reduzca el tamaño del problema progresivamente.
- En el análisis, el tiempo T(n) se expresa en función del tiempo para T(n-1), T(n-2)...→ **Ecuaciones de recurrencia.**

- **Ejemplo**. Torres de Hanoi

  Hanoi (origen,pivote,destino,discos)

  si discos=1

  mover(origen,destino)

  en otro caso

  Hanoi (origen,destino,pivote,discos-1)
  mover (origen,destino)
  Hanoi (pivote,origen,destino,discos-1)

Ecuaciones de recurrencia

- En general, las ecuaciones de recurrencia tienen la forma:

  $t(n) = b$ Para $0 \leq n \leq n_0$ **Casos base**

  $t(n) = f(t(n), t(n-1), ..., t(n-k), n)$ En otro caso

- **Tipos de ecuaciones de recurrencia:**

  Lineales homegéneas:

  $a_0t(n) + a_1t(n-1) + ... + a_kt(n-k) = 0$

  Lineales no homegéneas:

  $a_0t(n) + a_1t(n-1) + ... + a_kt(n-k) = p(n) + ...$

  No lineales:

  Ejemplo: $a_0t^2(n) + t(n-1)*t(n-k) + sqrt(t(n-2) + 1) = p(n)$

Ecuaciones lineales homogéneas

- La ecuación de recurrencia es de la forma:
  $a_0 t(n) + a_1 t(n-1) + ... + a_k t(n-k) = 0$; a_i constante

- **Solución:** $t(n) = x^n$

$t(n) = x \cdot t(n-1) = x \cdot x \cdot t(n-2) = x^3\, t(n-3) = ... = x^n \cdot t(0) \Rightarrow$
$t(n) = x^n$

$\rightarrow$ Indicar qué valores aparecen en los ejemplos que hemos visto: factorial, Hanoi, Fibonacci…

Ecuaciones lineales homogéneas

- Sean las soluciones $x = (s_1, s_2, ..., s_k)$, todas distintas.
- La solución será:

$$\mathbf{t(n) = c_1 \cdot s_1^n + c_2 \cdot s_2^n + ... + c_k \cdot s_k^n = \sum_{i=1}^{k} c_i \cdot s_i^n}$$

- Siendo $\mathbf{c_i}$ constantes, cuyos valores dependen de los casos base (condiciones iniciales).

- Son constantes que añadimos nosotros. Debemos resolverlas, usando los casos base de la ecuación recurrente, o casos que se obtienen a partir de los casos base, y tendremos que asegurarnos que se llegue a estos casos tras sucesivas llamadas recursivas.

Ecuaciones lineales homogéneas

- **Ejemplo**. El tiempo de ejecución de un algoritmo es:

$$t(n) = \begin{cases} 0 & \text{Si } n = 0 \\ 1 & \text{Si } n = 1 \\ 3 \cdot t(n-1) + 4 \cdot t(n-2) & \text{Si } n > 1 \end{cases}$$

$\rightarrow$ Encontrar una fórmula explícita para t(n), y calcular el orden de complejidad del algoritmo.

Ecuaciones lineales homogéneas, soluciones múltiples

- Si $\mathbf{s_i}$ tiene multiplicidad **m**, entonces tendremos soluciones básicas:

  $s_i^n \quad n \cdot s_i^n \quad n^2 \cdot s_i^n \quad ... \quad n^{m-1} \cdot s_i^n$

- Dadas las soluciones $x = (s_1, s_2, ..., s_k)$ siendo $\mathbf{s_k}$ de multiplicidad **m**, la solución será:

  $$\mathbf{t(n)} = c_1 \cdot s_1^n + c_2 \cdot s_2^n + ... + c_k \cdot s_k^n + c_{k+1} \cdot n \cdot s_k^n +$$
  $$+ c_{k+2} \cdot n^2 \cdot s_k^n + ... + c_{k+1+m} \cdot n^{m-1} \cdot s_k^n$$

→ Calcular t(n) y el orden de complejidad para:

$t(n) = 5\ t(n-1) - 8\ t(n-2) + 4\ t(n-3)$

$t(0) = 0,\ t(1) = 3,\ t(2) = 10$

Recurrencias no homogéneas

- ¿Qué pasa si tenemos algo como $t(n) = 2 \cdot t(n-1) + 1$?

- Si en la ecuación de recurrencia aparece un término de la forma $\mathbf{b^n \cdot p(n)}$ ($p(n)$ polinomio de n), entonces en la ecuación característica habrá un factor:

  $(x-b)^{\mathbf{Grado(p(n))+1}} \rightarrow$ Sol. **b** con multiplicidad Grado(p(n))+1

  $\rightarrow$ Obtener el tiempo de ejecución para el algoritmo de las torres de Hanoi y el Fibonacci recursivo

Recurrencias no homogéneas

- En general, tendremos recurrencias de la forma:

  $a_0t(n) + a_1t(n-1) + ... + a_kt(n-k) = b_1^n p_1(n) + b_2^n p_2(n) + ...$

- Y la ecuación característica será:

  $(a_0x^k + a_1x^{k-1} + ... + a_k)(x-b_1)^{G(p_1(n))+1}(x-b_2)^{G(p_2(n))+1}... = 0$

→ Calcular t(n) y O(t(n)).

$t(n) = 1 + n$ $\qquad$ $n = 0, 1$

$t(n) = 4t(n-2) + (n+5)3^n + n^2$ $\qquad$ Si $n>1$

Cambio de variable

$t(n) = a \cdot t(n/4) + b \cdot t(n/8) + ...$

Cambio de variable:

- Convertir las ecuaciones anteriores en algo de la forma $t'(k) = a \cdot t'(k-c_1) + b \cdot t'(k-c_2) + ...$
- Resolver el sistema en **k**.
- Deshacer el cambio, y obtener el resultado en **n**

Cambios típicos:

- $n = 2^k$; $k = \log_2 n$
- $n = 5k$; $k = n/5$

Cambio de variable

- **Ejemplo.** Resolver:

$t(n) = a$ si $n=1$

$t(n) = 2\, t(n/2) + b \cdot n$ si $n>1$, con $b>0$

$\rightarrow$ Resolver:

$t(n) = n$ si $n<b$

$t(n) = 3 \cdot t(n/b) + n^2 + 1$ en otro caso

Fórmulas maestras

- Dada una ecuación de recurrencia con valores constantes, se obtiene su orden dependiente de esos valores:

Ejemplo:

con a, b, c y d $\in R^+$ y e, $n_0 \in N^+$ si

$$f(n) = \begin{cases} d & \text{Si } n \leq n_0 \\ a \cdot f(n-e) + bn + c & \text{Si } n > n_0 \end{cases}$$

Se tiene: $a < 1 \Rightarrow f \in O(n)$

$a = 1 \Rightarrow f \in O(n^2)$

$a > 1 \Rightarrow f \in O(a^{n/e})$

→ Aplicarlo al cálculo del tiempo de las torres de Hanoi.

Cuestiones

→ ¿Qué podemos decir sobre el tiempo de ejecución la ordenación por Quicksort en el caso más desfavorable y en promedio?

→ Comparar experimentalmente el tiempo de ejecución de la función sort de Perl y la rutina del Quicksort.

→ ¿Qué podemos decir sobre el tiempo de ejecución de algunos de los algoritmos de manejo de cadenas vistos en los temas anteriores?