ALGORÍTMICA

Tema 1. Planteamiento General

Tema 2. La Eficiencia de los Algoritmos

Tema 3. Algoritmos “Divide y vencerás”

Tema 4. Algoritmos Voraces (“Greedy”)

Tema 5. Algoritmos para la Exploración de Grafos (“Backtraking”, “Branch and Bound”)

Tema 6. Algoritmos basados en Programación Dinámica

Tema 7. Otras Técnicas Algorítmicas de Resolución de Problemas

Tema 2: La Eficiencia de los Algoritmos

Bibibliografía:

G. BRASSARD, P. BRATLEY. Fundamentos de Algoritmia. Prentice Hall (1997.
J.L. VERDEGAY. Curso de Teoría de Algoritmos. Universidad de Granada (2004).

Objetivo General de la asignatura:

Dado un problema,

Seleccionar el algoritmo adecuado para resolverlo e implementarlo.

¿Qué es un algoritmo?

- Algoritmo (Diccionario R.A.E., 22a ed.):
  - 1. m. Conjunto ordenado y finito de operaciones que permite hallar la solución de un problema.
- Características que debe tener un algoritmo:
  - 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

Ejemplo

- Se tienen *n* bolas de igual tamaño, todas ellas de igual peso salvo una más pesada. Como único medio para identificar esta bola singular se dispone de una balanza romana clásica (sólo indica si algo pesa más, menos o igual que otra cosa).

Diseñar un algoritmo que permita determinar cuál es la bola más pesada

Ejemplo: Elementos duplicados

Sea A[1..n] un vector de enteros.

Determinar si existen dos índices distintos i y j, tales que A[i] = A[j]

Diseñar un algoritmo para solucionar el problema

Elementos duplicados?

Algoritmo 1

```
for (i=0;i<n;i++)
    for (j= i+1; j<n;j++)
        if A[i] == A[j]
            return true
return false
```

Algoritmo 2

- Ordenar A
- for (i=0; i<n; i++)
  if A[i] == A[i+1]
  return true
- return false

Comparación de algoritmos

- Es frecuente disponer de más de un algoritmo para resolver un problema dado:
- ¿Con cuál nos quedamos?
- Estudio de los recursos:
  - **Tiempo**
  - Espacio

Índice

Introducción

- Objetivo: analizar la eficiencia de un algoritmo en función del tamaño de las entradas.

- Empírica
  - Implementar cada algoritmo y tomar tiempos !!!
- Teórica
  - estamos interesados en contar cuantas instrucciones simples (asignaciones, comparaciones, etc.) se ejecutan, asumimos que independientemente de la máquina en que se ejecute, todas las operaciones simples consumen el mismo tiempo.

Análisis de Eficiencia

- Ventajas:
  - Mejor comprensión de los algoritmos
  - Diseñar algoritmos mejores
  - Determinar la escalabilidad

- El tiempo de ejecución de un algoritmo depende
  - El problema que tratamos de resolver
  - El lenguaje de programación utilizado
  - El compilador, el hardware utilizado
  - La habilidad del programador

¿Sería una buena idea considerar los seg como unidad de medida?

Eficiencia de algoritmos

- 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:

Dos implementaciones distintas de un mismo algoritmo no difieren en eficiencia más que en una constante multiplicativa.

Tamaño de datos?

- 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 bits (por ejemplo, Fibonacci)

Tipos de Análisis

- Peor de los Casos: Se corresponde con el peor tiempo. T(n) es el tiempo máximo sobre las entradas.
- Mejor Caso: Límite inferior en el tiempo. T(n) es el menor tiempo de todas las posibles entradas
- Caso Promedio: Es el tiempo medio esperado sobre todas las posibles entradas de tamaño n. Se considera una distribución de probabilidad sobre las entradas.

Tipos de Análisis

- Análisis Probabilístico: Es el tiempo de ejecución esperado para una entrada aleatoria. Se expresa tanto el tiempo de ejecución y la probabilidad de obtenerlo.

- Análisis Amortizado: El tiempo que se obtiene para un conjunto de ejecuciones, dividido por el número de ejecuciones.

Análisis Asintótico

- Dos algoritmos para un mismo problema:
- Algoritmo 1: $T(n) = 10^{-4}\ 2^n$ s

  n=38, T(n) = 1 año

- Algoritmo 2: $T(n) = 10^{-2}\ n^3$ s

  n=1000, T(n) = 1 año
- Se precisa **análisis asintótico**

Notación Asintótica

- Estudia el comportamiento del algoritmo cuando el tamaño de las entradas, n, es lo suficientemente grande, sin tener en cuenta lo que ocurre para entradas pequeñas y obviando factores constantes.

  comparar el tiempo de ejecución del algoritmo con la curva que define una función, f(n), para entradas de tamaño n.

- Notación O
- Notación Omega
- Notación Theta:

Notación Asintótica

- Notación O

$$T(n) \in O(f(n)) \text{ sii } \exists c > 0, n_o \in \mathcal{N} \text{ tal que } \forall n \geq n_o T(n) \leq cf(n)$$

- Notación Omega

$$T(n) \in \Omega(f(n)) \text{ sii } \exists c > 0, n_o \in \mathcal{N} \text{ tal que } \forall n \geq n_o \ T(n) \geq cf(n).$$

Notación Theta

$$T(n) \in \Theta(f(n)) \text{ sii } T(n) \in O(f(n)) \text{ y } T(n) \in \Omega(f(n))$$

Propiedades de Notación Asint.

Transitividad: $f(n) \in O(g(n))$ y $g(n) \in O(h(n)) \Rightarrow f(n) \in O(h(n))$.Idem para $\Theta y \Omega$.

Reflexiva: $f(n) \in O(f(n))$.Idem para $\Theta y \Omega$.

Simetrica: $f(n) \in \Theta(g(n)) sii g(n) \in \Theta(f(n))$.

Suma: Si $T1(n) \in O(f(n))$ y $T2(n) \in O(g(n))$,

entonces $T1(n) + T2(n) \in O(\max\{f(n), g(n)\})$.

Producto: Si $T1(n) \in O(f(n))$ y $T2(n) \in O(g(n))$,

entonces $T1(n) \times T2(n) \in O(f(n) \times g(n))$.

Ejemplos en O(.)

- $T(n) = (n + 1)^2$ es $O(n^2)$
- $T(n) = 3n^3 + 2n^2$ es $O(n^3)$
- $T(n) = 3^n$ no es $O(2^n)$

Flexibilidad en la notación: Emplearemos la notación O(f(n)) aun cuando en un número finito de valores de n, f(n) sea negativa o no esté definida. Ej.: n/log(n)

Órdenes más habituales

- Lineal: n
- Cuadrático: n^2
- Polinómico: n^k, con k
- Logarítimo: log(n)
- Exponencial: c^n

log(n)	n	n^2	n^5	2^n
1	2	4	32	4
2	4	16	1024	16
3	8	64	32768	256
4	16	256	1048576	65536
5	32	1024	33554432	4.29E+09
6	64	4096	1.07E+09	1.84E+19
7	128	16384	3.44E+10	3.4E+38
8	256	65536	1.1E+12	1.16E+77
9	512	262144	3.52E+13	1.3E+154
10	1024	1048576	1.13E+15	#NUM!

Notación O: Ejemplos

O(1) Constante: Algunos algoritmos de búsqueda, como por ejemplo Hashing o búsqueda del menor elemento en un Árbol Parcialmente Ordenado (Heap)

O(log n) logarítmico: Algoritmo de búsqueda binaria, inserción o borrado en un Heap

O(n) lineal: Búsqueda Secuencial.

O(n log(n)) : Algoritmos de ordenación Mergesort, Heapsort

$O(n^2)$ Cuadrático: Algoritmos de ordenación Burbuja, Inserción o Selección

$O(n^k)$, si $k>0$ Polinomial: Multiplicación de matrices (k=3, cúbico)

$O(b^n)$ Exponencial: Fibonacci, Torres de Hanoi.

O(n!) Factorial: Problemas de permutaciones

Operación elemental

- Operación de un algoritmo cuyo tiempo de ejecución se puede acotar superiormente por una constante.

  En nuestro análisis sólo contará el número de operaciones elementales y no el tiempo exacto necesario para cada una de ellas.

Consideraciones sobre operaciones elementales

- En la descripción de un algoritmo puede ocurrir que una línea de código corresponda a un número de variable de operaciones elementales.

  Por ejemplo, si A es un vector con n elementos, y queremos calcular

  x = max{A[k], 0 <= k < n}

  el tiempo para hacerlo depende de n, no es constante

Consideraciones sobre operaciones elementales

- Algunas operaciones matemáticas no deben ser tratadas como tales operaciones elementales.
  - Por ejemplo, el tiempo necesario para realizar sumas y productos crece con la longitud de los operandos.
- No obstante, consideraremos las operaciones suma, diferencia, producto, cociente, módulo, operaciones booleanas, comparaciones y asignaciones como elementales, salvo que explícitamente se establezca otra cosa.

Ej: Elemento mayoritario

```
bool  Mayoritario(vector<int> A; int prim, int ult){
  /* supone que el vector esta ordenado */
  int mitad, i;    bool existe = false;
  if (prim==ult) existe= true;
  else {
       mitad=(prim+ult)/2;
       for (i=mitad; i<ult && ! existe;i++)
            if (a[i]==a[i-mitad+1])
                 existe = true;
  }
  return existe;
}
```

¿Cual es la op. crítica?

Cual es la op. crítica?

```
bool  Mayoritario(vector<int> A; int prim, int ult){
  /* supone que el vector esta ordenado */
  int mitad, i;    bool existe = false;
  if (prim==ult) existe= true;
  else {
       mitad=(prim+ult)/2;
       for (i=mitad; i<ult && ! existe;i++)
            if (a[i]==a[i-mitad+1])
                 existe = true;
  }
  return existe;
}
```

Reglas para cálculo eficiencia

- Como regla general, empezar por la parte más interna del algoritmo y se avanza (mediante sucesivas aplicaciones de la regla de la suma y/o el producto) hacia las partes más externas.
  - Sentencias Simples
  - Secuencias
  - Sentencias Condicionales
  - Bucles
  - Funciones no recursivas
  - Funciones Recursivas

Sentencia Simple

- Su tiempo de ejecución está acotado superiormente por una constante, O(1).
  - Operaciones de lectura, escritura, asignaciones, etc. y de forma genérica todas aquellas operaciones en las que no sea relevante el tamaño del problema considerado

Secuencia de sentencias

- Se aplica la regla de la suma, que nos indica que el orden de todo el bloque es el máximo de los ordenes de eficiencia de dicho bloque.

```
cin >> x;
x = x+1;
z = 2*x;
```

Sentencia condicional

- *Si (condición)*
  Entonces Accion_Si
  Sino Accion_No.

- pertenece al orden del

  max{ O(cond),O(Acc_SI), O(Acc_NO) }.

Bucle

- Se aplica la regla del producto.

El tiempo asociado se obtiene como producto del orden del numero de iteraciones que realiza el bucle por el orden del tiempo de ejecución del conjunto de sentencias que forman el cuerpo del bucle.

Llamadas a funciones

- Se analiza primero el orden de eficiencia de las funciones a las que se llama y posteriormente , considerando las reglas anteriores, se pasa a calcular el tiempo de ejecución de la función que las llama.

- Problema: Funciones Recursivas.

Ejemplo: Evaluar un Polinomio

```
class Polinomio {
  private:
     vector<double>  coeficientes;
     // FA: a0 + a1 x + ... + an x^n ---> coef[i] = ai
public:
    double evalua_1 (double x) ;
}
```

Ejemplo: Eval. polinomio

```
double evalua_1 (double x) {
   double resultado= 0.0;
   for (int ter= 0; ter < coeficientes.size(); ter++) {
     double xn= 1.0;
     for (int j= 0; j < ter; j++)
              xn*= x;          // x elevado a n
     resultado += coeficientes[ter] * xn;
   }
```

```
double evalua_2 (double x) {
    double xn= 1.0;
    double resultado= coeficientes[0];
    for (int ter= 1; ter < coeficientes.size(); ter++) {
        xn*= x;
        resultado+= coeficientes[ter] * xn;
    }
    return resultado;
  }
```

Como $1+2x+3x^2 = 1 + x\,(2+3x)$ .....

```
double evalua_3 (double x) {
   double resultado= 0.0;
   for (int ter= coeficientes.size()-1; ter >= 0; ter--)
   {
     resultado= resultado * x +coeficientes[ter];
   }
   return resultado;
 }
```

- Evalua_2 y Evalua_3 tienen idéntico orden de complejidad, pero sus tiempos de ejecución serán distintos.
  - Evalua_3 ejecuta N multipl. y N sumas,
  - Evalua_3 requiere 2N multipl. y N sumas.
  - Si, como es frecuente, el tiempo de ejecución es notablemente superior para realizar una multiplicación, cabe razonar que el último algoritmo ejecutará en menos tiempo

grado	evalua_1	Evalua_2	Evalua_3
1	0	0	0
2	10	0	0
5	0	0	0
10	0	0	0
20	0	0	0
50	40	0	10
100	130	0	0
200	521	0	10
500	3175	10	10
1000	63632	872	580

Llamadas a funciones

- Se analiza primero el orden de eficiencia de las funciones a las que se llama y posteriormente , considerando las reglas anteriores, se pasa a calcular el tiempo de ejecución de la función que las llama.

- Problema: Funciones Recursivas.

Funciones recursivas

- Las funciones de tiempo que se obtienen son también recursivas

  Expresiones que representan el tiempo de ejecución de un algoritmo para entradas de tamaño n en función del tiempo de ejecución que se tiene para el mismo algoritmo para entradas de tamaño menor.

  Por ejemplo, $T(n) = T(n-1) + f(n)$.

Función factorial

```
1: int fact(int n) {
2:
3:   if (n <= 1)
4:      return 1;
5:   else
6:      return (n + fact(n – 1));
7:  }
```

Llamamos T(n) al tiempo de ejecución de fact(n).

Las líneas 3 y 4 son op. elementales

Ejemplo 2: Ordenar Vector

```
Void ordenaVector( vector<int> & V, int n)
// n es una posicion valida en el vector
{ if (n==0) return;
  else  {
    pos= encuentraMaximo(V,n)
    intercambia(V,pos,n);
    ordenaVector(V,n-1);
   }
}
```

Determ. Ec. Recurrencia

Solución Ecuaciones Recursivas

- Métodos:
  - Sustitución
  - Inducción
  - Árbol de Recursión
  - Fórmula Maestra
  - Ecuación Característica

Función factorial (II)

$$
\begin{aligned}
T(n) &= c + T(n - 1) \\
&= c + (c + T(n-2)) = 2c + T(n-2) \\
&= 2c + (c + T(n-3)) = 3c + T(n-3) \\
&\ldots \\
&= ic + T(n-i) \\
&\ldots \\
&= (n-1)c + T(n - (n-1)) = (n-1)c + d
\end{aligned}
$$

De donde T(n) es O(n)

Ejemplo

```
1:      int E(int n) {
2:      if (n == 1)
3:              return 0;
4:      else
5:              return E(n/2) + 1;
6:      }
```

$$T(n) = \begin{cases} 1, & n = 1 \\ 1 + T\left(\frac{n}{2}\right) & n > 1 \end{cases}$$

T(n) es $O(\log_2(n))$

Ejemplo de recurrencia

$$T(n) = T\left(\frac{n}{2}\right) + n^2, \quad n \geq 2, \quad T(1) = 1$$

- Cambio de variable: $n=2^m$; $n^2 = (2^m)^2=(2^2)^m=4^m$

$T(2m) = T(2^{m-1}) + 4^m =$

$= T(2^{m-2}) + 4^{m-1} + 4^m$

...

$= T(2^{m-i}) + [4^{m-(i-1)} + \ldots + 4^{m-1} + 4^m]$

Ejemplo

$$T(2^m) = T(1) + [4^1 + \cdots + 4^{m-2} + 4^{m-1} + 4^m]$$

$$= \sum_{i=0}^{m} 4^i = \frac{4 \bullet 4^{m+1} - 1}{4 - 1} = \frac{4}{3} 4^m - \frac{1}{3}$$

$$= [4^m = n^2] = \frac{4}{3} n^2 - \frac{1}{3}$$

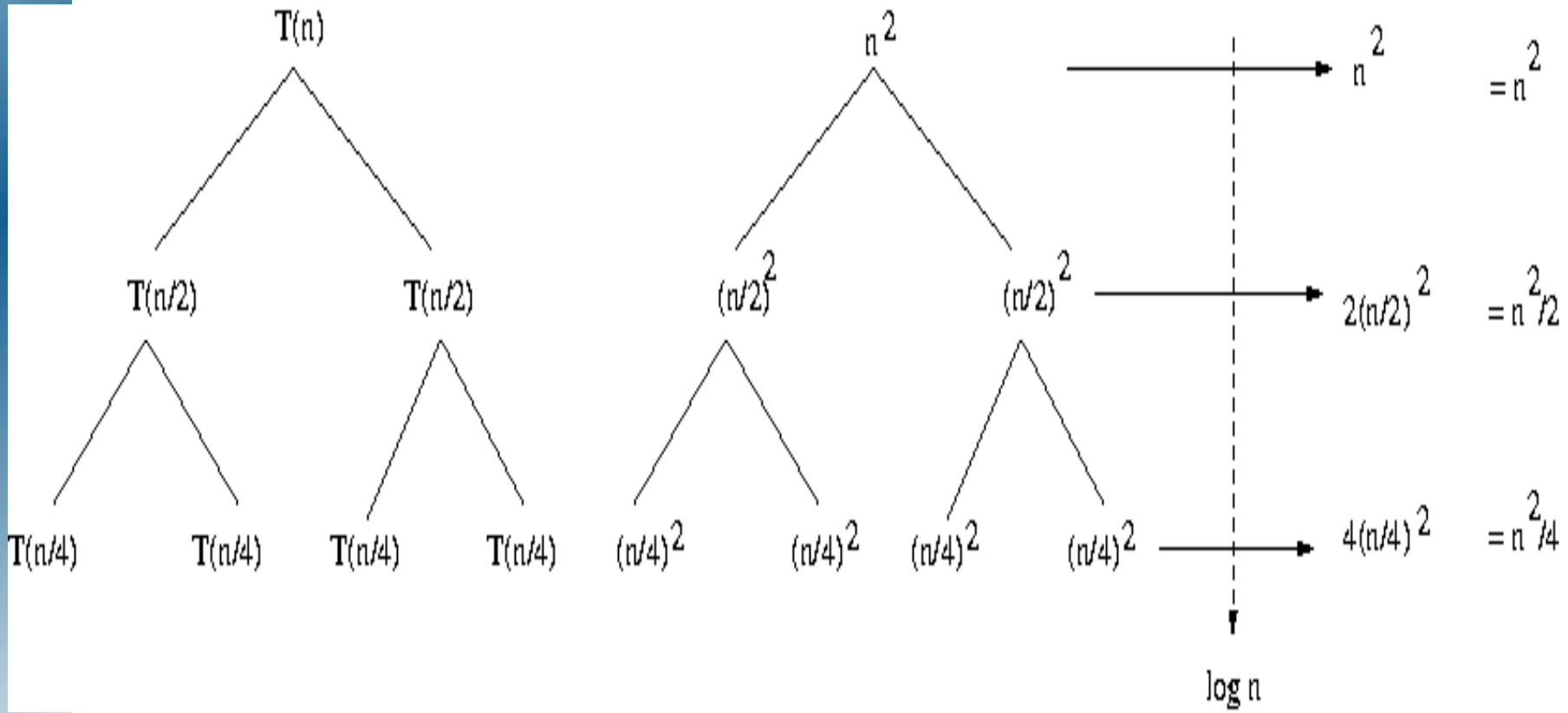
T(n) es $O(n^2)$

Arboles de recursión

Se representa gráficamente el proceso de recursión. Factores a tener en cuenta:

- El tamaño de las entradas usado como argumento en la siguiente relación de recurrencia.
- La suma del trabajo realizado en cada nivel de recursión.

Arboles de recursión



Fórmula maestra

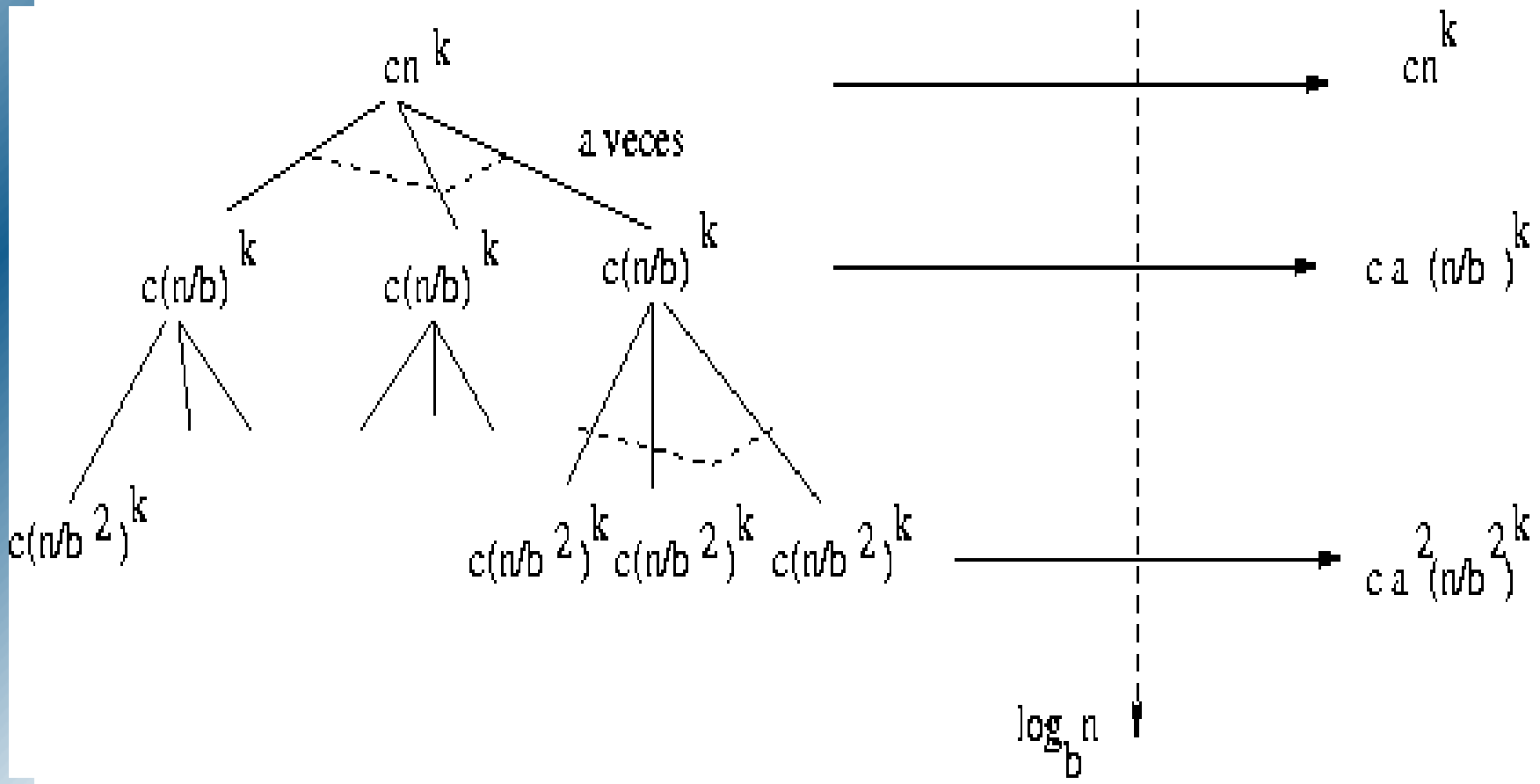
$$T(n) = aT(n/b) + cn^k, \quad \text{con } T(1) = c.$$

$$T(n) \in \Theta(n^k \qquad) \text{ si } a < b^k.$$

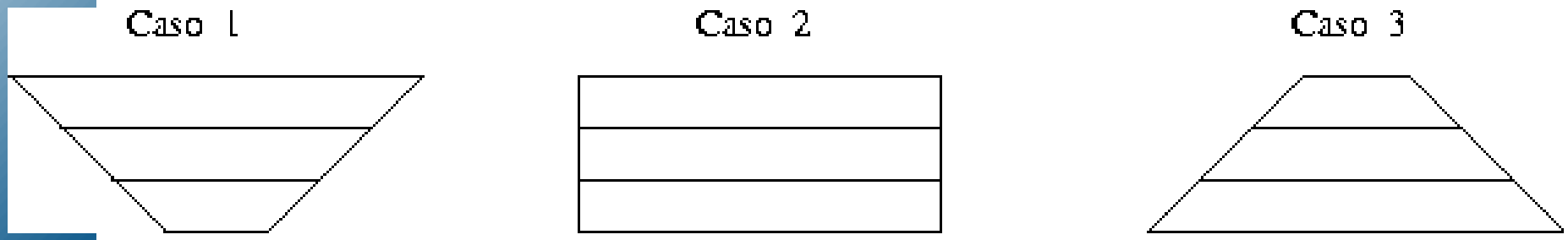
$$T(n) \in \Theta(n^k \log n) \text{ si } a = b^k.$$

$$T(n) \in \Theta(n^{\log_b a}) \text{ si } a > b^k.$$

Fórmula Maestra



Fórmula Maestra



- Intuitivamente, podemos considerar cada nivel de la recursión como un cajón de altura 1 y anchura igual al valor del costo computacional del nivel (excluyendo las llamadas recursivas). El objetivo final se podría considerar como el cálculo del área de la estructura completa

Recurrencia General II

$$T(n) = \begin{array}{ll} f(n) & si\ 0 \leq n < c \\ a\,T(n-c) + b\,n^k & si\ c \leq n \end{array}$$

Entonces

$$T(n) = O(n^k)\ si\ a < 1$$

$$T(n) = O(n^{k+1})\ si\ a = 1$$

$$T(n) = O(a^{n/c})\ si\ a > 1$$

Ecuación característica

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

1.- Construcción de la ecuación característica.
2.- Construcción del polinomio característico asociado a la recurrencia inicial.
3.- Factorización del polinomio mediante el cálculo de las raíces del mismo
4.- Representar la ecuación de recurrencia mediante la siguiente combinación líneal

$$p(x) = \prod_{i=1}^{l} (x - r_i)^{m_i}$$

5.- **Calcular las constantes**

Resolución de recurrencias

- Desarrollar; intentar encontrar la expresión general; resolver.
- Método de la ecuación característica

Recurrencias homogéneas

- Consideramos recurrencias *homogéneas lineales con coeficientes constantes*:

$$a_0 t_n + a_1 t_{n-1} + \cdots + a_k t_{n-k} = 0$$

- Lineales: no hay términos $t_{n-1}t_{n-j}, t^2_{n-i}$
- Homogénea: igualada a 0
- Con coeficientes constantes: a_i son constantes
- Ejemplo (sucesión de Fibonacci)

$$f_n = f_{n-1} + f_{n-2} \quad \Rightarrow \quad f_n - f_{n-1} - f_{n-2} = 0$$

- *Observación*: las combinaciones lineales de las soluciones de la recurrencia también son soluciones

Recurrencias homogéneas (II)

Suposición $t_n = x^n$

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

satisfecha si x=0 o (ecuación característica)

$$a_0 x^k + a_1 x^{k-1} + \cdots + a_k = 0$$

Polinomio característico

$$p(x) = a_0 x^k + a_1 x^{k-1} + \cdots + a_k$$

Recurrencias homogéneas (III)

(Teorema fundamental del Álgebra)

$$p(x) = \prod_{i=1}^{k} (x - r_i)$$

Consideremos una raíz del polinomio característico, r_i, $p(r_i) = 0$, r_i^n es solución de la recurrencia.

Además,

$$t_n = \sum_{i=1}^{k} c_i r_i^n$$

Cuando todos los r_i son distintos, éstas son las únicas soluciones

Ejemplo Fibonacci

$$f_n = \begin{cases} n, & \text{si } n = 0 \text{ ó } n = 1 \\ f_{n-1} + f_{n-2} & \text{en otro caso} \end{cases}$$

$$f_n - f_{n-1} - f_{n-2} = 0$$

$$p(x) = x^2 - x - 1$$

$$r_1 = \frac{1+\sqrt{5}}{2} \quad y \quad r_2 = \frac{1-\sqrt{5}}{2}$$

Solución Fibonacci

Solución general:

$$f_n = c_1 r_1^n + c_2 r_2^n$$

$$\begin{aligned} c_1 &+ c_2 &= 0 \\ r_1 c_1 &+ r_2 c_2 &= 1 \end{aligned}$$

$$c_1 = \frac{1}{\sqrt{5}} \quad y \quad c_2 = -\frac{1}{\sqrt{5}}$$

$$f_n = \frac{1}{\sqrt{5}} \left[\left(\frac{1+\sqrt{5}}{2} \right)^n - \left(\frac{1-\sqrt{5}}{2} \right)^n \right]$$

Ejemplo

$$t_n = \begin{cases} 0, & n = 0 \\ 5, & n = 1 \\ 3t_{n-1} + 4t_{n-2}, & \text{en otro caso} \end{cases}$$

$$t_n - 3t_{n-1} - 4t_{n-2} = 0$$

$$x^2 - 3x - 4 = (x+1)(x-4)$$

A partir de las condiciones iniciales: $c_1 = -1$ y $c_2 = 1$

$$t_n = c_1(-1)^n + c_2 4^n$$

Raíces múltiples

Supongamos que las raíces del polinomio característico NO son todas distintas. Sean r_i con multiplicidad m_i, i=1,…,l, las soluciones de p(x).

Entonces

$$t_n = \sum_{i=1}^{l} \sum_{j=0}^{m_i - 1} c_{ij} n^j r_i^n$$

Ejemplo

$$t_n = \begin{cases} n, & \text{si } n = 0, 1 \text{ ó } 2 \\ 5t_{n-1} + 8t_{n-2} + 4t_{n-3} & \text{en otro caso} \end{cases}$$

$$t_n - 5t_{n-1} - 8t_{n-2} - 4t_{n-3} = 0$$

$$x^3 - 5x^2 - 8x - 4 = (x-1)(x-2)^2$$

$$t_n = c_1(1)^n + c_2 2^n + c_3 n 2^n$$

A partir de las condiciones iniciales:
c1 = -2, c2 = 2, c3 = 1/2

$$t_n = 2^{n+1} + n2^{n-1} - 2$$

Recurrencias no homogéneas

$$a_0 t_n + a_1 t_{n-1} + \ldots + a_k t_{n-k} = b^n p(n)$$

- b es una constante
- p(n) es un polinomio de n de grado d
- Ejemplo: $t_n - 2t_{n-1} = 3^n$
- Ecuación característica:

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

- El polinomio característico es:

$$a_0 x^n + a_1 x^{n-1} + \cdots + a_k x^{n-k} = b^n p(n)$$

- Se procede igual que en el caso homogéneo, salvo que las condiciones iniciales se obtienen de la propia recurrencia

Ejemplo

$$t_n = \begin{cases} 0, & \text{si } n = 0 \\ 2t_{n-1} + 1 & \text{en otro caso} \end{cases}$$

$$t_n - 2t_{n-1} = 1 \qquad p(x) = (x-2)(x-1)$$

Solución general : $t_n = c_1 2^n + c_2$

$$\begin{aligned} c_1 + c_2 &= 0 \\ 2c_1 + c_2 &= 1 \end{aligned} \Rightarrow \begin{aligned} c_1 &= 1 \\ c_2 &= -1 \end{aligned}$$

$$t_n = 2^n - 1$$

Ejemplo

$$t_n = 2t_{n-1} + n$$

$$p(x) = (x-2)(x-1)^2$$

Solución general : $t_n = c_1 2^n + c_2 1^n + c_3 n 1^n$

$$\begin{array}{rcrcrcl} c_1 & + & c_2 & & & = & 0 \\ 2c_1 & + & c_2 & + & c_3 & = & 1 \\ 4c_1 & + & c_2 & + & 2c_3 & = & 4 \end{array} \quad \Rightarrow \quad \begin{array}{l} c_1 = 2 \\ c_2 = -2 \\ c_3 = -1 \end{array}$$

$$t_n = 2^{n+1} - n - 2$$

$$\mathbf{O(2^n)}$$

Generalización

$a_0\, t_n + a_1\, t_{n-1} + \ldots + a_k\, t_{n-k} = b_1^n p_1(n) + b_2^n p_2(n) + \ldots$

donde las b_i son constantes distintas y los $p_i(n)$ son polinomios en n de grado d_i respectivamente.

Ecuación característica

$$(a_0 x^k + a_1 x^{k-1} + \ldots + a_k)(x-b_1)^{d1+1}\,(x-b_2)^{d2+1} \ldots = 0$$

Generalización

Ejemplo

$$t_n = 2t_{n-1} + n + 2^n,\ n \geq 1,\ \text{con}\ t_0 = 0.$$

$b_1 = 1,\ p_1(n) = n,\ b_2 = 2,\ p_2(n) = 1.$

- Ecuacion característica: $(x-2)(x-1)^2\ (x-2) = 0$,
- Solucion general:

$$t_n = c_1\ 1^n + c_2\ n1^n + c_3\ 2^n + c_4\ n2^n$$

Cambio de variable

- **Calcular el orden de T(n) si n es potencia de 2, y**

$$T(n) = 4T(n/2) + n,\ n > 1$$

- Reemplazamos n por 2^k (de modo que k = lg n) para obtener $T(2^k) = 4T(2^{k-1}) + 2^k$. Esto puede escribirse,

$$t_k = 4t_{k-1} + 2^k$$

- si $t_k = T(2^k) = T(n)$. (n es una potencia de 2)

Cambio de variable

$$t_k = 4t_{k-1} + 2^k$$

- Ecuación característica es $(x-4)(x-2) = 0$

  y entonces $t_k = c_1\, 4^k + c_2\, 2^k$.
- Poniendo n en lugar de k, tenemos

$$T(n) = c_1\, n^2 + c_2\, n$$

T(n) esta por tanto es $O(n^2$ /n es una potencia de 2)

Cambio de variable

- Encontrar el orden de T(n) si n es una potencia de 2 y si

  $T(n) = 4T(n/2) + n^2$, $n > 1$

- Obtenemos sucesivamente

  $T(2^k) = 4T(2^{k-1}) + 4^k$, y $t_k = 4t_{k-1} + 4^k$

- Ecuacion característica: $(x-4)^2 = 0$,

  $t_k = c_1 4^k + c_2 k4^k$, **$T(n) = c_1 n^2 + c_2 n^2 \lg n$**

- $O(n^2 \log n$ / n es potencia de 2).

Cambio de variable

- Calcular el orden de T(n) si n es una potencia de 2

$$T(n) = 2T(n/2) + n\lg n,\ n > 1$$

- Obtenemos

$$T(2^k) = 2T(2^{k-1}) + k2^k$$

$$t_k = 2t_{k-1} + k2^k$$

- Ecuación característica es $(x-2)^3 = 0$, y asi,

$$t_k = c_1 2^k + c_2 k2^k + c_3 k^2 2^k$$

$$T(n) = c_1 n + c_2 n\lg n + c_3 n\lg^2 n$$

- T(n) es $O(n\log^2 n$ / n es potencia de 2).

Cambio de variable

- Calcular el orden de T(n) si n es potencia de 2 y $T(n) = 3T(n/2) + cn$ (c es constante, $n \geq 1$).
- Obtenemos sucesivamente,

$$T(2^k) = 3T(2^{k-1}) + c2^k$$
$$t_k = 3t_{k-1} + c2^k$$

- Ecuación característica: $(x-3)(x-2) = 0$, y asi,

$$t_k = c_1 3^k + c_2 2^k$$
$$T(n) = c_1 3^{\lg n} + c_2 n$$

- y como $a^{\lg b} = b^{\lg a}$, $T(n) = c_1 n^{\lg 3} + c_2 n$

y finalmente T(n) es $O(n^{\lg 3}$ / n es potencia de

Transformaciones del rango

- $T(n) = nT^2(n/2)$, $n > 1$, $T(1) = 6$ y n potencia de 2.
- Cambiamos la variable: $t_k = T(2^k)$, y asi

$$t_k = 2^k t^2_{k-1} , k > 0; t_0 = 6.$$

- Esta recurrencia no es lineal, y uno de los coeficientes no es constante.
- Para transformar el rango, creamos una nueva recurrencia tomando $V_k = \lg t_k$, lo que da,

$$V_k = k + 2 V_{k-1} , k > 0; V_0 = \lg 6.$$

- Ecuación característica: $(x-2)(x-1)^2 = 0$ y asi,

$$V_k = c_1 2^k + c_2 1^k + c_3 k1^k$$

Teoría de Algoritmos

Tema 1. Planteamiento General

Tema 2. La Eficiencia de los Algoritmos

Tema 3. Algoritmos "Divide y vencerás"

Tema 4. Algoritmos Voraces ("Greedy")

Tema 5. Algoritmos para la Exploración de Grafos ("Backtraking", "Branch and Bound")

Tema 6. Algoritmos basados en Programación Dinámica

Tema 7. Otras Técnicas Algorítmicas de Resolución de Problemas