Sección TQ - 1er Semestre/2018

Análisis de Algoritmos (parte II)

Prof. Cristian Cappo

1/Marzo/2018

En esta clase

- Ejercicio de análisis de algoritmo
- Proceso del diseño y análisis de algoritmos
- Uso de cálculo para calcular relaciones asintóticas
- Definición alternativas de cotas
- Otras notaciones asintóticas
- Propiedades de funciones asintóticas
- Algoritmos con estructura recurrente
- Análisis de un algoritmo Divide y Vencerás
- Ecuación de recurrencia
- Resolución de ecuaciones de recurrencia

Ejemplo de análisis de algoritmo

Considere el siguiente código: dada una secuencia A= $\langle a_1, a_2,\ldots,a_n \rangle$, retornar verdadero si A contiene dos valores iguales a_i=a_j (con $i \neq j$)

```
boolean tieneIgual( int [] A ) {
    for ( int i = 0; i < A.length; i++) {
        int j = i+1;
        while ( j < A.length) {
            if ( A[i] == A[j] )
                return true;
            j++;
        }
    return false;
}
```

- ¿Cuál es la T(n)? Considerar que *n = A.length* y el análisis de **peor caso**
- ¿Cuáles son las cotas asintóticas: O, Ω y θ?
- ¿Puede mejorar la cota? ¿Cómo?

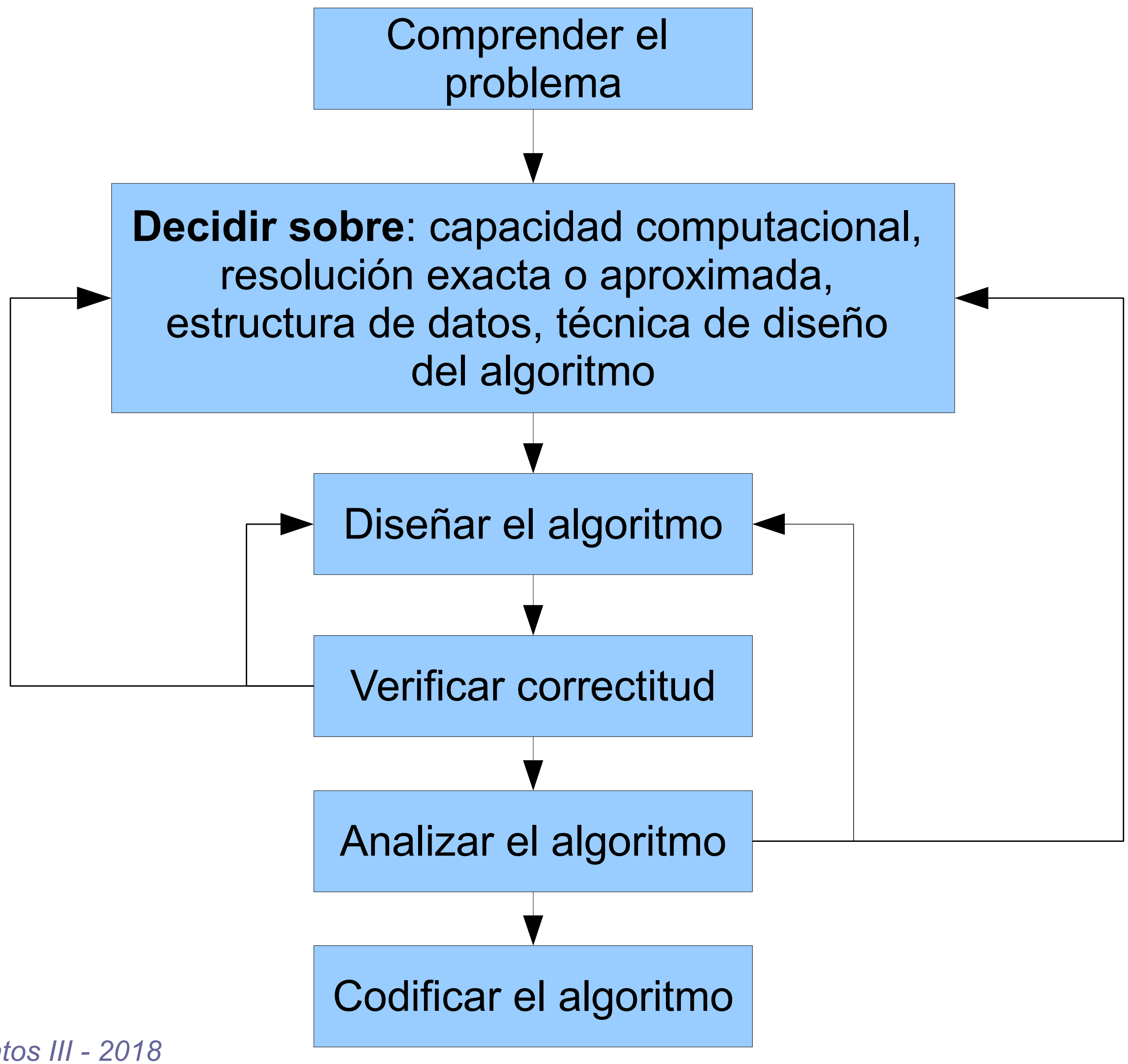
Solución

- $T(n) = 3.5\ n^2 + 2.5\ n + 2$ (ver ejemplo corriendo)
- $T(n) = O(n^2)$
- $T(n) = \Omega(n^2)$
- $T(n) = \theta\ (n^2)$
- Puede mejorarse la cota, claro. Por ejemplo: puede ordenar antes con un algoritmo de cota *n log n* (*mergeSort o quickSort u otro)*
- Así la cota será dominada por la ordenación ya que determinar si hay iguales simplemente será un solo recorrido por el arreglo ya ordenado.

```
sort(A);
for(int i = 0;i < A.length-1; i++) /* Cota θ(n) en el peor caso */
   if ( A[i]==A[i+1] )
      return true;
return false;
```

```java
public static boolean tieneIgual( int [] A ) {
     int i = 0 ;
     boolean res = false;
     int count = 1; /* 1 asig */

     int n = A.length;     //por claridad hacemos esto

     while ( i < n ) {     //for ( int i = 0; i < A.length;i++ ) {

        int j = i+1;
        count+=3; /* 1 inc, 1 suma y 1 cmp(del for) */

        while ( j < n ) {
           count += 7; /* 2 cmp (1 del while), 2 acc, 1 ret, 1 sum, 1 asig */
           if ( A[i] == A[j] )
               res = true;
           j++;
        }
        count+=1; /* 1 cmp por salida de while */
        i++;
        count+=2; /* 1 asig, 1 suma ) */
     }
     count+=1; /* 1 cmp por salida de while */

     /* usamos la formula que nos dio el analisis utilizando el peor caso*/
     System.out.printf("%5d\t%10d\t%10.0f\n", n, count, 3.5*n*n + 2.5*n + 2);

     return res; /* No contamos esta instruccion pq imprimimos antes */
  }
```

Diseño de un algoritmo y el proceso de análisis



Otras notaciones asintóticas

Notación o (o chica)

Dada las funciones *f(n)* y *g(n)*, se dice que f(n) "está en $o(g(n))$" si existen las constantes positivas $c > 0$ y n_0 tal que

$$f(n) < cg(n) \; para \; n \geq n_0$$

Por ejemplo $2n = o(n^2)$ pero $2n^2 \neq o(n^2)$

Otros ejemplos:

$$n^{1.9999} = o(n2)$$
$$n^2/ \lg n = o(n2)$$
$$n^2/ 1000 \neq o(n2)$$

Otras notaciones asintóticas

Notación ω (omega chica)

Dada las funciones *f(n) y g(n),* se dice que *f(n)* "está en $\omega(g(n))$" si existen las constantes positivas $c > 0$ y n_0 tal que

$$f(n) > cg(n) \; para \; n \geq n_0$$

Por ejemplo $n^2/2 = \omega(n)$ pero $n^2/2 \neq \omega(n^2)$

Otros ejemplos: $n^{2.0001} = \omega(n^2)$

$n^2/\lg n = \omega(n^2)$

$n^2 \neq \omega(n^2)$

Usando límites para comparar tasas de crecimiento

Aunque las definiciones de O, Θ, Ω, o, ω son indispensables, un método simple para comparar las tasas de crecimiento de dos funciones es el uso del cálculo del límite del ratio de estas funciones. Los tres casos son:

$$\lim_{n\to\infty} \frac{t(n)}{g(n)} = \begin{cases} 0 & t(n) < g(n) \\ c > 0 & t(n) = g(n) \\ \infty & t(n) > g(n) \end{cases}$$

Note que los dos primeros casos significan que $t(n) \in O(g(n))$

los dos últimos casos significan que $t(n) \in \Omega(g(n))$

y el segundo caso significa que $t(n) \in \Theta(g(n))$

Aplicar límites

- Puede utilizar la regla de L'Hôpital-Bernoulli

$$\lim_{n\to\infty} \frac{t(n)}{g(n)} = \lim_{n\to\infty} \frac{t'(n)}{g'(n)}$$

- Y la fórmula de Stirling para n!

$$n! \approx \sqrt{2\pi n} \left(\frac{n}{e}\right)^n$$

Ejemplos, comparar el orden de crecimiento de las siguientes funciones

$$a)\quad \frac{1}{2}n(n-1) \quad vs \quad n^2$$

$$b)\quad \log_2 n \quad vs \quad \sqrt{n}$$

$$c)\quad n! \quad vs \quad 2^n$$

$$a)\quad \lim_{n\to\infty}\frac{\frac{1}{2}n(n-1)}{n^2}=\frac{1}{2}\lim_{n\to\infty}\frac{n^2-n}{n^2}=\frac{1}{2}\lim_{n\to\infty}(1-\frac{1}{n})=\frac{1}{2} \quad\Rightarrow\quad \frac{1}{2}n(n-1)\in\Theta(n^2)$$

$$b)\quad \lim_{n\to\infty}\frac{\log_2 n}{\sqrt{n}}=\lim_{n\to\infty}\frac{(\log_2 n)'}{(\sqrt{n})'}=\lim_{n\to\infty}\frac{(\log_2 e)\frac{1}{n}}{\frac{1}{2\sqrt{n}}}=2\log_2\lim_{n\to\infty}\frac{\sqrt{n}}{n}=2\log_2\lim_{n\to\infty}\frac{(\sqrt{n})'}{(n)'}$$

$$=2\log_2 e\lim_{n\to\infty}\frac{1}{2\sqrt{n}}=0 \quad\Rightarrow\quad \log_2 n\in O(\sqrt{n}) \quad\Rightarrow\quad \log_2 n\in o(\sqrt{n})$$

$$c)\quad \lim_{n\to\infty}\frac{n!}{2^n}=\lim_{n\to\infty}\frac{\sqrt{2\pi n}(\frac{n}{e})^n}{2^n}=\lim_{n\to\infty}\sqrt{2\pi n}\frac{n^n}{2^n e^n}=\lim_{n\to\infty}\sqrt{2\pi n}\left(\frac{n}{2e}\right)^n=\infty \quad\Rightarrow\quad n!\in\Omega(2^n) \quad\Rightarrow\quad n!\in\omega(2^n)$$

Definiciones alternativas de cotas

Un algoritmo de tiempo *t(n)* esta en:

O(f(n)) sii

$$\lim_{n \to \infty} \frac{t(n)}{f(n)} = C \quad , \quad 0 \leq C < \infty$$

o(f(n)) sii

$$\lim_{n \to \infty} \frac{t(n)}{f(n)} = 0$$

Θ(f(n)) sii

$$\lim_{n \to \infty} \frac{t(n)}{f(n)} = C \quad , \quad 0 < C < \infty$$

Ω(f(n)) sii

$$\lim_{n \to \infty} \frac{t(n)}{f(n)} = C \quad , \quad 0 < C \leq \infty$$

ω(f(n)) sii

$$\lim_{n \to \infty} \frac{t(n)}{f(n)} = \infty$$

Algunas propiedades de las cotas

- $f(n) \in O(f(n))$ *(reflexividad – aplicable a las otras cotas)*
- $f(n) \in \Theta(g(n))$ **sii** $g(n) \in \Theta(f(n))$ *(simetría)*
- $f(n) \in O(g(n))$ **sii** $g(n) \in \Omega(f(n))$ *(simetría traspuesta)*
- **Si** $f(n) \in O(g(n))$ y $g(n) \in O(h(n))$, **entonces** $f(n) \in O(h(n))$ *(transitividad – aplicable a las otras cotas)*
- **Si** $f_1(n) \in O(g_1(n))$ y $f_2(n) \in O(g_2(n))$, **entonces** $f_1(n) + f_2(n) \in O(\mathbf{max}\{g_1(n), g_2(n)\})$
- **Si** $f(n) \in O(g(n))$ y $f(n) \in \Omega(g(n))$, **entonces** $f(n) \in \Theta(g(n))$

Ejemplo de notaciones asintóticas

Ejemplos de funciones en $O(n^2)$	**Ejemplo de funciones en** $\Omega(n^2)$
n^2	n^2
$n^2 + n$	$n^2 + n$
$n^2 + 1000n$	$n^2 - n$
$1000\, n^2 + 1000\, n$	$1000\, n^2 + 1000\, n$
También	$1000\, n^2 - 1000\, n$
n	***También***
$n/1000$	n^3
$n^{1.9999}$	$n^{2.0001}$
$n^2 / \lg \lg \lg n$	$n^2 \lg \lg \lg n$
	2^{2n}

O, Θ, Ω como funciones anónimas

- Podemos usar la notaciones asintóticas para representar funciones desconocidas. Ejemplo:

  $$f(n) = 10n^2 + O(n)$$

  Significa que $f(n)$ es igual a $10n^2$ más una función que no conocemos o que no nos importa conocer y que es asintóticamente al menos lineal en n.

- Ejemplos:

  $n^2 + 4n - 1 = n^2 + \theta(n)$? ✓

  $n^2 + \Omega(n) - 1 = O(n^2)$? ✗

  $n^2 + O(n) - 1 = O(n^2)$? ✓

  $n \log n + \theta(\sqrt{n}) - 1 = O(n\sqrt{n})$? ✓

Recordar que

$$O \approx \geq$$

$$\Omega \approx \leq$$

$$\Theta \approx =$$

$$o \approx >$$

$$\omega \approx <$$

Algunas series útiles para el análisis

$$\sum_{1}^{n} 1 = n \quad \in \quad \Theta(n)$$

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} \quad \in \quad \Theta(n^2)$$

$$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6} \quad \in \quad \Theta(n^3)$$

$$\sum_{i=1}^{n} a^i = \frac{a^{n+1}-1}{a} - 1 \quad para\, a > 1$$

$$En\ particular \sum_{i=1}^{n} 2^i = 2^{n+1} - 1 \quad \in \quad \Theta(2^n)$$

Serie Armónica

$$\sum_{i=1}^{n} \frac{1}{i} = \ln n + O(1)$$

Algunas propiedades básicas de las sumatorias

$$\sum (a_i \pm b_i) = \sum a_i \pm \sum b_i$$

$$\sum c\, a_i = c \sum a_i$$

$$\sum_{l \le i \le u} a_i = \sum_{l \le i \le m} a_i + \sum_{m+1 \le i \le u} a_i$$

Como estimar una suma discreta

- Reemplazar la suma con una integral

$1 + 2 + \ldots + N.$

$$\sum_{i=1}^{N} i \sim \int_{x=1}^{N} x\,dx \sim \frac{1}{2} N^2$$

$1 + 1/2 + 1/3 + \ldots + 1/N.$

$$\sum_{i=1}^{N} \frac{1}{i} \sim \int_{x=1}^{N} \frac{1}{x} dx = \ln N$$

$$\sum_{i=1}^{N} \sum_{j=i}^{N} \sum_{k=j}^{N} 1 \sim \int_{x=1}^{N} \int_{y=x}^{N} \int_{z=y}^{N} dz\,dy\,dx \sim \frac{1}{6} N^3$$

Algoritmos con estructura recurrente

Condiciones a tener en cuenta para una solución recurrente:

- Definir el caso base (obligatorio)
- Avanzar en cada llamada recursiva hacia el caso base (obligatorio)
- Asumir que la recursión funciona
- No realizar cálculos repetidos (*no obligatorio pero recomendable*)
  ¿Cuál es la solución recursiva de la serie fibonacci?

Serie Fibonacci

- Una secuencia de números: $0,1,1,2,3,5,8,13,21,34,\ldots$
- Leornardo da Pisa (a. 1170 – a. 1250)
  hijo de Guglielmo “Bonaccio”

  alias *Leornardo Fibonacci*

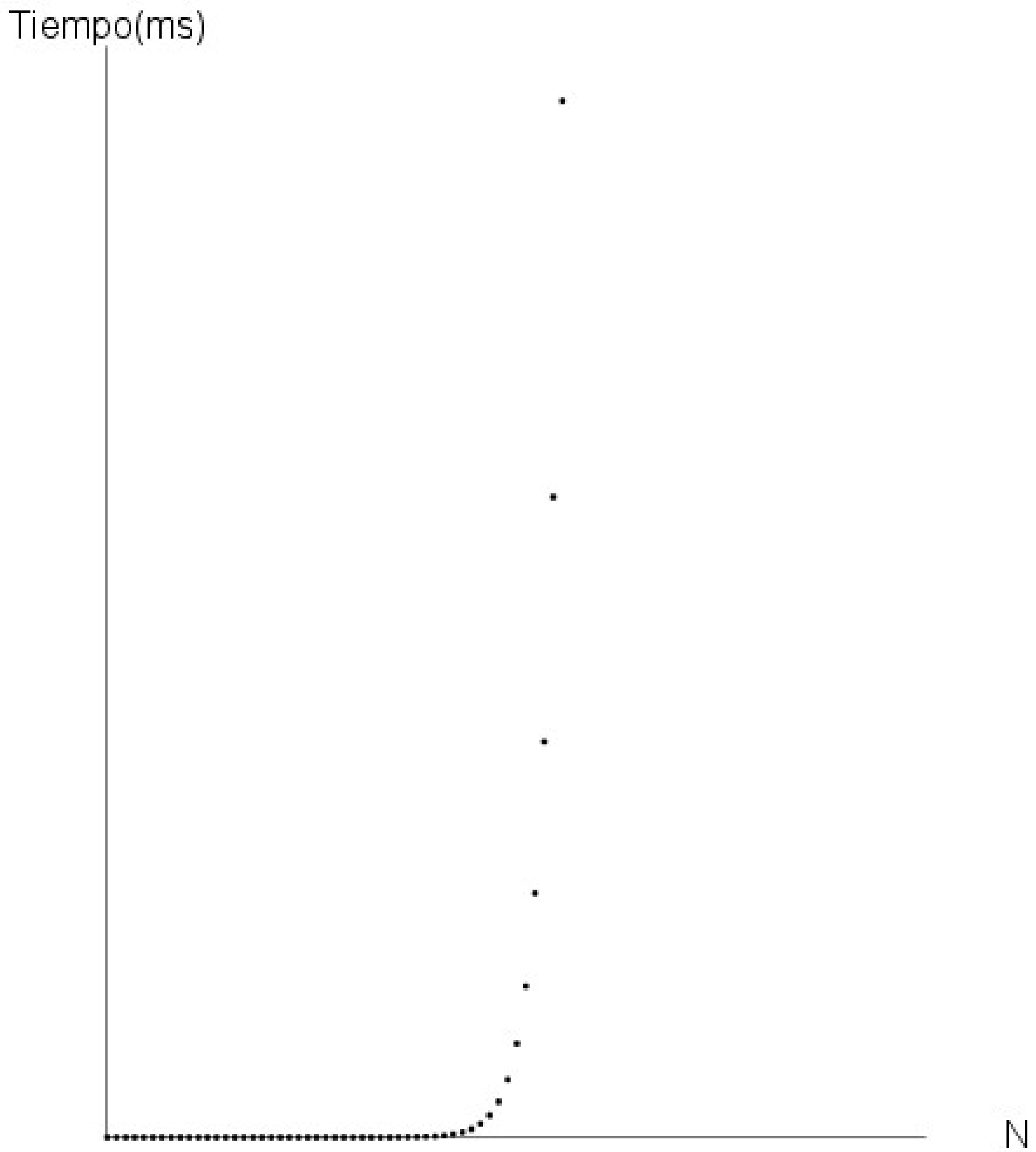
- Definición matemática:

$$F_n = \begin{cases} 0 & si \quad n=0 \\ 1 & si \quad n=1 \\ F_{n-1}+F_{n-2} & si \quad n>1 \end{cases}$$

Algoritmo para la Serie Fibonacci

En Java

```
public static int F ( int n ) {
    if ( n == 0 )
        return 0;
    else if ( n == 1 )
        return 1;
    else
        return F(n-1) + F(n-2);
}
```


Tiempo(ms)
N

Algoritmo para la Serie Fibonacci

```
public static long F (long n)
{
if ( n == 0 )
   return 0;
else if ( n == 1 )
   return 1;
else
   return F(n-1) + F(n-2);
}
```

¿Cuánto tiempo lleva?

$T(0) = 2 \; ; \; T(1) = 3$

$T(n) = T(n-1) + T(n-2) + 3 \Rightarrow T(n) >= F_n$

$T(n) >= F_n = F_{n-1} + F_{n-2}$

$F_n >= F_{n-1} >= F_{n-2} >= F_{n-3} >= \ldots$

$F_n >= 2F_{n-2} >=2(2F_{n-4}) >= 2(2(2F_{n-6}))>=\ldots >= 2^{n/2}$

Esto significa que $T(n) \geq (\sqrt{2})^n \approx (1.4)^n$

Técnica de diseño de algoritmos Divide y Vencerás

- InsertSort es un algoritmo incremental
- Un algoritmo que usa la estrategia DyV tiene las siguientes partes:
  - **Dividir**: el problema en subproblemas independientes.
  - **Conquistar**: los subproblemas recursivamente, el caso base se resuelve por fuerza bruta.
  - **Combinar**: las soluciones de los subproblemas para obtener la solución completa.

MergeSort como ejemplo de DyV

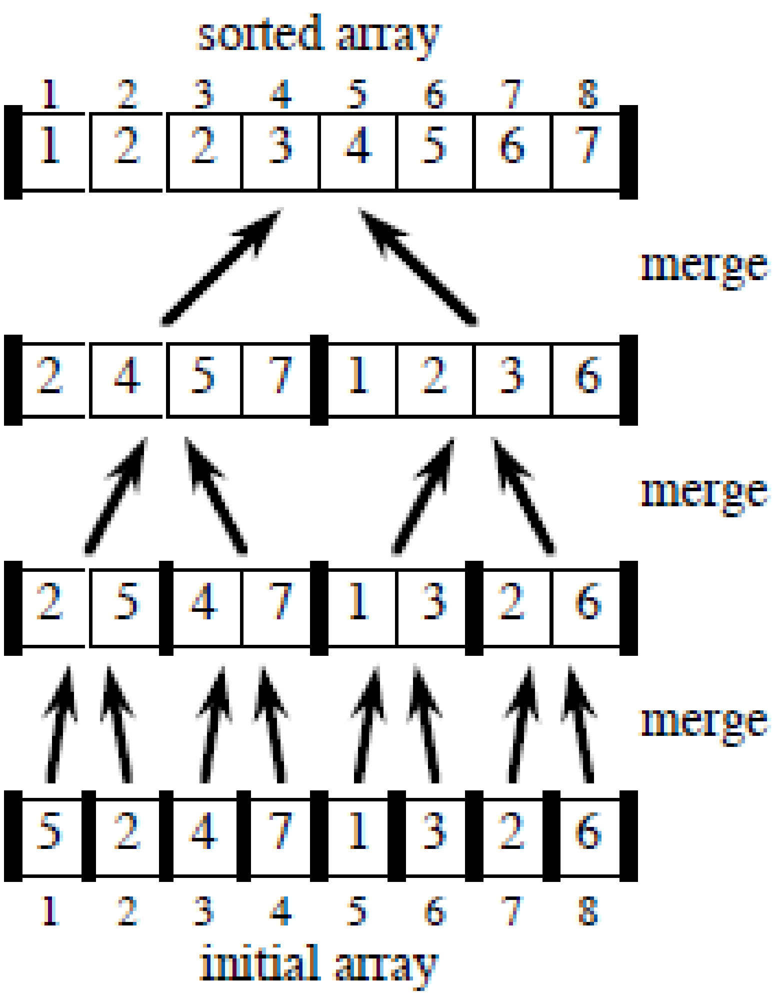
- Es un algoritmo de ordenación que usa la estrategia DyV. Su cota es menor que InsertSort.
- **Idea**: para ordenar el arreglo A[p..r] (inicialmente p=1 y r=n) hacemos lo siguiente:
  - **Dividir**: partir en 2 subarreglos A[p..q] y A[q+1..r] donde q es el punto medio de A[p..r]
  - **Conquistar**: ordenar recursivamente los subarreglos A[p..q] y A[q+1..r].
  - **Combinar**: mezclar los 2 subarreglos ordenados y producir un solo arreglo A[p..r].
  - La recursión continua hasta que el subarreglo tenga un solo elemento, el cual esta ordenado.

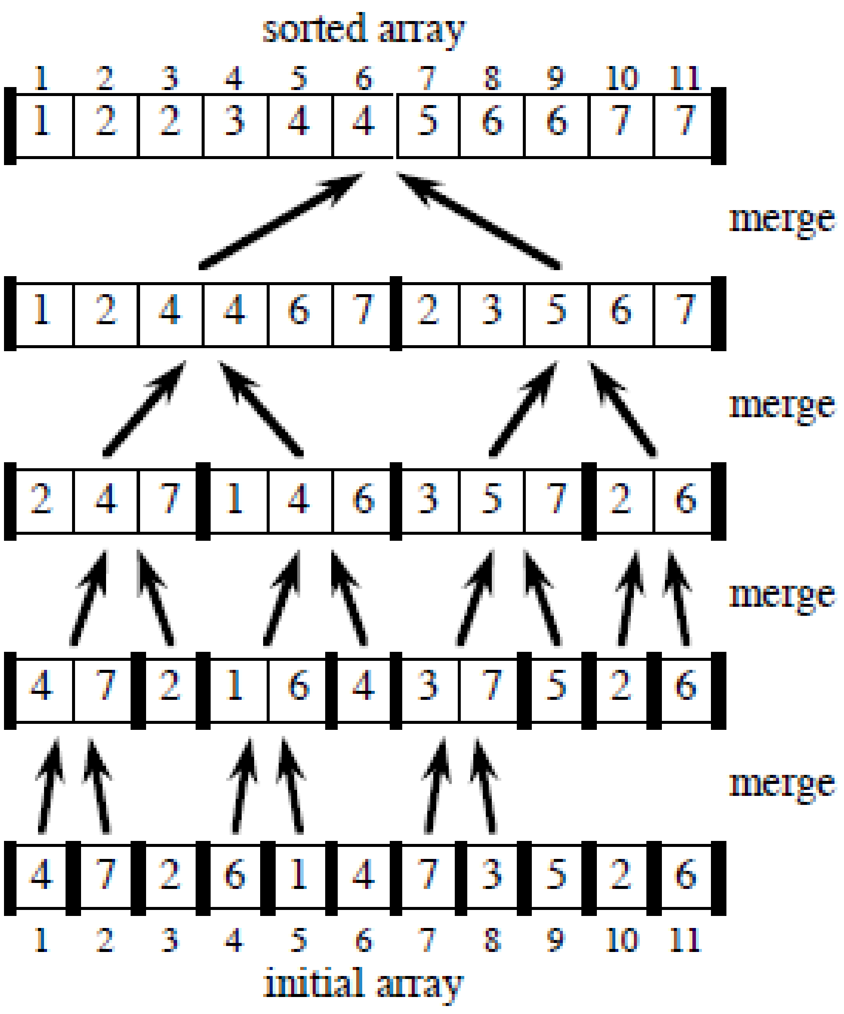
Algoritmo MergeSort

```
MERGE-SORT(A, p , r)

if ( p < r ) then
    q = ⌊(p + r)/2⌋          → Dividir
    MERGE-SORT(A,p,q)        → Conquistar
    MERGE-SORT(A,q+1,r)      → Conquistar
    MERGE(A,p,q,r)           → Combinar
```

Llamada inicial : `MERGE-SORT(A,1,n)`

Ejemplos


sorted array
1 2 3 4 5 6 7 8
1 2 2 3 4 5 6 7
merge
2 4 5 7 1 2 3 6
merge
2 5 4 7 1 3 2 6
merge
5 2 4 7 1 3 2 6
1 2 3 4 5 6 7 8
initial array


sorted array
1 2 3 4 5 6 7 8 9 10 11
1 2 2 3 4 4 5 6 6 7 7
merge
1 2 4 4 6 7 2 3 5 6 7
merge
2 4 7 1 4 6 3 5 7 2 6
merge
4 7 2 1 6 4 3 7 5 2 6
merge
4 7 2 6 1 4 7 3 5 2 6
1 2 3 4 5 6 7 8 9 10 11
initial array

Operación de Merge (combinar)

```
MERGE(A, p , q, r)

   n1 = q - p + 1
   n2 = r - q
   crear arreglos L[1..n1+1] y R[1..n2+1]

   for i=1 to n1 { L[i] = A[p+i-1] }
   for j=1 to n2 { R[j] = A[q+j] }
   L[n1+1] = ∞ ; R[n2+1] = ∞

   i=1 ; j=1
   for k=p to r {
      if ( L[i] <= R[j] ) {
           A[k] = L[i]
           i=i+1
      } else {
           A[k] = R[j]
           j=j+1
      }
   }
```

Analizar el tiempo de este algoritmo...

Su tiempo esta en ***O(n)***

Análisis de un algoritmo DyV

- Usamos una ecuación de recurrencia para describir su tiempo de ejecución
- Siendo ***T(n)*** el tiempo para resolver un problema de tamaño ***n***:
  - Si el problema es pequeño (***n*** <= ***c***, ***c*** es constante), es el caso base y nos lleva un tiempo constante ***O(1)***
  - De otra forma, dividimos en ***a*** subproblemas, cada uno de tamaño 1/**b** del original.
  - El tiempo necesario para la división es ***D(n)***
  - Existen ***a*** subproblemas, cada uno de ***n/b***, entonces cada subproblema toma ***T(n/b)*** y el tiempo total requerido es ***aT(n/b)***.
  - El tiempo necesario para combinar las soluciones es ***C(n)***

Ecuación de recurrencia de un algoritmo DyV

Así, la recurrencia de un algoritmo DyV se define como:

$$\mathrm{T}(n) = \begin{cases} O(1) & n <= c\ , c \text{ es constante} \\ a\,\mathrm{T}\left(\dfrac{n}{b}\right) + D(n) + C(n) & \text{en otro caso} \end{cases}$$

Analizando MergeSort

- Por simplicidad asumimos que ***n*** es potencia de 2.
  - Caso base, *n = 1*
  - Si *n >= 2*
    - Dividir: calcular q lleva => D(n) = O(1)
    - Conquistar: resolver recursivamente 2 subproblemas de tamaño n/2 => 2 T(n/2)
    - Combinar: mezclar los dos subarreglos de n/2 toma O(n) así C(n) = O(n)
  - *D(n)=O(1)* y *C(n)=O(n)*, sumados es *O(n)* (el máximo)
  - La ecuación de recurrencia de MergeSort es:

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

¿Cómo resolver la recurrencia?

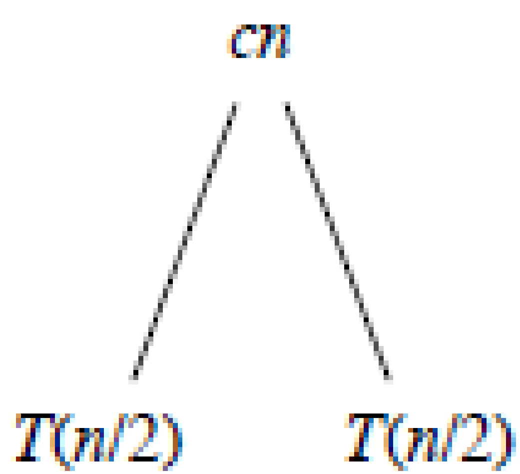
- Usando el Árbol de recurrencia
- Utilizando el Teorema Maestro
- Recurriendo al método de sustitución

Resolver la ecuación de recurrencia por el Árbol de recurrencia (MergeSort)

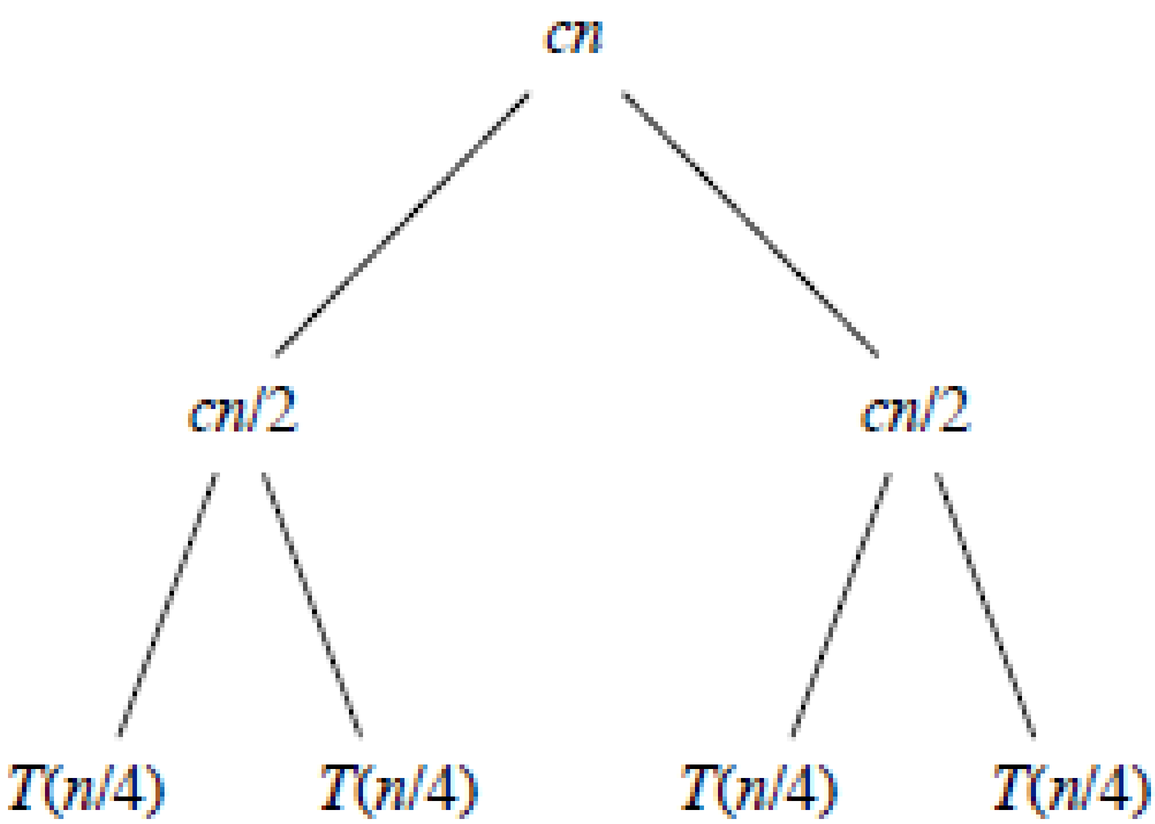
- Consideramos:

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

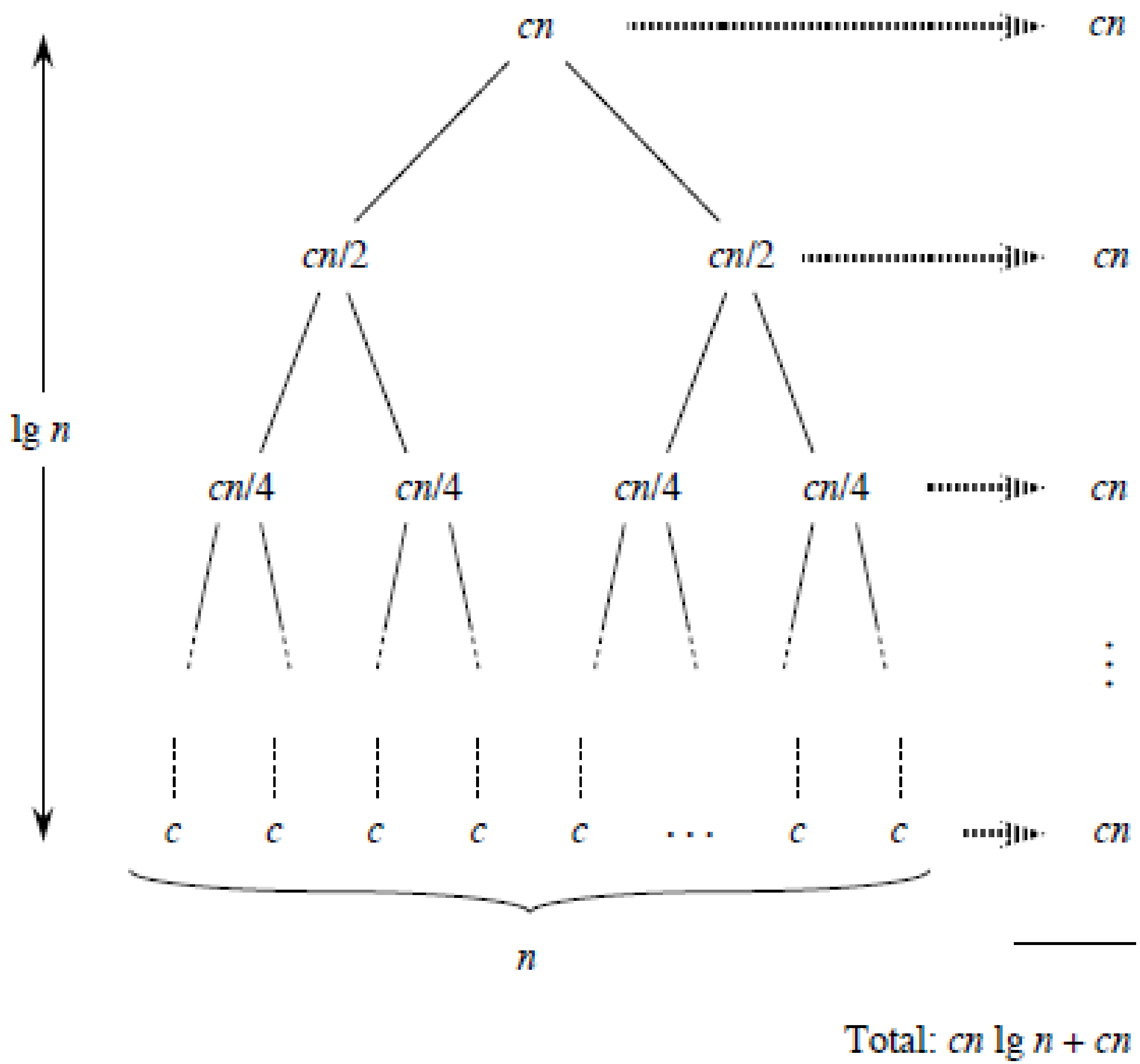
- Dibujamos el árbol de llamadas recursivas, abajo la primera llamada:



- Por cada subproblema de tamaño n/2 tenemos un costo de cn/2



- Expandimos hasta que n=1



Cada nivel tiene un costo de *cn*

- El nivel superior tiene un costo de *cn*
- El siguiente nivel 2 subproblemas, cada uno con costo *cn/2*
- El siguiente nivel 4 subproblemas, cada uno con costo *cn/4*
- Cada siguiente nivel tiene el doble de subproblemas pero mitad de tamaño
- Por tanto, el costo por cada nivel se mantiene a *cn*

- Existen *log n + 1* niveles (la altura es *log n*)
- Usando demostración por inducción
  - Caso base *n=1* => 1 nivel , *log 1 + 1 = 0+1 = 1*
  - Hipótesis inductiva, un árbol de tamaño 2^i tiene $\log 2^i + 1$ niveles
  - Ya que asumimos que tenemos problemas de tamaño potencia de 2, el siguiente tamaño a 2^i será 2^{i+1}
  - Un árbol de tamaño 2^{i+1} tiene un nivel más que el del tamaño 2^i , es decir tiene *i+2* niveles.
  - Debido a que $\log 2^{i+1} = (i+1)+1 = i+2$, llegamos a la hipótesis inductiva.

- El costo total es la suma de los costos de cada nivel. Tenemos *log n + 1* niveles cada uno con costo *cn*, así el costo total es *c n log n + cn*
- Ignorando el término de menor orden y las constantes, llegamos que MergeSort tiene un costo ***O(n log n)***

Teorema maestro para resolver recurrencias [CLRS – 4.3 pag 73 (2nd edition)]

- Usado para la forma de recurrencia siguiente:

  T(n) = a T(n/b) + f(n)

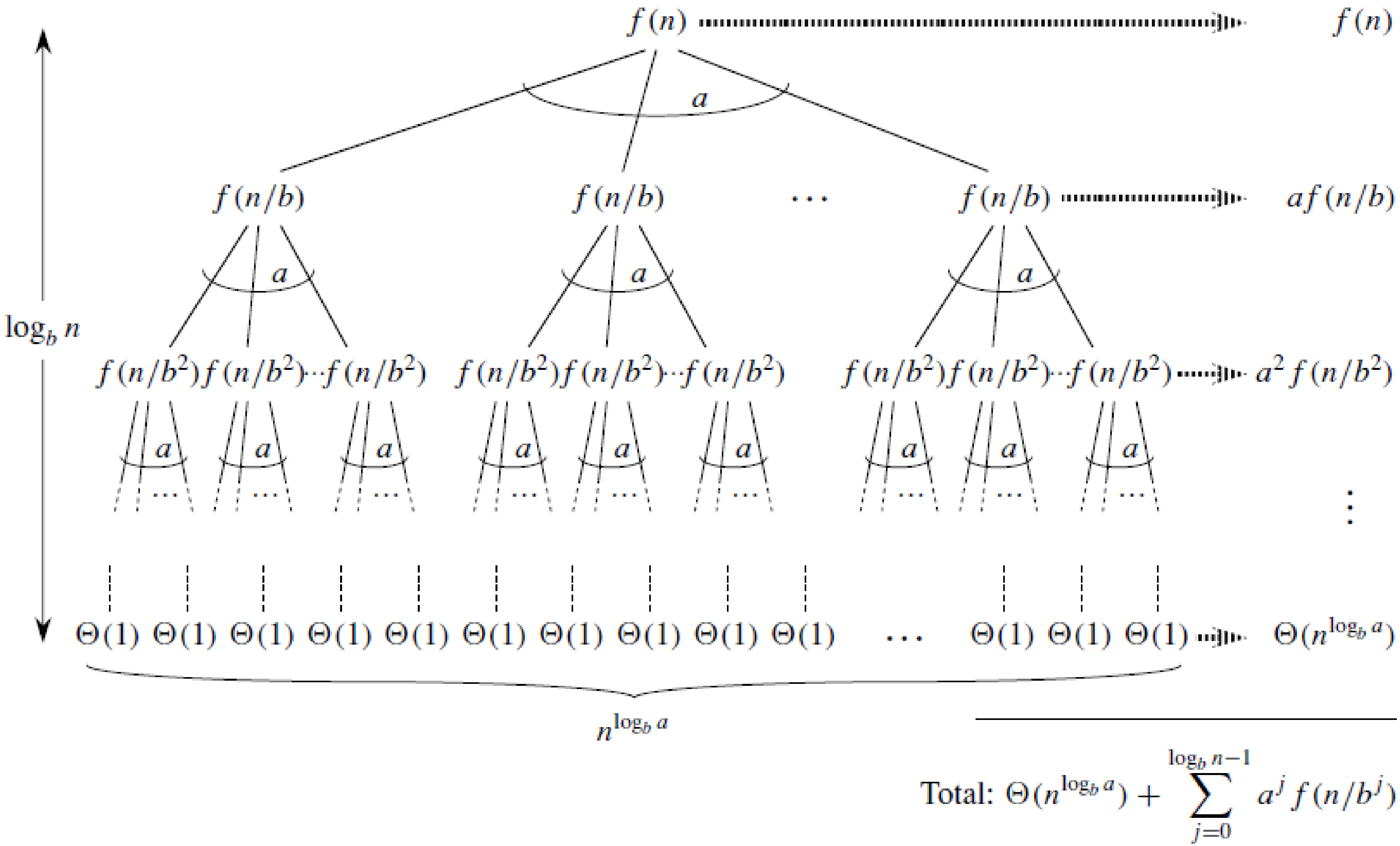
  donde a >= 1, b >=1 y f(n) > 0

- El mecanismo de aplicación es básicamente comparar

$$n^{\log_b a} \quad vs \quad f(n)$$

Teorema maestro

Árbol de recursión generado por la ecuación de recurrencia mostrada anteriormente



Teorema Maestro - casos

- **Caso 1**:

$$f(n) = O(n^{\log_b a - \epsilon}) \quad para \quad alguna \quad \epsilon > 0$$

$$(f(n)\, es\ polinomialmente\ menor\ que\ n^{\log_b a})$$

$$Solución: \quad T(n) = \Theta(n^{\log_b a})$$

- **Caso 2**:

$$f(n) = \Theta(n^{\log_b a} \log^k n) \quad donde\ k \geq 0$$

$$(f(n)\, esta\ dentro\ de\ un\ factor\ polilogaritmico\ de\ n^{\log_b a}, no\ menos)$$

$$Solución: \quad T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$$

$$Intuitivamente: el\ costo\ es\ \Theta(n^{\log_b a} \log^k n)\, en\ cada\ nivel,$$

$$y\ hay\ \Theta(\log n)\, niveles.$$

$$El\ caso\ simple: k = 0 \Rightarrow f(n) = \Theta(n^{\log_b a}) \Rightarrow T(n) = \Theta(n^{\log_b a} \log n)$$

Teorema Maestro – casos (2)

- **Caso 3**:

$$f(n) = \Omega(n^{\log_b a + \epsilon}) \quad para \quad alguna \quad \epsilon > 0$$
$$y\ f(n)\ satisface\ la\ condición\ de\ regularidad$$
$$a\ f\left(\frac{n}{b}\right) \leq c\ f(n),\ donde\ c < 1\ y\ n\ suficientemente\ grande$$

$$(f(n)\ es\ polinomialmente\ mayor\ que\ n^{\log_b a})$$
$$Solución: \quad T(n) = \Theta(f(n))$$

$$Intuitivamente: el\ costo\ es\ dominado\ por\ la\ raiz.$$

Ejemplos de aplicación del Teorema maestro

- **Ejemplo 1**

$$T(n) = 5T(\frac{n}{2}) + \Theta(n^2)$$

$$n^{\log_2 5} \quad vs \quad n^2$$

$$Ya\ que \log_2 5 - \epsilon = 2\ para\ alguna\ constante\ \epsilon > 0,$$
$$corresponde\ al\ caso\ 1 \Rightarrow T(n) = \Theta(n^{\log_2 5})$$

- **Ejemplo 2**

$$T(n) = 27\mathrm{T}\left(\frac{n}{3}\right) + \Theta(n^3 \log n)$$

$$n^{\log_3 27} \quad vs \quad n^3 \log n$$

$$corresponde\ al\ caso\ 2\ con\ k = 1 \Rightarrow T(n) = \Theta\left(n^3 \log^2 n\right)$$

- **Ejemplo 3**

$$T(n) = 5\mathrm{T}(\frac{n}{2}) + \Theta(n^3)$$

$$n^{\log_2 5} \quad vs \quad n^3$$

$$Ya\ que \log_2 5 + \epsilon = 3\ para\ alguna\ constante\ \epsilon > 0,$$

$$chequeamos\ la\ condición\ de\ regularidad$$

$$a\ f(\frac{n}{b}) = 5(\frac{n}{2})^3 = 5\frac{n^3}{8} \leq c\,n^{3,} \quad para\ c = \frac{5}{8} < 1$$

$$corresponde\ al\ caso\ 3 \Rightarrow T(n) = \Theta(n^3)$$

Forma simple de aplicar el TM

$$T(N) = \begin{cases} \Theta(1) & \text{Si N = 1} \\ aT(N/b) + O(N^d) & \text{Si N > 1} \end{cases}$$

Con a > 0, b > 1 y d >= 0

$$T(N) = \begin{cases} O(N^d) & \text{Si } d > \log_b a \\ O(N^d \log N) & \text{Si } d = \log_b a \\ O(N^{\log_b a}) & \text{Si } d < \log_b a \end{cases}$$

Ejercicios para la clase

Resolver las siguientes recurrencias

1) $T(n) = 16T(\frac{n}{4}) + n^2$

2) $T(n) = 7T(\frac{n}{2}) + n^2$

3) $T(n) = 2T(\frac{n}{4}) + \sqrt{n}$

4) $T(n) = 7T(\frac{n}{3}) + n^2$

5) $T(n) = T(n-1) + n$

6) $T(n) = 3T(\frac{n}{2}) + n\log_2 n$

7) $T(n) = T(n-1) + 1$

Bibliografía

[CLRS 2009] Cormen T., Leiserson C., Rivest R., Stein C. *Introduction to Algorithms*. 3rd Edition. 2009.
MIT Press. *Capítulo 2 y 3*.