

# Tema 3

## ALGORITMIA

Parte III. *Recurrencia* . 2016/2017



Irene Martínez Masegosa

Depto. de Informática

<mailto:irene@ual.es>

3.3-1

### Contents

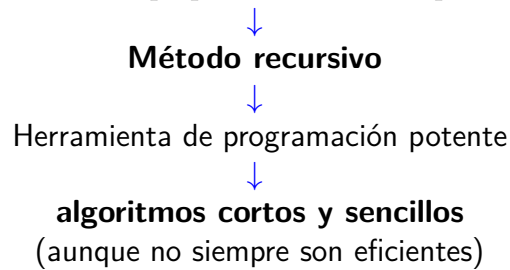
1	Algoritmos	2
2	Análisis	10
3	Árboles de Recursión	14
4	Método Maestro	17
5	Recurrencias Básicas	21
6	Aplicaciones	21
7	Problemas	25
A	Conclusiones	27

3.3-2

- El concepto de recurrencia es fundamental en matemáticas y en computación
- Un programa “recursivo” es un programa que se llama a sí mismo del mismo modo que una función matemática se define en términos de sí misma.

## Motivación

Resolver un problema mediante recursión significa que la solución depende de soluciones más pequeñas del mismo problema.



3.3-3

## Un poco de Historia...



[http://www.nytimes.com/2011/10/26/science/26mccarthy.html?\\_r=0](http://www.nytimes.com/2011/10/26/science/26mccarthy.html?_r=0)

- El profesor John McCarthy del Massachusetts Institute of Technology, MIT, fue el primero en considerar el uso de la recursión en los lenguajes de programación.
- Defendió que se incluyese en el diseño del lenguaje Algol60, que fue precursor de lenguajes como Pascal, PL/I y C.
- Además desarrolló el lenguaje Lisp, que introdujo estructuras de datos recursivas junto a procedimientos y funciones recursivos.
- El valor de la recursión fue muy apreciado durante el intenso periodo de desarrollo de algoritmos en los años 70.
- Actualmente todos los lenguajes de programación soportan recursión (Java, C++,...)

3.3-4

## 1 Algoritmos con recurrencia

### ¿Qué es la recurrencia?

#### Recursive Algorithm

Un **algoritmo recursivo** es aquel que resuelve un problema resolviendo una o más instancias **más pequeñas del mismo problema**.

*Ejemplo: Sucesión de Fibonacci*

$$f(n) = \begin{cases} 0; & \text{si } n = 0 \\ 1; & \text{si } n = 1 \\ f_{n-1} + f_{n-2} & \text{si } n \geq 2 \end{cases}$$

```

funcion FibREC(n)
  IF n < 2 devolver n
  ELSE
    devolver FibREC(n - 1) + FibREC(n - 2)

```

$$\begin{aligned}
 \text{FibREC}(6) &= \text{FibREC}(5) + \text{FibREC}(4) \\
 &= \dots \\
 &= \dots \text{FibREC}(1) + \text{FibREC}(0)
 \end{aligned}$$

Números de Fibonacci: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...

3.3-5

### Construcción de un algoritmo recursivo

- *Método recursivo*: se define en términos de una instancia más pequeña de sí mismo:  $\Rightarrow$  Debe haber algún **caso base** que se pueda resolver **sin llamadas recursivas**.

Un **algoritmo recursivo** consta de:

- **Caso base**: (sin llamadas recursivas) implementación para valores específicos de los argumentos.
- **Etapas de reducción**: (llamadas recursivas) asume que el método funciona para valores más pequeños de sus argumentos y lo utiliza para implementar otros valores.
  - Las llamadas recursivas deben progresar hacia un caso base.  
(Un fallo en el progreso hacia el caso base hace que el programa no funcione)

3.3-6

### Ejemplo. Máximo Común Divisor Algoritmo de *Euclides*

$$\text{GCD}(p, q) = \begin{cases} p; & \text{si } q = 0 & \leftarrow \text{Caso base} \\ \text{GCD}(q, p \% q) & \text{otro caso} & \leftarrow \text{Reducción (converge c. base)} \end{cases}$$

```

funcion GCD(p, q)
  IF q == 0 devolver p;    ← caso base
  ELSE
    devolver GCD(q, p % q);

```

$$\begin{aligned}
 \text{GCD}(4032, 1272) &= \text{GCD}(1272, 216) && \leftarrow 4032 = 3 \times 1272 + 216 \\
 &= \text{GCD}(216, 192) \\
 &= \text{GCD}(192, 24) \\
 &= \text{GCD}(24, 0) \\
 &= 24
 \end{aligned}$$

3.3-7

## Algoritmos recursivos conocidos

Ejemplo: Cálculo del Factorial

$$Factorial(n) = \begin{cases} 1; & \text{si } n = 0 \\ n \times Factorial(n-1) & \text{si } n \geq 1 \end{cases}$$

```
funcion Factorial(n)
  IF n == 0 devolver 1;      ← caso base
  ELSE
    devolver n × Factorial(n-1);
```

```
Factorial(5) = 5 × Factorial(4)
             = 4 × Factorial(3)
             = 3 × Factorial(2)
             = 2 × Factorial(1)
             = 1 × Factorial(0)      ← caso base
             return 1
             return 2 × 1 = 2
             return 3 × 2 = 6
             return 4 × 6 = 24
             return 5 × 24 = 120
             = 120
```

3.3-8

## Algoritmos recursivos conocidos

Ejemplo: Suma de los  $n$  primeros enteros

$$suma(n) = \begin{cases} 1; & \text{si } n = 1 \\ n + suma(n-1) & \text{si } n > 1 \end{cases}$$

```
funcion SUMAenteros(n)
  IF n == 1 devolver 1
  ELSE
    devolver SUMAenteros(n-1) + n
```

```
SUMA(5) = 5 + SUMA(4)
         = 4 + SUMA(3)
         = 3 + SUMA(2)
         = 2 + SUMA(1)
         = 1          ← caso base
         return 1
         return 2 + 1 = 3
         return 3 + 3 = 6
         return 4 + 6 = 10
         return 5 + 10 = 15
         = 15
```

3.3-9

## ¿Cómo funciona?

- Es necesario entender cómo funciona la recursión en un ordenador.
- Las llamadas recursivas a métodos se implementan mediante **activation frames**.

### Activation frame

Es la unidad básica de **almacenamiento** en una invocación de un método durante la ejecución. En el *frame* se almacenan:

- las **variables locales** del método
- los **parámetros actuales**
- **variables temporales** del compilador incluida la variable de la instrucción `return()`
- **Dirección de retorno**: es la instrucción que debe ejecutarse cuando “vuelva” de la llamada recursiva

3.3-10

### Pila de LLamadas Recursivas

- Cuando se produce una llamada a un método recursivo el compilador genera código para ocupar espacio de una zona de almacenamiento llamada **pila** (memoria de pila)
- A esta zona de memoria se accede a través de un registro especial llamado **frame pointer**.
- Al ejecutar una invocación del método recursivo, a través de ese puntero se sabe dónde están almacenadas sus *variables locales*, *parámetros de entrada*, y el valor de *return*.
- Cada invocación de método tiene asociado un **único frame**.
- Se producen varias llamadas recursivas:
  - se genera un *frame* para cada llamada y
  - se van guardando como una pila.
- Cuando la rutina “vuelve” de la llamada: recupera su *frame*, y con él sus variables locales, etc.

3.3-11

### Ejemplos. Algoritmos GCD y Factorial.

Traza de llamadas recursivas y *frames* en los algoritmos:

**GCD:** <http://introcs.cs.princeton.edu/lectures/23demo-gcd.pps>

**Factorial:** <http://introcs.cs.princeton.edu/lectures/23demo-factorial.pps>

3.3-12

### Ejemplo: Cálculo del Factorial

Factorial(3)

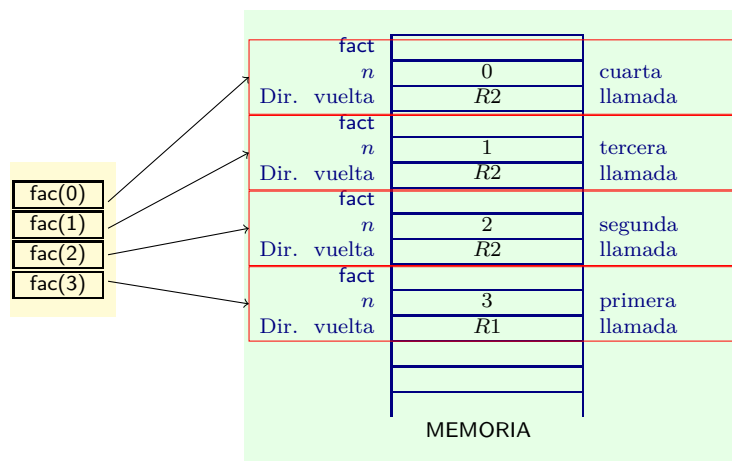
```
public class Factorial {
    public static int fact(int n) {
        if (n == 0) return 1;
        else return n * fact(n-1);
    }

    public static void main(String[] args) {
        System.out.println(fact(3));
    }
}
```

<http://introcs.cs.princeton.edu/lectures/23demo-factorial.pps>

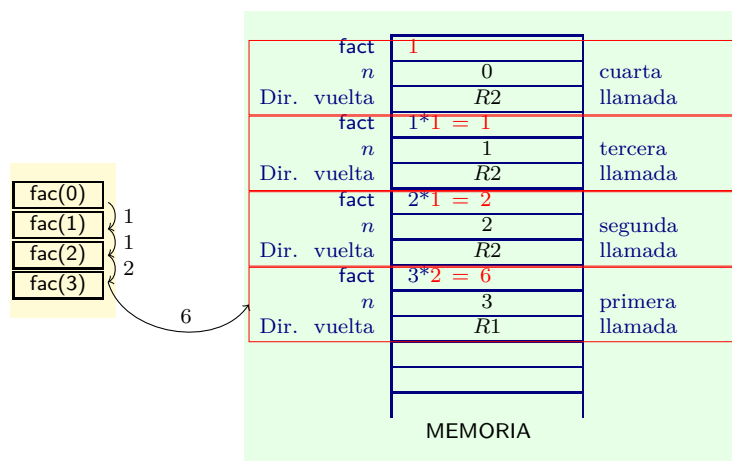
3.3-13

## Pila de llamadas recursivas



3.3-14

## Devolución de valores en las llamadas recursivas



3.3-15

## Reglas Básicas de la Recursión

### Reglas Básicas:

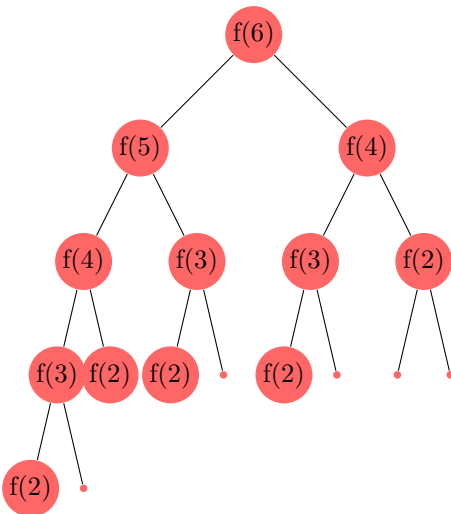
1. **Casos bases:** se debe tener siempre al menos un caso base que pueda resolverse sin recursión
2. **Progreso:** las llamadas recursivas deben progresar hacia un caso base.
3. **Corrección:** asumir siempre que toda llamada recursiva interna funciona correctamente.
4. **No recalcular términos:** nunca hay que duplicar trabajo resolviendo la misma instancia de un problema en llamadas recursivas separadas.

3.3-16

## Ejemplo Recursión Ineficiente: Términos de la sucesión de Fibonacci

```

funcion FibREC(n)
  IF n < 2 devolver n
  ELSE
    devolver FibREC(n-1) + FibREC(n-2)
  
```



- $f(4)$  se calcula 2 veces
- $f(3)$  se calcula 3 veces
- $f(2)$  se calcula 5 veces

3.3-17

## Ejemplo Recursión Ineficiente: Coeficientes Binomiales

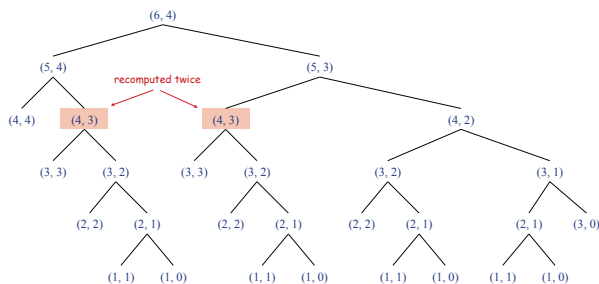
*Cálculo del coeficiente binomial*

$$\binom{n}{k} = \begin{cases} 1 & \text{si } k = 0, \quad k = n \\ \binom{n-1}{k-1} + \binom{n-1}{k} & \text{si } 0 < k < n \\ 0 & \text{otro caso} \end{cases}$$

```

funcion BinomREC(n,k)
  IF ( k = 0 OR k = n )   Devolver 1
  ELSE
    Devolver( BinomREC(n-1,k-1) + BinomREC(n-1,k) )
  
```

$$\begin{aligned}
 \text{BinomREC}(6,4) &= \text{BinomREC}(5,4) + \text{BinomREC}(5,3) \\
 &= \dots \\
 &= \dots \text{BinomREC}(1,1) + \text{BinomREC}(1,0)
 \end{aligned}$$



Copyright: Algorithms 4<sup>th</sup> by Sedgwick R, Wayne K., 2002-2010

$$\rightarrow T(n,k) \in \Omega\left(\binom{n}{k}\right)$$

3.3-18

## Inducción Matemática

- La programación recursiva está directamente relacionada con la **inducción matemática**.
- La inducción es una técnica para demostrar teoremas sobre funciones discretas.
- Probamos que un teorema es cierto para un entero  $N$ , y para otros posibles valores de  $N$ , en **dos etapas**:
  - **Caso Base**: probamos que la afirmación es cierta para unos valores sencillos de  $N$  (generalmente 0 ó 1)
  - **Etapla de Inducción**: es la parte principal de la demostración.
    - \* Establecemos la **hipótesis de inducción**, en la que consideramos que el teorema es cierto para un  $k$  arbitrario **menor que**  $N$ ,
    - \* A partir de ésta probamos que el teorema es cierto para  $k + 1$ ,
    - \* Y por inducción, el teorema queda demostrado para todos los casos de  $N$ .

3.3-19

## Inducción Matemática. Ejemplo

**Teorema:** La suma de los  $n$  primeros números impares es  $n^2$

$$\begin{aligned}
 1 &= 1 \\
 1 + 3 &= 4 \\
 1 + 3 + 5 &= 9 \\
 1 + 3 + 5 + 7 &= 16 \\
 1 + 3 + 5 + 7 + 9 &= 25
 \end{aligned}$$

- Caso base: cierto para  $n=1$
- Etapa de Inducción:
  - . Sea  $T(n)$  la suma de los primeros  $n$  núm impares:  $1 + 3 + 5 + \dots + (2n - 1)$
  - . Hipótesis: supondremos  $T(n - 1) = (n - 1)^2$
  - . Entonces:

$$\begin{aligned}
 T(n) &= T(n - 1) + (2n - 1) \\
 &= (n - 1)^2 + (2n - 1) \\
 &= n^2 - 2n + 1 + (2n - 1) \\
 &= n^2
 \end{aligned}$$

Ejercicio: Demostrar que la suma de los  $n$  primeros enteros vale  $\frac{n(n+1)}{2}$

Nota: Ver Weiss

3.3-20

## ¿Por qué funciona la Recursión?

- Las demostraciones por inducción nos muestran que podemos demostrar la certeza de una afirmación considerando que es cierta para casos más sencillos.



- Se puede demostrar la corrección de los algoritmos recursivos usando inducción matemática
- Ejemplo:  
Imprimir un número no negativo  $N$  en base 10.

```
void ImprimeBaseDiez( $n$ )
1.  IF  $n \geq 10$ 
2.      ImprimeBaseDiez( $n/10$ )
3.      Imprimir( $n\%10$ )
```

– Ejemplo: 1369, debe imprimirse como 1, 3, 6, 9

```
ImprimeBaseDiez(1369) :    ImprimeBaseDiez(136)    (a la vuelta Imprime 9)
                        :    ImprimeBaseDiez(13)     (a la vuelta Imprime 6)
                        :    ImprimeBaseDiez(1)       (a la vuelta Imprime 3)
                        :    Imprimir(1)              ← caso base
                        :    Imprimir(3)
                        :    Imprimir(6)
                        :    Imprimir(9)
```

- Demostración [Weiss 7.3]

3.3-21

## 2 Análisis de Algoritmos Recursivos

### Ecuación de Recurrencia

- En un algoritmo recursivo, la función  $T(n)$  que establece su complejidad viene dada por una **ecuación de recurrencia**.

#### Recurrencia

Una recurrencia es una ecuación o desigualdad que describe una función en términos de sus valores para entradas más pequeñas.

- Sintetiza la relación de dependencia entre el **tiempo de ejecución** de un algoritmo para una **entrada de tamaño  $n$**  y el de **entradas de tamaños menores**.
- Por ejemplo:  $T(n) = T(n-1) + T(n-2)$

3.3-22

### ¿Cómo hacemos el Análisis del Algoritmo Recursivo?

1. A partir del código del algoritmo calculamos el **tiempo** empleado en las **llamadas recursivas** y el del **caso/s base**:
  - hay que tener en cuenta el **número** de llamadas recursivas
  - y el **tamaño de la entrada** en cada una de ellas
2. Planteamos una **ecuación de recurrencia** que recoja todos los tiempos
3. **Resolver** la ecuación, con un método adecuado

3.3-23

## Ejemplo. Ecuación de Recurrencia

Ejemplo: Sucesión de Fibonacci con algoritmo recursivo

```
funcion FibREC(n)
  IF n < 2 devolver n
  ELSE devolver FibREC(n-1) + FibREC(n-2)
```

- Para  $n < 2$ ,  $T(n)$  es constante:  $a$   
En otro caso:
  - dos llamadas recursivas que requieren:  $T(n-1)$  y  $T(n-2)$
  - sumar  $f_{n-1}$  y  $f_{n-2}$  y comprobaciones:  $\rightarrow h(n)$
- **Ecuación de recurrencia:**

$$T(n) = \begin{cases} a & \text{si } n < 2 \\ T(n-1) + T(n-2) + h(n) & \text{otro caso} \end{cases}$$

- **Solución** a la ecuación :  $T_{FibREC}$  es **exponencial** en  $n$  [Brassard 4.7.2]

3.3-24

## Cómo plantear la Ecuación de Recurrencia

- Sea  $n$  el tamaño del problema.
- Analizando para el **caso peor**,  $T(n)$  se define en función de  $n$  como la **SUMA** de:

### Coste RECURSIVO

1. Coste de todas las llamadas recursivas:

$$T(k_1) + T(k_2) + \dots + T(k_i) + \dots$$

- siendo  $k_i$  el **tamaño del problema en la llamada recursiva  $i$** ,
- $k_i < n$

### Coste NO RECURSIVO

2. Calcular costes de códigos no recursivos:

- coste de los **casos base**
- coste bloques que no son caso base,
- coste de llamadas a funciones NO-recursivas

3. Aplicar regla de la SUMA y obtener el coste MÁXIMO:  $f(n)$

$$T(n) = \text{Coste RECURSIVO} + \text{Coste NO RECURSIVO}$$

$$T(n) = \overbrace{T(k_1) + T(k_2) + \dots + T(k_i) + \dots} + \overbrace{f(n)}$$

3.3-25

## Ejemplo. Algoritmo de ordenación Mergesort

```

MergeSort(i, j, T)
1.  n ← j - i + 1      // Núm. de elementos a ordenar
2.  IF (n == 1) Devolver T
3.  ELSE
    (a) S ← n div 2
    (b) MergeSort(i, S, T)
    (c) MergeSort(S + 1, j, T)
    (d) Mezclar(i, S, j, T)    // Mezclar los dos subvectores

```

$$T(n) = \begin{cases} a & \text{si } n = 1 \\ \underbrace{T(n/2)}_{\text{mitad izquierda}} + \underbrace{T(n/2)}_{\text{mitad derecha}} + \underbrace{n}_{\text{mezcla}} & \text{otro caso} \end{cases}$$

- Coste recursivo= Ordenar mitad izda + Ordenar mitad derecha =  $T(n/2) + T(n/2)$
- Coste no recursivo= Coste caso base + Coste Mezcla subvectores ordenados =  $a + \Theta(n) = \Theta(n)$
- Ecuación de recurrencia:  $T(n) = T(n/2) + T(n/2) + n$

$$T(n) = 2T(n/2) + n$$

3.3-26

## Ejemplo. Algoritmo Factorial

```

funcion Factorial(n)
  IF n == 0 devolver 1;    ← caso base
  ELSE
    devolver n × Factorial(n - 1);

```

$$T(n) = \begin{cases} a & \text{si } n = 0 \\ \underbrace{T(n-1)}_{\text{1 sola llamada}} + \underbrace{b}_{\text{operaciones elementales}} & \text{otro caso} \end{cases}$$

- Coste recursivo: tiempo invertido en 1 llamada de tamaño  $n - 1 = T(n - 1)$
- Coste no recursivo: Coste caso base + Coste op. elementales =  $a + b = \Theta(1)$
- Ecuación de recurrencia:  $T(n) = T(n - 1) + 1$

3.3-27

## Ejemplo. Sucesión de Fibonacci

```

funcion FibREC(n)
  IF n < 2 devolver n
  ELSE
    devolver FibREC(n - 1) + FibREC(n - 2)

```

$$T(n) = \begin{cases} a & \text{si } n < 2 \\ \underbrace{T(n-1)}_{\text{Primera llamada}} + \underbrace{T(n-2)}_{\text{Segunda llamada}} + \underbrace{b}_{\text{op. elem}} & \text{otro caso} \end{cases}$$

- Coste recursivo= tiempo invertido en 1 llamada de tamaño  $n-1$ :  $T(n-1)$ , más el invertido en otra llamada de tamaño  $n-2$ :  $T(n-2) = T(n-1) + T(n-2)$
- Coste no recursivo= Coste caso base + Coste suma=  $a + b = \Theta(1)$
- Ecuación de recurrencia:  $T(n) = T(n-1) + T(n-2) + 1$

3.3-28

### Ejemplo: Búsqueda Binaria Recursiva

```
funcion int BinREC(A,x,i,j);
    // Busca x en el array A, en el intervalo A[i] - A[j]

    1. IF j < i    Devolver(-1)
    2. medio ← (i + j)/2
    3. IF x ≤ A[medio]
        Devolver( BinREC(A,x,i,medio) )

    4. ELSE
        Devolver( BinREC(A,x,medio+1,j) )
```

- Coste no-recursivo: pasos 1 y 2 + comparación en paso 3: *cte*
- Coste recursivo: Hay dos llamadas recursivas en un bloque IF-ELSE :
  - Tenemos que analizar la parte IF y la parte ELSE y tomar el tiempo de la que tarda más
  - es decir, para calcular el tiempo, solo se considera la parte recursiva del IF o la del ELSE
  - Si  $n$  es el número de elementos considerados del array  $n = j - i$ 
    - la parte del IF : 1 llamada de tamaño  $n/2 = T(n/2)$
    - y la parte del ELSE : 1 llamada de tamaño  $n/2 = T(n/2)$
  - Por ello **coste recursivo**=  $T(n/2)$

$$T(n) = \begin{cases} a & \text{si } n = 1 \\ b + T(\frac{n}{2}) & \text{otro caso} \end{cases} \quad a, b : \text{cte}$$

- Ecuación de recurrencia:  $T(n) = T(n/2) + 1$

3.3-29

### Métodos para resolver Recurrencias

- **Método de Sustitución, Ecuación Característica, ...**
  - Se verán en la asignatura EDA2 [Brassard 4.7]
- **Árboles de Recursión**
  - No es un método formal pero sí es intuitivo
- **Método Maestro**
  - Se aplica en un determinados tipos de recurrencias

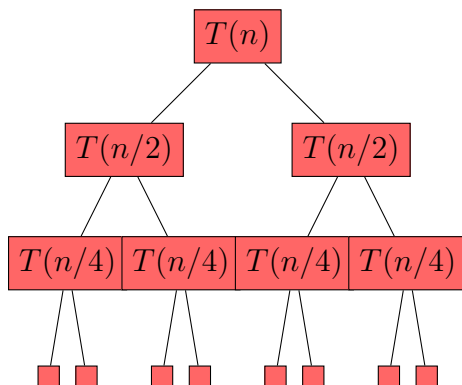
3.3-30

### 3 Árboles de Recursión

#### Recursion Trees

#### Árboles de Recursión

Son una herramienta para analizar las llamadas que realizan los algoritmos recursivos. Nos permiten estudiar la carga de trabajo de un algoritmo (tiempo de ejecución, número de comparaciones, etc.)



- Cada nodo representa el tamaño del problema
- Añadimos el tiempo de ejecución **no-recursivo** de cada nodo
- Sumamos estos tiempos por niveles = coste del nivel  $k$
- Suma de todos los niveles = **coste del algoritmo**

3.3-31

#### Cálculo de $T(n)$ con un árbol de recursión

- Calcular el **máximo número de niveles** = altura del árbol =  $numNIVELES$
- Sea  $costeNR(k)$  el **coste NO recursivo del nivel  $k$** , es decir, la suma del coste de todos los nodos del nivel  $k$  del árbol
- Entonces:

$$T(n) \leq \sum_{k=0}^{numNIVELES-1} costeNR(k)$$

3.3-32

#### Ejemplo. Algoritmo de ordenación MergeSort

- Ordena un array de  $n$  elementos con dos llamadas recursivas de subarrays de tamaño  $n/2$  (suponemos  $n$  potencia de 2)
- Mezcla los subarrays en un tiempo de orden  $\leq \Theta(n)$

$$T(n) = \begin{cases} a & \text{si } n = 1 \\ \underbrace{T(n/2)}_{\text{mitad izquierda}} + \underbrace{T(n/2)}_{\text{mitad derecha}} + \underbrace{n}_{\text{mezcla}} & \text{otro caso} \end{cases}$$

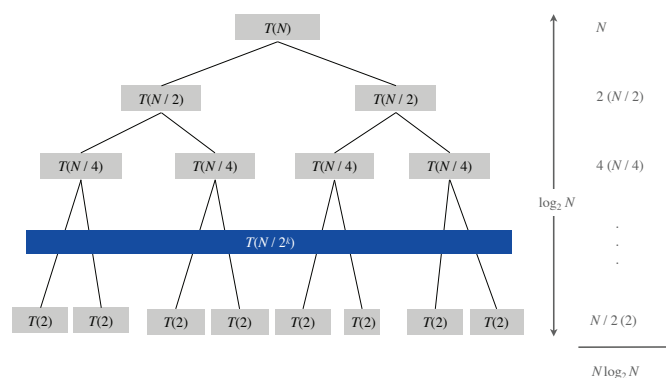
- Ecuación de recurrencia :

$$T(n) = 2T(n/2) + n$$

- El **coste no recursivo** es  $n$

3.3-33

## MergeSort. Árbol de Recursión



- El **número de niveles** del árbol es :  $\log_2 n$   
(vamos dividiendo por 2 el tamaño del problema)
- El coste no recursivo de cada nodo es  $n$ , y sumando este coste para cada nivel obtenemos que el **coste no recursivo para el nivel  $k$**  es:

$$2^k \cdot \frac{n}{2^k}$$

- Calculamos:

$$T(n) \leq \sum_{k=0}^{numNIVELES-1} costeNR(k)$$

$$T(n) \leq \sum_{k=0}^{numNIVELES-1} 2^k \cdot \frac{n}{2^k} = \sum_{k=0}^{\log n - 1} n = \sum_{k=1}^{\log n} n = n \log n$$

- El algoritmo está en  $\mathcal{O}(n \log n)$

3.3-34

## Resumen de pasos para crear Árbol de Recursión

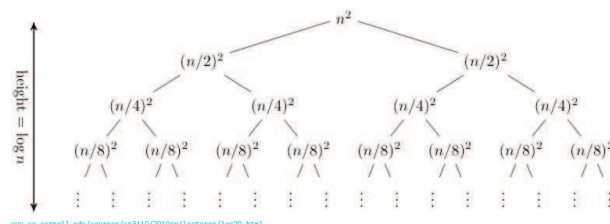
1. Empezar con nodo raíz asociándole su tamaño
2. Expandir el resto de los nodos:
  - Determinar el coste no-recursivo del nodo (a partir de la ecuación)
  - Crear los nodos descendientes para las llamadas recursivas
  - Los caso base (nodos hoja) son nodos sin coste no-recursivo de valor 1
3. Calcular el máximo número de niveles del árbol.
4. Sumar el coste de todos los niveles. Al plantear la sumatoria podemos encontrar, en general, patrones como:
  - Series geométricas en las que los niveles superiores son más costosos
  - Todos los niveles con aproximadamente el mismo coste.
  - Series geométricas en las que los niveles inferiores son más costosos

3.3-35

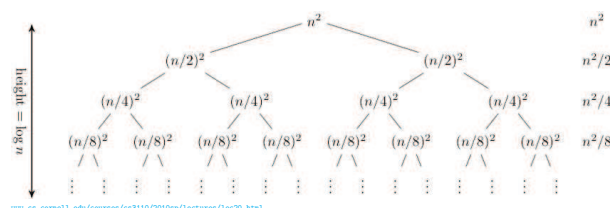
## Ejemplo. Árbol de Recursión

El árbol de recursión nos permite visualizar lo que ocurre cuando iteramos una recurrencia.

- Por ejemplo, sea la recurrencia:  $T(n) = 2T(n/2) + n^2$



- Sumamos por niveles para obtener el coste total:



- El **número de niveles** del árbol es :  $\log_2 n$
- El coste no recursivo de cada nodo es  $n^2$ , y sumando este coste para cada nivel obtenemos que el **coste no recursivo para el nivel  $k$**  es:

$$\frac{n^2}{2^k}$$

- Calculamos:

$$T(n) \leq \sum_{k=0}^{numNIVELES-1} costeNR(k)$$

$$T(n) \leq \sum_{k=0}^{numNIVELES-1} \frac{n^2}{2^k} = \sum_{k=0}^{\log n - 1} \frac{n^2}{2^k} = n^2 \sum_{k=0}^{\log n - 1} \frac{1}{2^k}$$

$$T(n) \leq n^2 \left(1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \dots\right)$$

- Resultado: una sucesión geométrica que suma (en límite)  $O(n^2)$ , y por tanto  $T(n) \in O(n^2)$

## 4 Método Maestro

### Master Method

#### Método Maestro

- Es un método para resolver recurrencias de la forma:

$$T(n) = aT(n/b) + f(n)$$

- $a \geq 1$  y  $b > 1$  son constantes y  $f(n)$  función asintóticamente positiva
- La recurrencia describe el tiempo de ejecución de un algoritmo que resuelve un problema de tamaño  $n$  en:
  - $a$  llamadas recursivas de subproblemas de tamaño  $n/b$ , siendo:
  - $T(n/b)$ : el tiempo empleado en resolver cada llamada **recursiva**, y
  - $f(n)$  el coste no-recursivo del algoritmo

3.3-37

#### Soluciones de la Recurrencia del Método Maestro

##### Teorema Maestro

Sean  $a \geq 1$  y  $b > 1$  constantes,  $f(n)$  una función, y sea  $T(n)$  definida en los enteros no negativos por la recurrencia:

$$T(n) = aT(n/b) + f(n)$$

- Entonces  $T(n)$  puede acotarse asintóticamente como:

$$T(n) \in \begin{cases} \Theta(n^{\log_b a}) & \text{si } f(n) \in \mathcal{O}(n^{\log_b a - \epsilon}) \\ \Theta(n^{\log_b a} \log n) & \text{si } f(n) \in \Theta(n^{\log_b a}) \\ \Theta(f(n)) & \text{si } f(n) \in \Omega(n^{\log_b a + \epsilon}) \end{cases} \quad (*)$$

(\*) En el tercer caso, es necesario que  $af(n/b) \leq cf(n)$

- $\epsilon > 0$ ,  $c < 1$ : constantes
- $n$ : tamaño del problema
- $a$  llamadas recursivas de subproblemas de tamaño  $n/b$

3.3-38

#### Interpretación del Teorema Maestro

- En los tres casos se comparan las funciones  $f(n)$  y  $n^{\log_b a}$
- Intuitivamente **la solución a la recurrencia está determinada por la mayor de las dos funciones**:
  - Caso 1: solución:  $T(n) \in \Theta(n^{\log_b a})$ , ya que  $n^{\log_b a}$  es la mayor.
  - Caso 2: las dos funciones son del mismo tamaño, multiplicamos por un factor logarítmico y la solución es:

$$T(n) \in \Theta(n^{\log_b a} \log n) = \Theta(f(n) \log n)$$



- Caso 3:  $T(n) \in \Theta(f(n))$  , ya que  $f(n)$  es la mayor
- Pero esta condición no es suficiente. **Además se debe cumplir:**
  - Caso 1:  $f(n)$  debe ser **polinómicamente menor** que  $n^{\log_b a}$ .  
Esto es, debe ser menor *por un factor de*  $n^\epsilon$  ,  $\epsilon > 0$  .
  - Caso 3:  $f(n)$  debe ser **polinómicamente mayor** que  $n^{\log_b a}$ , y además satisfacer la restricción de que  $af(n/b) \leq cf(n)$

3.3-39

### Ejemplo 1. Solución de Recurrencias

1.  $T(n) = 9T(n/3) + n$ 
  - .  $a = 9$  llamadas recursivas de tamaño  $n/3$
  - .  $b = 3$
  - .  $f(n) = n$
  - . Calculamos  $n^{\log_b a} = n^{\log_3 9} = n^2 \in \Theta(n^2)$   
y lo comparamos con  $f(n)$ :
    - Como  $f(n) \in \mathcal{O}(n^2)$ , comprobamos las condiciones del caso 1
  - Se debe cumplir que:
    - $f(n) \in \mathcal{O}(n^{\log_b a - \epsilon}) = \mathcal{O}(n^{\log_3 9 - \epsilon})$ , es decir
    - $f(n) \in \mathcal{O}(n^{2-\epsilon})$ ,
    - y es cierto para  $\epsilon = 1$ ,
  - Podemos aplicar el Caso 1 y por tanto la solución a la recurrencia es:

$$T(n) \in \Theta(n^{\log_3 9}) \Rightarrow T(n) \in \Theta(n^2)$$

3.3-40

### Ejemplo 2. Solución de Recurrencias

2.  $T(n) = T(2n/3) + 1$ 
  - .  $a = 1$  una llamada recursiva de tamaño  $2n/3$ .
  - .  $b = 3/2$
  - .  $f(n) = 1$
  - . Calculamos  $n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1 \in \Theta(1)$   
y comparamos con  $f(n)$ :
    - Como  $f(n) \in \Theta(1) = \Theta(n^{\log_b a})$ , entonces aplicamos Caso 2.
  - Por tanto, la solución a la recurrencia es:

$$T(n) \in \Theta(n^{\log_b a} \log n) \Rightarrow T(n) \in \Theta(\log n)$$

3.3-41

### Ejemplo 3. Solución de Recurrencia

3.  $T(n) = 3T(n/4) + n \log n$

.  $a = 3$  ,  $b = 4$

.  $f(n) = n \log n$

. Calculamos  $n^{\log_b a} = n^{\log_4 3} \in \mathcal{O}(n^{0.793})$

y comparamos con  $f(n)$

– Como  $f(n) = n \log n \in \Omega(n^{\log_b a})$  , es decir  $f(n) \in \Omega(n^{0.793})$  , comprobamos el caso 3

• Se deben cumplir dos condiciones:

i)  $f(n) \in \Omega(n^{\log_b a + \epsilon}) = \Omega(n^{\log_4 3 + \epsilon})$ ,  
cierto para  $\epsilon \approx 0.2$

ii) Condición de regularidad para  $f(n)$ . Para  $n$  suficientemente grande,

$$af(n/b) \leq cf(n)$$

$$af(n/b) = 3(n/4) \log(n/4) \leq (3/4)n \log n = cn \log n$$

cierto para  $c = 3/4 < 1$ .

• La solución a la recurrencia es:

$$T(n) \in \Theta(f(n)) \Rightarrow T(n) \in \Theta(n \log n)$$

3.3-42

### Cuándo NO es posible aplicar el Método Maestro

• Hay que destacar que los tres casos del teorema NO cubren todas las posibilidades para  $f(n)$ .

• Si la función  $f(n)$  está en una de estas situaciones, **no podemos usar el método maestro para resolver la recurrencia**:

– Caso 1:  $f(n)$  es **menor** que  $n^{\log_b a}$ , pero no lo es polinómicamente.

– Caso 3:  $f(n)$  es **mayor** que  $n^{\log_b a}$ , pero no polinómicamente, o NO se satisface la restricción de que  $af(n/b) \leq cf(n)$

3.3-43

### Ejemplo 4. Solución de Recurrencia

4.  $T(n) = 2T(n/2) + n \log n$

.  $a = 2$  ,  $b = 2$

.  $f(n) = n \log n$

• Calculamos  $n^{\log_b a} = n^{\log_2 2} = n$

y comparamos con  $f(n)$

– Como  $f(n) = n \log n \in \Omega(n^{\log_b a})$  , es decir  $f(n) \in \Omega(n)$  , comprobamos el caso 3

- Tenemos que encontrar un  $\epsilon > 0$  que cumpla que:
  - $f(n) \in \Omega(n^{\log_b a + \epsilon})$  es decir,  $n \log n \in \Omega(n^{1+\epsilon})$ ,
  - $\epsilon = ??$
  - No podemos acotar:  $f(n) \in \Omega(n^{1+\epsilon})$  para  $\epsilon > 0$
- $f(n) = n \log n$  es asintóticamente mayor que  $n^{\log_b a} = n$  pero **no polinómicamente mayor**
- NO es posible aplicar el Caso 3: la relación

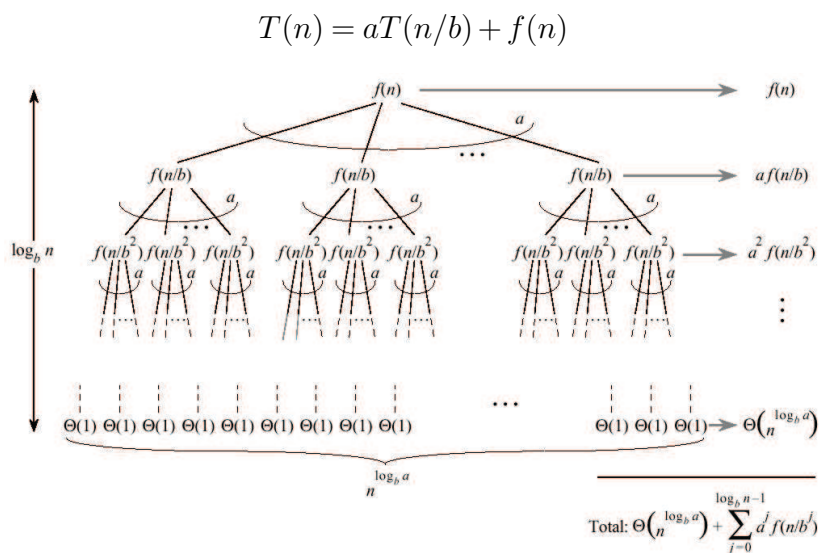
$$\frac{f(n)}{n^{\log_b a}} = \frac{n \log n}{n} = \log n$$

es asintóticamente menor que  $n^\epsilon$ , para cualquier constante positiva  $\epsilon$

- NO podemos resolver esta recurrencia con el Método Maestro.

3.3-44

### Árbol de Recursión del Teorema Maestro



- número de niveles = altura del árbol =  $h = \log_b n$
- número de hojas =  $a^h = a^{\log_b n} = n^{\log_b a}$
- Coste:

$$T(n) = \Theta(n^{\log_b a}) + \sum_{j=0}^{\log_b n - 1} a^j f(n/b^j)$$

$T(n) =$  coste de hacer  $n^{\log_b a}$  subproblemas de tamaño 1 +  
coste de los nodos interiores

3.3-45

## 5 Recurrencias Básicas

### Fórmulas Básicas en el Análisis de Algoritmos Recursivos [Sedgewick 2.5]

Algunas fórmulas habituales utilizadas en el análisis de algoritmos recursivos que no pueden ser calculadas mediante el Método Maestro.

#### **Fórmula 1**

Modeliza programas que recorren la entrada para eliminar un ítem:

$$T(n) = T(n-1) + n \quad \text{para } n \geq 2 \quad \text{y } T(1) = 1$$

**Solución:**

$$T(n) \approx \frac{n^2}{2}$$

*Demostración:*  $T_n = T_{n-1} + n$

$$\begin{aligned} T_n &= T_{n-1} + n \\ &= T_{n-2} + (n-1) + n \\ &= T_{n-3} + (n-2) + (n-1) + n \\ &= \dots \\ &= T_1 + 2 + \dots + (n-2) + (n-1) + n \\ &= 1 + 2 + \dots + (n-2) + (n-1) + n \\ &= \frac{n(n+1)}{2} \end{aligned}$$

#### **Fórmula 2**

$$T(n) = T(n-1) + 1 \quad \text{para } n \geq 2 \quad \text{y } T(1) = 1$$

**Solución:**

$$T(n) \approx n$$

*Demostración:*  $T_n = T_{n-1} + 1$

$$\begin{aligned} T_n &= T_{n-1} + 1 \\ &= (T_{n-2} + 1) + 1 \\ &= T_{n-2} + 2 \\ &= \dots \\ &= T_{n-i} + i \\ &= T_1 + n - 1 = 1 + n - 1 \\ &= n \end{aligned}$$

3.3-46

## 6 Aplicaciones de la recurrencia en diseño de Algoritmos

### Diseño de Algoritmos

- Los algoritmos recursivos se utilizan en los paradigmas de diseño de algoritmos más importantes:
  - *Divide and Conquer* : algoritmos “Divide y Vencerás”
  - *Dynamic Programming* : algoritmos de “Programación Dinámica”
  - *Backtracking* : algoritmos de “Vuelta Atrás”

3.3-47

## Paradigma Divide y Vencerás

- Consiste en **descomponer** el problema en un conjunto de subproblemas más pequeños
- Se **resuelven** estos subproblemas y se **combinan** las soluciones para obtener la solución para el problema original

```

Divide_y_Venceras(P: problema)
  descomponer(P, p1, p2, ..., pm)
  FOR i = 1, 2, ..., m
    si = Divide_y_Venceras(pi)
  solucion = combinar(s1, s2, ..., sm)

```

- Los subproblemas deben ser **disjuntos** (sin superposiciones)
- Para resolver los subproblemas se realizan llamadas recursivas al mismo algoritmo
- **Tiempo de ejecución**: podemos aplicar el **método maestro** en la mayoría de los casos

3.3-48

## Divide y Vencerás. Ejemplo 1

### MergeSort

```

MergeSort(i, j, T)
1. n ← j - i + 1 // Núm. de elementos a ordenar
2. IF (n == 1) Devolver T
3. ELSE
  (a) S ← n div 2
  (b) MergeSort(i, S, T)
  (c) MergeSort(S + 1, j, T)
  (d) Mezclar(i, S, j, T) // Mezclar los dos subvectores

```

- Número de subproblemas en cada pasada: 2
- Tamaño de los subproblemas:  $n/2$
- Coste de mezclar subvectores ordenados:  $\Theta(n)$
- Ecuación de recurrencia :

$$T(n) = 2T(n/2) + n$$

- Resolvemos con método maestro:

- $a = 2$  ,  $b = 2$
- $f(n) = \Theta(n)$  : caso 2

$$T(n) \in \Theta(n \log n)$$

3.3-49

## Divide y Vencerás. Ejemplo 2

### Buscar el Máximo de un Array

Dado un array, buscar de forma recursiva el máximo valor:

- Caso base: Si el array es de tamaño 1 es inmediato
- Para  $n > 1$ ,
  - divide el array en dos subarrays de tamaño menor que  $n$
  - busca el máximo en cada parte de forma recursiva, y
  - devuelve el mayor de ambas partes.

```

BuscaMAX(left,right,T[n])
  S ← n div 2
  IF (left==right) Devolver (T[1])
  u ← BuscaMAX(left,S,T)
  v ← BuscaMAX(right+1,j,T)
  IF (u > v) Devolver u
  ELSE Devolver v
  
```

- $n$ : tamaño del problema
- Número de subproblemas en cada pasada: 2
- Tamaño de los subproblemas:  $n/2$
- Cada llamada recursiva tiene un coste constante:  $\Theta(1)$
- Ecuación de recurrencia :

$$T(n) = 2T(n/2) + 1$$

- Resolvemos con método maestro:
  - $a = 2$  ,  $b = 2$
  - $f(n) = \Theta(1)$  ,  $n^{\log_b a} = n^{\log_2 2} = n$
  - caso 1

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

3.3-50

## Divide y Vencerás. Ejemplo 3

### Búsqueda Binaria Recursiva

```

funcion BinREC(T[], first, last, x): int
  1. IF last < first Devolver(-1)
  2. mid ← (first + last)/2
  3. IF x ≤ T[mid] THEN
    Devolver( BinREC(T[], first, mid, x) )
  4. ELSE
    Devolver( BinREC(T[], mid + 1, last, x) )
  
```

- Es un caso particular en el que en cada llamada nos quedamos con **sólo 1 subproblema de tamaño  $n/2$**

- El tiempo de cada llamada es **constante**,  $f(n) = \Theta(1)$
- $a = 1$ ,  $b = 2$ ,
- $n^{\log_b a} = n^{\log_2 1} = 1$  : caso 2

$$T(n) \in \Theta(\log n)$$

3.3-51

### Ejemplo. Elevar un número a un Exponente

*Exponenciación. Solución Divide y Vencerás*

Calcular  $x = a^n$ .  $a, n$  enteros,  $n > 0$

- Se basa en que  $a^n = (a^{\frac{n}{2}})^2$  si  $n$  par
- Produce la recurrencia:

$$a^n = \begin{cases} a & \text{si } n = 1 \\ (a^{\frac{n}{2}})^2 & \text{si } n \text{ es par} \\ a \cdot a^{n-1} & \text{otro caso} \end{cases}$$

```

expoDyV ( $a, n$ )
  IF  $n = 0$     Devolver(1)
  IF  $n = 1$     Devolver( $a$ )
  IF  $n$  es par,
    Devolver [expoDyV( $a, \frac{n}{2}$ )]2
  ELSE
    Devolver  $a \cdot \text{expoDyV}(a, n - 1)$ 

```

Ejemplo:  $a^{29} = a \cdot a^{28} = a \cdot (a^{14})^2 = a \cdot ((a^7)^2)^2 = \dots$

- Asociamos la recurrencia:

$$T(n) = \begin{cases} 1 & \text{si } n = 1 \\ T(n/2) + 1 & \text{si } n \text{ es par} \\ T(n-1) + 1 & \text{otro caso} \end{cases}$$

- **NO podemos usar el método maestro** para resolver la recurrencia
- Podemos ver la solución de esta recurrencia en [Brassard 7.7] :
- Si las operaciones son elementales:  $T(n) \in \Theta(\log n)$
- Nota: el algoritmo clásico es  $\Theta(n)$  :

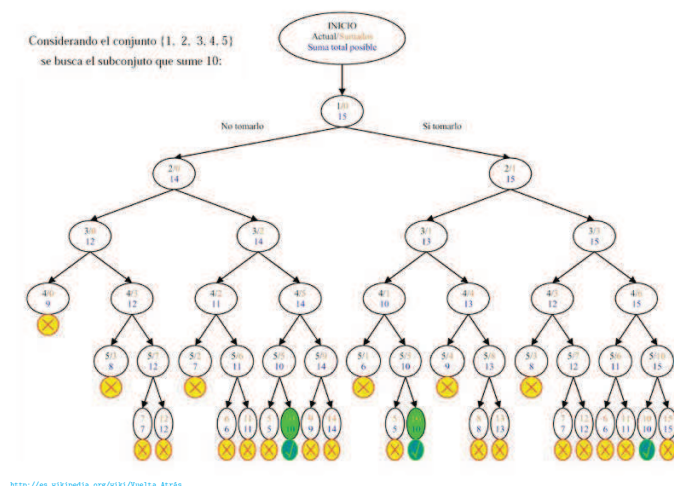
```

expoClasico ( $a, n$ )
   $valor = 1$ 
  FOR  $i = 1$  TO  $n$ 
     $valor = valor \times a$ 
  Devolver  $valor$ 

```

3.3-52

## Otras aplicaciones. Algoritmos de Vuelta Atrás



En el transcurso de la búsqueda si se encuentra un estado incorrecto (en amarillo), se retrocede hasta la decisión anterior, y se continua la búsqueda por los caminos que aún no han sido explorados, y que puedan conducir a una solución.

3.3-53

## Aplicaciones. Algoritmos de Búsqueda de soluciones en juegos

Ejemplo: Algoritmo de tres en raya

- Dada una posición del tablero se usa la estrategia “vuelta atrás” para elegir el mejor movimiento
- La rutina que elige el mejor movimiento realiza llamadas recursivas para evaluar todos los posibles movimientos
- Ver ejemplo en *Weiss 7.7*

3.3-54

## 7 Problemas de la Recursión

### Problemas de usar Recursión

- Demasiada recursión puede ser peligrosa
- Es una buena solución pero NO siempre es apropiada: nunca se debe usar como sustitución de un bucle simple.
- COSTE MEMORIA: hay que tener en cuenta el coste de memoria de pila que se requiere en cada llamada recursiva. Aunque un método recursivo sólo utilice variables escalares, puede requerir un espacio de  $\Theta(n)$  si realiza  $n$  llamadas recursivas.

### • Errores más comunes:

- No incluir el **caso base**
- Las llamadas recursivas siempre deben **progresar hacia un caso base**: en otro caso, la recursión no terminará.
- Evitar la **superposición** de llamadas recursivas o se pueden generar algoritmos de coste **exponencial**. (Ejemplo: alg. de Fibonacci)
- El tiempo empleado por las llamadas recursivas ha de calcularse usando **fórmulas recursivas**, no son llamadas con tiempo lineal.

3.3-55



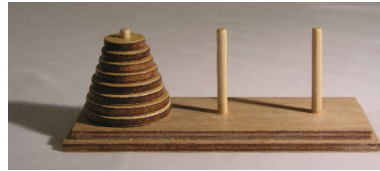
## Y para terminar, la leyenda de las Torres de Hanoi...

### Problema:

Se tienen tres varillas verticales. En una de las varillas se apilan  $n$  discos de tamaño creciente. No hay dos discos iguales.

El juego consiste en pasar todos los discos de la varilla ocupada a una de las otras, siguiendo unas reglas:

- Mover un disco cada vez
- El disco solo se puede poner sobre otro mayor



[http://en.wikipedia.org/wiki/Tower\\_of\\_Hanoi](http://en.wikipedia.org/wiki/Tower_of_Hanoi)

La leyenda dice que unos monjes tienen la tarea de resolver este problema con 64 discos. El mundo acabará el día que lo resuelvan...

### Cuánto tiempo nos queda???

3.3-56

### Solución recursiva del problema...

```

HANOI( $n$ , varillaA, varillaB, varillaC)
  IF( $n=1$ )
    Imprimir (pasar disco de A a B)
  ELSE
    HANOI( $n-1$ , A, C, B)
    Imprimir (pasar disco de A a B)
    HANOI( $n-1$ , C, B, A)
  
```

La **solución recursiva** de este problema, para  $n$  discos es una recurrencia del tipo:

$$T(n) = 2T(n-1) + 1 \quad \text{para } n > 1 \text{ y } T(1) = 1$$

Cuya solución es

$$T(n) = 2^n - 1$$

### Exponencial !!

3.3-57

### Solución recursiva del problema...

Si los monjes mueven un disco por segundo,

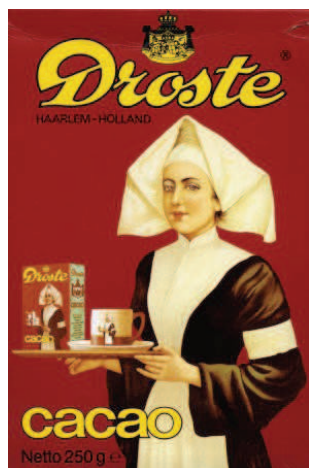
- les llevaría más de una semana resolver el problema con 20 discos
- más de 31 años finalizar un problema con 30 discos,
- más de 348 siglos si la torre tiene 40 discos, (asumiendo que no cometen errores)
- Y el problema de 64 discos les llevaría más de **5.8 billones de siglos**

(La solución iterativa también necesita  $2^n - 1$  movimientos).

<http://mazeworks.com/hanoi/index.htm>

3.3-58

## Imágenes con recursión..



Anuncio de cacao con una imagen recursiva. La mujer muestra un paquete idéntico al del propio anuncio, conteniendo así a otra mujer que muestra otro paquete más pequeño, de forma recursiva.

3.3-59

## Triangulo de Sierpinski

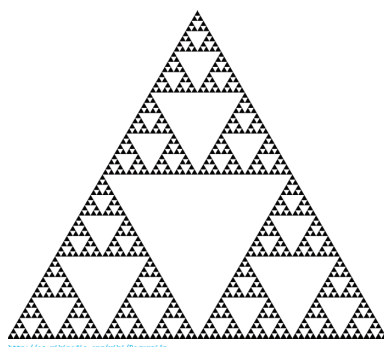


Imagen recursiva formada por un triángulo. Cada triángulo está compuesto de otros más pequeños, compuestos a su vez de la misma estructura recursiva.

3.3-60

# A Conclusiones

## Conclusiones

1. La recursión es una técnica básica para resolver problemas que pueden definirse **en función de sus propios términos** o en función de varios casos más sencillos.
2. Al definir un algoritmo recursivo es necesario **incluir siempre un caso base** que termine las llamadas recursivas.
3. Podemos demostrar la corrección de un algoritmo recursivo por medio de **Inducción Matemática**.
4. El **tiempo de ejecución** de los algoritmos recursivos se obtiene asociando **Ecuaciones de Recurrencia** que modelizan las llamadas recursivas.

5. El **Teorema maestro** permite resolver ecuaciones de recurrencia de la forma  $T(n) = aT(n/b) + f(n)$ .
6. Los **Árboles de Recursión** permiten analizar gráficamente las llamadas recursivas y su coste.
7. **Inconvenientes**: los algoritmos que usan recursión a veces no son eficientes. Hay que evitar repetir cálculos en las llamadas recursivas. Coste de memoria elevado.
8. **Aplicaciones**: Algoritmos “Divide y Vencerás”, “Vuelta Atrás”, recorrido de árboles y grafos, ...

3.3-61

## References

- [1] [Básica] Weiss M.A. *Estructuras de datos en Java*. (capítulo 7) Addison-Wesley, 2000 /2004
- [2] Brassard G., Bratley P. *Fundamentos de Algoritmos*. (capítulo 4) Prentice Hall, 1997 /2004
- [3] Sedgewick R. *Algorithms*. (capítulos 2 y 5) Addison-Wesley, 2002  
<http://introcs.cs.princeton.edu/23recursion/>

3.3-62