UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escola Tècnica Superior d'Enginyeria Informàtica etsinf

Tema 1. Recursión

Parte I

Programación (PRG)

Curso 2022/23

Departamento de Sistemas Informáticos y Computación

Índice y bibliografía

1. Introducción
2. Métodos recursivos
3. Recursividad y pila de llamadas
4. Etapas del diseño recursivo
5. Algunos ejemplos

- N. Prieto y otros. *Empezar a programar usando Java*. Tercera edición. Editorial UPV, 2016. Capítulo 10.
- M.A. Weiss. *Estructuras de datos en Java: compatible con Java 2*. Addison-Wesley, 2000. Capítulo 7.
- R. Sedgewick, K. Wayne. *Introduction to Programmig in Java. An Interdisciplinary Approach*. Pearson – Addison-Wesley 2009. Capítulo 2.3.
- Asignatura *Matemática Discreta* del Grado en Informática, ETSINF, UPV. Tema 1 (Métodos de demostración: inducción).

Introducción

- A grandes rasgos, se dice que un algoritmo es ***recursivo*** si viene definido o descrito en términos de la aplicación de él mismo a uno o más casos del problema "más simples" en alguna medida.
- Ejemplo. Función factorial de *n*, definida para $n \geq 0$.

$$n! = \prod_{k=1}^{n} k$$

es decir: $n! = n \cdot \underbrace{(n-1) \cdot \cdots \cdot 2 \cdot 1}_{(n-1)!}$ cuando $n \geq 1$

$n! = 1$ cuando $n = 0$

Se puede observar que en el caso de los $n \geq 1$ (para los que $n - 1 \geq 0$):

$$n! = n \cdot (n-1)!$$

Introducción

- En resumen, se tiene:

$$n! = \begin{cases} 1 & n = 0 \\ n \cdot (n-1)! & n > 0 \end{cases}$$

En efecto:

$0! = 1$ } Caso $n = 0$

$1! = 1 = 1 \cdot 0! = 1$
$2! = 2 \cdot 1 = 2 \cdot 1! = 2$
$3! = 3 \cdot 2 \cdot 1 = 3 \cdot 2! = 6$
$4! = 4 \cdot 3 \cdot 2 \cdot 1 = 4 \cdot 3! = 24$
$5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 5 \cdot 4! = 120$
...
en general, $n! = n \cdot (n-1)!$

} Caso $n > 0$

Introducción

- La traza anterior ilustra un hecho importante:

$$n! = \begin{cases} 1 & n = 0 \quad (1) \\ n \cdot (n-1)! & n > 0 \quad (2) \end{cases}$$

se puede considerar una definición *efectiva* de $n!$, es decir un método o algoritmo para calcular $n!$, pues:

  - Para $n = 0$, (1) dice directamente que $n!$ vale 1.
  - Para los $n > 0$, (2) dice cómo obtener el valor de $n!$, suponiendo que se ha calculado $(n-1)!$.

En el caso (2) se está haciendo la suposición o *hipótesis* de que a su vez se sabe calcular $(n-1)!$, y que la propia definición indica cómo hacerlo de manera efectiva. Hipótesis que es correcta pues basta con aplicar (2) un número finito de veces:

$$n! \leftarrow (n-1)! \leftarrow (n-2)! \leftarrow \ldots \leftarrow 4! \leftarrow 3! \leftarrow 2! \leftarrow 1! \leftarrow 0!$$

(… 24 6 2 1 1)

Aplicación de un **número finito** de veces de (2) desde el caso directo (1)

- El hecho de que el número de veces que hay disminuir un entero $n>0$ cualquiera para llegar a 0 sea finito, está en el fundamento de las *demostraciones por inducción sobre los naturales* (*MAD*, tema 1).

Introducción

- Con más precisión que en la definición inicial, se dice que un algoritmo es ***recursivo*** si viene dado por un par de casos complementarios y excluyentes para los datos del problema:
  - ***Caso directo o base***: Caso más simple, en alguna medida, para el que la resolución es directa.
  - ***Caso recursivo o general***: La resolución del problema viene expresada en términos de la resolución de uno o más **subproblemas del mismo tipo** y **más pequeños** en alguna medida (cantidad entera ≥0, para que el *razonamiento inductivo* subyacente sea correcto).

- Ejemplos:
  - Definición recursiva de $n!$, $n \geq 0$

$$n! = \begin{cases} 1 & n = 0 \\ n \cdot (n-1)! & n > 0 \end{cases}$$

  - Definición recursiva de *fibonacci(n)*, $n \geq 0$

$$fibonacci(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ fibonacci(n-1) + fibonacci(n-2) & n > 1 \end{cases}$$

Métodos recursivos

La resolución iterativa de *n*! se puede ver también bajo este punto de vista:

```
/** Calculo de n!. Precondicion: n >= 0 */
public static int factorial(intn) {
 int f = 1, i = 0;
    while (i < n) { i++;
        f = f * i;
    }
    return f;
}
```

núm. pasada	i	f
0	0	1
1	1	1
2	2	2
3	3	6
4	4	24
5	5	120
…	…	…
$n-1$	$n-1$	$(n-1)!$
n	n	$n!$

- Por inducción se demuestra que la siguiente relación entre las variables:

  `f = i!`

  es general, se mantiene sea cual sea el número de pasadas ejecutadas.

- Por una parte, ello implica que cuando se llega a **`i`** = *n*, en **`f`** se tiene el valor de *n*!
- También muestra que en el cálculo de *n*!, para $n > 0$, se ha usado el cálculo previamente resuelto de $(n-1)!$.

Métodos recursivos

- Java permite declarar *métodos recursivos*, entendiendo por tales aquellos en los que se cita al propio método que se está declarando:

Condicional que discrimina si a **n** se le ha de aplicar el caso base o el caso general.

```
/** Calculo de n!. Precondicion: n >= 0. */
public static int factorial(int n) {
    if (n == 0) { return 1; }
    else { return n * factorial(n - 1); }
}
```

- Caso base: $n = 0$

$$0\,! = 1$$

- Caso general: $n > 0$

$$n\,! = n \cdot (n-1)\,!$$

Recursividad y pila de llamadas

- El mecanismo que da soporte a las declaraciones de métodos recursivos es la pila de registros de activación.

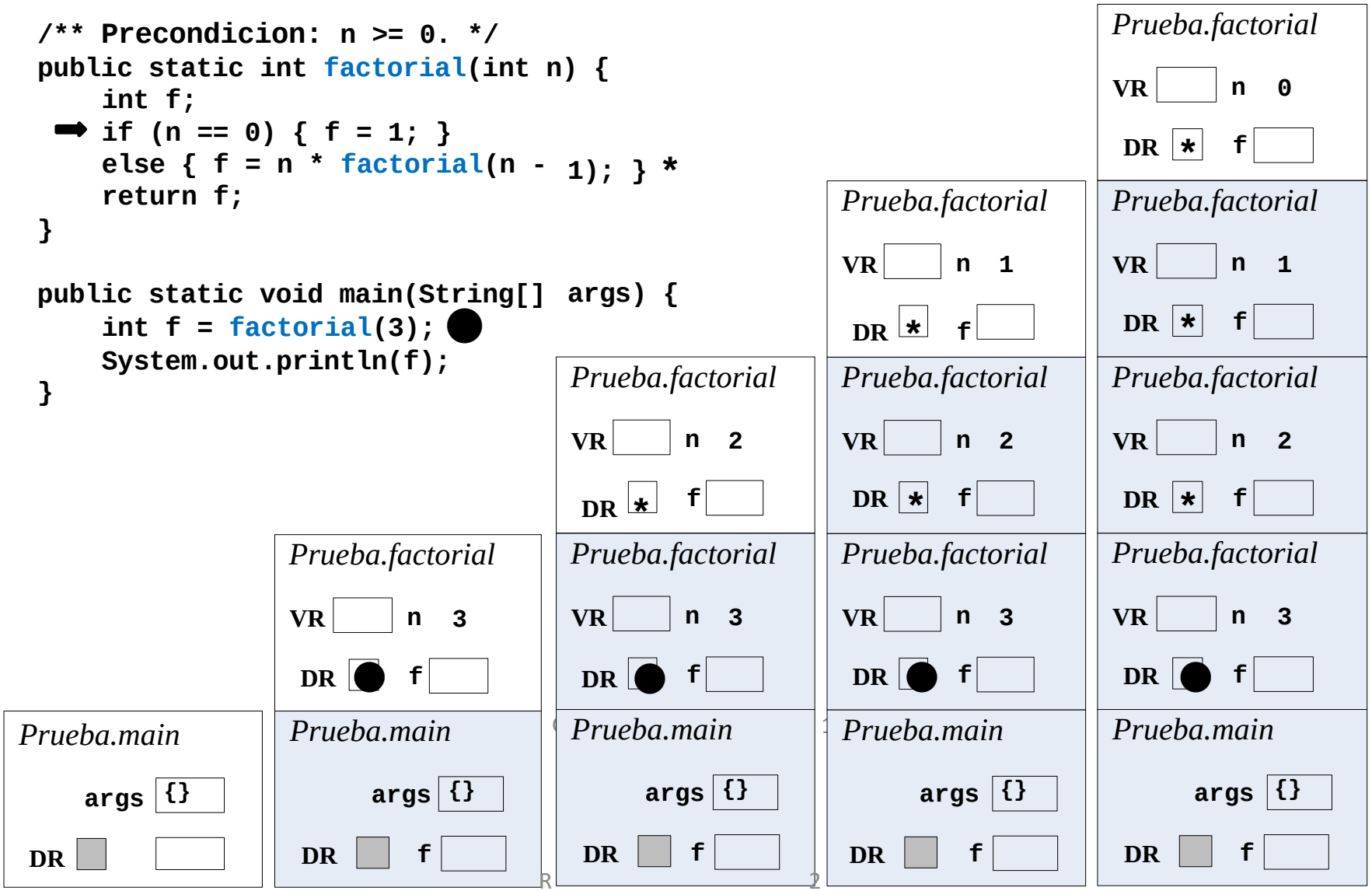
- Cada llamada recursiva al método tiene asociado el registro de activación para sus propios datos, pero se aplica el mismo método o algoritmo (el mismo código Java) a todos ellos; el código del método deberá distinguir si los datos se encuentran en el caso base o en el caso general.

Recursividad y pila de llamadas

Ejemplo. Programa **Prueba**, con los métodos:

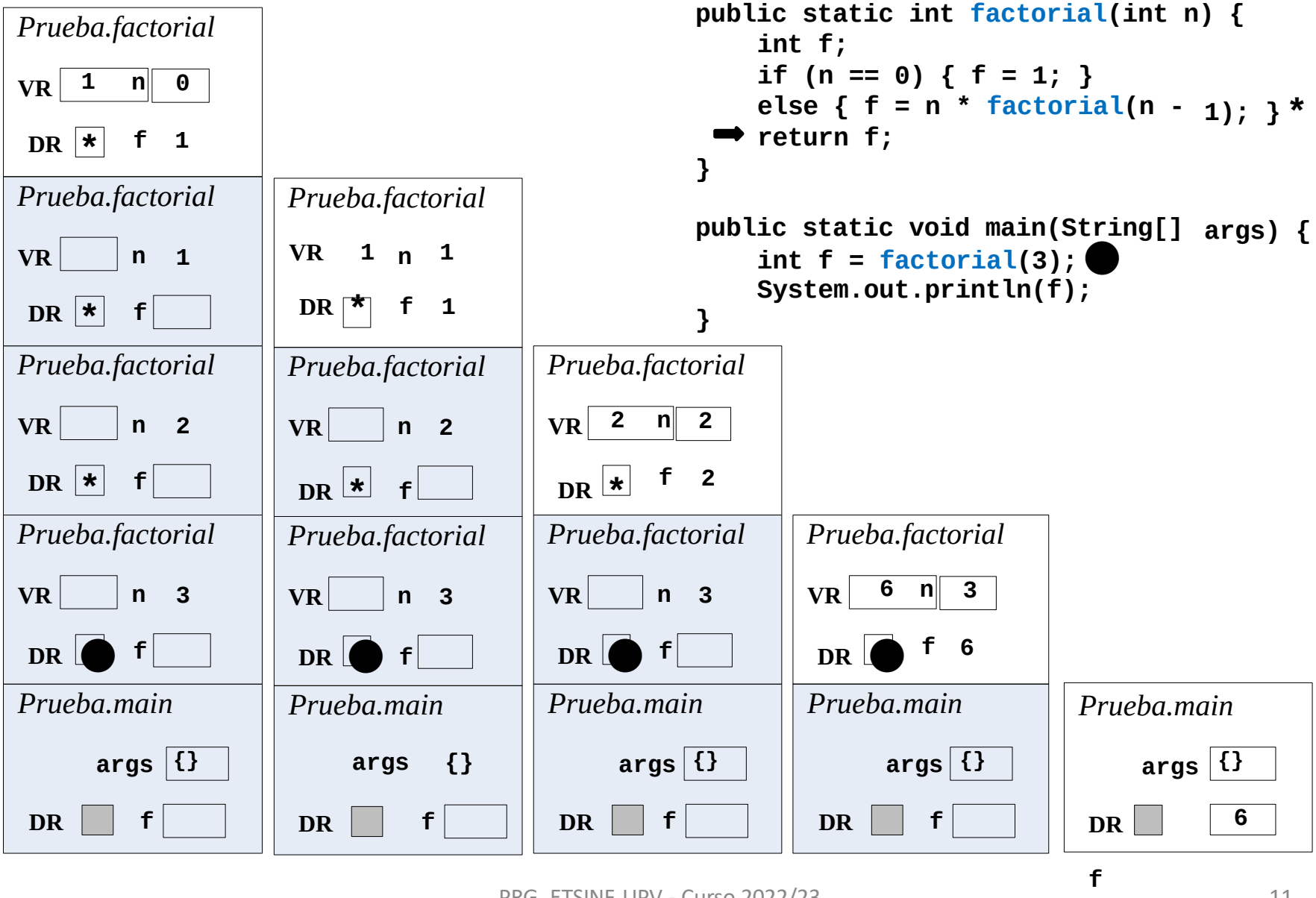
```
/** Precondicion: n >= 0. */
public static int factorial(int n) {
    int f;
    if (n == 0) { f = 1; }
    else { f = n * factorial(n - 1); } *
    return f;
}

public static void main(String[] args) {
    int f = factorial(3);
    System.out.println(f);
}
```



f

Recursividad y pila de llamadas




```
/** Precondicion: n >= 0. */
public static int factorial(int n) {
    int f;
    if (n == 0) { f = 1; }
    else { f = n * factorial(n - 1); } *
 ➡  return f;
}

public static void main(String[] args) {
    int f = factorial(3); ●
    System.out.println(f);
}
```

Recursividad y pila de llamadas

- La siguiente traza resume la evolución de los parámetros en las sucesivas llamadas del ejemplo, así como el valor que se calcula en los retornos del método.

n	**f (VR)**	Relación entre **n**, **f**	
3	6 ×3	3 · 2 = 6 = 3!	
2	2 ×2	2 · 1 = 2 = 2!	**n>0**
1	1 ×1	1 · 1 = 1 = 1!	
0	1	1 = 0!	**n=0**

```
/** Calculo de n!. Precondicion: n >= 0. */
public static int factorial(int n) {
    int f;
    if (n == 0) { f = 1; }
    else { f = n * factorial(n - 1); }
    return f;
}                              (n-1)!

public static void main(String[] args){
    int f = factorial(3);
    System.out.println(f);
}
```

Llamada *inicial* al método

Etapas del diseño recursivo

1. ***Enunciado del problema.*** Perfil del método y precondición de los datos.
2. ***Análisis por casos.*** Determinación de los siguientes casos complementarios y excluyentes:
   - Caso base o directo: Caso más *simple* o *pequeño* en alguna medida de los datos, con solución directa.
   - Caso general o recursivo: Reducción del problema a uno o más subproblemas
     a) del mismo tipo,
     b) y más pequeños en alguna medida.
3. ***Validación del razonamiento recursivo***:
   a) Asegurarse de que los datos que se le pasen a los subproblemas continúan cumpliendo la precondición.
   b) Asegurarse de que los nuevos datos son más pequeños en alguna medida (cantidad entera, positiva, acotada inferiormente).

- En la transcripción al lenguaje Java, una instrucción **`if ... else`** deberá discriminar si el valor de los parámetros del método se encuentran en el caso base o en el caso general.

Algunos ejemplos – Potencia *n*-ésima

1. ***Enunciado del problema.*** Dados unos enteros a y $n \geq 0$, calcular la potencia a^n.

```
/** Calculo de a elevado a n. Precondicion: n >= 0. */
public static int potencia(int a, int n)
```

2. ***Análisis de casos.***
   - Caso base: Si $n = 0$, entonces $a^n = 1$.
   - Caso general: Si $n > 0$, entonces $a^n = \underbrace{a \cdot \ldots \cdot a}_{n \text{ veces}} = a^{n-1} \cdot a$.

   Son casos complementarios y excluyentes.

Escrito en Java:

```
/** Calculo de a elevado a n. Precondicion: n >= 0. */
public static int potencia(int a, int n) {
    if (n == 0) { return 1; }
    else { return potencia(a, n - 1) * a; }
}
```

3. ***Validación del diseño.*** En el caso $n > 0$:
   - A la llamada recursiva se le continúa pasando un exponente $n - 1 \geq 0$
   - El nuevo exponente, de valor $n - 1$, cumple $n > n - 1 \geq 0$, es decir, tiene un valor entero más pequeño que el anterior, sin hacerse menor que 0.

Algunos ejemplos – Potencia *n*-ésima

- La siguiente traza ejemplo de evolución de los parámetros en las sucesivas llamadas, ilustra el comportamiento del método y la reducción recursiva que sigue.
- Sean 3 y 4 los valores respectivos de **a** y **n** en la llamada inicial:

a	**n**	**VR**	Relación entre **a**, **n** y la potencia	
3	**4**	81 ×3	$81 = 27 \cdot 3 = 3^4$	**n**>0
3	**3**	27 ×3	$27 = 9 \cdot 3 = 3^3$	
3	**2**	9 ×3	$9 = 3 \cdot 3 = 3^2$	
3	**1**	3 ×3	$3 = 1 \cdot 3 = 3^1$	
3	**0**	1	$1 = 3^0$	**n=0**

```
/** Calculo de a elevado a n. Precondicion:  n>=0. */
public static int potencia(int a, int n) {
    if (n == 0){ return 1; }
    else { return potencia(a, n - 1) * a; }
}
```

a elevado a (n-1)

Algunos ejemplos – Número de cifras

1. ***Enunciado del problema.*** Dado un entero $n \geq 0$, calcular el número de cifras de n.

```
/** Calculo del numero de cifras de n. Precondicion: n >= 0. */
public static int numCifras(int n)
```

2. ***Análisis de casos.***
   - Caso base: Si $n < 10$, entonces el número de cifras es 1.
   - Caso general: Si $n \geq 10$, entonces n / 10 tiene una cifra menos. Si se calcula el número de cifras de n / 10, sólo falta sumarle 1.

   Son casos complementarios y excluyentes.

Escrito en Java:

```
/** Calculo del numero de cifras de n.
 *  Precondicion: n >= 0. */
public static int numCifras(int n) {
    if (n < 10) { return 1; }
    else { return numCifras(n / 10) + 1; }
}
```

3. ***Validación del diseño.*** En el caso $n \geq 10$:
   - A la llamada recursiva se le continúa pasando un entero ≥ 0.
   - El nuevo valor n / 10 del parámetro cumple $n > n / 10 \geq 0$, es decir, tiene una cifra menos, es un valor entero más pequeño, sin hacerse menor que 0.

Algunos ejemplos – Número de cifras

- Traza ejemplo de evolución de los parámetros en las sucesivas llamadas.
  Sea 23850 el valor de **n** en la llamada inicial:

n	VR	
23850	5 + 1	**n**≥10
2385	4 + 1	
238	3 + 1	
23	2 + 1	
2	1	**n**<10

```
/** Calculo del numero de cifras de n.
 *  Precondicion: n >= 0.
 */
public static int numCifras(int n) {
    if (n < 10) { return 1; }
    else { return numCifras(n / 10) + 1; }
}
```

Algunos ejemplos – Cociente de la división entera

1. ***Enunciado del problema.*** Dados unos enteros $a \geq 0$, $b > 0$ calcular a / b.

```
/** Calculo de a / b. Precondicion: a >= 0, b > 0. */
public static int cociente(int a, int b)
```

2. ***Análisis de casos.***
   - Caso base: Si $a < b$, entonces a / b *es 0*
   - Caso general: Si $a \geq b$, entonces $a / b = (a - b) / b + 1$.

   Son casos complementarios y excluyentes.

   Escrito en Java:

3. ***Validación del diseño.*** En el caso $a \geq b$:
   - A la llamada recursiva se le continúa pasando un dividendo ≥ 0.
   - El nuevo valor $a - b$ es un valor entero más pequeño que a, sin llegar a ser menor que 0.

Algunos ejemplos – Cociente de la división entera

- La siguiente traza ejemplo de evolución de los parámetros en las sucesivas llamadas, ilustra el comportamiento del método y la reducción recursiva que sigue.
- Sean 23 y 5 los valores respectivos de **a** y **b** en la llamada inicial:

a	**b**	**VR**	Relación entre **a**, **b** y el cociente	
23	5	4	23 = **4** ·5+3	**a ≥ b**
18	5	3 (+1)	18 = **3** ·5+3	
13	5	2 (+1)	13 = **2** ·5+3	
8	5	1 (+1)	8 = **1** ·5+3	
3	5	0 (+1)	3 = **0** ·5+3	**a < b**

```
/** Calculo de a / b.
 *  Precondicion: a >= 0, b > 0.
 */
public static int cociente(int a, int b) {
    if (a < b) { return 0; }
    else { return cociente(a - b, b) + 1; }
}
```

(a-b)/b

Algunos ejemplos – mcd

- Ejemplo: Considerar el problema de determinar el ***máximo común divisor*** (mcd) de dos números enteros, *A*, *B* mayores que 0.

  El algoritmo, debido a Euclides (300 a.C.), es uno de los más antiguos conocidos.

- Algoritmo MCD. Sean *A* y *B* los valores iniciales de unas variables **a** y **b**, entonces :

  Mientras **a** y **b** sean diferentes, cambiar el mayor de los dos por la diferencia del mayor menos el menor. Cuando tengan el mismo valor, dicho valor es el mcd de *A* y *B*.

Algunos ejemplos – mcd

Traza ejemplo con valor inicial de las variables
$\mathbf{a} = 247, \mathbf{b} = 152$

- El mcd de **a** y **b** se mantiene constante;
- cuando $\mathbf{a} = \mathbf{b} = 19$ su mcd es 19.

$\Downarrow$

El mcd de 247 y 152 es 19.

Estado de las variables			
Núm. de pasadas ejecutadas	**a**	**b**	
0 (inicio)	247	152	**a ≠ b**
1	95	152	
2	95	57	
3	38	57	
4	38	19	
5	19	19	**a = b**

Método Java iterativo

```
/** Calculo del mcd de a y b.
 *  Precondicion: a > 0, b > 0. */
public static int mcd(int a, int b) {
    while ( a != b ) {
        if (a > b) { a = a - b; }
        else { b = b - a; }
    }
    return a;
}
```

Algunos ejemplos – mcd

- El algoritmo de Euclides anterior se basa en la siguiente propiedad:

$$\left.\begin{array}{ll} MCD(a,b) = MCD(a-b,b) & \text{cuando } a > b, \\ MCD(a,b) = MCD(a,b-a) & \text{cuando } b > a \end{array}\right\} \; a \neq b$$

- Cuando finalmente $a = b$, $MCD(A,B) = MDC(a,b) = a = b$

que dan el caso general y caso base de la versión recursiva.

Escrito en Java:

```
/** Calculo del mcd de a y b.
 *  Precondicion: a > 0, b > 0. */
public static int mdc(int a, int b) {
    if (a > b) { return mcd(a - b, b); }
    else if (a < b) { return mcd(a, b - a); }
    else { return a; }
}
```

Algunos ejemplos – mcd

- Validación del razonamiento recursivo:
  - *a*, *b* se mantienen >0 en todas las llamadas.
  - En el caso general $a \neq b$, el mayor de los dos parámetros disminuye, por tanto disminuye también el número de veces que el nuevo máximo contiene al mcd.

- Traza ejemplo:

a	**b**	**VR**	Relación entre **a**, **b** y el mcd	
247	152	19	247 = **13·19**, 152 = 8·**19**	
95	**152**	19	95 = 5·19, 152 = **8·19**	
95	57	19	95 = **5·19**, 57 = 3·19	**a ≠ b**
38	**57**	19	38 = 2·19, 57 = **3·19**	
38	19	19	38 = **2·19**, 19 = 1·19	
19	**19**	19	**19 = 1 · 19**, **19 = 1 · 19**	**a = b**

Algunos ejemplos – Dibujo de triángulo

1. ***Enunciado del problema.*** Dado un entero $n > 0$, escribir en la salida estándar n líneas de asteriscos, de longitudes sucesivas n, $n-1$, $n-2$, ..., 1.

```
/** Dibuja un triangulo invertido de n lineas. Precondicion: n > 0. */
public static void dibujaTri(int n)
```

Ejemplo de la salida para $n = 4$:

```
****
***
**
*
```

2. ***Análisis de casos.***
   - Caso base: $n = 1$, se escribe '*'
   - Caso general: Si $n > 1$, se escribe primero una línea y después un triángulo "más pequeño".

```
/** Precondicion: n >= 1. */
public static void dibujaTri(int n){
    if (n == 1) { System.out.println('*'); }
    else {
        for(int i = 1; i <= n; i++) {
            System.out.print('*');
        }
        System.out.println();
        dibujaTri(n - 1);
    }
}
```

3. ***Validación del diseño.*** En el caso $n > 1$:
   - A la llamada recursiva se le continúa pasando un número de líneas $n - 1 \geq 1$ (cumple la precondición).
   - El nuevo parámetro es un valor entero $n - 1 < n$ (decreciente, y acotado inferiormente).