UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escola Tècnica Superior d'Enginyeria Informàtica etsinf

# Tema 1. Recursión

Programación (PRG)

Departamento de Sistemas Informáticos y Computación

# Contenidos

1. Introducción
2. Diseño de un método recursivo
3. Tipos de recursión
4. Recursividad y pila de llamadas
5. Algunos ejemplos
6. Recursión con arrays: recorrido y búsqueda
   - Esquemas recursivos de recorrido
   - Esquemas recursivos de búsqueda
   - Búsqueda binaria recursiva
7. Recursión versus iteración

# Introducción

- Se denomina recursivo a cualquier ente (definición, proceso, estructura, etc.) que se define en función de sí mismo.
- Una función es recursiva si su resolución requiere la solución previa de la función para un caso más sencillo. Por ejemplo, la función factorial:

$$\text{factorial}(x) = \begin{cases} 1 & \text{si } x = 0 \\ x * \text{factorial}(x-1) & \text{en otro caso} \end{cases}$$

- Un algoritmo es recursivo si obtiene la solución de un problema en base a los resultados que él mismo proporciona para casos más sencillos del mismo problema, esto es, si se invoca a sí mismo ...
- Así, para resolver un problema complejo mediante un algoritmo recursivo se descompone el problema en una serie de problemas más simples que se resuelven empleando el mismo algoritmo, para luego componer la solución del problema complejo en base a las soluciones de los problemas más simples.

# Introducción

- Hay dos situaciones distintas a la hora de resolver el problema planteado:
  - Caso base: el problema es lo suficientemente simple para ser resuelto de manera trivial.
  - Caso general: el problema requiere el uso de la resolución de una instancia más simple para hallar su solución.

- Una vez planteado un algoritmo recursivo es necesario verificar:
  1. La terminación del algoritmo, es decir, que en algún momento se llegará al caso base a partir de cualquier caso general planteado inicialmente.
  2. La corrección del algoritmo, es decir, que para cualquier subproblema que se dé, la solución del algoritmo es correcta (suele requerir una demostración matemática por inducción sobre algún parámetro del algoritmo).

# Diseño de un método recursivo

- Los algoritmos recursivos se implementan fácilmente usando métodos recursivos.

```
/** n>=0 */
public static int factorial(int n) {
  if (n==0)       // Condición del caso base
     return 1;    // Instrucciones del caso base
  else /* n>0 */                  // Condición del caso general
     return n * factorial(n-1); // Instrucciones del caso general
}
```

- La estructura fundamental consiste en tener una instrucción condicional que permita distinguir entre caso base y general.
- Las llamadas recursivas deben efectuarse cada vez para casos estrictamente más simples.

# Diseño de un método recursivo - Etapas

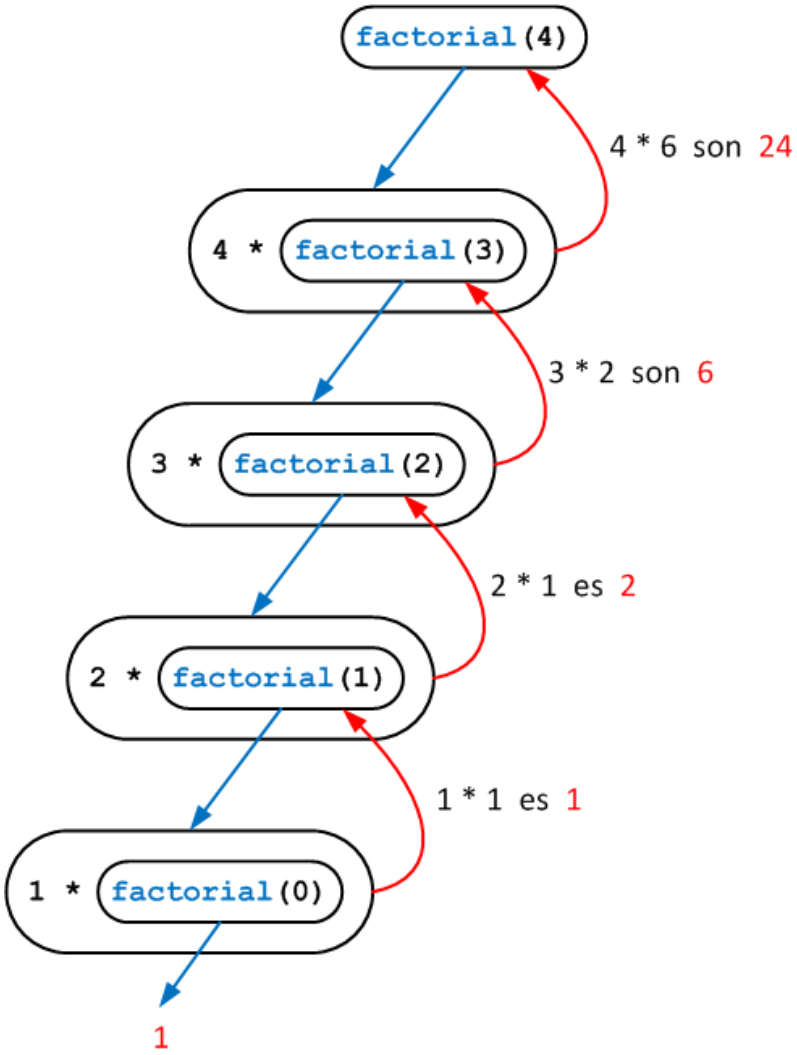
1. Enunciado del problema. Consiste en declarar la cabecera del método que se va a construir, y establecer tanto las condiciones de entrada para las que deberá ejecutarse el algoritmo, como el resultado esperado de dicha ejecución.
2. Análisis de casos. Consiste en hacer explícito el caso base y el caso general de la recursión, estableciendo para cada caso las instrucciones pertinentes para resolver el problema. Hay que comprobar que se cubre cualquier caso posible, esto es, que ante cualquier posible entrada del problema, se efectúa el caso base o el general.
3. Transcripción del algoritmo a un método en Java.
4. Validación del diseño. Determinar que en cada nueva llamada recursiva se cumple que:
   - El nuevo problema que se debe resolver es estrictamente más cercano al caso base que el problema original, y
   - los datos de entrada de la nueva llamada cumplen las condiciones de entrada en las que se ha establecido que se deberá ejecutar el algoritmo.

# Tipos de recursión

- Recursión Lineal: hay a lo sumo una sola llamada recursiva en cada ejecución del algoritmo, generando una secuencia de llamadas.
  - Final: el resultado de la llamada recursiva es el propio resultado de la llamada actual; es decir, la solución del caso más simple es a su vez la del caso más complejo, con lo que no hay combinación de resultados.
  - No final: el resultado de la llamada recursiva se emplea para calcular el resultado de la llamada actual, arrojando un resultado posiblemente distinto al de la llamada recursiva.
- Recursión Múltiple: hay más de una llamada recursiva en cada ejecución del algoritmo, y sus resultados han de combinarse para obtener la solución del caso actual. Se genera un árbol de llamadas.

# Tipos de recursión

- Por ejemplo, el método `factorial` es un método lineal (se hace una única llamada recursiva a `factorial(n-1)` en cada ejecución) no final (el resultado de la llamada se multiplica por n para devolverse).
- Para `factorial(4)` se genera la secuencia de llamadas de la figura.

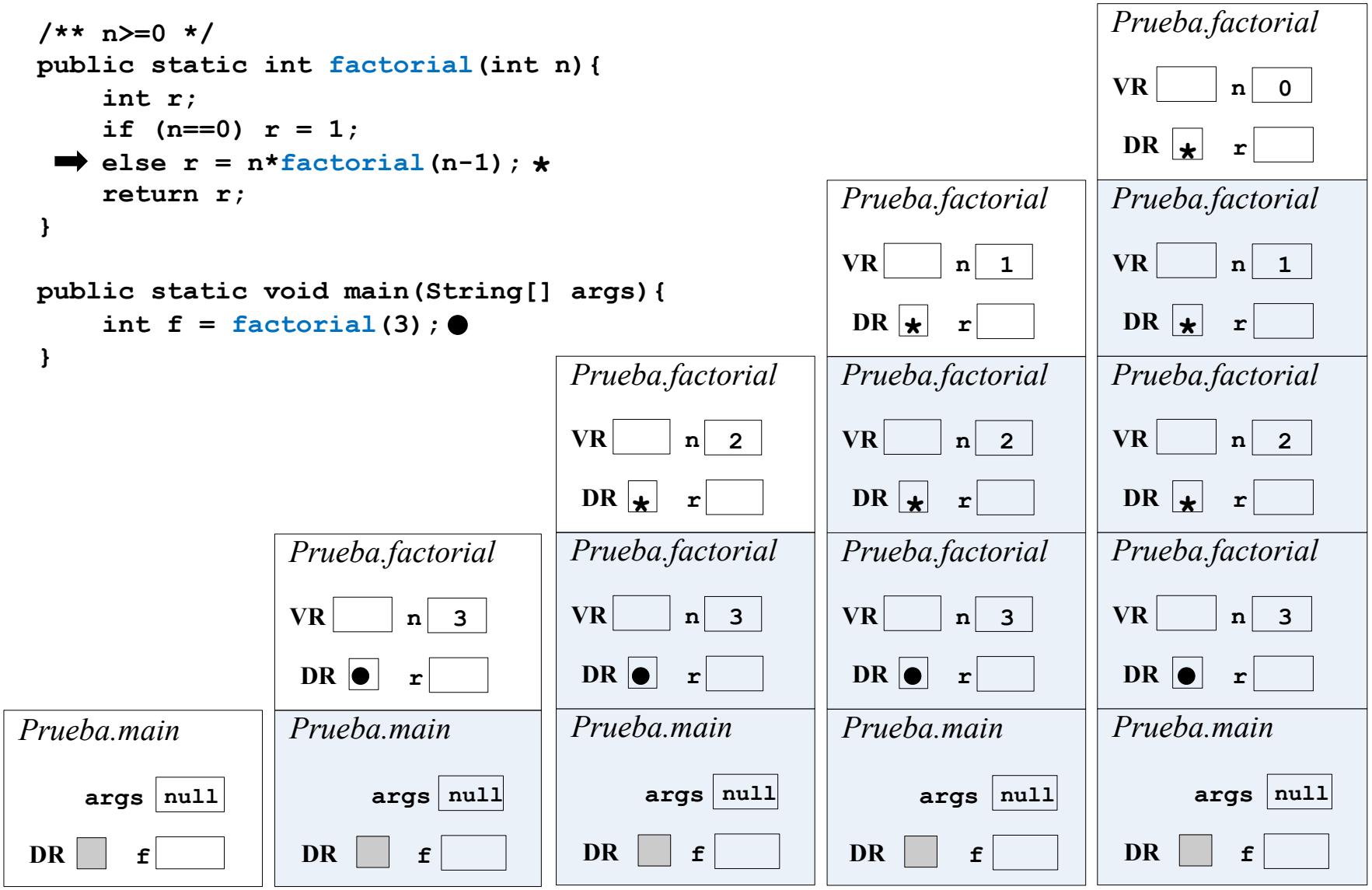


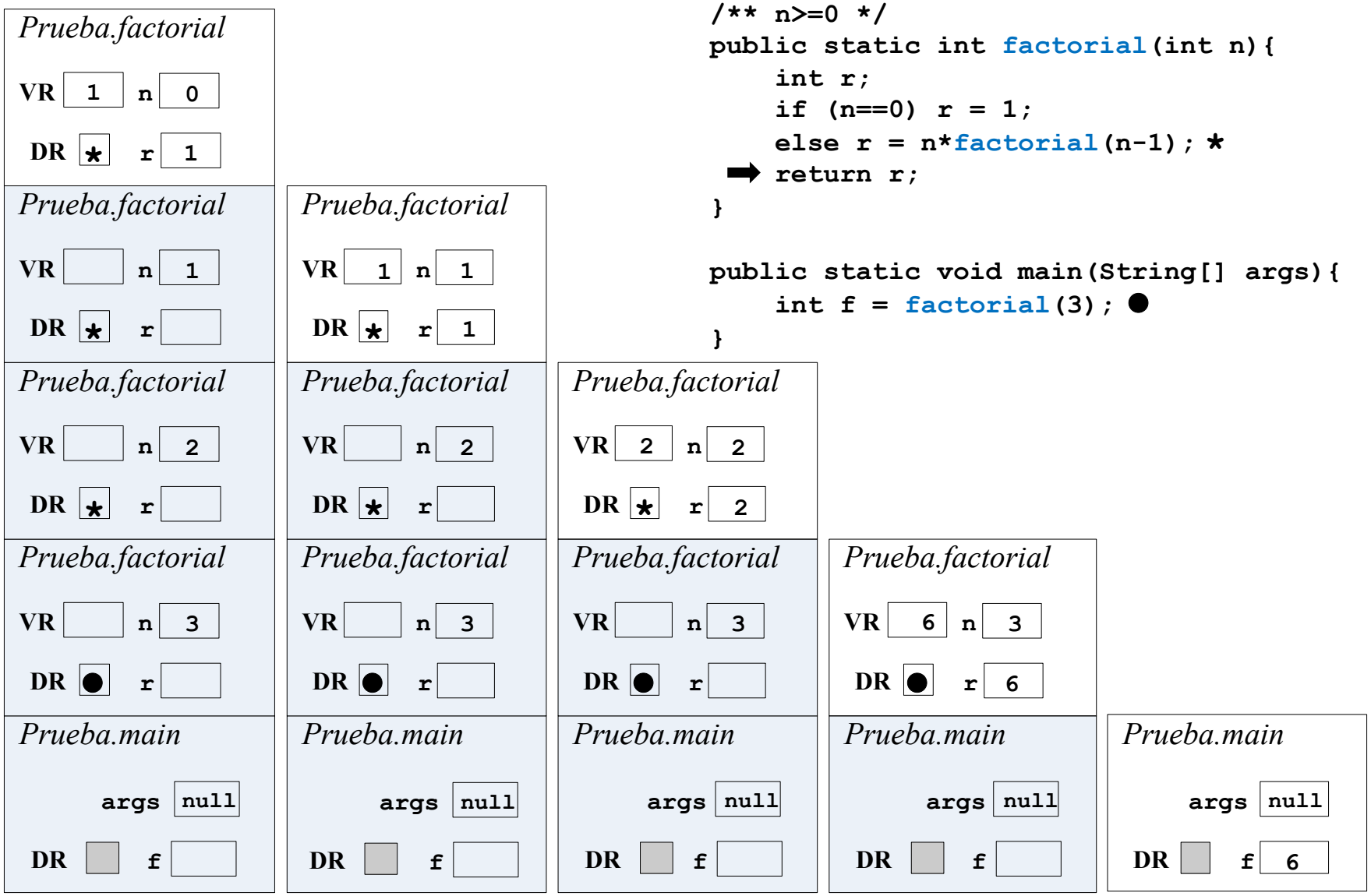
# Recursividad y pila de llamadas

- Cada llamada recursiva a un mismo método está asociada a un registro de activación propio que se apila en la pila de llamadas. Es decir, existen tantos registros de activación como llamadas pendientes. De todos ellos, sólo está activo el que está en el tope de la pila.
- Cuando una ejecución de un método finaliza, deja de existir su registro de activación (se desapila). En el caso recursivo, la ejecución puede reanudarse en una ejecución inmediatamente anterior del mismo método, que habrá dejado de estar pendiente.
- En la recursión se hace un uso intensivo de la pila de llamadas.
- Puede llegar a provocar serios problemas por agotamiento de la memoria, produciéndose un desbordamiento de la pila (stack overflow).
- La causa habitual del desbordamiento de la pila es la recursión infinita, provocando la excepción `StackOverflowError`.

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

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






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

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

# Recursividad y pila de llamadas

- Si se compara el uso de la pila que provoca la llamada `factorial(n)` en la versión iterativa y recursiva del método, se puede concluir que dicho uso es mayor en el caso recursivo que en el iterativo.
- En la versión iterativa de `factorial` coexisten simultáneamente en memoria, como mucho, el registro de activación del método `factorial` y el registro de activación del método `main`.
- En la pila de llamadas de la versión recursiva de `factorial` pueden llegar a coexistir simultáneamente n+1 registros de activación de las distintas llamadas a `factorial` más el registro de activación del método `main`.
- Es decir, el consumo de memoria del método `factorial` iterativo es siempre el mismo mientras que el del `factorial` recursivo depende del valor de n.

# Algunos ejemplos – Potencia n-ésima

1. Enunciado del problema. Dado un número natural $n \geq 0$ y un número real $a \neq 0$, calcular la potencia $a^n$.

```
/** n>=0 y a!=0 */
public static double potencia(double a, int n)
```

2. Análisis de casos.
   - Caso base: Si $n=0$, entonces $a^n = 1$.
   - Caso general: Si $n>0$, entonces $a^n = a^{n-1} * a$.
3. Transcripción del algoritmo a un método en Java.

```
/** n>=0 y a!=0 */
public static double potencia(double a, int n) {
        if (n==0) return 1;
        else return potencia(a,n-1)*a;
}
```

recursión lineal no final

4. Validación del diseño.
   - En el caso general, en cada llamada el valor del segundo parámetro decrece en 1; así, en algún momento llegará a ser 0, alcanzando el caso base y finalizando el algoritmo.
   - Además, siempre se cumple que el valor de $n \geq 0$ y $a \neq 0$.

# Algunos ejemplos – Resto de la división entera

1. Enunciado del problema. Dados dos números naturales a≥0 y b>0, calcular el resto de su división entera a/b.

```
/** a>=0 y b>0 */
public static int resto(int a, int b)
```

2. Análisis de casos.
   - Caso base: Si a<b, entonces el resto de a/b es a.
   - Caso general: En otro caso, el resto de a/b es el resto de (a-b)/b.
3. Transcripción del algoritmo a un método en Java.

```
/** a>=0 y b>0 */
public static int resto(int a, int b) {
        if (a<b) return a;
        else return resto(a-b,b);
}
```

recursión lineal final

4. Validación del diseño.
   - En el caso general, en cada llamada el valor del primer parámetro va decreciendo; en algún momento será inferior al segundo parámetro, alcanzando el caso base y finalizando el algoritmo.
   - Además, siempre se cumple que a≥0 y b>0.

# Algunos ejemplos – Algoritmo de Euclides I

1. Enunciado del problema. Dados dos números naturales a>0 y b>0, calcular su máximo común divisor (m.c.d.) siguiendo el algoritmo de Euclides.

```
/** a>0 y b>0 */
public static int mcd(int a, int b)
```

2. Análisis de casos.
   - Caso base: Si a=b, el m.c.d. es b.
   - Caso general: Si a>b, el m.c.d. es el m.c.d. de a-b y b.
     Si a<b, el m.c.d. es el m.c.d. de a y b-a.
3. Transcripción del algoritmo a un método en Java.

```
/** a>0 y b>0 */
public static int mcd(int a, int b) {
        if (a==b) return b;
        else if (a>b) return mcd(a-b,b);
             else return mcd(a,b-a);
}
```

recursión lineal final

4. Validación del diseño.
   - En el caso general, en cada llamada el valor del primer o segundo parámetro va decreciendo; en algún momento llegarán a ser iguales y, en ese caso, se alcanzará el caso base, finalizando el algoritmo.
   - Además, siempre se cumple que a>0 y b>0.

# Algunos ejemplos – Algoritmo de Euclides II

1. Enunciado del problema. Dados dos números naturales a>0 y b>0, calcular su máximo común divisor (m.c.d.) siguiendo el algoritmo de Euclides.

```
/** a>0 y b>0 */
public static int euclides(int a, int b)
```

2. Análisis de casos.
   - Caso base: Si el resto de a/b es 0, el m.c.d. es b.
   - Caso general: En otro caso, el m.c.d. es el m.c.d. de b y el resto de a/b.
3. Transcripción del algoritmo a un método en Java.

```
/** a>0 y b>0 */
public static int euclides(int a, int b) {
        if (a%b==0) return b;
        else return euclides(b,a%b);
}
```

recursión lineal final

4. Validación del diseño.
   - En el caso general, en cada llamada el valor del segundo parámetro va decreciendo; en algún momento llegará a ser el m.c.d. de los valores originales y, en ese caso, se alcanzará el caso base, finalizando el algoritmo.
   - Además, siempre se cumple que a>0 y b>0.

# Algunos ejemplos – Sucesión de Fibonacci

1. Enunciado del problema. Calcular el término n-ésimo de la sucesión de Fibonacci (asumiendo que el término 0-ésimo es el primero de la sucesión).

```
/** n>=0 */
public static int fibonacci(int n)
```

2. Análisis de casos.
   - Caso base: Si n≤1, el valor del término n-ésimo es n.
   - Caso general: En otro caso, es la suma de los términos (n-1)-ésimo y (n-2)-ésimo .
3. Transcripción del algoritmo a un método en Java.

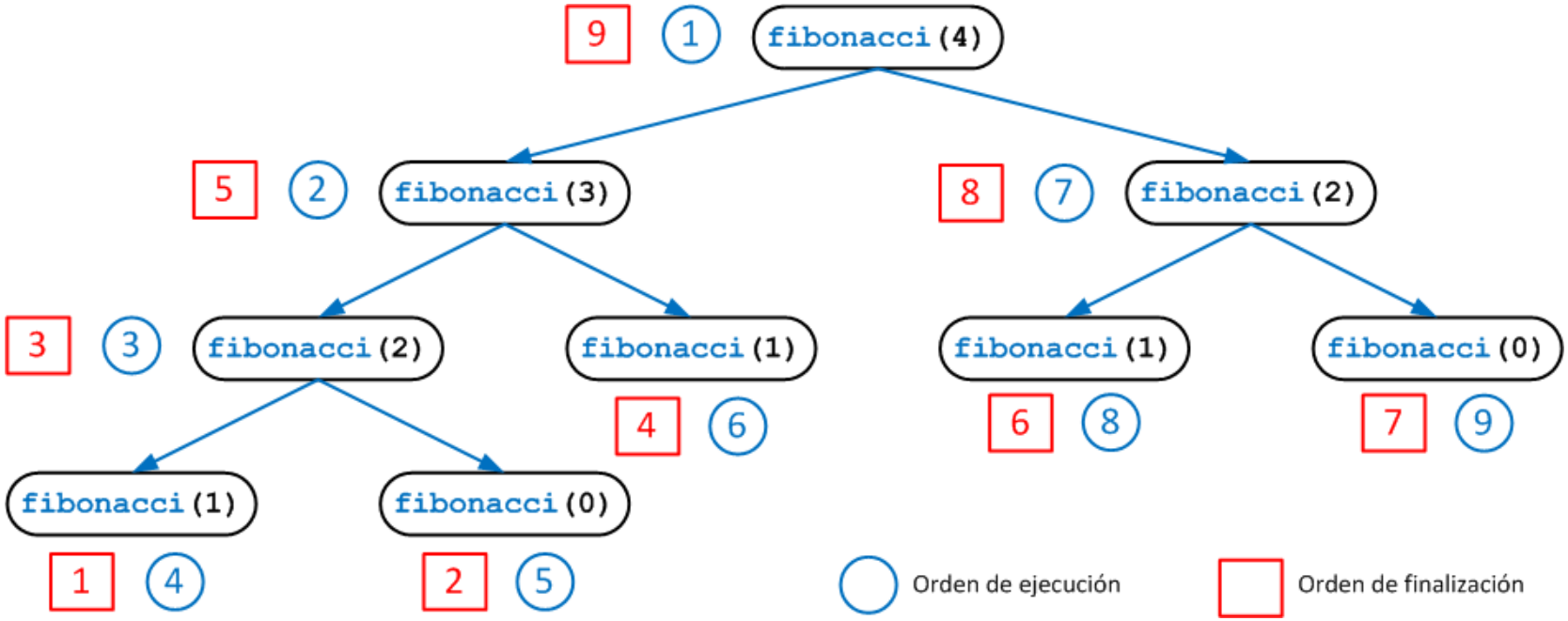
```
/** n>=0 */
public static int fibonacci(int n) {
   if (n<=1) return n;
   else return fibonacci(n-1) + fibonacci(n-2);
}
```

recursión múltiple

4. Validación del diseño.
   - En el caso general, en cada llamada se va tendiendo a términos inferiores de la sucesión; en algún momento se cumplirá la condición del caso base, finalizando el algoritmo.
   - Además, siempre se cumple que n>=0.

# Algunos ejemplos – Sucesión de Fibonacci

- El método `fibonacci` es un método recursivo múltiple (en el caso general se realizan dos llamadas recursivas y el resultado se construye a partir de la suma de los resultados de las dos llamadas).
- Para `fibonacci(4)` se genera el árbol de llamadas de la figura.



# Recursión con arrays: recorrido y búsqueda

- Dada la declaración de un array `a` de `num` elementos de tipo `tipoBase`:

  ```
  tipoBase[] a = new tipoBase[num];
  ```

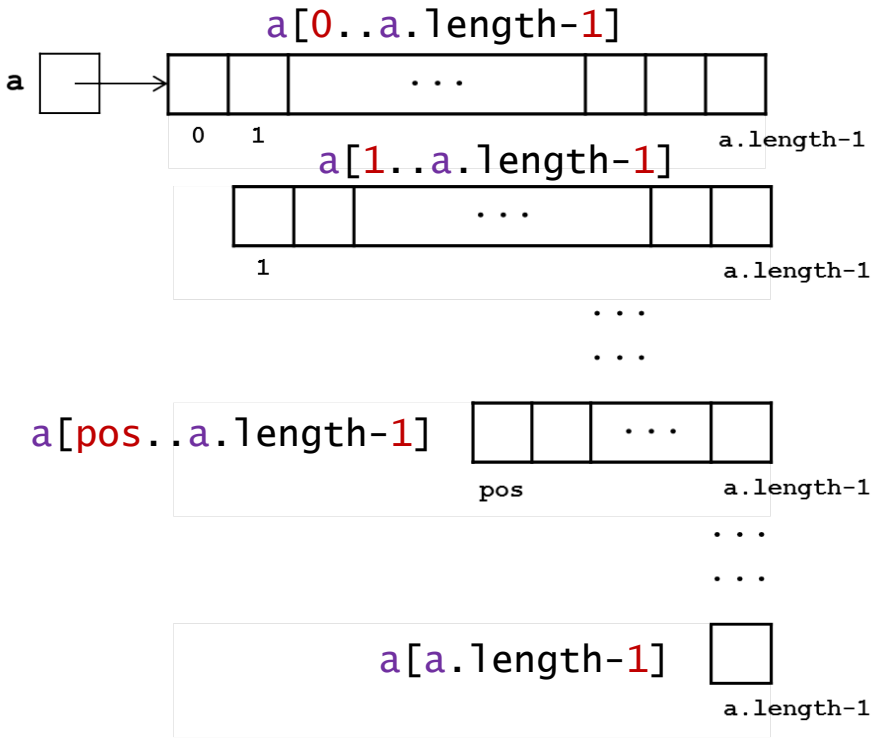
  se puede considerar el array `a` como una secuencia:

  `a[0], a[1], a[2], …, a[a.length-2], a[a.length-1]`

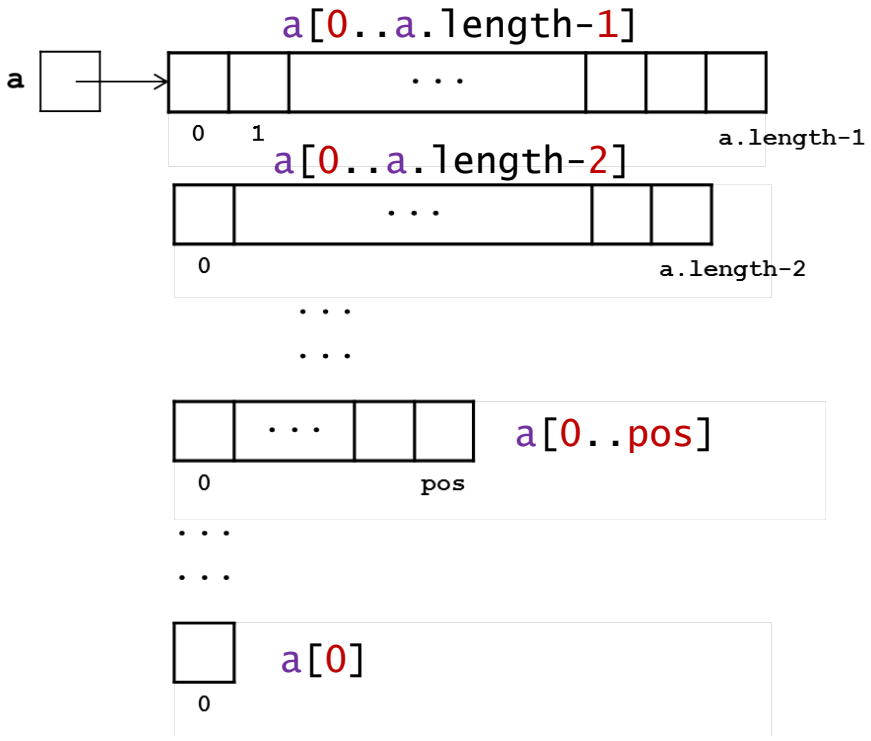
  denotada como `a[0..a.length-1]`.
- Dicha secuencia se puede definir recursivamente como la secuencia formada por `a[0]` y la subsecuencia (subarray) definida por el resto de componentes de `a`, esto es, `a[1..a.length-1]`.
- A su vez, el subarray `a[1..a.length-1]` puede definirse recursivamente del mismo modo y así, sucesivamente, hasta que el subarray correspondiente esté vacío, sin componentes. Es lo que se denomina descomposición recursiva ascendente.
- Análogamente, el array `a` se puede definir de forma recursiva considerando el subarray `a[0..a.length-2]` y su última componente `a[a.length-1]`, lo que se denomina descomposición recursiva descendente.

# Recursión con arrays: recorrido y búsqueda

descomposición recursiva ascendente




descomposición recursiva descendente



# Recursión con arrays: recorrido y búsqueda

- Los parámetros formales de un método recursivo que realice un recorrido o una búsqueda sobre un (sub)array `a` serán el propio (sub)array `a` y las posiciones que marcan el `inicio` y el `fin` del recorrido o la búsqueda a realizar sobre él en cada llamada.
- Se denomina talla de un problema al valor o conjunto de valores asociados a los datos de entrada que representan una medida de la dificultad para su resolución.
- En el caso de problemas con arrays, la talla es el número de componentes del (sub)array `a[inicio..fin]`, esto es, `t = fin-inicio+1`.
- Además, habrá que determinar el tipo de descomposición recursiva:
  - Ascendente: `inicio` se incrementa en cada llamada hasta que supera a `fin` en el caso base: `inicio=fin+1` (`inicio>fin`).

    Para una talla inicial `t = a.length`, se cumple:

    $$0 \leq \texttt{inicio} \leq \texttt{a.length} \text{ y } \texttt{fin} = \texttt{a.length}-1$$
  - Descendente: `fin` se decrementa en cada llamada hasta que es menor que `inicio` en el caso base: `fin=inicio-1` (`fin<inicio`).

    Para una talla inicial `t = a.length`, se cumple:

    $$\texttt{inicio} = 0 \text{ y } -1 \leq \texttt{fin} \leq \texttt{a.length}-1$$

# Esquemas recursivos de recorrido

- Esquema recursivo de recorrido ascendente de un array `a` desde una posición `izq` a una posición `der`, 0≤ `izq` ≤ `der` < `a.length`.

```
/** 0<=inicio<=der+1 y fin=der */
public static void recorrer(tipoBase[] a, int inicio, int fin) {
    if (inicio>fin) tratarVacio();
    else {
           tratar(a[inicio]);
           recorrer(a, inicio+1, fin);
         }
}
```

- `tratarVacio()`: operación a realizar para un (sub)array sin elementos.
- `tratar(a[inicio])`: operación a realizar con el elemento que ocupa la posición `inicio` del array.
- Llamada inicial: `recorrer(a,izq,der)`, esto es, `inicio=izq` y `fin=der`.

# Esquemas recursivos de recorrido

- Esquema recursivo de recorrido ascendente de todos los elementos de un array `a`, es decir, en la llamada inicial `inicio`=`0` y `fin`=`a.length-1`.
- Se suele definir un método público homónimo, denominado guía o lanzadera, que realiza la llamada inicial, con el fin de ocultar la estructura recursiva del array `a` que muestran los parámetros `inicio` y `fin` de la cabecera del método recursivo `recorrer` anterior que ahora se define privado.

```
public static void recorrer(tipoBase[] a) {
        recorrer(a, 0, a.length-1);
}
```

# Esquemas recursivos de recorrido

- Esquema recursivo de recorrido ascendente simplificado.
- Se sustituye cualquier referencia al parámetro formal `fin` por su valor en la llamada inicial `der`, como sigue:

```
/** 0<=inicio<=der+1 */
public static void recorrer(tipoBase[] a, int inicio) {
    if (inicio==der+1) tratarVacio();
    else {
            tratar(a[inicio]);
            recorrer(a, inicio+1);
         }
}
```

- Llamada inicial: `recorrer(a,izq)`, esto es, `inicio=izq`.

# Esquemas recursivos de recorrido

- Esquema recursivo de recorrido descendente de un array `a` desde una posición `izq` a una posición `der`, 0≤ `izq` ≤ `der` < `a.length`.

```
/** inicio=izq y izq-1<=fin<a.length */
public static void recorrer(tipoBase[] a, int inicio, int fin) {
    if (fin<inicio) tratarVacio();
    else {
           tratar(a[fin]);
           recorrer(a, inicio, fin-1);
         }
}
```

- `tratarVacio()`: operación a realizar para un (sub)array sin elementos.
- `tratar(a[fin])`: operación a realizar con el elemento que ocupa la posición `fin` del array.
- Llamada inicial: `recorrer(a,izq,der)`, esto es, `inicio=izq` y `fin=der`.

# Esquemas recursivos de recorrido

- Si se trata de un recorrido de todos los elementos del array `a`, en la llamada inicial `inicio=0` y `fin=a.length-1`, se puede definir también, en este caso, un método guía idéntico al del esquema ascendente.

- Esquema recursivo de recorrido descendente simplificado en el que `inicio` desaparece como parámetro formal:

```
/** izq-1<=fin<a.length */
public static void recorrer(tipoBase[] a, int fin) {
    if (fin==izq-1) tratarVacio();
    else {
            tratar(a[fin]);
            recorrer(a, fin-1);
          }
}
```

- Llamada inicial: `recorrer(a,der)`, esto es, `fin=der`.

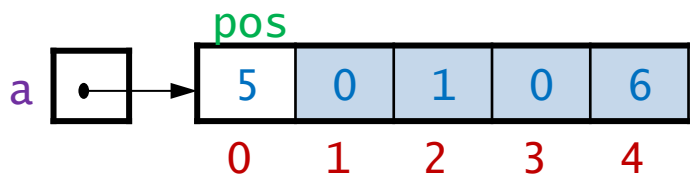
# Algunos ejemplos - Recorrido

1. Enunciado del problema. Determinar la suma de todos los elementos de un array de enteros a[0..a.length-1].

Recorrido recursivo ascendente

```
/** 0<=pos<=a.length */
public static int sumaRecAsc(int[] a, int pos)
```

2. Análisis de casos.
   - Caso base: Si pos=a.length, entonces la suma de 0 elementos es 0.
   - Caso general: En otro caso, la suma será la suma de a[pos] y la de a[pos+1..a.length-1].
3. Transcripción del algoritmo a un método en Java.




```
/** 0<=pos<=a.length */
public static int sumaRecAsc(int[] a, int pos) {
        if (pos==a.length) return 0;
        else return a[pos] + sumaRecAsc(a, pos+1);
}
```

   - La llamada inicial debe ser: `int suma = sumaRecAsc(a,0);`
4. Validación del diseño.
   - En el caso general, en cada llamada la talla del problema decrece en 1 porque pos se incrementa en 1; así, en algún momento pos llegará a ser a.length, alcanzando el caso base y finalizando el algoritmo.
   - En cualquiera de los dos casos, el valor de pos siempre cumple la precondición.

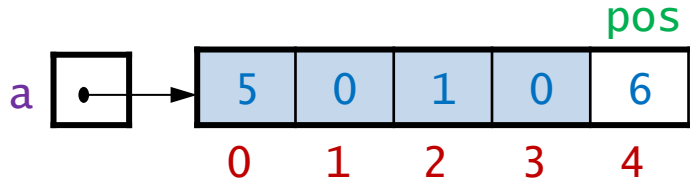
# Algunos ejemplos - Recorrido

1. Enunciado del problema. Determinar la suma de todos los elementos de un array de enteros a[0..a.length-1].

Recorrido recursivo descendente

```
/** -1<=pos<a.length */
public static int sumaRecDesc(int[] a, int pos)
```

2. Análisis de casos.
   - Caso base: Si pos=-1, entonces la suma de 0 elementos es 0.
   - Caso general: En otro caso, la suma será la suma de a[pos] y la de a[0..pos-1].
3. Transcripción del algoritmo a un método en Java.




```
/** -1<=pos<a.length */
public static int sumaRecDesc(int[] a, int pos) {
        if (pos==-1) return 0;
        else return a[pos]+sumaRecDesc(a, pos-1);
}
```

- La llamada inicial debe ser: `int suma = sumaRecDesc(a,a.length-1);`

4. Validación del diseño.
   - En el caso general, en cada llamada la talla del problema decrece en 1 porque pos se decrementa en 1; así, en algún momento pos llegará a ser -1, alcanzando el caso base y finalizando el algoritmo.
   - En cualquiera de los dos casos, el valor de pos siempre cumple la precondición.

# Algunos ejemplos - Recorrido

1. Enunciado del problema. Contar las apariciones de un entero `x` en un array de enteros `a[0..a.length-1]`.

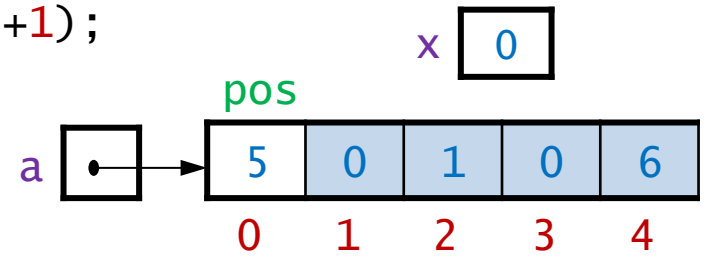
Recorrido recursivo ascendente

```
/** 0<=pos<=a.length */
public static int contarRecAsc(int[] a, int x, int pos)
```

2. Análisis de casos.
   - Caso base: Si `pos=a.length`, entonces no hay elementos y devuelve `0`.
   - Caso general: En otro caso, habrá que contar las apariciones de `x` en `a[pos+1..a.length-1]` y sumar `1` si `a[pos]` es `x`.

3. Transcripción del algoritmo a un método en Java.

```
/** 0<=pos<=a.length */
public static int contarRecAsc(int[] a, int x, int pos) {
   if (pos==a.length) return 0;
   else { int cont = contarRecAsc(a, x, pos+1);
          if (a[pos]!=x) return cont;
          else return 1 + cont;
        }
}
```




   - La llamada inicial debe ser: `int num = contarRecAsc(a,x,0);`

4. Validación del diseño. Similar a la validación del recorrido ascendente anterior.

# Algunos ejemplos - Recorrido

1. Enunciado del problema. Contar las apariciones de un entero `x` en un array de enteros `a[0..a.length-1]`.

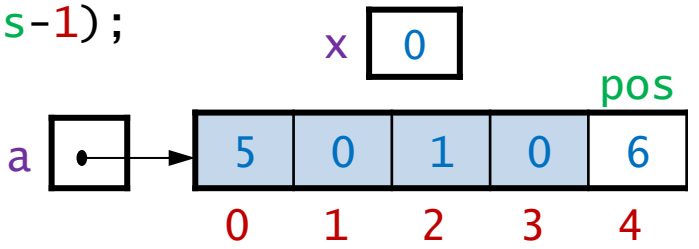
   Recorrido recursivo descendente

```
/** -1<=pos<a.length */
public static int contarRecDesc(int[] a, int x, int pos)
```

2. Análisis de casos.
   - Caso base: Si `pos=-1`, entonces no hay elementos y devuelve `0`.
   - Caso general: En otro caso, habrá que contar las apariciones de `x` en `a[0..pos-1]` y sumar `1` si `a[pos]` es `x`.

3. Transcripción del algoritmo a un método en Java.

```
/** -1<=pos<a.length */
public static int contarRecDesc(int[] a, int x, int pos) {
   if (pos==-1) return 0;
   else { int cont = contarRecDesc(a, x, pos-1);
          if (a[pos]!=x) return cont;
          else return 1 + cont;
        }
}
```




   - La llamada inicial debe ser: `int num = contarRecDesc(a,x,a.length-1);`

4. Validación del diseño. Similar a la validación del recorrido descendente anterior.

# Esquemas recursivos de búsqueda

- Esquema recursivo de búsqueda ascendente de la posición de un elemento que cumpla cierta propiedad en un array `a` desde una posición `izq` a una posición `der`, $0 \leq$ `izq` $\leq$ `der` $<$ `a.length`.

```
/** 0<=inicio<=der+1 y fin=der */
public static int buscar(tipoBase[] a, int inicio, int fin) {
    int resMetodo = -1;
    if (inicio<=fin)
       if (propiedad(a[inicio])) resMetodo = inicio;
       else resMetodo = buscar(a, inicio+1, fin);
    return resMetodo;
}
```

- `propiedad(a[inicio])`: comprueba si el elemento que ocupa la posición `inicio` del array cumple la propiedad enunciada.
- Llamada inicial: `int res=buscar(a,izq,der)`, esto es, `inicio=izq` y `fin=der`.

# Esquemas recursivos de búsqueda

- Si se trata de una búsqueda en todo el array a, en la llamada inicial inicio=0 y fin=a.length-1, se puede definir también un método guía similar al del esquema de recorrido.

```
public static int buscar(tipoBase[] a) {
        return buscar(a, 0, a.length-1);
}
```

- Esquema recursivo de búsqueda ascendente simplificado.

```
/** 0<=inicio<=der+1 */
public static int buscar(tipoBase[] a, int inicio) {
    int resMetodo = -1;
    if (inicio<=der)
       if (propiedad(a[inicio])) resMetodo = inicio;
       else resMetodo = buscar(a, inicio+1);
    return resMetodo;
}
```

- Llamada inicial: `int res=buscar(a,izq)`, esto es, `inicio=izq`.

# Esquemas recursivos de búsqueda

- Esquema recursivo de búsqueda descendente de la posición de un elemento que cumpla cierta propiedad en un array `a` desde una posición `izq` a una posición `der`, $0 \leq$ `izq` $\leq$ `der` $<$ `a.length`.

```
/** inicio=izq y izq-1<=fin<a.length */
public static int buscar(tipoBase[] a, int inicio, int fin) {
    int resMetodo = -1;
    if (inicio<=fin)
       if (propiedad(a[fin])) resMetodo = fin;
       else resMetodo = buscar(a, inicio, fin-1);
    return resMetodo;
}
```

- `propiedad(a[fin])`: comprueba si el elemento que ocupa la posición `fin` del array cumple la propiedad enunciada.
- Llamada inicial: `int res=buscar(a,izq,der)`, esto es, `inicio=izq` y `fin=der`.

# Esquemas recursivos de búsqueda

- Si se trata de una búsqueda en todo el array a, en la llamada inicial inicio=0 y fin=a.length-1, se puede definir también, en este caso, un método guía idéntico al del esquema ascendente.

- Esquema recursivo de búsqueda descendente simplificado.

```
/** izq-1<=fin<a.length */
public static int buscar(tipoBase[] a, int fin) {
    int resMetodo = -1;
    if (izq<=fin)
       if (propiedad(a[fin])) resMetodo = fin;
       else resMetodo = buscar(a, fin-1);
    return resMetodo;
}
```

- Llamada inicial: `int res=buscar(a,der)`, esto es, `fin=der`.

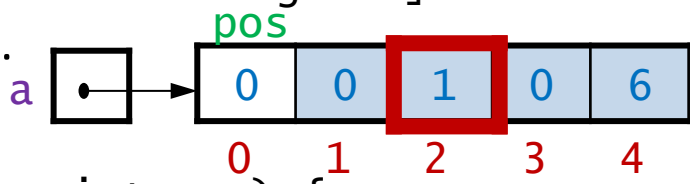
# Algunos ejemplos - Búsqueda

1. Enunciado del problema. Obtener la posición del primer elemento no nulo de un array de enteros a[0..a.length-1].

Búsqueda recursiva ascendente

```
/** 0<=pos<=a.length */
public static int encontrarRecAsc(int[] a, int pos)
```

2. Análisis de casos.
   - Caso base: Si pos=a.length, entonces no se encuentra y devuelve -1.
   - Caso general: En otro caso, bien a[pos] es el primer no nulo y devuelve pos o bien no lo es y la búsqueda continúa en a[pos+1..a.length-1].
3. Transcripción del algoritmo a un método en Java.




```
/** 0<=pos<=a.length */
public static int encontrarRecAsc(int[] a, int pos) {
        if (pos==a.length) return -1;
        else if (a[pos] !=0) return pos;
        else return encontrarRecAsc(a, pos+1);
}
```

   - La llamada inicial debe ser: `int noNulo = encontrarRecAsc(a,0);`
4. Validación del diseño.
   - En el caso general, en cada llamada la talla del problema decrece en 1 porque pos se incrementa en 1; así, en algún momento pos llegará a ser a.length, alcanzando el caso base y finalizando el algoritmo.
   - En cualquiera de los dos casos, el valor de pos siempre cumple la precondición.

# Algunos ejemplos - Búsqueda

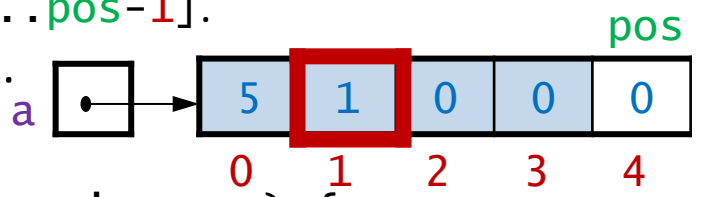
1. Enunciado del problema. Obtener la posición del último elemento no nulo de un array de enteros a[0..a.length-1].

Búsqueda recursiva descendente

```
/** -1<=pos<a.length */
public static int encontrarRecDesc(int[] a, int pos)
```

2. Análisis de casos.
   - Caso base: Si pos=-1, entonces no se encuentra y devuelve -1.
   - Caso general: En otro caso, bien a[pos] es el último no nulo y devuelve pos o bien no lo es y la búsqueda continúa en a[0..pos-1].

3. Transcripción del algoritmo a un método en Java.




```
/** -1<=pos<a.length */
public static int encontrarRecDesc(int[] a, int pos) {
        if (pos==-1) return -1;
        else if (a[pos] !=0) return pos;
        else return encontrarRecDesc(a, pos-1);
}
```

- La llamada inicial debe ser: `int noNulo=encontrarRecDesc(a,a.length-1);`

4. Validación del diseño.
   - En el caso general, en cada llamada la talla del problema decrece en 1 porque pos se decrementa en 1; así, en algún momento pos llegará a ser -1, alcanzando el caso base y finalizando el algoritmo.
   - En cualquiera de los dos casos, el valor de pos siempre cumple la precondición.

# Búsqueda binaria recursiva

1. Enunciado del problema. Obtener la posición de un entero `x` en un array de enteros `a[0..a.length-1]` ordenado ascendentemente.

```
/** 0<=ini<=a.length y -1<=fin<a.length */
public static int encBinRec(int[] a, int x, int ini, int fin)
```

2. Análisis de casos.
   - Caso base: Si `ini>fin`, entonces no se encuentra y devuelve `-1`.
   - Caso general: Si `ini≤fin`, para el subarray `a[ini..fin]`,
     - Determinar la posición del elemento central `mitad = (ini+fin)/2`.
     - Acceder al elemento central de `a[ini..fin]`, `a[mitad]`, comprobando que si:
     - `a[mitad]==x`, entonces la búsqueda acaba con éxito y devuelve `mitad`.
     - `a[mitad]>x`, entonces la búsqueda continúa en `a[ini..mitad-1]`.
     - `a[mitad]<x`, entonces la búsqueda continúa en `a[mitad+1..fin]`.

# Búsqueda binaria recursiva

3. Transcripción del algoritmo a un método en Java.

```
/** 0<=ini<=a.length y -1<=fin<a.length */
public static int encBinRec(int[] a, int x, int ini, int fin) {
  if (ini>fin) return -1;
  else { int mitad = (ini+fin)/2;
         if (a[mitad]==x) return mitad;
         else if (a[mitad]>x) return encBinRec(a,x,ini,mitad-1);
         else return encBinRec(a,x,mitad+1,fin);
       }
}
```

- La llamada inicial debe ser: `int pos = encBinRec(a,x,0,a.length-1);`

4. Validación del diseño.
   - En el caso general, en cada llamada la talla del problema se divide por 2 (división entera); así, en algún momento `ini>fin`, alcanzando el caso base y finalizando el algoritmo.
   - En cualquiera de los dos casos, los valores de `ini` y `fin` siempre cumplen la precondición.

# Recursión versus iteración

- Tanto recursión como iteración hacen uso de estructuras de control: la recursión usa como instrucción principal una instrucción de selección (condicional) y la iteración usa como instrucción principal una instrucción de repetición (bucle).
- Ambas requieren una condición de terminación: la condición del caso base en la recursión y la condición de la guarda del bucle en la iteración.
- Ambas se aproximan gradualmente a la terminación: la iteración conforme se acerca al cumplimiento de una condición y la recursión conforme se divide el problema en otros más pequeños, acercándose también al cumplimiento de una condición, la del caso base.
- Ambas pueden tener (por error) una ejecución potencialmente infinita.
- Se puede demostrar que la solución algorítmica de cualquier problema algorítmicamente resoluble, se puede expresar recursivamente y también iterativamente.
- En este sentido, se dice que recursión e iteración son equivalentes, y por ello, alternativos.

# Recursión versus iteración

- En general, no es posible afirmar qué es lo más conveniente o sencillo.
- Es frecuente encontrar problemas para los que la solución iterativa es más sencilla de estructurar que la recursiva.
- Además, la recursión supone, en general, más carga computacional (espacio en memoria) que la iteración.
- En otros casos, la versión recursiva refleja de manera más natural, concisa y elegante la solución al problema que la versión iterativa, lo que hace que sea más fácil de depurar y entender.
- Se puede concluir que recursión e iteración, además de alternativos, son complementarios.