Para poder entender lo que es la recursividad, antes hay que entender lo que es la recursividad

Recursividad

Eva Lucrecia Gibaja Galindo

Dpto. Informática y Análisis Numérico

Introducción.Problemas recursivos

- Una definición de un problema *T* se dice recursiva cuando *T* se define en términos de versiones más pequeñas de sí mismo
- Para que una definición recursiva esté completamente identificada, es necesario tener un **caso base** que no se calcule utilizando casos anteriores
- Ejemplos:
  - El factorial de un número (ej. 5!= 5*4*3*2*1=5*4!):

    $n! = n*(n-1)!$

    $0! = 1$
  - La función potencia

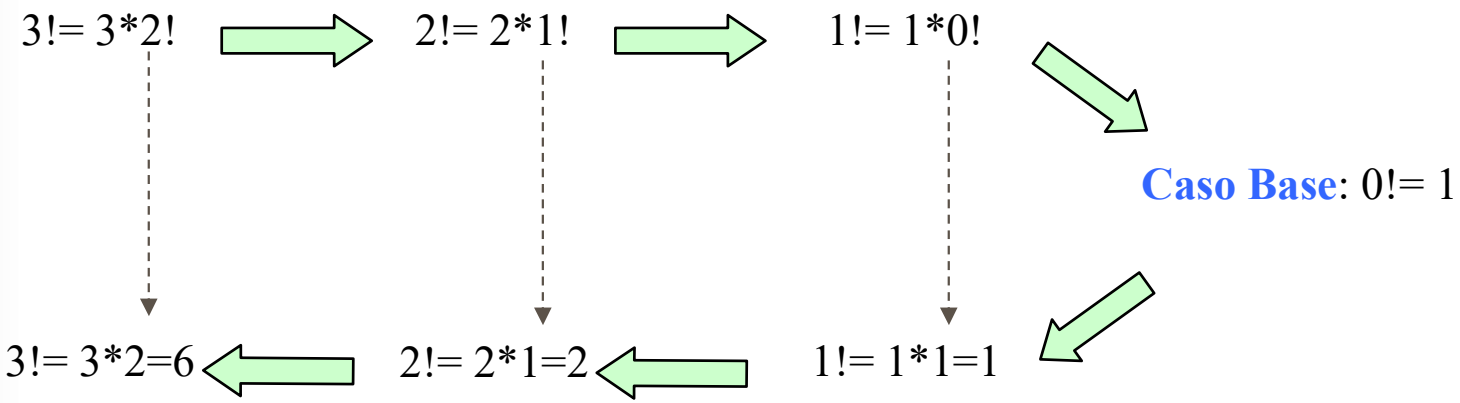
    $x^n = x*x^{n-1}$

    $x^0 = 1$

casos base

Introducción.Problemas recursivos

- **Ejemplo**. Calcular el factorial con n=3
  - Para calcular 3! debemos hacer 3*2!
  - Para calcular 2! debemos hacer 2*1!
  - Para calcular 1! debemos hacer 1*0!
  - 0! es, por definición de caso base igual a 1
  - 1! = 1*0! = 1
  - 2! = 2*1! = 2
  - 3! = 3*2! = 6



Introducción.Problemas recursivos

- Siguiendo el ejemplo del factorial, podemos distinguir dos procesos fundamentales en la recursividad:
  1. Se van produciendo llamadas para calcular *n!* con *n* decreciendo sucesivamente. La resolución de *n!* se queda en espera, hasta que se calcule *(n-1)!*
  2. Se llega a un caso base y se produce una vuelta atrás en la secuencia anterior, para terminar la resolución de las distintas llamadas que se habían quedado en espera

Introducción.Algoritmos recursivos

- Un algoritmo o un módulo se dice que es recursivo si realiza una o más llamadas a si mismo
- Un módulo recursivo consta de dos partes:
  - **Condición de parada o caso base**: Establece el momento de terminación del procedimiento recursivo
  - **Cuerpo del módulo**: Establece los cálculos que se deben realizar para resolver el problema entre los cuales se incluyen las llamadas recursivas

Introducción.Algoritmos recursivos

- La recursividad se puede clasificar como:
  - **Directa**: El algoritmo contiene una llamada a sí mismo
    - **Simple o lineal**: El nombre del algoritmo aparece una sola vez en el cuerpo del algoritmo
    - **Doble, múltiple o no lineal**: El nombre aparece dos (o más) veces en el cuerpo del algoritmo
  - **Indirecta**: El algoritmo *P* contiene una llamada a otro algoritmo *Q*, que a su vez contiene una llamada directa o indirecta a *P*

main.c;

recur.c;recur.h

Ejemplo. Factorial

```
int factorial(int n)
{
   int resultado;
   if (n==0)
      resultado = 1; /*Caso base*/
   else
      resultado = n*factorial(n-1); /*Cuerpo*/
   return(resultado);
}
```

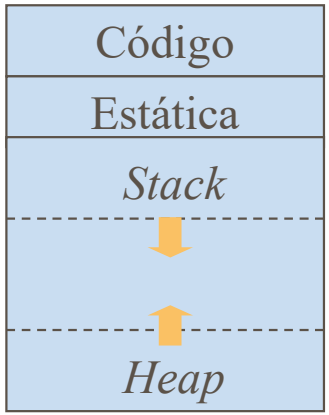
Ejemplo. Factorial

Factorial		
n	Valor matemático	Valor programa
1	1	1
2	2	2
3	6	6
4	24	24
5	120	120
6	720	720
7	5040	5040
8	40320	40320
9	362880	362880
10	3628800	3628800
11	39916800	39916800
12	479001600	479001600
13	6227020800	1932053504
14	87178291200	1278945280
15	1307674368000	2004310016
16	20922789888000	2004189184
17	355687428096000	-288522240

479.001.600 <2.147.483.647(INT_MAX)< 6.227.020.800

Ejecución de un módulo recursivo

- La memoria del programa en tiempo de ejecución se divide en:
  - **Código**→ Instrucciones del programa en código máquina
  - **Datos** → Variables estáticas y globales
  - **Montículo (*heap*)** → Variables dinámicas
  - **Pila (*stack*)** → **Registro de activación** del módulo: Variables locales y parámetros de la función que está siendo ejecutada



Ejecución de un módulo recursivo

Llamada a una función

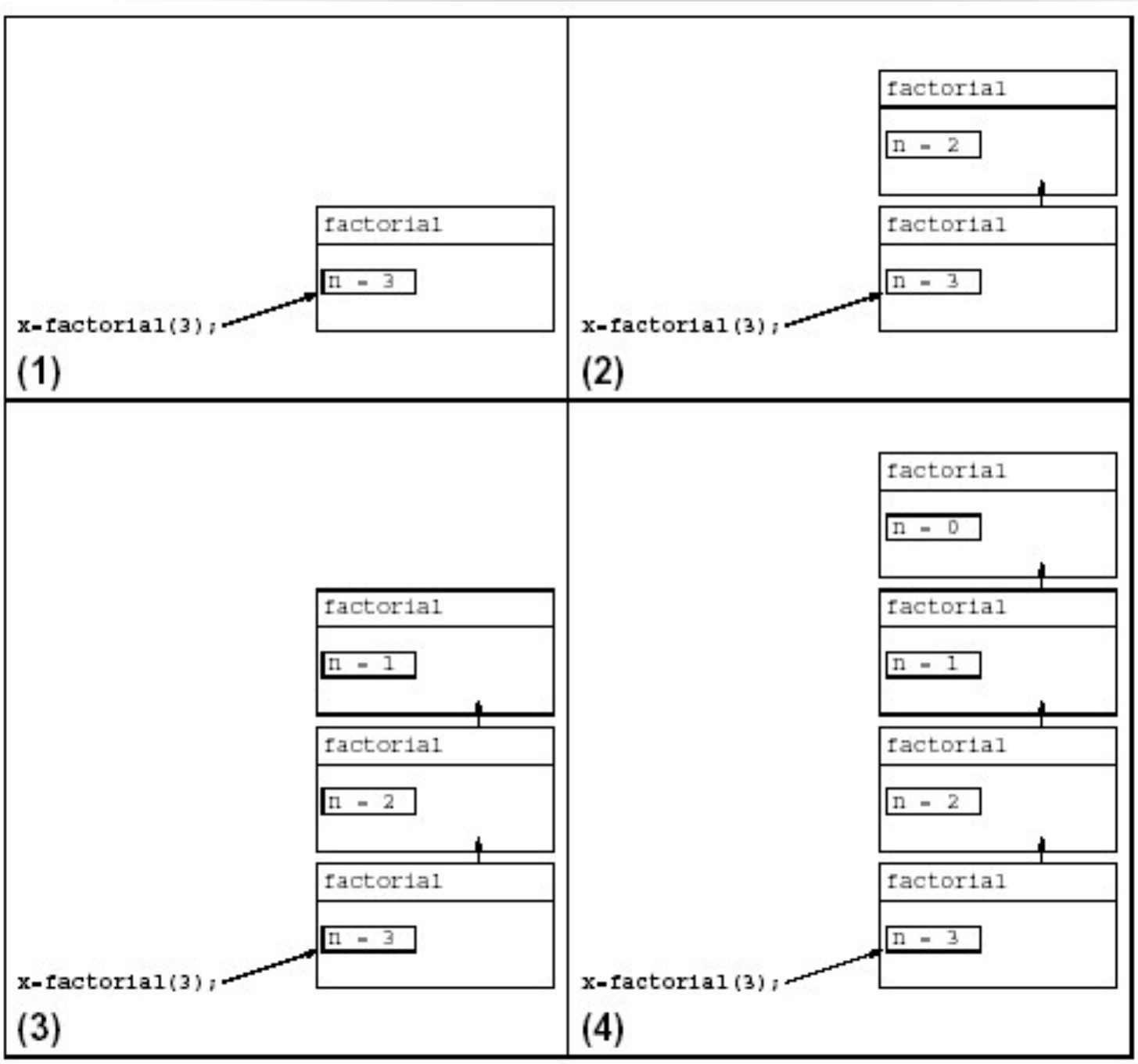
1. Se reserva espacio en la pila para los parámetros de la función y sus variables locales
2. Se guarda en la pila la dirección de la línea de código que ha llamado a la función
3. Se almacenan en pila los parámetros de la función y sus valores
4. Al terminar, se libera la memoria asignada en la pila y se vuelve a la instrucción actual

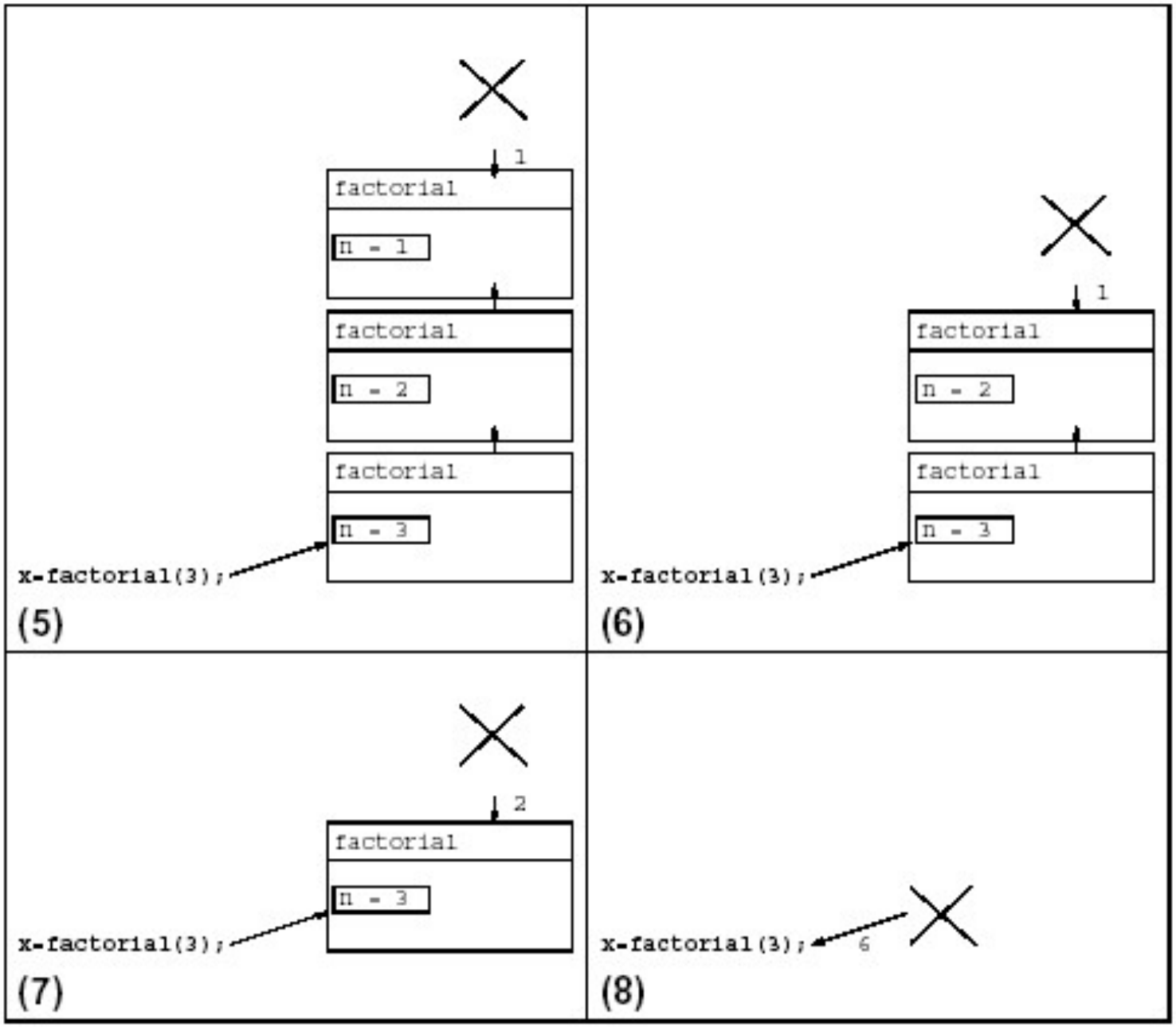
Llamada a una función recursiva

- Cada llamada recursiva genera una nueva zona de memoria en la pila independiente (un nuevo registro de activación) del resto de llamadas con sus correspondientes objetos locales:
  1. La función se ejecuta normalmente hasta la llamada recursiva
  2. En ese momento se crean en la pila nuevos parámetros y variables locales.
  3. El nuevo ejemplar de función comienza a ejecutarse
  4. Se crean más copias hasta llegar al los casos base, donde se resuelve directamente el valor
  5. Se sale liberando la memoria hasta llegar a la primera llamada (última en cerrarse)

Ejemplo: factorial (3)

1. Dentro del factorial, cada llamada *n*factorial(n-1)* genera una nueva zona de memoria en la pila, siendo *n-1* el correspondiente parámetro actual para esta zona de memoria y queda pendiente la evaluación de la expresión y la ejecución del *return*
2. El proceso anterior se repite hasta que la condición del caso base se hace cierta
   - Se ejecuta la sentencia *return 1*
   - Empieza la vuelta atrás de la recursión, se evalúan las expresiones y se ejecutan los que estaban pendientes


factorial
n = 3
x=factorial(3);
(1)
factorial
n = 2
factorial
n = 3
x=factorial(3);
(2)
factorial
n = 1
factorial
n = 2
factorial
n = 3
x=factorial(3);
(3)
factorial
n = 0
factorial
n = 1
factorial
n = 2
factorial
n = 3
x=factorial(3);
(4)


1
factorial
n = 1
factorial
n = 2
factorial
n = 3
x=factorial(3);
(5)
1
factorial
n = 2
factorial
n = 3
x=factorial(3);
(6)
2
factorial
n = 3
x=factorial(3);
(7)
x=factorial(3);
6
(8)

Cálculo de la potencia

$$x^n = x * x^{n-1} \text{ si } n \geq 1$$

$$x^0 = 1$$

```
int potencia(int base, int expo) {
   if (expo==0)
      return 1;
   else
      return base * potencia(base,expo-1);
}
```

$3^5 = 243$

1. llamadas recursivas

$3^5 = 3 * 3^4$ $\quad 3^5 = 3 * 81$

$3^4 = 3 * 3^3$ $\quad 3^4 = 3 * 27$

$3^3 = 3 * 3^2$ $\quad 3^3 = 3 * 9$

$3^2 = 3 * 3^1$ $\quad 3^2 = 3 * 3$

$3^1 = 3 * 3^0$ $\quad 3^1 = 3 * 1$

2. deshacer recursividad

$3^0 = 1$

main.c;

recur.c;recur.h

Suma recursiva

3+5=8

3+5= 1+suma(3, 4)

3+4 =1+suma(3, 3)

3+3 =1+suma(3, 2)

3+2 = 1+suma(3, 1)

3+1 = 1+suma(3, 0)

3

3+5= 1+7

3+4 =1+6

3+3 =1+5

3+2 = 1+4

3+1 = 1+3

$$suma(a,b) = 1 + suma(a,b-1)$$

$$suma(a,0) = a$$

```
int suma(int a, int b) {
   if (b==0)
      return a;
   else
      return 1+suma(a,b-1);
}
```

*Sumar **a+b** es sumar **b** veces un **1** al número **a***

Producto recursivo

3*5=15

3*5= 3+producto(3, 4) 3*5= 3+12

3*4=3+producto(3, 3) 3*4 =3+9

3*3=3+producto(3, 2) 3*3 =3+6

3*2=3+producto(3, 1) 3*2 =3+3

3*1=3+producto(3, 0) 3*1 = 3+0

0

$$producto(a,b) = a + producto(a, b-1)$$

$$producto(a,0) = 0$$

```
int producto(int a, int b) {
   if (b==0)
      return 0;
   else
      return a+producto(a,b-1);
}
```

Producto como sumas sucesivas:

*Multiplicar **a*b** es sumar **b** veces **a***

Suma de los dígitos de un número

```
int sumaDigitos(int num)
{
     if(num<10)
        return (num);
     else
        return(num%10+sumaDigitos(num/10));
}
```

2483

3+sumaDigitos(248) 17=3+14

8+sumaDigitos(24) 8+6

4+sumaDigitos(2) 4+2

2

El operador de módulo (%) permite obtener el último dígito

¿Es un número potencia de una base?

```
int esPotencia(int num, int base)
{     if (num==1)
        return 1;
      else
        if (num%base!=0)
           return 0;
        else
              return esPotencia(num/base, base);
}
```

esPotencia(12, 2)	esPotencia(8,2)
esPotencia(6,2)	esPotencia(4,2)
esPotencia(3,2) → falso	esPotencia(2,2)
	esPotencia(1,2) →cierto

main.c;

recur.c;recur.h

Recursión doble

Sucesión de Fibonacci

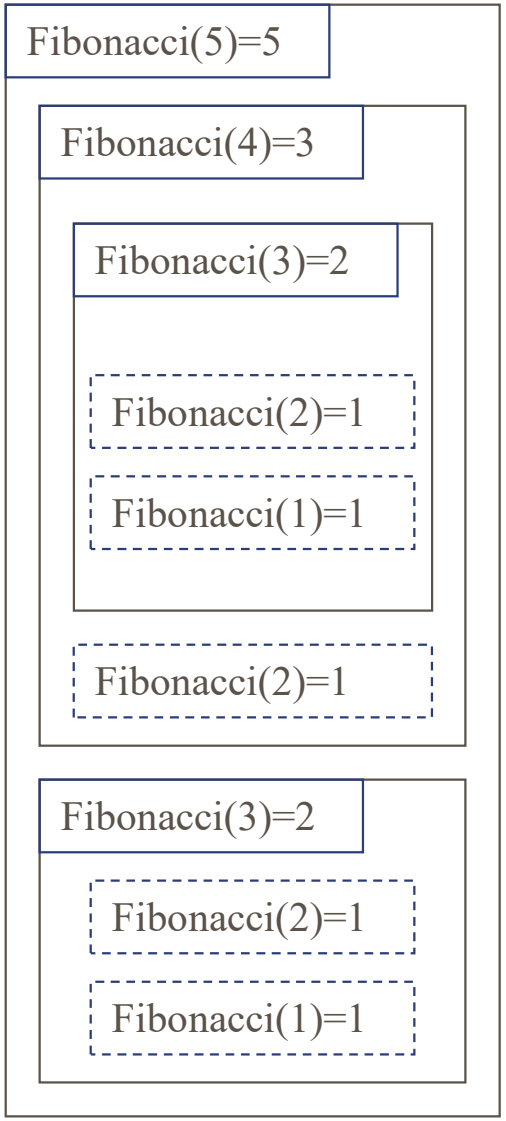
$$Fib(1) = Fib(2) = 1$$

$$Fib(n) = Fib(n-1) + Fib(n-2)$$

```
int fibonacci(int n)
{
  if((n==1)||(n==2))
  {
    return(1);
  }
  else
  {
    return(fibonacci(n-1)+fibonacci(n-2));
  }
}
```

n	1	2	3	4	5	6
Fib(n)	1	1	2	3	5	8

Esta serie fue concebida originalmente como modelo para el crecimiento de una granja de conejos (reproducción de conejos) por el italiano del s. XVI Fibonacci



main.c;

recur.c;recur.h

Fibonacci iterativo (mucho más eficiente)

$$Fib(1) = Fib(2) = 1$$
$$Fib(n) = Fib(n-1) + Fib(n-2)$$

n	1	2	3	4	5	6
Fib(n)	1	1	2	3	5	8

i	actual=0	ant1=1	ant2=1
3	1+1=2	2	1
4	2+1=3	3	2
5	3+2=5	5	3
6	3+3=8	8	5

```
int fibonacciIterativo(int n)
{
  int ant1 = 1; //Anterior
  int ant2 = 1; //Anteanterior
  int actual=0;
  int i;

  if((n==1)||(n==2))
  {
    actual = 1; //no calcular nada
  }
  else
  {
    for(i=3; i<=n; i++)
    {
        //Suma los dos anteriores
        actual = ant1+ant2;

        //actualiza ant2 y ant1
        ant2 = ant1;
        ant1 = actual;
    }
  }
  return(actual);
}
```

main.c;

recur.c;recur.h

Suma recursiva de los elementos de un vector

*El parámetro **n** es la **posición** hasta la que queremos sumar (6) y no el **número de elementos** (7) → La llamada es **sumaV(V, 6)***

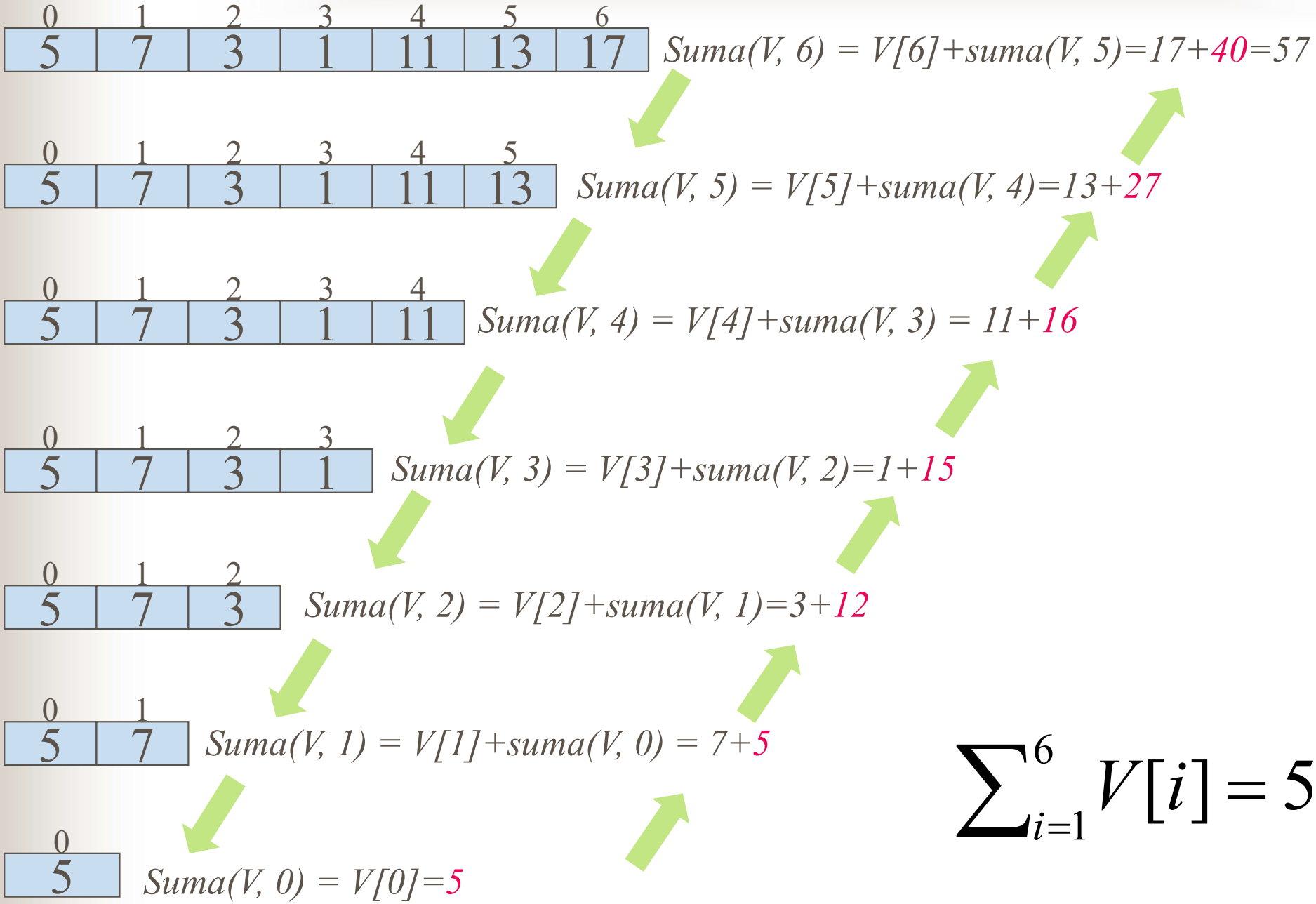
5	7	3	1	11	13	17

$$SumaV(V,n) = V[n] + SumaV(V,n-1)$$

$$SumaV(V,0) = V[0]$$

```
int SumaV (int *V, int n) {
   if (n==0)
      return V[0];
   else
      return V[n]+SumaV(V,n-1);
}
```

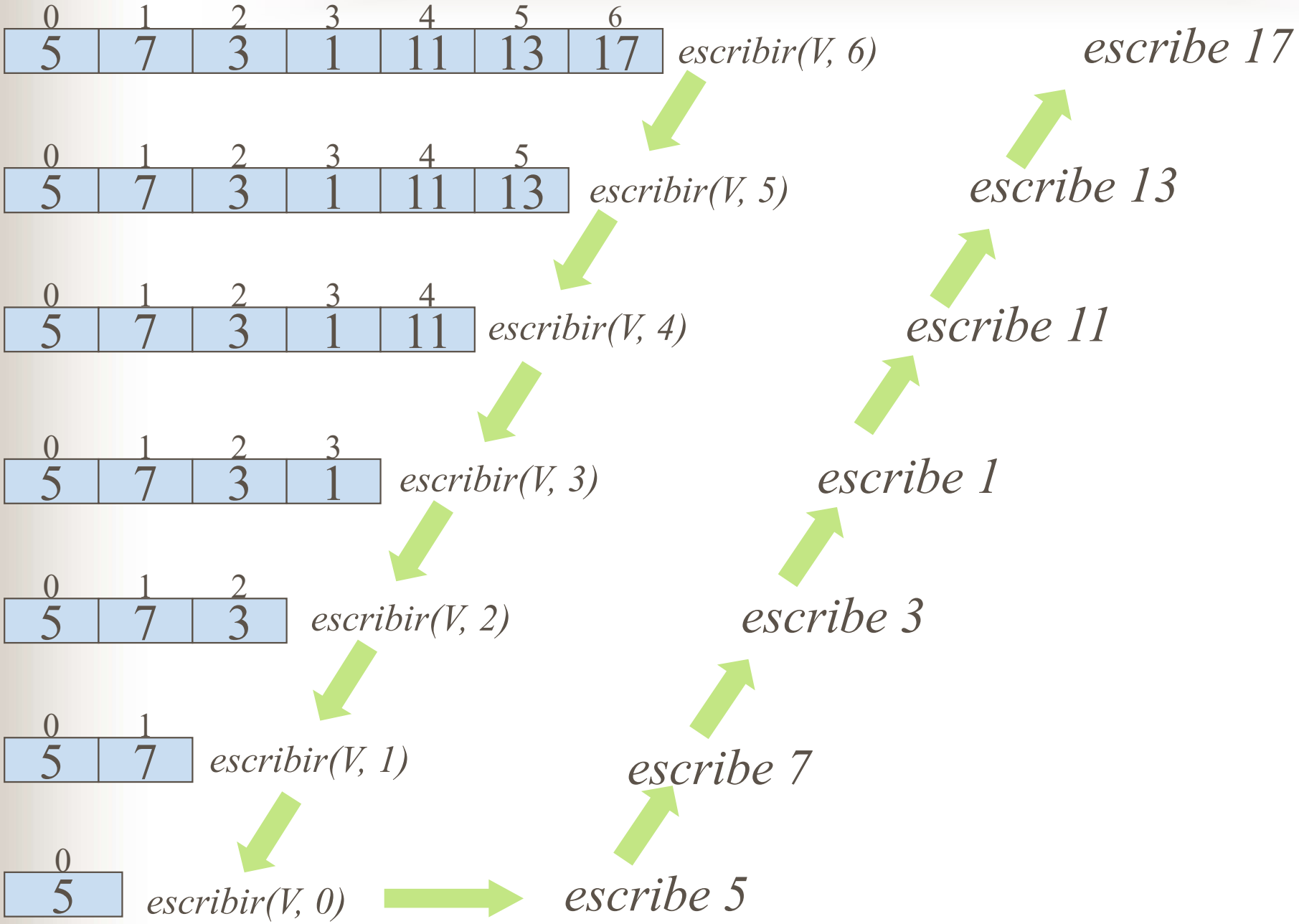
Recorrido recursivo de un vector


0 1 2 3 4 5 6
5 7 3 1 11 13 17
Suma(V, 6) = V[6]+suma(V, 5)=17+40=57
0 1 2 3 4 5
5 7 3 1 11 13
Suma(V, 5) = V[5]+suma(V, 4)=13+27
0 1 2 3 4
5 7 3 1 11
Suma(V, 4) = V[4]+suma(V, 3) = 11+16
0 1 2 3
5 7 3 1
Suma(V, 3) = V[3]+suma(V, 2)=1+15
0 1 2
5 7 3
Suma(V, 2) = V[2]+suma(V, 1)=3+12
0 1
5 7
Suma(V, 1) = V[1]+suma(V, 0) = 7+5
0
5
Suma(V, 0) = V[0]=5

$$\sum_{i=1}^{6} V[i] = 57$$

Escribir un vector

```
void escribirVector(int* V, int posicion)
{
  if(posicion>=0)
  {
    escribirVector(V, posicion-1);
    printf("\n\tV[%d]=%d", posicion, V[posicion]);
  }
}
```

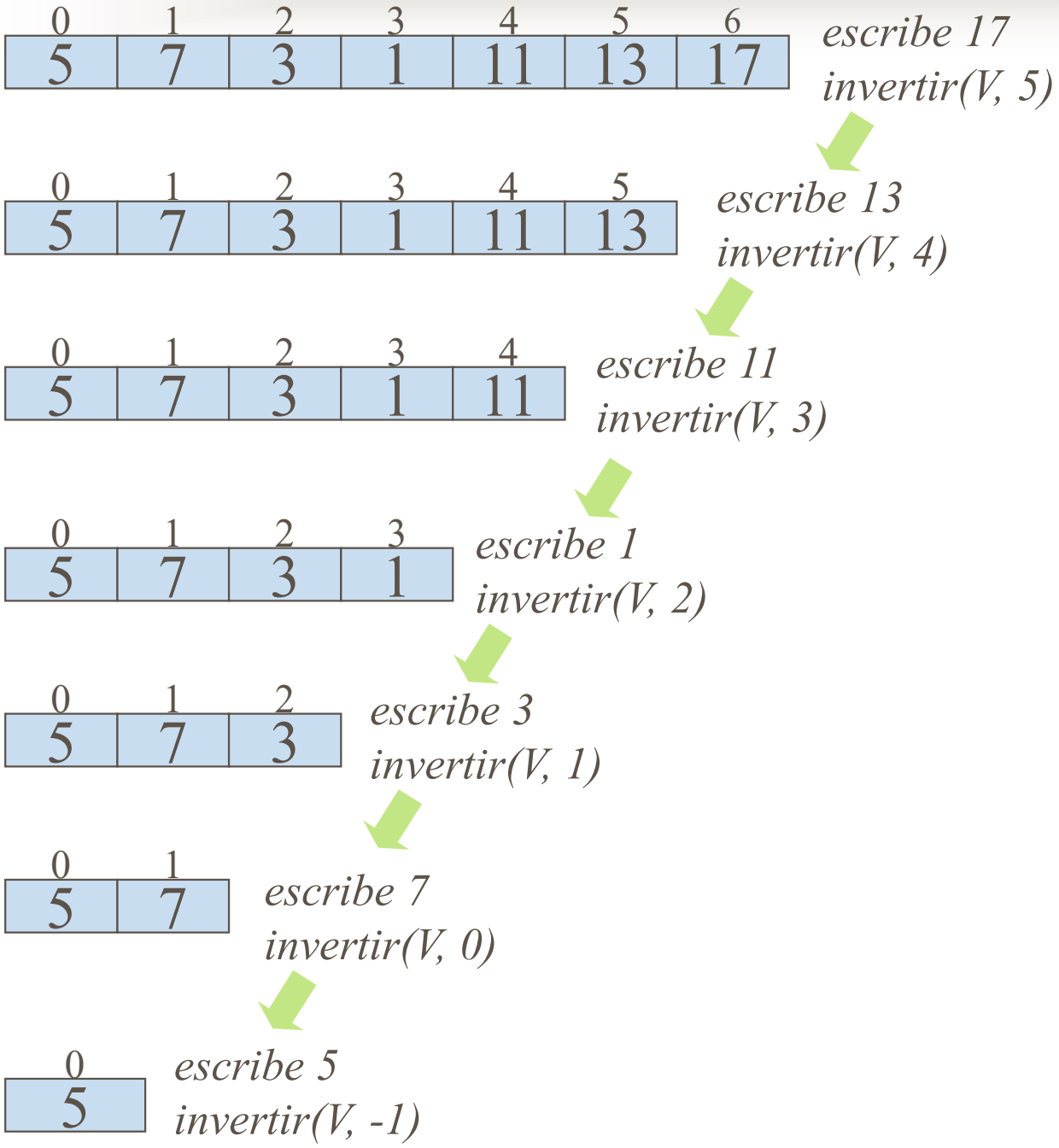

0 1 2 3 4 5 6
5 7 3 1 11 13 17
escribir(V, 6)
escribe 17
0 1 2 3 4 5
5 7 3 1 11 13
escribir(V, 5)
escribe 13
0 1 2 3 4
5 7 3 1 11
escribir(V, 4)
escribe 11
0 1 2 3
5 7 3 1
escribir(V, 3)
escribe 1
0 1 2
5 7 3
escribir(V, 2)
escribe 3
0 1
5 7
escribir(V, 1)
escribe 7
0
5
escribir(V, 0)
escribe 5

main.c;

recur.c;recur.h

Invertir un vector

```
void escribirVectorInvertido(int* V, int posicion)
{
  if(posicion>=0)
  {
    printf("\n\tVinvertido[%d]=%d", posicion, V[posicion]);
    escribirVectorInvertido(V, posicion-1);
  }
}
```


0 1 2 3 4 5 6
5 7 3 1 11 13 17
escribe 17
invertir(V, 5)
0 1 2 3 4 5
5 7 3 1 11 13
escribe 13
invertir(V, 4)
0 1 2 3 4
5 7 3 1 11
escribe 11
invertir(V, 3)
0 1 2 3
5 7 3 1
escribe 1
invertir(V, 2)
0 1 2
5 7 3
escribe 3
invertir(V, 1)
0 1
5 7
escribe 7
invertir(V, 0)
0
5
escribe 5
invertir(V, -1)

main.c;

recur.c;recur.h

búsqueda lineal recursiva

```
int busquedaLineal(int* V, int posicion, int elemento)
{
  if(posicion<0)
  {
    return(0); //Primer caso base
  }
  else
  {
    if(V[posicion]==elemento)
    {
      return(1); //Segundo caso base
    }
    else
    {
      return(busquedaLineal(V, posicion-1, elemento));
    }
  }
}
```

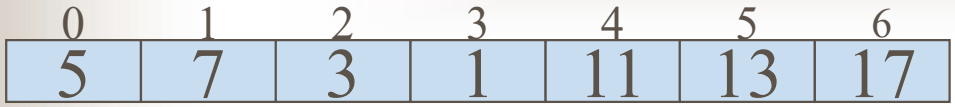
$$BusquedaLineal(V,n,b) = (V[n] == b) \text{ ó } (b \in \{V[0],\ldots,V[n-1]\})$$

$$BusquedaLineal(V,n,b) = Verdad \text{ si } V[n] = b\}$$

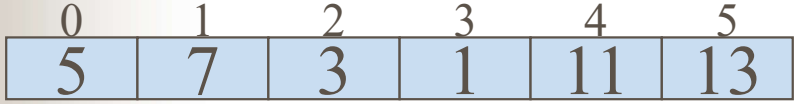
$$BusquedaLineal(V,n,b) = Falso \text{ si } V[0] \neq b$$

Dos casos base:

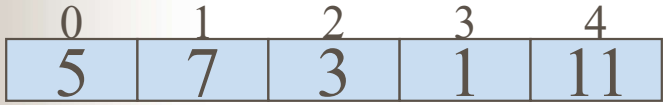
1. **Se ha recorrido todo el vector sin encontrar el elemento**
2. **Se ha encontrado el elemento**



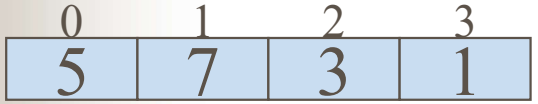
busqueda(V, 6, 7) =>Encontrado=>return(1)



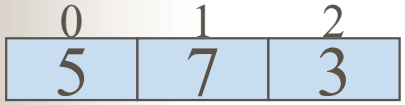
busqueda(V, 5, 7) =>Encontrado=>return(1)



busqueda(V, 4, 7)=>Encontrado=>return(1)

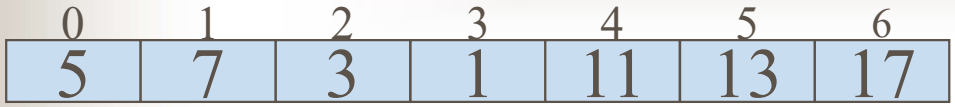


busqueda(V, 3, 7)=>Encontrado=>return(1)

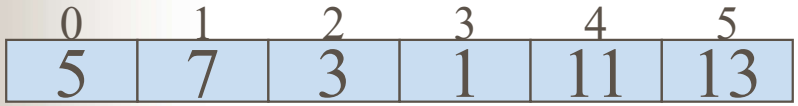


busqueda(V, 2, 7)=>Encontrado=>return(1)

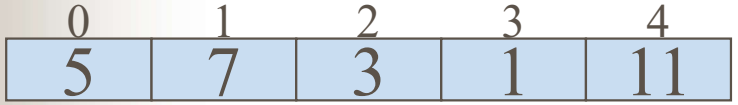
busqueda(V, 1, 7)=>Encontrado=>return(1)



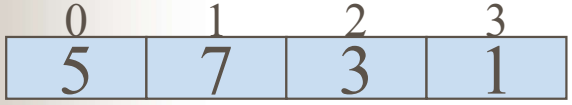
busqueda(V, 6, 2) =>!Encontrado=>return(0)



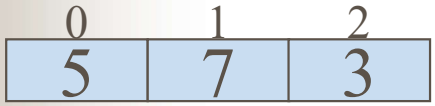
busqueda(V, 5, 2) =>!Encontrado=>return(0)



busqueda(V, 4, 2)=>!Encontrado=>return(0)

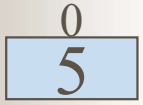


busqueda(V, 3, 2)=>!Encontrado=>return(0)



busqueda(V, 2, 2)=>!Encontrado=>return(0)

busqueda(V, 1, 2)=>!Encontrado=>return(0)



busqueda(V, 0, 2)=>!Encontrado=>return(0)

busqueda(V, -1, 2)=>!Encontrado=>return(0)

main.c;

recur.c;recur.h

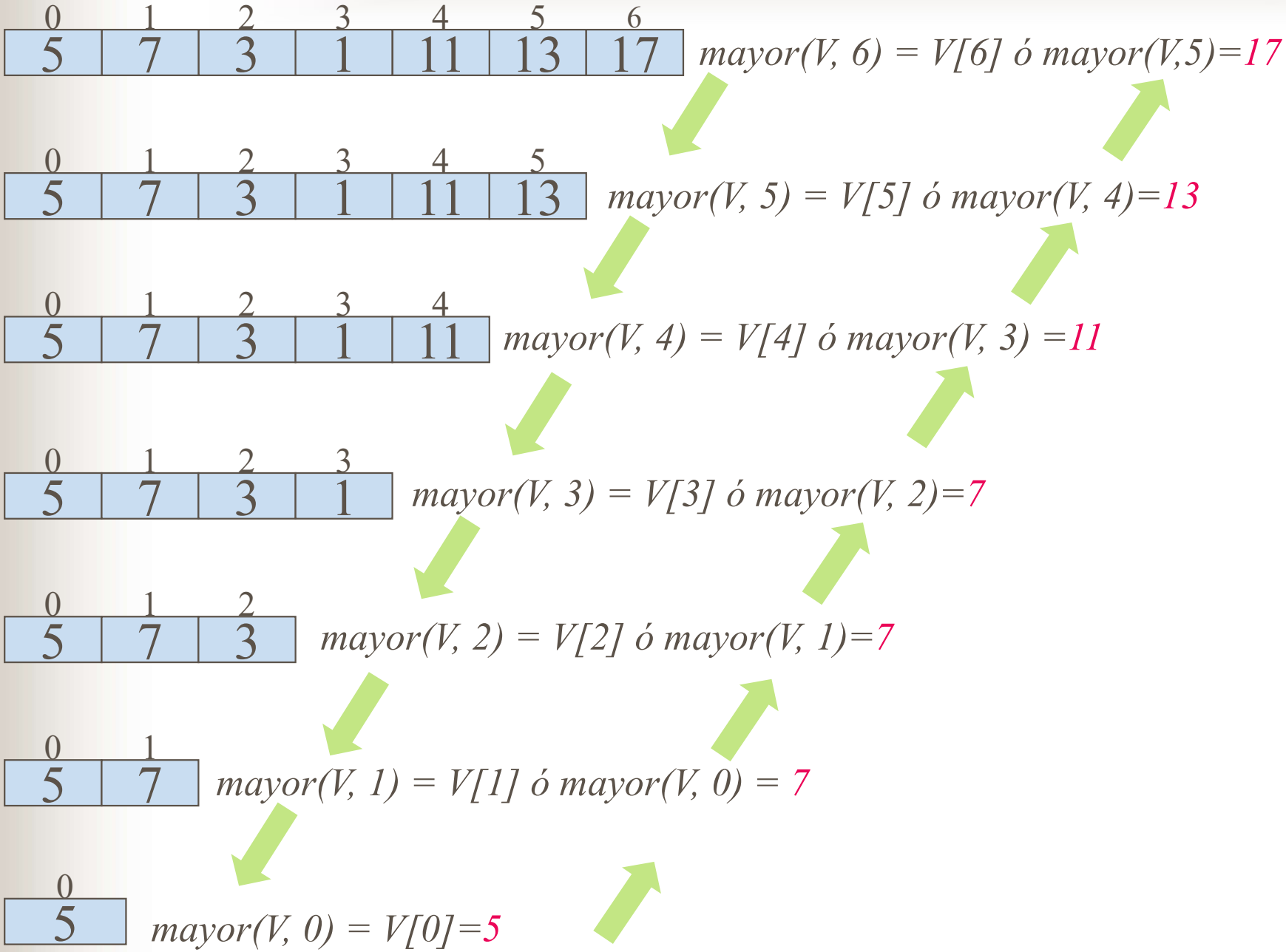
Elemento mayor de un vector

$$Mayor1(V,n) = V[n] \text{ ó } Mayor1(V,n-1)$$

$$Mayor1(V,0) = V[0]$$

```
int Mayor1 (int *V, int n) {
   int aux;

   if (n==0)
      return V[0];
   else {
      aux = Mayor1 (V, n-1);
      if (V[n]> aux)
         return V[n];
      else
         return aux;
}
```


0 1 2 3 4 5 6
5 7 3 1 11 13 17
mayor(V, 6) = V[6] ó mayor(V,5)=17
0 1 2 3 4 5
5 7 3 1 11 13
mayor(V, 5) = V[5] ó mayor(V, 4)=13
0 1 2 3 4
5 7 3 1 11
mayor(V, 4) = V[4] ó mayor(V, 3) =11
0 1 2 3
5 7 3 1
mayor(V, 3) = V[3] ó mayor(V, 2)=7
0 1 2
5 7 3
mayor(V, 2) = V[2] ó mayor(V, 1)=7
0 1
5 7
mayor(V, 1) = V[1] ó mayor(V, 0) = 7
0
5
mayor(V, 0) = V[0]=5

main.c;

recur.c;recur.h

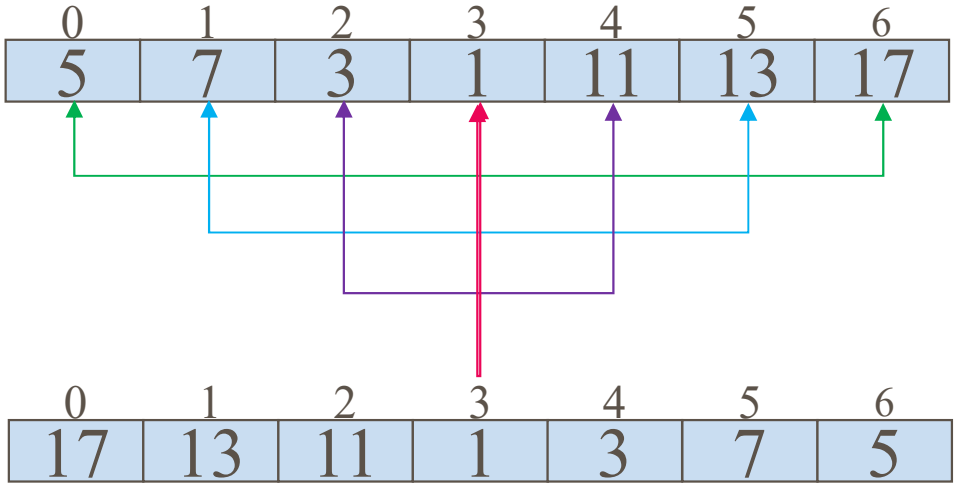
Los dos elementos mayores de un vector

```
void dosMayores(int* V, int posicion, int* a, int* b)
{
  if(posicion==1)
  {
    if(V[0]>V[1]){
      *a=V[0];
      *b=V[1];
    }
    else{
      *a=V[1];
      *b=V[0];
    }
  }
  else{
    dosMayores(V, posicion-1, a, b);
    if(V[posicion]>*a) //Caso1:-V[pos]--a--b
    {
      *b=*a;
      *a=V[posicion];
    }
    else if(V[posicion]>*b)//Caso2: a--V[pos]--b
    {
      *b=V[posicion];
    }
  }
}
```

Invertir un vector

Izda y dcha son las posiciones que se van a intercambiar → La llamada es ***invertirVector(V, 0, 6)***

```
void invertirVector(int* V, int izda, int dcha)
{       int aux;
        if (izda <= dcha)
        {   aux=V[izda];
            V[izda] = V[dcha];
            V[dcha] =aux;
            invertirVector(V, izda+1, dcha-1);
        }
}
```

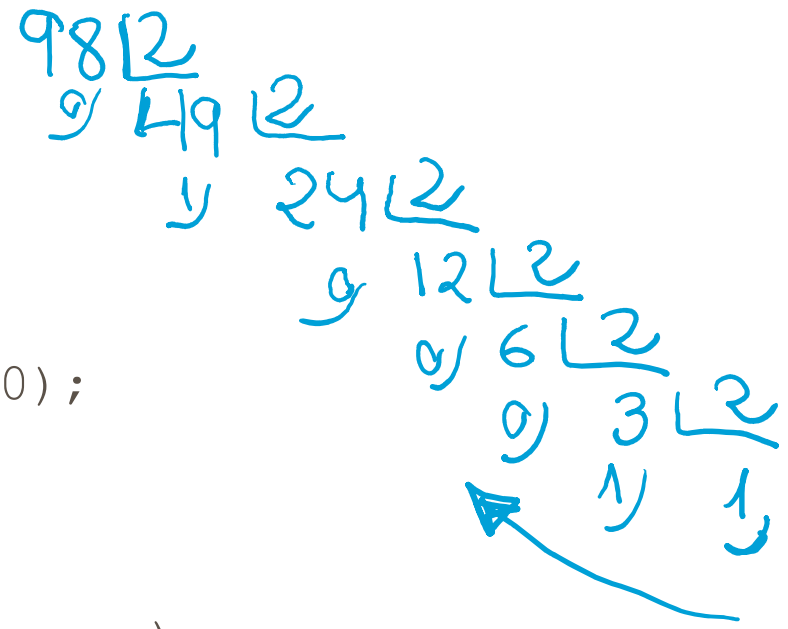


Determinar si una cadena es palindromo

```
int palindromo(char* cad, int izda, int dcha)
{
        if (izda > dcha)
           return(1);
        else
        {
            if (cad[izda]==cad[dcha])
               {
                  return(palindromo(cad, izda+1, dcha-1));
               }
            else
               {
                  return(0);
               }
        }
}
```

Pasar de decimal a binario

```
int decimalto2(int numero, char* res)
{
    if(numero==1)
     sprintf(res, "%d", 1);
    else
      if (numero==0)
      {
         sprintf(res, "%d", 0);
      }
      else
      {
         decimalto2(numero/2, res);
         sprintf(res, "%s%d", res, numero%2);
      }
}
```



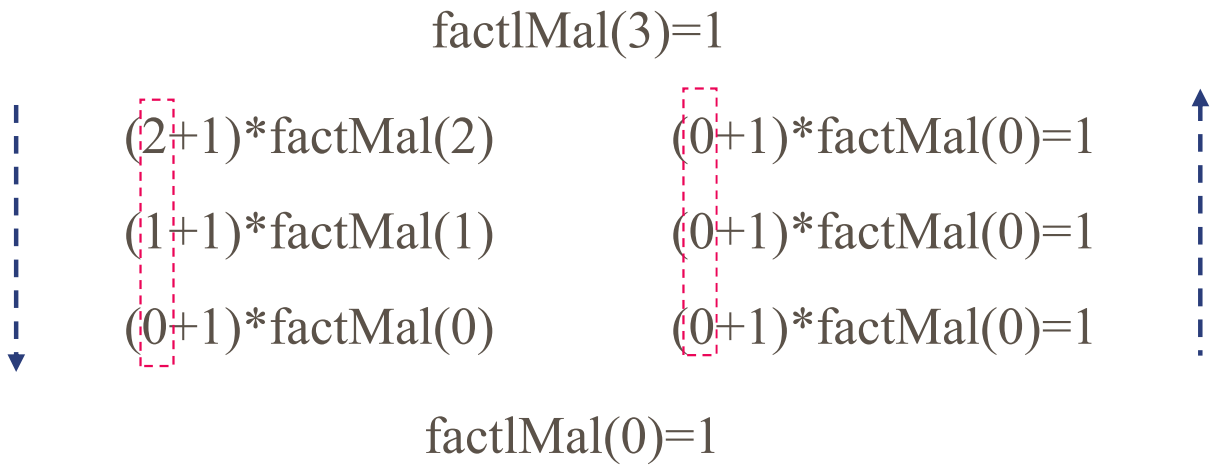
Paso de parámetros en recursión

- Hay que tener cuidado en el paso de parámetros en los módulos recursivos, ya que un paso inadecuado puede ocasionar que la función no actúe como nosotros esperamos
  - *n* es un parámetro que se pasa por referencia. Por tanto, en cada llamada recursiva, el valor de la variable es compartido por todas las llamadas recursivas y las modificaciones que se hagan en una de ellas afectarán al resto de llamadas

```
int factorial_mal (int *n)
{
  if (*n == 0) // caso base, terminar la recursion
    return (1);
  else { // continua la recursividad
    *n--;
    return ((*n+1) * factorial_mal (n));
  }
}
```

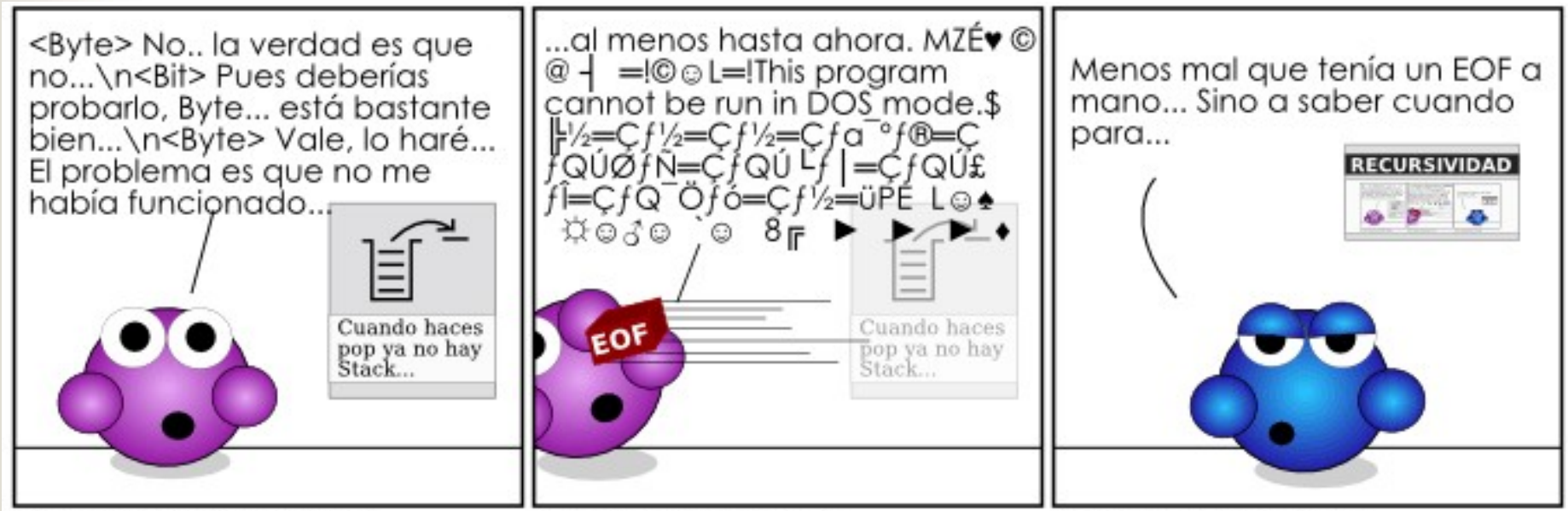
Paso de parámetros en recursión

```
int factorial_mal (int *n)
{
  if (*n == 0) // caso base, terminar la recursion
    return (1);
  else { // continua la recursividad
    *n--;
    return ((*n+1) * factorial_mal (n));
  }
}
```



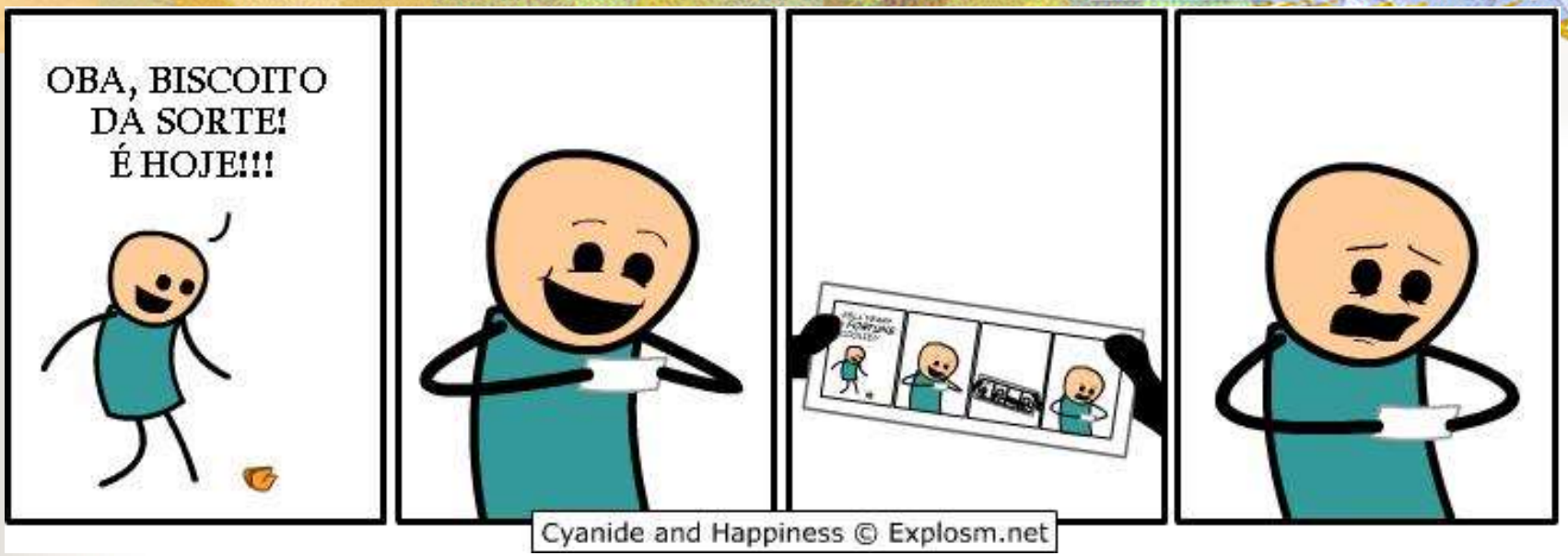
Conclusión, recursividad ¿Si o no?

- La utilización indiscriminada de la recursión para solucionar todo tipo de problemas no es una buena estrategia general, hay situaciones en que su uso no es recomendable
- **Cuando evitar la recursividad**. Si el problema planteado se resuelve fácilmente de forma iterativa, siempre se preferirá esta aproximación a una solución recursiva
  - La implementación es más rápida (no hay que controlar el paso de parámetros ni el funcionamiento en sí de la recursividad)
  - El algoritmo recursivo necesita memoria para almacenar los entornos de las distintas llamadas, y el iterativo no
  - La llamada a un subprograma siempre requiere un tiempo de ejecución adicional (salto a la zona de código del subprograma, inclusión del registro de activación en la pila de control, paso de parámetros, etc.)
  - La iteración evita más fácilmente tener que calcular dos veces el mismo valor
- **Cuando utilizar la recursividad**. Hay problemas que presentan una resolución a través de un algoritmo recursivo inmediata, siendo la solución no recursiva mucho más compleja. ¿Cómo podremos detectar dichos problemas? Sólo la experiencia nos permitirá *reconocer* dichos problemas

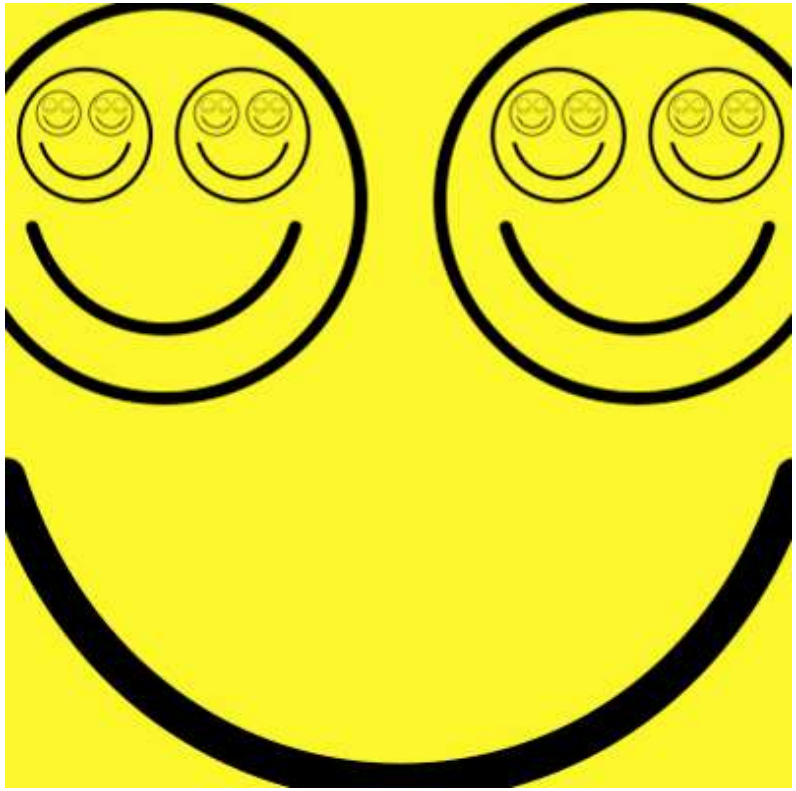


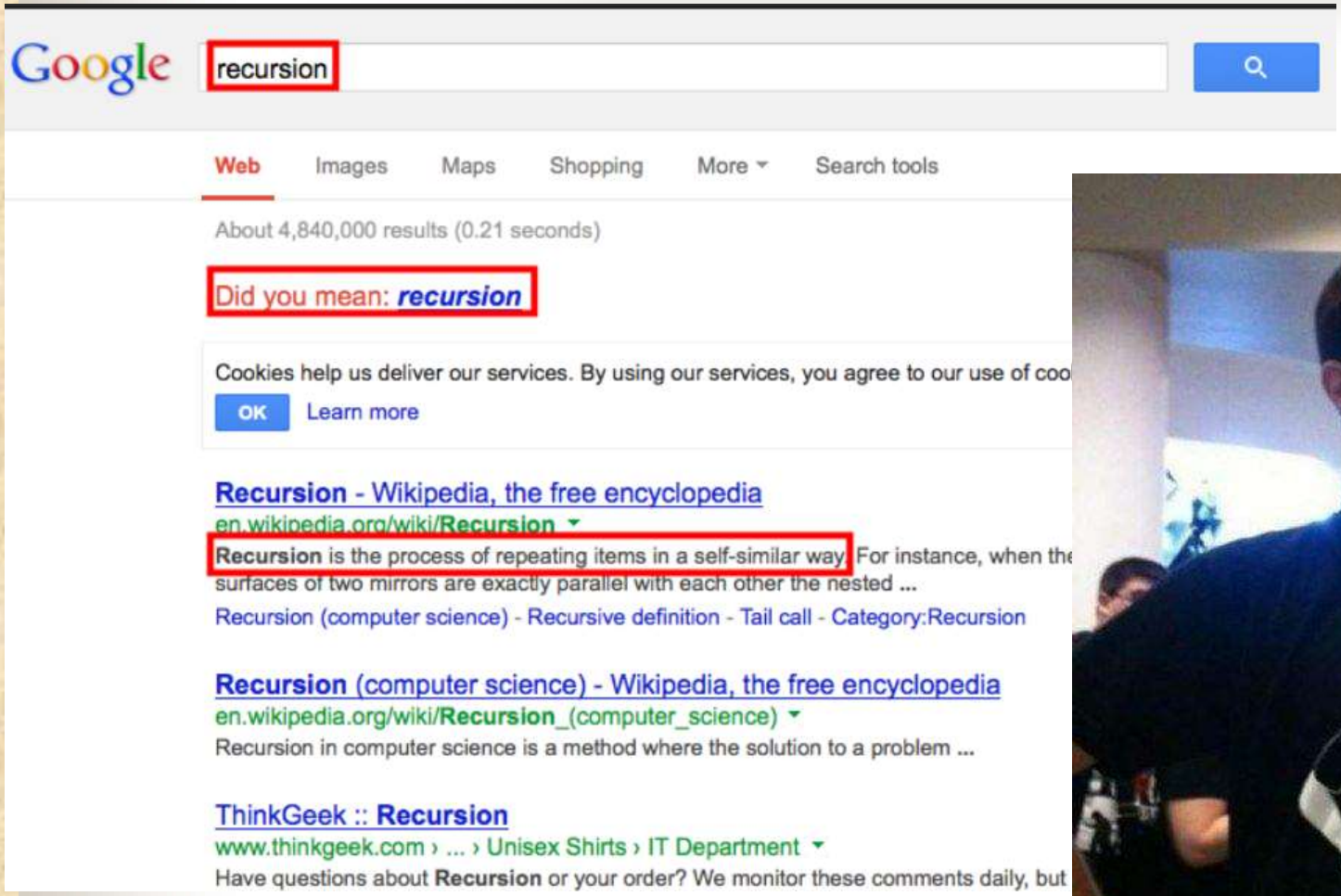
La tira cómica de Bit y Byte http://tira.emezeta.com Idea: Geekdraz y Manz


¿Eres Feliz, cariño?
Soy Feliz si tú eres Feliz
Y yo sólo soy Feliz si tú también lo eres
Espera, creo que en algún momento hemos olvidado la condición de salida de esta recursión


OBA, BISCOITO DA SORTE! É HOJE!!!
Cyanide and Happiness © Explosm.net


MARTILLO ROMPECRISTALES
ROMPER EL CRISTAL PARA ACCEDER AL MARTILLO




Google
recursion
Web
Images
Maps
Shopping
More
Search tools
About 4,840,000 results (0.21 seconds)
Did you mean: recursion
Cookies help us deliver our services. By using our services, you agree to our use of coo
OK
Learn more
Recursion - Wikipedia, the free encyclopedia
en.wikipedia.org/wiki/Recursion
Recursion is the process of repeating items in a self-similar way. For instance, when the
surfaces of two mirrors are exactly parallel with each other the nested ...
Recursion (computer science) - Recursive definition - Tail call - Category:Recursion
Recursion (computer science) - Wikipedia, the free encyclopedia
en.wikipedia.org/wiki/Recursion_(computer_science)
Recursion in computer science is a method where the solution to a problem ...
ThinkGeek :: Recursion
www.thinkgeek.com › ... › Unisex Shirts › IT Department
Have questions about Recursion or your order? We monitor these comments daily, but