

# **TEMA 7: RECURSIVIDAD**

ALGORITMOS Y ESTRUCTURAS DE DATOS

M. Colebrook Santamaría

J. Riera Ledesma

J. Hernández Aceituno

# Objetivos

- Concepto de recursividad
- Diseño de algoritmos recursivos
- Ejecución de un módulo recursivo
- Ejemplo de funciones recursivas
- Ejemplos más complejos
- Simulación de recursividad mediante una pila
- ¿Recursividad o iteración?

# Concepto de recursividad (1)

- La recursividad constituye una de las herramientas más potentes en programación.
- Una función que se llama a sí misma se denomina **recursiva**.
- Por ejemplo, tenemos la siguiente definición recursiva para calcular el **factorial** de un número entero:

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

- **Ventaja:** no es necesario definir la secuencia de pasos exacta para resolver el problema.
- **Desventaja:** podría ser menos eficiente.

# Concepto de recursividad (2)

- Para que una **definición recursiva** esté completamente identificada es necesario tener un **caso base**, que no se calcule utilizando casos anteriores, y que la división del problema converja a ese caso base.
- En el ejemplo anterior del factorial, el caso base es:  $0! = 1$
- Si aplicamos la definición recursiva del factorial a  $n = 4$ , obtenemos la siguiente traza:

$$\begin{aligned} 4! &= 4 \cdot 3! \\ &= 4 \cdot 3 \cdot 2! \\ &= 4 \cdot 3 \cdot 2 \cdot 1! \\ &= 4 \cdot 3 \cdot 2 \cdot 1 \cdot 0! \\ &= 4 \cdot 3 \cdot 2 \cdot 1 \cdot 1 \\ &= 4 \cdot 3 \cdot 2 \cdot 1 \\ &= 4 \cdot 3 \cdot 2 \\ &= 4 \cdot 6 \\ &= 24 \end{aligned}$$

# Concepto de recursividad (3)

- Otro ejemplo:

$$x^n = \begin{cases} 1 & \text{si } n = 0 \\ x \cdot x^{n-1} & \text{si } n > 0 \end{cases}$$

- Si aplicamos esta definición con  $x = 2$ , y  $n = 4$ , tenemos que:

$$\begin{aligned} 2^4 &= 2 \cdot 2^3 \\ &= 2 \cdot 2 \cdot 2^2 \\ &= 2 \cdot 2 \cdot 2 \cdot 2^1 \\ &= 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2^0 \\ &= 2 \cdot 2 \cdot 2 \cdot 2 \cdot 1 \\ &= 2 \cdot 2 \cdot 2 \cdot 2 \\ &= 2 \cdot 2 \cdot 4 \\ &= 2 \cdot 8 \\ &= 16 \end{aligned}$$

# Diseño de algoritmos recursivos (1)

- Para poder resolver un problema de forma recursiva, el primer paso será el diseño de un **algoritmo recursivo**.
- Para ello, hay que descomponer el problema de forma que su solución quede definida en **función de ella misma**, pero para un tamaño menor, además de incluir la tarea para el **caso base**.
- Por tanto, tendremos que diseñar:
  - Caso base
  - Caso general
  - Solución en términos de dichos casos

# Diseño de algoritmos recursivos (2)

- **Caso base:** son los casos del problema que se resuelven con un segmento de código sin recursividad.

Siempre debe existir **al menos** un caso base

El número y forma de los casos base son arbitrarios. La solución será mejor cuanto más simple y eficiente resulte el conjunto de casos seleccionados.

# Diseño de algoritmos recursivos (3)

- **Casos generales:** si el problema es suficientemente complejo, la solución se expresa de forma recursiva como la **unión** de:
  1. La solución de **uno o más subproblemas**, de igual naturaleza pero **menor tamaño**.
  2. Un conjunto de pasos adicionales. Estos pasos adicionales, junto con las soluciones a los subproblemas, componen la solución al problema general que queremos resolver.

Los casos generales siempre deben **avanzar** hacia un caso base.



# Diseño de algoritmos recursivos (4)

```
unsigned int factorial(unsigned int n)
{
    unsigned int resultado;

    // caso base
    if (n == 0) resultado = 1;
    // caso general
    else      resultado = n * factorial(n - 1);

    return resultado;
}
```

# Diseño de algoritmos recursivos (5)

```
unsigned int factorial(unsigned int n)
{
    // caso base
    if (n == 0) return 1;
    // caso general
    else return n * factorial(n - 1);
}
```

- ¿Se puede simplificar aún más este código?

# Ejecución de un módulo recursivo (1)

- En general, en la **pila del sistema** se almacena el entorno asociado a las distintas funciones que se van activando.
- En particular, en un **módulo recursivo**, cada llamada recursiva genera una **nueva zona de memoria** en la **pila del sistema**, la cual es independiente del resto de llamadas.

# Ejecución de un módulo recursivo (1)

- Por ejemplo, la ejecución del factorial:

1. Dentro del factorial, cada llamada del tipo:

$n * \text{factorial}(n - 1)$

genera una nueva zona de memoria en la pila del sistema.

2. El proceso anterior se repite hasta que la condición del caso base se cumple:

- Se ejecuta la sentencia de retorno del valor 1.
- Empieza la vuelta atrás de la recursión, se evalúan las expresiones y se ejecutan los **return** que estaban pendientes.

# Ejecución de un módulo recursivo (2)

```
factorial(3); // n = 3
```

```
return 3 * factorial(2); // n = 2
```

```
return 2 * factorial(1); // n = 1
```

```
return 1 * factorial(0); // n = 0
```

```
return 1;
```

```
// 1 * 1 = 1
```

```
// 2 * 1 = 2
```

```
// 3 * 2 = 6
```

```
factorial(3)  = 3 * factorial(2)
               = 3 * 2 * factorial(1)
               = 3 * 2 * 1 * factorial(0)
               = 3 * 2 * 1 * 1 = 6
```

# Ejemplos de funciones recursivas (1)

- Cálculo de la potencia de un número real  $x$  y un entero no negativo  $n$ :

$$x^n = \begin{cases} 1 & \text{si } n = 0 \\ x \cdot x^{n-1} & \text{si } n > 0 \end{cases}$$

# Ejemplos de funciones recursivas (1)

- Cálculo de la potencia de un número real  $x$  y un entero no negativo  $n$ :

$$x^n = \begin{cases} 1 & \text{si } n = 0 \\ x \cdot x^{n-1} & \text{si } n > 0 \end{cases}$$

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

# Ejemplos de funciones recursivas (2)

- Cálculo del producto de un entero  $a$  por otro entero no negativo  $b$ :

$$\text{producto}(a, b) = \begin{cases} 0 & \text{si } b = 0 \\ a + \text{producto}(a, b - 1) & \text{si } b > 0 \end{cases}$$



# Ejemplos de funciones recursivas (2)

- Cálculo del producto de un entero  $a$  por otro entero no negativo  $b$ :

$$\text{producto}(a, b) = \begin{cases} 0 & \text{si } b = 0 \\ a + \text{producto}(a, b - 1) & \text{si } b > 0 \end{cases}$$

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

# Ejemplos de funciones recursivas (3)

- Suma recursiva de los elementos de un vector:

$$\sum_{i=0}^{n-1} V[i] = \text{sumatorio}(V, n) = \begin{cases} V[0] & \text{si } n = 1 \\ V[n-1] + \text{sumatorio}(V, n-1) & \text{si } n > 1 \end{cases}$$

# Ejemplos de funciones recursivas (3)

- Suma recursiva de los elementos de un vector:

$$\sum_{i=0}^{n-1} V[i] = \text{sumatorio}(V, n) = \begin{cases} V[0] & \text{si } n = 1 \\ V[n-1] + \text{sumatorio}(V, n-1) & \text{si } n > 1 \end{cases}$$

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

```
int n = 10, *V = new int[n];  
// rellenar V  
cout << sumatorio(V, n) << endl;  
delete[] V;
```

# Ejemplos de funciones recursivas (4)

- Búsqueda recursiva del máximo elemento de un vector:

$$\text{máximo}(V, n) = \begin{cases} V[0] & \text{si } n = 1 \\ \max(V[n - 1], \text{máximo}(V, n - 1)) & \text{si } n > 1 \end{cases}$$

# Ejemplos de funciones recursivas (4)

- Búsqueda recursiva del máximo elemento de un vector:

$$\text{máximo}(V, n) = \begin{cases} V[0] & \text{si } n = 1 \\ \max(V[n - 1], \text{máximo}(V, n - 1)) & \text{si } n > 1 \end{cases}$$

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

# Ejemplos de funciones recursivas (5)

- Sucesión de Fibonacci:

$$\text{Fib}(n) = \begin{cases} 1 & \text{si } n \leq 1 \\ \text{Fib}(n - 1) + \text{Fib}(n - 2) & \text{si } n > 1 \end{cases}$$

# Ejemplos de funciones recursivas (5)

- Sucesión de Fibonacci:

$$\text{Fib}(n) = \begin{cases} 1 & \text{si } n \leq 1 \\ \text{Fib}(n - 1) + \text{Fib}(n - 2) & \text{si } n > 1 \end{cases}$$

```
unsigned fib(unsigned n) {  
    if (n <= 1) return 1;  
    else      return fib(n-1) + fib(n-2);  
}
```

# Ejemplos de funciones recursivas (6)

```
unsigned fib(unsigned n) { // enfoque recursivo
    if (n <= 1) return 1;
    else      return fib(n-1) + fib(n-2);
}
```

```
unsigned fib(unsigned n) { // enfoque iterativo
    unsigned anterior1 = 1, anterior2 = 1, actual = 1;
    for (unsigned i = 2; i <= n; i++) {
        actual = anterior1 + anterior2;
        anterior1 = anterior2;
        anterior2 = actual;
    }
    return actual;
}
```



# Ejemplos de funciones recursivas (7)

- Coeficiente binomial:

$$C(n, k) = \begin{bmatrix} n \\ k \end{bmatrix} = \begin{cases} 1 & \text{si } k = 0 \text{ ó } k = n \\ C(n - 1, k - 1) + C(n - 1, k) & \text{en otro caso} \end{cases}$$

# Ejemplos de funciones recursivas (7)

## ■ Coeficiente binomial:

$$C(n, k) = \begin{bmatrix} n \\ k \end{bmatrix} = \begin{cases} 1 & \text{si } k = 0 \text{ ó } k = n \\ C(n - 1, k - 1) + C(n - 1, k) & \text{en otro caso} \end{cases}$$

```
unsigned int C(unsigned int n, unsigned int k) {  
    if (k == 0 || k == n) return 1;  
    else return C(n-1, k-1) + C(n-1, k);  
}
```

# Ejemplos más complejos: BBR (1)

- **Búsqueda binaria recursiva (BBR):** dado un vector ordenado  $v$  de forma ascendente, la búsqueda de un elemento  $x$  entre dos posiciones ( $i$ : izquierda,  $d$ : derecha) se puede realizar comparando el valor buscado  $x$  con el elemento central  $c = (i+d)/2$ .
  - Si  $v[c] == x$ , se devuelve la posición  $c$ .
  - Si  $x < v[c]$ , la búsqueda continúa en el subvector izquierdo  $v[i..c-1]$ .
  - Si  $v[c] < x$ , la búsqueda continúa en el subvector derecho  $v[c+1..d]$ .

# Ejemplos más complejos: BBR (2)

- La función de BBR retorna la **posición** en el vector **v** en la que se encuentra **x** o, si no se encuentra, retorna **-1**. La cabecera de la función es:

```
int BBR(int v[], int i, int d, int x)
```

- Por tanto, el algoritmo recursivo debería ser:
  1. Si  $i \leq d$ , calcular la posición  $c = \lfloor (i+d)/2 \rfloor$ .
  2. Si  $v[c] == x$ , retornar  $c$  (caso base de **éxito**).
  3. Si  $x < v[c]$ , buscar  $x$  en el subvector izquierdo:  
 $\text{BBR}(v, i, c-1, x)$
  4. Si  $v[c] < x$ , buscar  $x$  en el subvector derecho:  
 $\text{BBR}(v, c+1, d, x)$
  5. Si  $i > d$ , retornar  $-1$  (caso base de **fracaso**)

# Ejemplos más complejos: BBR (3)

- Vamos a aplicar el algoritmo anterior a este vector  $v$  para encontrar el valor  $x = 21$ .

	0	1	2	3	4	5	6	7	8	9
$v:$	1	2	3	5	8	13	21	34	55	89
$v[c] < x:$	$i$				$c$					$d$
$x < v[c]:$						$i$		$c$		$d$
$v[c] < x:$						$c=i$	$d$			
ÉXITO:							$c=i=d$			

# Ejemplos más complejos: BBR (4)

- Si volvemos a aplicar el mismo algoritmo para buscar el valor  $x = 22$ , nos retornaría **-1**.

	0	1	2	3	4	5	6	7	8	9
v:	1	2	3	5	8	13	21	34	55	89
v[c] < x:	i				c					d
x < v[c]:						i		c		d
v[c] < x:						c=i	d			
v[c] < x:							c=i=d			
<b>FRACASO:</b>							d	i		

# Ejemplos más complejos: BBR (5)

```
int BBR(int v[], int i, int d, int x)
{
    if (i > d) return -1;
    int c = (i + d) / 2;
    if (v[c] == x) return c;
    if (x < v[c]) return BBR(v, i, c - 1, x);
    if (v[c] < x) return BBR(v, c + 1, d, x);
}
```

# Ejemplos más complejos: OR (1)

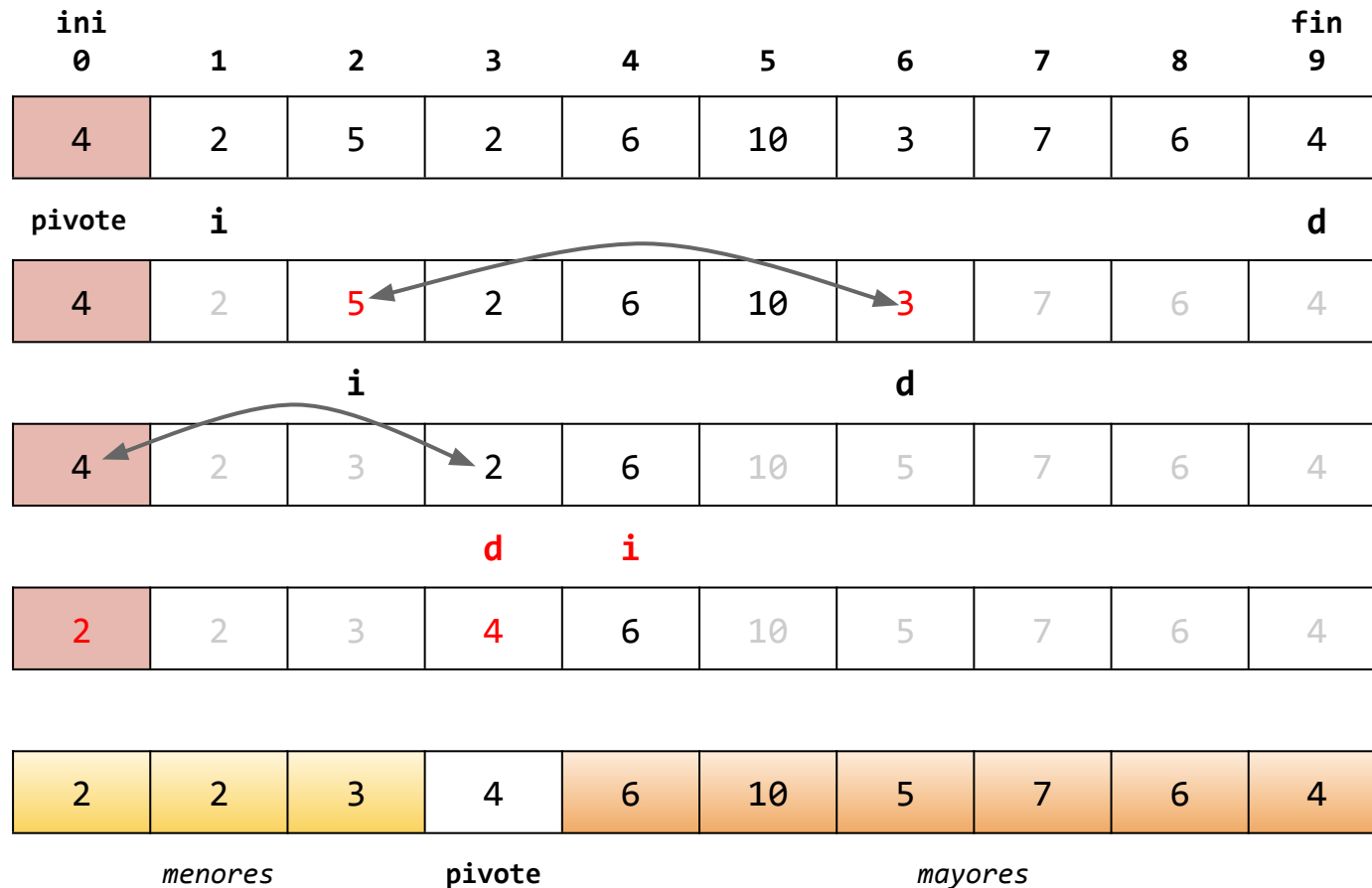
## ■ Ordenación Rápida (OR):

1. Se toma un **elemento arbitrario** del vector, al que denominaremos **pivote**. Sea  $p$  su valor.
2. Se recorre el vector de **izquierda a derecha** hasta encontrar un elemento situado en una posición  $i$  tal que  $v[i] > p$ .
3. Se recorre el vector de **derecha a izquierda** hasta encontrar un elemento situado en una posición  $d$  tal que  $v[d] < p$ .
4. Una vez localizados, se intercambian los elementos situados en las casillas  $i$  y  $d$ , de forma que los valores quedarían  $v[i] < p < v[d]$ .
5. Repetir hasta que los dos recorridos se encuentren ( $d < i$ ).
6. Finalmente, colocamos el pivote en el sitio que le corresponde (si se tomó  $v[0]$  como  $p$ , basta intercambiarlo con  $v[d]$ ). El vector queda dividido en dos subvectores, a los que se aplicaría el mismo proceso.



## Ejemplos más complejos: OR (2)

■ Ejemplo:



# Ejemplos más complejos: OR (3)

## Implementación iterativa

```
int partir(int *v, int ini, int fin)
{ int i = ini + 1, d = fin, p = v[ini];
  while (i <= d) {
    while (i <= d && v[i] <= p) i++;
    while (i <= d && p <= v[d]) d--;
    if (i < d) swap(v[i++], v[d--]);
  }
  swap(v[ini], v[d]);
  return d;
}
```

# Ejemplos más complejos: OR (4)

## Implementación Recursiva

```
void OR(int *v, int i, int d) {  
    if (i < d) {  
        int pivote = partir(v, i, d);  
        OR(v, i, pivote - 1);  
        OR(v, pivote + 1, d);  
    }  
}  
  
int *v = new int[n];  
// cargar datos en v  
OR(v, 0, n - 1);  
// mostrar los datos de v ordenados  
delete[] v;
```

# Simulación de recursividad mediante una pila (1)

```
unsigned factorial(unsigned n) {  
    unsigned resultado, siguiente;  
    if (n == 0) resultado = 1;  
    else {  
        cout << "push " << n << endl;  
        siguiente = factorial(n - 1);  
        cout << "pop " << siguiente << endl;  
        cout << "pop " << n << endl;  
        resultado = n * siguiente;  
    }  
    cout << "push " << resultado << endl;  
    return resultado;  
}
```

factorial(3):

```
push 3  
push 2  
push 1  
push 1  
pop 1  
pop 1  
push 1  
pop 1  
pop 2  
push 2  
pop 2  
pop 3  
push 6
```

- ¿Cómo puedo simular este código con una pila?

# Simulación de recursividad mediante una pila (2)

```
unsigned factorial(unsigned n) {  
    stack_t<unsigned> pila;  
    unsigned resultado;  
    while (n > 0)  
        pila.push(n--);  
    pila.push(1); // factorial(0)  
    while (!pila.empty()) {  
        resultado = pila.top(); pila.pop(); // siguiente  
        if (!pila.empty()) {  
            n = pila.top(); pila.pop();  
            pila.push(n * resultado);  
        }  
    }  
    return resultado;  
}
```

# ¿Recursividad o iteración?

A la hora de diseñar un algoritmo recursivo, debemos tener en cuenta:

- El **coste computacional** (tiempo+espacio) asociado a una llamada a una función y el retorno a la función que hace la llamada.
- Algunas soluciones recursivas pueden hacer que la solución para un determinado tamaño del problema se calcule **varias veces**.
- Muchos problemas recursivos tienen como caso base la resolución del problema para un tamaño muy reducido. En ocasiones resulta **excesivamente** pequeño.
- La **solución iterativa** (igual de eficiente) puede ser muy compleja de encontrar.
- La **solución recursiva** es muy concisa, legible y elegante.

# Referencias

- ★ Olsson, M. (2018), “C++ 17 Quick Syntax Reference”, Apress. Disponible en PDF en la BBTK-ULL:  
<https://doi.org/10.1007/978-1-4842-3600-0>
- ★ Stroustrup, B. (2002), “El Lenguaje de Programación C++”, Addison Wesley.
- ★ C++ Syntax Highlighting (código en colores):  
[tohtml.com/cpp](http://tohtml.com/cpp)
- ★ Ecuaciones editadas con:  
[s1.daumcdn.net/editor/fp/service\\_nc/pencil/Pencil\\_chromestore.html](http://s1.daumcdn.net/editor/fp/service_nc/pencil/Pencil_chromestore.html)