



# Fundamentos de Programación

## Tema 5 Recursividad

Javier Abad (abad@decsai.ugr.es)

Dpto. de Ciencias de la Computación e Inteligencia Artificial

<http://decsai.ugr.es>

# Contenidos

---

1. Concepto de recursividad
2. Diseño de algoritmos recursivos
3. Funciones recursivas
  1. Definición
  2. Ejecución
  3. Traza
4. Ejemplos
5. ¿Recursividad o iteración?
6. Problemas de eficiencia en la recursividad
  1. Recursividad de cola
  2. Repetición de cálculos
  3. Casos base excesivamente pequeños
  4. Excesivas llamadas recursivas
7. Ejemplo avanzado: Quicksort

# Objetivos

---

- ▶ Entender el concepto de recursividad.
- ▶ Conocer los fundamentos del diseño de algoritmos recursivos.
- ▶ Comprender la ejecución de algoritmos recursivos.
- ▶ Aprender a realizar trazas de algoritmos recursivos.
- ▶ Comprender las ventajas e inconvenientes de la recursividad frente a las soluciones iterativas.
- ▶ Entender los problemas de eficiencia que puede generar una solución recursiva.

# Concepto de recursividad

- ▶ La recursividad constituye una de las herramientas más potentes en programación. Es un concepto ya conocido. Por ejemplo:

- ▶ Definición recursiva:

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

- ▶ Demostración por inducción:
  - ▶ Demostrar la proposición para un caso base
  - ▶ Se demuestra que si se supone cierta la proposición para un tamaño,  $n$ , entonces también lo es para tamaños mayores que  $n$ .

## Recursividad

En programación de ordenadores, una función que se llama a sí misma se denomina recursiva.

# Concepto de recursividad

---

- ▶ En programación, podremos usar la recursividad si la solución de un problema se formula en términos de un problema de la misma naturaleza, aunque de menor tamaño, y conocemos la solución no recursiva para un determinado caso.
- ▶ Ventajas e inconvenientes de la recursividad:
  - ▶ Ventajas: No es necesario definir la secuencia de pasos exacta para resolver el problema. Podemos considerar que "tenemos resuelto" el problema (de menor tamaño).
  - ▶ Desventajas: Tenemos que encontrar una solución recursiva y, además, podría ser menos eficiente.

# Concepto de recursividad

- ▶ Para que una definición recursiva esté completamente identificada es necesario tener un caso base que no se calcule utilizando casos anteriores y que la reducción del problema converja a ese caso base.

$$0! = 1$$

- ▶ Ejemplos:

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

Sucesión de Fibonacci de orden 2:

$$Fib(0) = 0$$

$$Fib(1) = 1$$

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

# Concepto de recursividad

- Ejemplo: cálculo del factorial con  $n=3$

$$3! = 3 * 2!$$


$$2! = 2 * 1!$$

$$1! = 1 * 0!$$

Caso base  $\Rightarrow 0! = 1$

$$3! = 3 * 2 = 6$$

$$2! = 2 * 1 = 2$$

$$1! = 1 * 1 = 1$$

# Diseño de algoritmos recursivos

---

- ▶ El primer paso será la identificación del algoritmo recursivo, es decir, la descomposición del problema en subproblemas de menor tamaño (aunque de la misma naturaleza del problema original) y la composición la solución final a partir de las subsoluciones obtenidas.

Por lo tanto, debemos diseñar:

- ▶ **Casos base:** son los casos del problema que se resuelve con un segmento de código no recursivo.

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

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



# Diseño de algoritmos recursivos

---

- ▶ **Casos generales:** debemos encontrar la solución a un problema (o varios) de la misma naturaleza, pero de menor tamaño (subproblemas).
- ▶ **Los casos generales siempre deben avanzar hacia un caso base.**  
Esto es, la llamada recursiva se hace a un subproblema más pequeño y, en última instancia, los casos generales deberán alcanzar un caso base.
- ▶ **Composición:** una vez halladas las soluciones a los subproblemas, ejecutaremos un conjunto de pasos adicionales que, junto con las subsoluciones, componen la solución al problema general que queremos resolver.

# Diseño de algoritmos recursivos

- ▶ Ejemplo: solución recursiva al cálculo de potencias enteras,  $x^n$ 
  - ▶ Caso base:  $x^0 = 1$
  - ▶ Caso general:  $x^{n-1}$
  - ▶ Composición:  $x^n = x * x^{n-1}$
- ▶ Ejemplo: solución recursiva a la sumatoria  $\sum_{i=1}^n i$ 
  - ▶ Caso base:  $\sum_{i=1}^1 i = 1$
  - ▶ Caso general:  $\sum_{i=1}^{n-1} i$
  - ▶ Composición:  $\sum_{i=1}^n i = n + \sum_{i=1}^{n-1} i$
- ▶ Ejemplo: solución recursiva al producto de enteros,  $a * b$ 
  - ▶ Caso base:  $0 * b = 0, a * 0 = 0$
  - ▶ Caso general:  $a * (b-1)$
  - ▶ Composición:  $a * b = a + a * (b-1)$

# Funciones recursivas: definición

- ▶ ¿Cómo se traduce a un lenguaje de programación la solución recursiva de un problema? A través de una función que, dentro del código que la define, realiza una o más llamadas a sí misma. La clave está, claro, en los argumentos de la llamada.

## Función recursiva

Una función que se llama a sí misma.

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

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

**Solución estructurada**

```
int factorial(int n){

    if (n==0) //Caso base
        return 1;
    else      //Caso general
        return n * factorial(n-1);
}
```

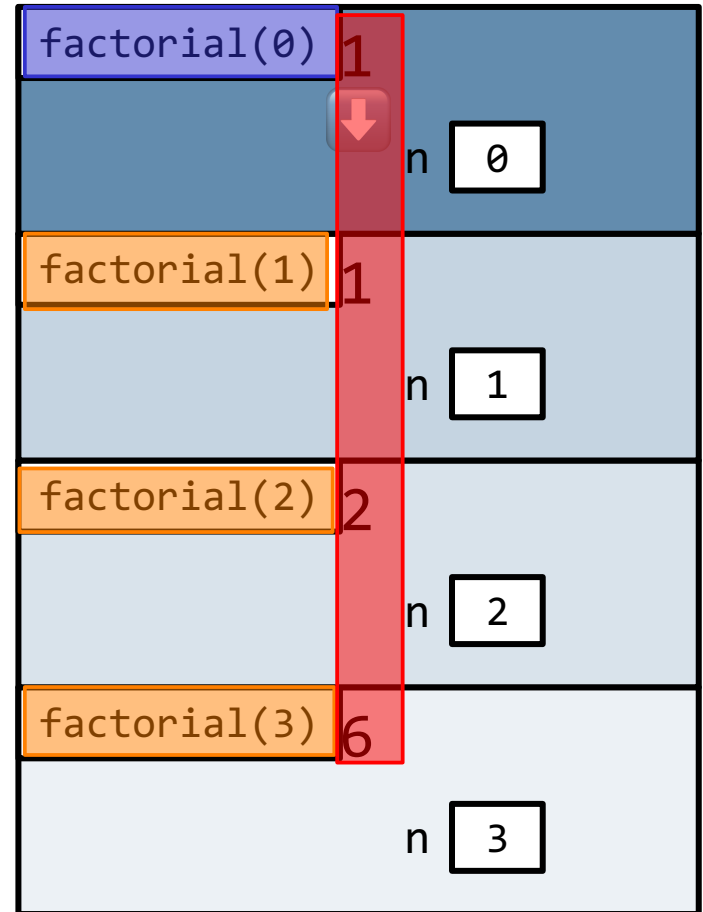
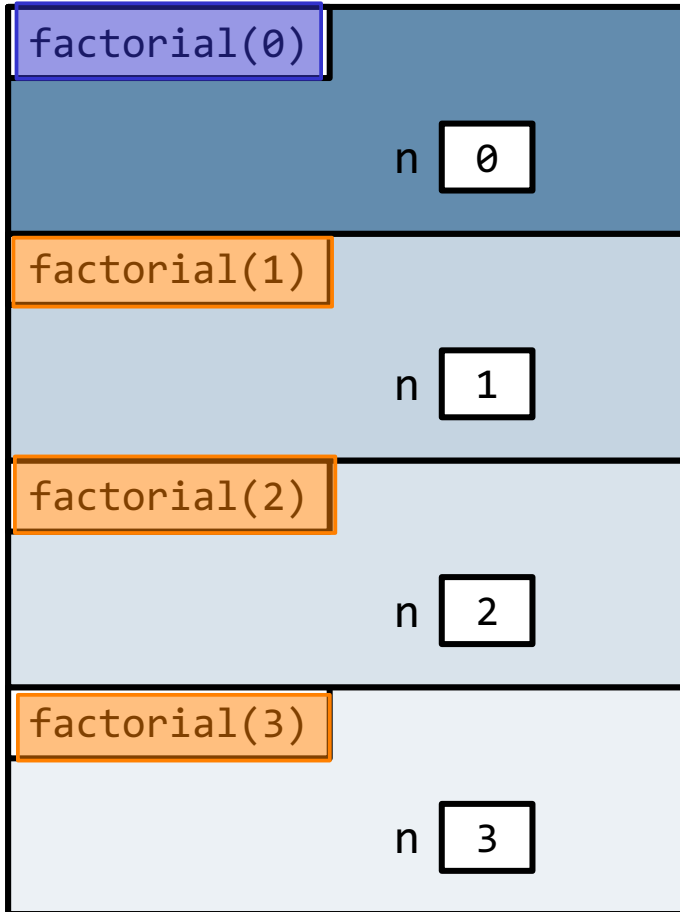
**Solución no estructurada**

# Funciones recursivas: ejecución

- ▶ En general, en la pila se almacena el marco 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 independiente del resto de llamadas.
- ▶ Ejemplo: ejecución del factorial con  $n=3$ 
  1. Dentro de `factorial()`, cada llamada `return 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 se verifica la condición del caso base. Entonces:
    1. Se ejecuta la sentencia `return 1;`
    2. Comienza la vuelta atrás de la recursión, se evalúan las expresiones y se ejecutan los `return` pendientes.

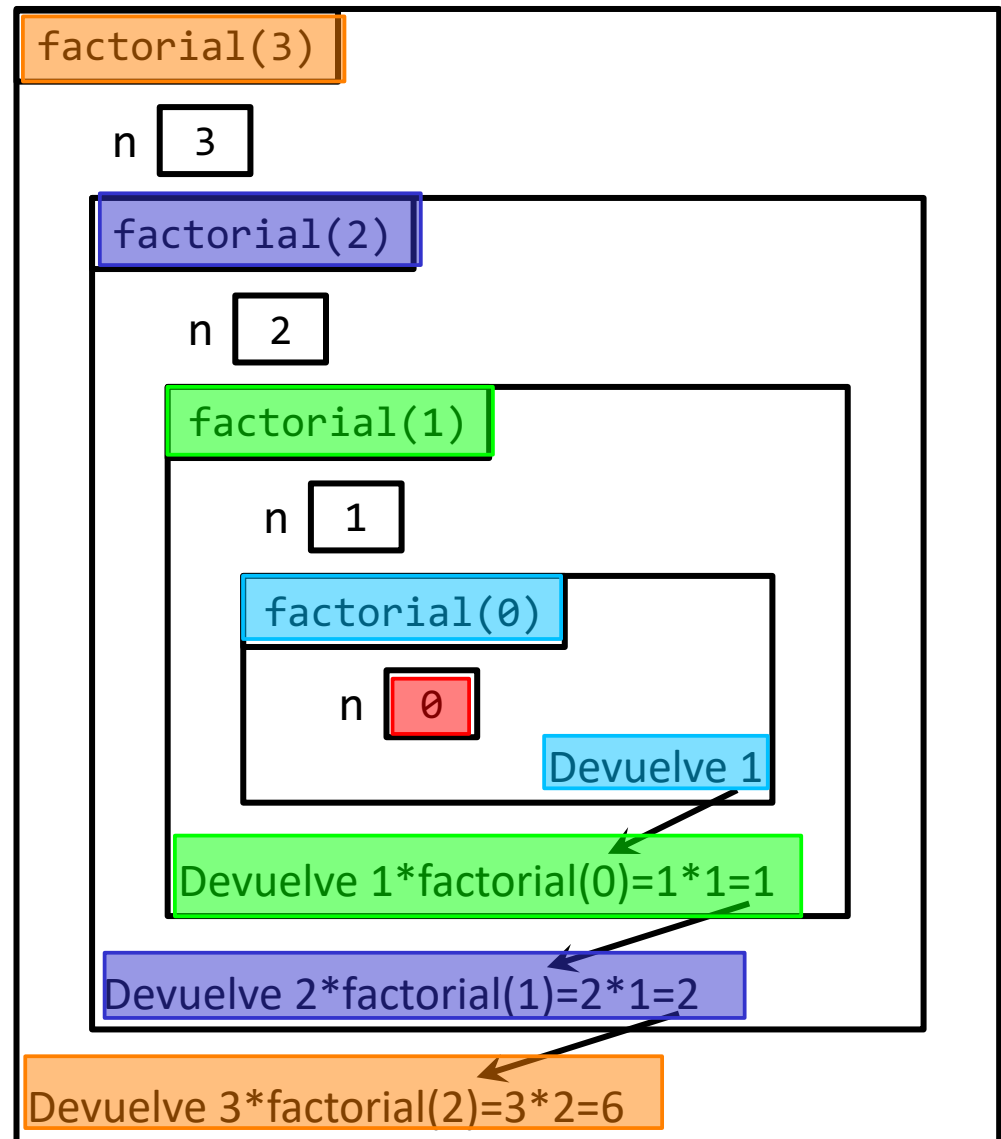
# Funciones recursivas: ejecución

## ► Gráficamente:



# Funciones recursivas: traza

- ▶ Cada una de las llamadas al módulo recursivo se representan en cascada, así como sus respectivas zonas de memoria y los valores que devuelven.
- ▶ Ejemplo: `factorial(3)`



# Ejemplos

## ► Potencia de un número real elevado a una potencia entera,

► Caso base( $n=0$ ):  $x^0 = 1$

► Caso general:  $x^{n-1}$

► Composición:  $x^n = x * x^{n-1}$

```
double potencia(double base, int exponente){
```

```
    if (exponente==0)
        return 1;
```

```
    else
```

```
        return base * potencia(base, exponente-1);
```

```
}
```



**Precondición:**  $\text{exponente} \geq 0$

# Ejemplos

- ▶ Sumatoria  $\sum_{i=1}^n i$ 
  - ▶ Caso base:  $\sum_{i=1}^1 i = 1$
  - ▶ Caso general:  $\sum_{i=1}^{n-1} i$
  - ▶ Composición:  $\sum_{i=1}^n i = n + \sum_{i=1}^{n-1} i$

```
int sumatoria(int n){  
    if (n==1)  
        return 1;  
    else  
        return n + sumatoria(n-1);  
}
```

**Precondición:**  $n \geq 1$



# Ejemplos

## ► Producto de enteros, $a * b$

► Casos base:  $0 * b = 0$ ,  $a * 0 = 0$

► Caso general:  $a * (b - 1)$

► Composición:  $a * b = a + a * (b - 1)$

```
int producto(int a, int b){
```

```
    if (a==0 || b==0)
        return 0;
```

```
    else
```

```
        return a + producto(a, b-1);
```

```
}
```



**Precondición:**  $a \geq 0$  y  $b \geq 0$

# Ejemplos

- Suma de los elementos de un vector

$$SumaVector(v, n) = \begin{cases} v[0] & \text{si } n = 0 \\ v[n] + SumaVector(v, n-1) & \text{si } n > 0 \end{cases}$$

```
int SumaVectorHasta(int v[], int pos){
```

```
    if (pos==0)
        return v[0];
```

```
    else return v[pos] + SumaVectorHasta(v, pos-1);
}
```

**Precondición:**  $n \geq 0$  y  $n \leq \text{util\_v}$

# Ejemplos

## ► Búsqueda lineal en un vector

```
int BusquedaLineal(int v[], int izqda, int drcha, int buscado){  
    if (izqda>drcha)  
        return -1;  
    else if (v[izqda] == buscado)  
        return izqda;  
    else  
        return BusquedaLineal(v, izqda+1, drcha, buscado);  
}
```



**¡Importante!:** En los problemas de búsqueda tendremos dos casos base: encontrado y no encontrado.

# Ejemplos

---

- Máximo de un vector entre dos posiciones dadas

```
int maximo(int v[], int izqda, int drcha){  
    int pos_maximo_resto;  
    if (izqda == drcha)  
        return izqda;  
    else{  
        pos_maximo_resto = maximo(v, izqda+1, drcha);  
        return v[izqda]>v[pos_maximo_resto] ? izqda : pos_maximo_resto;  
    }  
}
```



**Precondición:**  $izqda \leq drcha$

# Ejemplos

## ► Búsqueda binaria

```
int BusquedaBinaria(int v[], int izqda, int drcha, int buscado){  
    int centro;
```

```
    if (izqda>drcha)  
        return -1;
```

```
    else {
```

```
        centro = (izqda + drcha)/2;
```

```
        if (v[centro] == buscado)  
            return centro;
```

```
        else if (v[centro] > buscado)
```

```
            return BusquedaBinaria(v, izqda, centro-1, buscado);
```

```
        else
```

```
            return BusquedaBinaria(v, centro+1, drcha, buscado);
```

```
    }
```

```
}
```

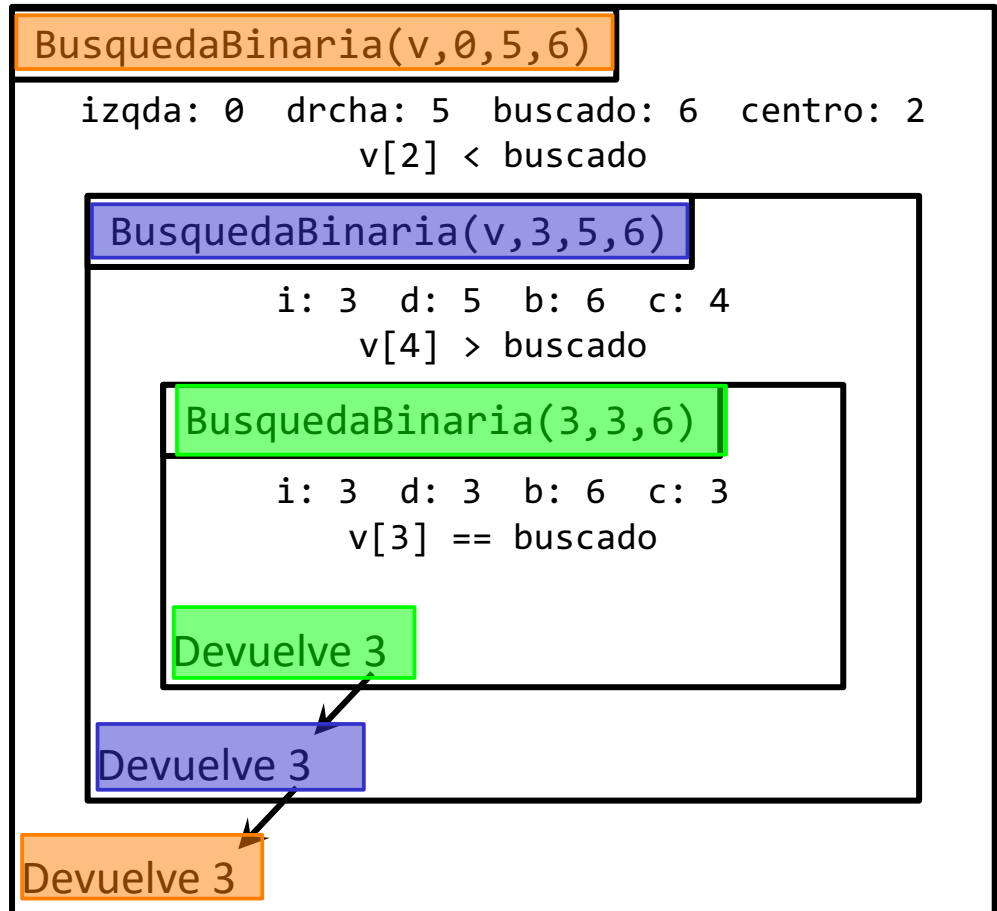


# Ejemplos

## ► Búsqueda binaria: traza del algoritmo

v

1	3	4	6	8	9	...	0
0	1	2	3	4	5		100

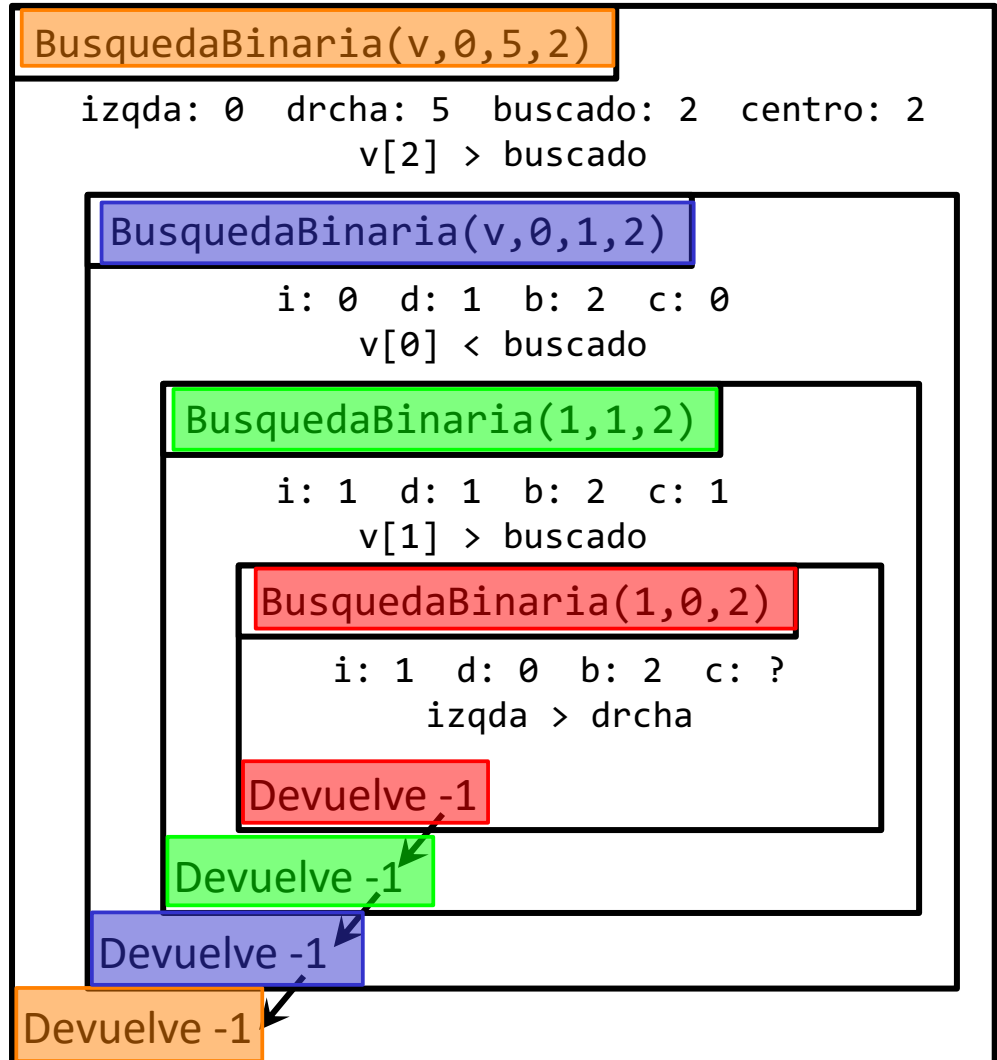


# Ejemplos

## ► Búsqueda binaria: traza del algoritmo

v

1	3	4	6	8	9	...	100
0	1	2	3	4	5		100



# ¿Recursividad o iteración?

---

- ▶ Inconvenientes de la recursividad:
  - ▶ La carga computacional (tiempo y memoria) asociada a la llamada a una función y al retorno a la función llamadora.
  - ▶ Algunas soluciones recursivas pueden llevar a que la solución para un determinado tamaño del problema se calcule varias veces.
  - ▶ Muchas soluciones recursivas tienen como caso base un problema de tamaño reducido, en ocasiones excesivamente reducido.
- ▶ Ventajas de la recursividad:
  - ▶ La solución recursiva suele ser concisa, legible y elegante.
  - ▶ Tal es el caso, por ejemplo, del recorrido de estructuras complejas como árboles, listas, grafos, etc, cuya definición y estructura es intrínsecamente recursiva. Se verá otras asignaturas del grado.



# Problemas de eficiencia en la recursividad

---

Vamos a analizar algunos de los problemas que nos harán descartar una solución recursiva en favor de su homóloga iterativa:

- ▶ Recursividad de cola.
- ▶ Repetición de cálculos.
- ▶ Casos base excesivamente pequeños.
- ▶ Excesivas llamadas recursivas por una reducción del problema demasiado pequeña.

# Recursividad de cola

- ▶ Diremos que una función tiene ***recursividad de cola*** si sólo se produce una llamada recursiva y después de ésta no se ejecuta ningún código.
- ▶ Estas funciones pueden traducirse fácilmente a un algoritmo no recursivo, mucho más eficiente. De hecho, algunos compiladores realizan automáticamente esta conversión cuando se activan las opciones de optimización.
- ▶ Así, por ejemplo, las funciones de búsqueda binaria y búsqueda lineal tienen recursividad de cola. En otros de los ejemplos vistos se realiza alguna acción adicional después de la llamada recursiva.
- ▶ En ambos casos, es muy sencillo encontrar la versión iterativa de estos algoritmos.

# Repetición de cálculos

- Algunas soluciones recursivas implican una inaceptable repetición de cálculos. Un ejemplo muy evidente es la sucesión de Fibonacci, que resulta ser tan ineficiente como elegante.

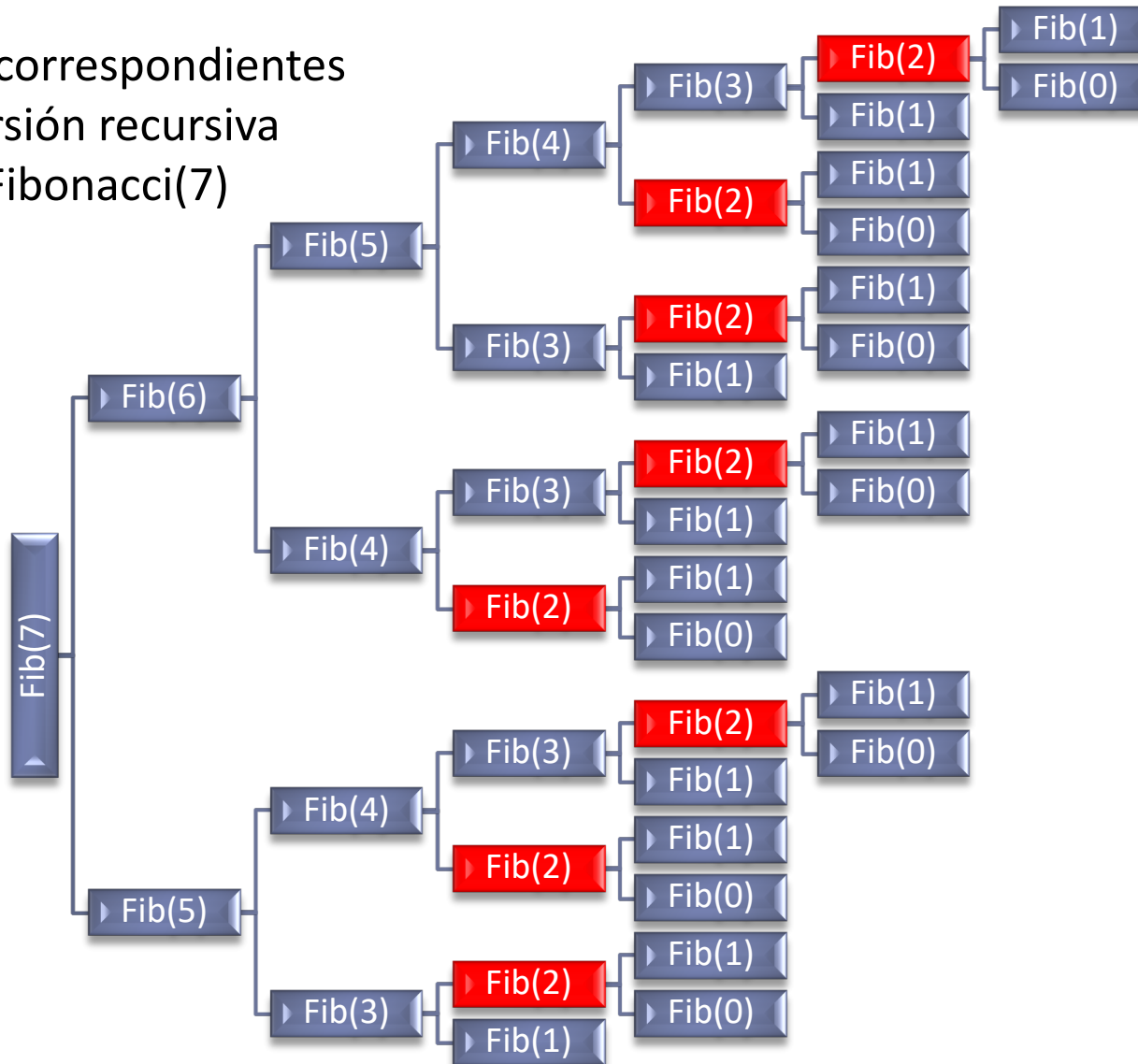
```
int fibonacci(int n){  
    if (n==0)  
        return 0;  
    else if (n==1)  
        return 1;  
    else  
        return fibonacci(n-1) +  
            fibonacci(n-2);  
}
```



n	Nº de llamadas	n	Nº de llamadas
0	1	15	1973
1	1	20	21891
2	$n_1 + n_0 + 1 = 3$	25	242785
3	$n_2 + n_1 + 1 = 5$	30	2692537
4	$n_3 + n_2 + 1 = 9$	35	29860703
5	$n_4 + n_3 + 1 = 15$	40	331160281
6	$n_5 + n_4 + 1 = 25$	45	3672623805
7	$n_6 + n_5 + 1 = 41$	50	40730022147
8	$n_7 + n_6 + 1 = 67$	55	451702867433
9	$n_8 + n_7 + 1 = 109$	60	5009461563921

# Repetición de cálculos

Llamadas correspondientes  
a la versión recursiva  
de Fibonacci(7)



# Repetición de cálculos

- ▶ La versión iterativa puede ser menos elegante, pero es, sin duda, infinitamente más eficiente:

```
int fibonacci(int n){
    int actual=0, n_1=1, n_2=0, i;

    if (n == 0)
        actual = 0;
    else if (n == 1)
        actual = 1;
    else
        for (i=2; i<=n; i++){
            actual = n_1 + n_2;
            n_2 = n_1;
            n_1 = actual;
        }
    return actual;
}
```

# Casos base excesivamente pequeños

---

- ▶ Muchos problemas recursivos tienen como caso base un problema de un tamaño reducido, en muchas ocasiones en exceso.
- ▶ Para resolverlo, suele aplicarse un algoritmo iterativo cuando se ha llegado a un caso base suficientemente pequeño.
- ▶ Ejemplo: La búsqueda binaria.
  - ▶ Aplicaremos el algoritmo recursivo en general y el iterativo cuando el tamaño del subvector en el que realizamos la búsqueda es pequeño.
  - ▶ Lo que hacemos es modificar el caso base, que en este caso es la búsqueda secuencial.

# Casos base excesivamente pequeños

```
int BusquedaBinaria(int v[], int izqda, int drcha, int buscado){  
    const int UMBRAL=10;  
    int centro;  
  
    if (drcha-izqda+1 <= UMBRAL)  
        return BusquedaSecuencial(v,izqda, drcha, buscado);  
    else {  
        centro = (izqda + drcha)/2;  
        if (v[centro] == buscado)  
            return centro;  
        else if (v[centro] < buscado)  
            return BusquedaBinaria(v, centro+1, drcha, buscado);  
        else  
            return BusquedaBinaria(v, izqda, centro-1, buscado);  
    }  
}
```



# Excesivas llamadas recursivas

---

- ▶ La mayoría de los problemas que hemos estudiado en este tema tienen una solución recursiva, pero es absurda.
- ▶ En general, no suele tener sentido una solución recursiva cuando los subproblemas a resolver no son significativamente menores que el problema original.
- ▶ Algunos ejemplos: la potencia, el producto, el factorial, la sumatoria...
- ▶ Corremos un serio peligro: que un gran número de llamadas recursivas desborde la pila. No podemos olvidar que **la pila es un recurso relativamente limitado**, y es bastante sencillo desbordarla con demasiadas llamadas a función anidadas.
- ▶ Además, el tiempo requerido para realizar una llamada es considerable, lo que debemos tener también en cuenta.



# Ejemplo avanzado: Quicksort

- ▶ Es un método de ordenación conceptualmente recursivo, cuya potencia se basa en dividir (aproximadamente) a la mitad el tamaño de los subproblemas.
- ▶ Elegir un elemento, al que llamaremos pivote.
- ▶ Reubicar los elementos de manera que los elementos menores que el pivote queden a su izquierda y los mayores que el pivote, a su derecha. Al hacerlo, el pivote queda en su posición definitiva en la lista ordenada.
- ▶ La lista queda dividida en dos sublistas, a las que podemos aplicar de nuevo el proceso de forma recursiva.
- ▶ Al terminar, todos los elementos estarán ordenados.
- ▶ Lo ideal sería que el pivote fuera la mediana, porque así los dos subproblemas serían justo de la mitad del tamaño original, pero resulta costoso, por lo que lo elegiremos aleatoriamente.

# Ejemplo avanzado: Quicksort

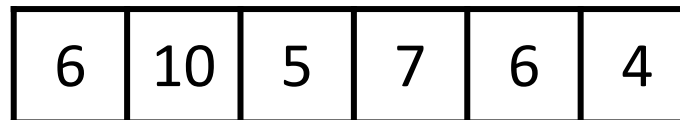
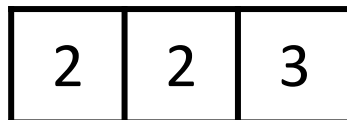
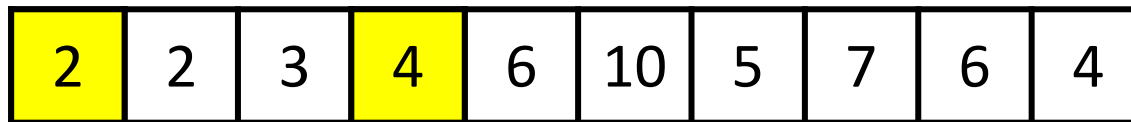
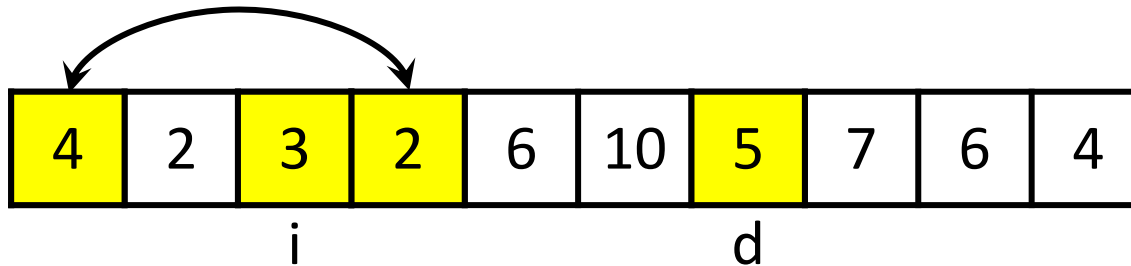
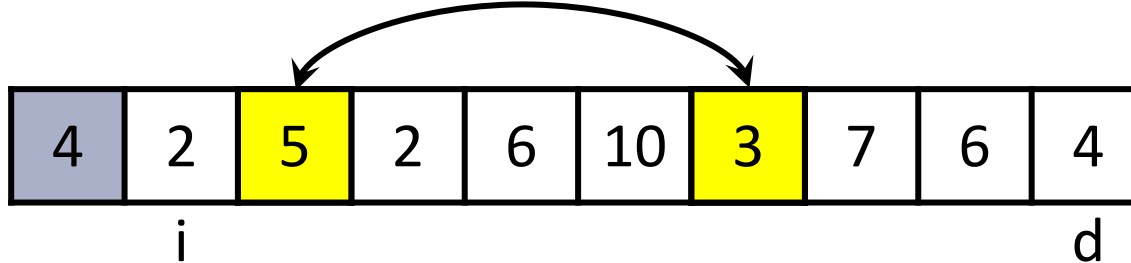
```
void quicksort(int v[], int izqda, int drcha){  
    int pos_pivote;  
    if(izqda<drcha){  
        pos_pivote = partir(v, izqda, drcha);  
        quicksort(v, izqda, pos_pivote-1);  
        quicksort(v, pos_pivote+1, drcha);  
    }  
}
```



```
//Función auxiliar de intercambio  
template <class T>  
void intercambiar(T & v1, T & v2){  
    T aux(v1);  
    v1 = v2;  
    v2 = aux;  
}
```

# Ejemplo avanzado: Quicksort

- Primera partición de Quicksort:



Siguientes llamadas recursivas

# Ejemplo avanzado: Quicksort

```
int partir(int v[], int primero, int ultimo){  
    int pivote = v[primero],  
    izqda = primero+1,  
    drcha = ultimo;
```



```
while(izqda<=drcha){ //Mientras no se crucen los índices  
    //Busco por la izquierda el primer elemento mayor que el pivote
```

```
while((izqda<=drcha) && (v[izqda]<pivote))  
    izqda++;
```

```
//Busco por la derecha el primer elemento menor que el pivote
```

```
while((izqda<=drcha) && (v[drcha]>=pivote))  
    drcha--;
```

```
//Si no se han cruzado los índices, intercambiar
```

```
if (izqda<drcha)  
    intercambiar(v[izqda++], v[drcha--]);
```

```
}
```

```
//Al terminar la partición, situar el pivote en su posición
```

```
intercambiar(v[primero], v[drcha]);
```

```
return drcha; //Devolvemos la nueva posición del pivote
```

```
}
```

# Ejemplo avanzado: Quicksort

---

- ▶ Quicksort es un algoritmo de ordenación muy eficiente, especialmente en comparación con los algoritmos básicos estudiados anteriormente.
- ▶ La clave de su eficiencia radica en la elección del pivote. La posible mejora pasa por diseñar buenas estrategias para elegir pivotes que dejen más o menos los mismos elementos a su izquierda que a su derecha.
- ▶ Además, Quicksort es fácilmente paralelizable, ya que las diferentes llamadas recursivas pueden ejecutarse en diferentes CPUs.

# Ejemplo avanzado: Quicksort

- Podemos mejorar quicksort evitando aplicarlo a subproblemas de pequeño tamaño.

```
void quicksort(int v[], int izqda, int drcha){
```

```
    const int umbral = 10;
```

```
    int pos_pivote;
```

```
    if(drcha-izqda+1 < umbral)
```

```
        seleccion(v, izqda, drcha);
```

//Si el problema es pequeño

//usamos un método simple

```
    else if(izqda < drcha){
```

```
        pos_pivote = partir(v, izqda, drcha);
```

```
        quicksort(v, izqda, pos_pivote-1);
```

```
        quicksort(v, pos_pivote+1, drcha);
```

```
    }
```

```
}
```