



# Programación 2

## Diseño recursivo

Fernando Orejas

Transparencias basadas en las de Ricard Gavaldà

1. Principios del diseño recursivo
2. Inmersión
3. Recursividad lineal final y algoritmos iterativos

## *Principios del diseño recursivo*

# Algoritmos recursivos

Un algoritmo recursivo no es más que la implementación directa de una definición inductiva.

# Factorial

$$n! = 1 * 2 * \dots * (n-1) * n$$

## Definición inductiva

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

```
int factorial(int n){//solución recursiva  
    // Pre: n >= 0  
    // Post: devuelve el factorial de n  
  
    if (n == 0) return 1;  
    else return n*factorial(n-1);  
}
```

## Suma de los elementos de una pila

Si  $P = e_1 \ e_2 \ \dots \ e_n$

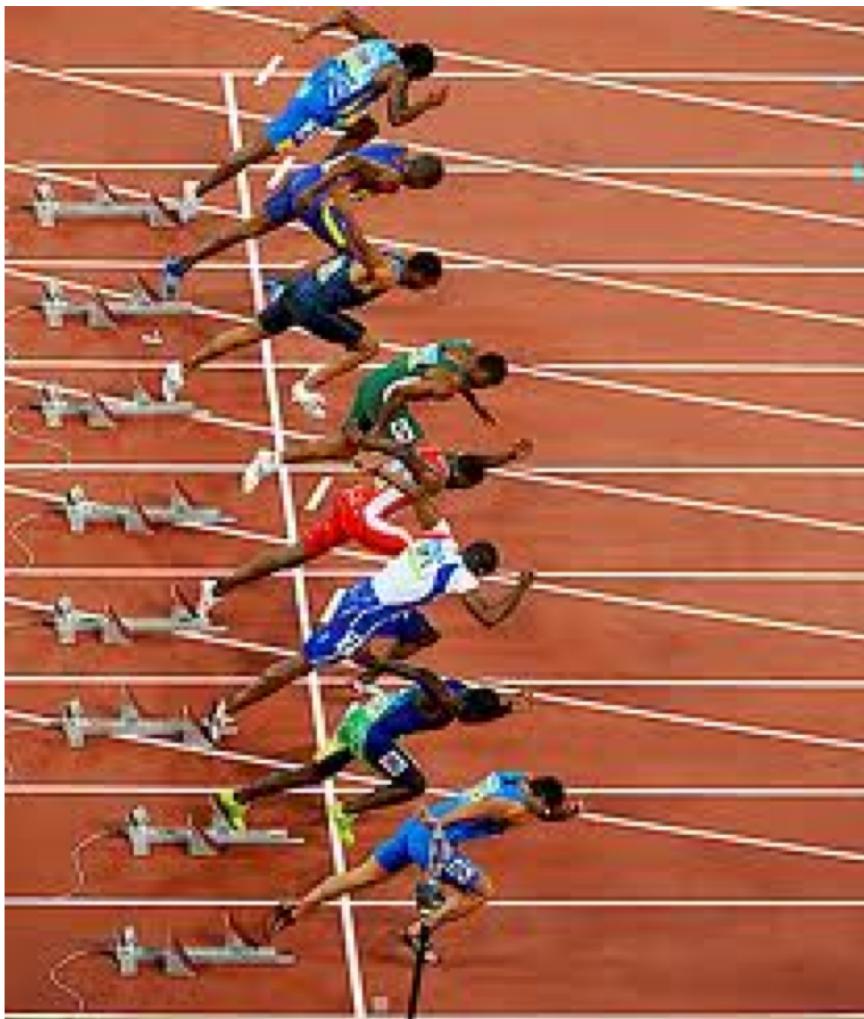
$\text{Suma}(P) = e_1 + e_2 + \dots + e_n$

$$\text{Suma}(P) = \begin{cases} 0 & \text{si } P \text{ está vacía} \\ P.\text{top}() + \text{Suma}(P.\text{pop}()) & \text{en otro caso} \end{cases}$$

```
// Pre: true
// Post: devuelve la suma de los valores de P

int Suma(Stack <int> P) {
    if (P.empty()) return 0;
    else return P.top() + Suma(P.pop());
}
```

# Diseño recursivo



1. Caso inicial o básico



## 2. Caso general

### 3. Análisis de terminación



## Terminación del factorial

En el caso general de  $\text{factorial}(n)$  hacemos una llamada a  $\text{factorial}(n-1)$ , como se supone que  $n \geq 0$ , cada nueva llamada nos acerca al caso inicial o base.

# Verificación de un algoritmo recursivo

Hemos de demostrar que para cada valor de los parámetros que cumpla la Pre:

- El algoritmo termina.
- Los resultados cumplen la postcondición

## Terminación de un algoritmo recursivo

Usaremos una función entera  $T$  de *tamaño*, cuyos parámetros son los de la función recursiva, que cumpla:

- Si  $T(\dots)$  es 0, o negativo, estamos en un caso base.
- Las llamadas recursivas hacen que  $T$  decrezca

# Corrección parcial de un algoritmo recursivo

Hemos de demostrar:

- Si  $X$  es un caso inicial: directamente.
- Hipótesis de inducción: Si  $X'$  es *más pequeño* que  $X$  y cumple Pre, entonces  $f(X')$  cumple Post.
- En el caso general: comprobamos que todo  $X'$  usado en las llamadas recursivas cumple Pre y suponemos que  $f(X')$  cumple Post, y demostramos que los calculos adicionales nos garantizan que el resultado de la función cumple Post.

# Igualdad de pilas

```
// Pre: p1 = P1, p2 = P2
// Post: nos dice si P1 y P2 son iguales

bool iguales(Stack <int>& p1, Stack <int>& p2) {
    if (p1.empty() or p2.empty())
        return p1.empty() and p2.empty();
    else if (p1.top() != p2.top()) return false;
    else {
        p1.pop(); p2.pop();
        return iguales(p1,p2);
    }
}
```

// Exponenciación

// Pre: y >= 0  
// Post: Retorna  $x^y$

```
int Potencia(int x, int y) {  
    if (y == 0) return 1;  
    else if (y % 2 == 0)  
        return potencia(x*x, y/2);  
    else  
        return x*potencia(x, y-1);  
}
```

*Inmersión*

## Inmersión de una función en otra

Hacer una inmersión de una función  $f$ , quiere decir definir una función  $g$ , con más parámetros y que generaliza  $f$ .

Dos tipos de inmersiones:

- Inmersión con *datos adicionales* (debilitamiento de la Post)
- Inmersión con *resultados adicionales* (fortalecimiento de la Pre)

```
// Pre: true
// Post: devuelve la suma de los valores de P

int Suma(Stack <int> P) {
    if (P.empty()) return 0;
    else return P.top() + Suma(P.pop());
}
```

```
// Pre: v.size() > 0
/* Post: devuelve la suma de los valores de un
vector v */
```

```
int Suma(const vector <int> &v);
```

¿Cómo hacemos una definición recursiva?

## Inmersión con datos adicionales

```
/* Pre: v.size() > 0, 0 <= n < v.size() */
/* Post: devuelve la suma de los valores de
v[0..n] */

int i_Suma(const vector <int> &v, int n){
    if (n == 0) return v[0];
    else return v[n]+i_Suma(v,n-1);
}
```

## Inmersión con datos adicionales

```
/* Pre: v.size() > 0, 0 <= n < v.size() */
/* Post: devuelve la suma de los valores de
   v[0..n] */

int i_Suma(const vector <int> &v, int n){
    if (n == 0) return v[0];
    else return v[n]+i_Suma(v,n-1);
}

int Suma(const vector <int> &v){
    return i_Suma(v,v.size()-1);
}
```

## Inmersión con resultados adicionales

```
/* Pre: v.size() > 0, 0 <= n < v.size(), s es la
suma de v[0..n] */
/* Post: devuelve la suma de los valores de v */

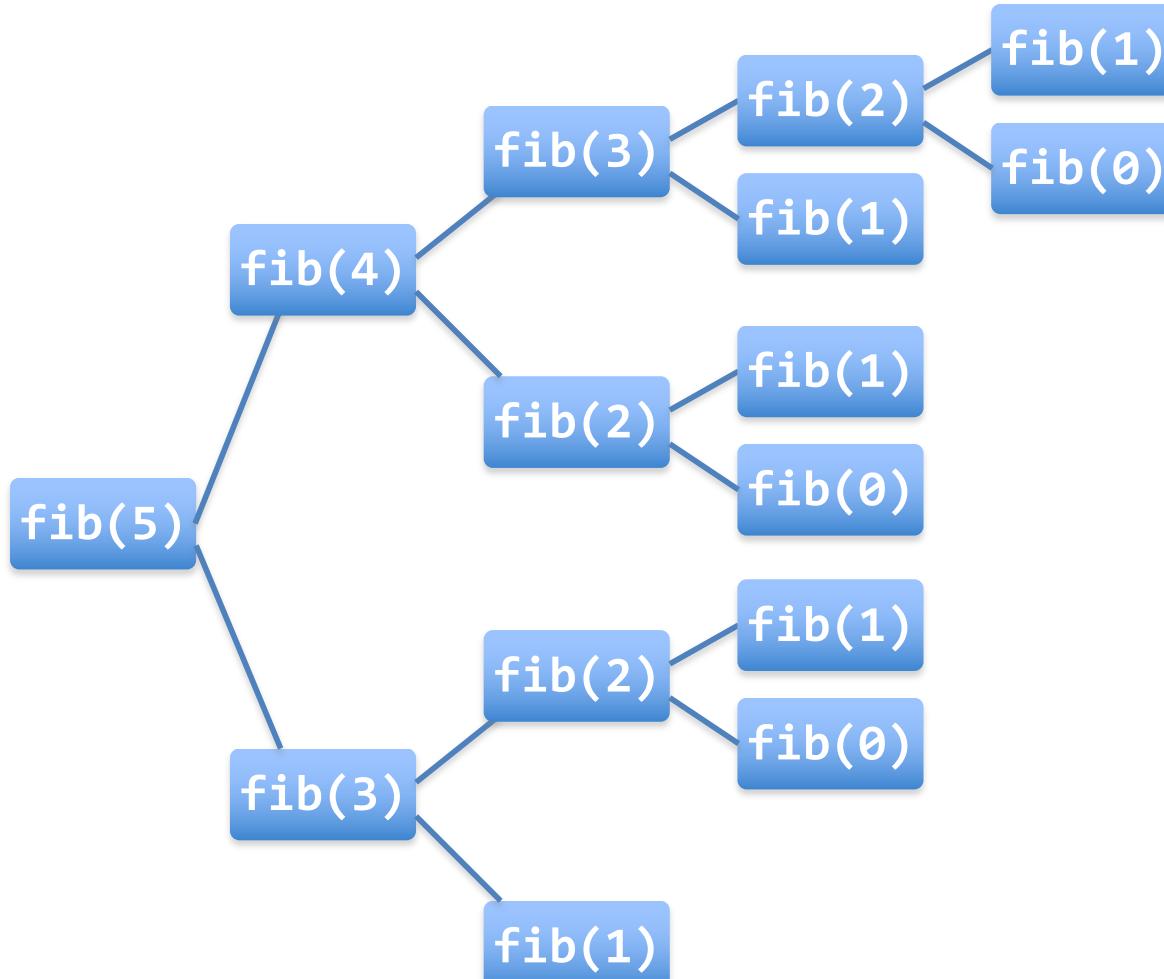
int i_Suma(const vector <int> &v, int n, int s){
    if (n == v.size()-1) return s;
    else return i_Suma(v,n+1,s+v[n+1]);
}

int Suma(const vector <int> &v) {
    return i_Suma(v,0,v[0]);
}
```

# Números de Fibonacci

```
int fib (int n) {  
    // Pre: n >= 0  
    // Post: Retorna el número de Fibonacci de orden n.
```

```
int fib (int n) { // versión recursiva  
    // Pre: n >= 0  
    // Post: Devuelve el número de Fibonacci de orden n.  
  
    if (n == 0 or n == 1) return 1;  
    else return fib(n-2) + fib(n-1);  
}
```



¡¡Muy ineficiente!!

```
/* Pre: 0 < i <= n, i fact es el número de fibonacci de  
orden i, fant es el número de fibonacci de orden i-1 */  
/* Post: Retorna el número de Fibonacci de orden n */
```

```
int fib1 (int n; int i, int fact, int fant) {  
    if (n == i) return fact;  
    else return fib1(n, i+1, fact+fant, fact);  
}
```

```
/* Pre: 0 < i <= n, i fact es el número de fibonacci de
orden i, fant es el número de fibonacci de orden i-1 */
/* Post: Retorna el número de Fibonacci de orden n */
```

```
int fib1 (int n; int i, int fact, int fant) {
    if (n == i) return fact;
    else return fib1(n, i+1, fact+fant, fact);
}
```

```
int fib (int n) {
    if (n == 0) return 1;
    else return fib1(n,1,1,1);
}
```

# *Recursividad lineal final y algoritmos iterativos*

## Recursividad lineal final (tail recursion)

- Un algoritmo recursivo es *lineal* si cada llamada genera una sola llamada recursiva
- Una función recursiva lineal es *final* si la última instrucción que se ejecuta es la llamada recursiva y el resultado obtenido es el resultado de esa llamada final

```
int factorial(int n){  
  
    // Pre: n >= 0  
    // Post: devuelve el factorial de n  
  
    if (n == 0) return 1;  
    else return n*factorial(n-1);  
}
```

```
// Pre: n >= i >= 0, f = i!
// Post: devuelve el factorial de n

int i_factorial(int n, int i, int f){
    if (n == i) return f;
    else return i_factorial(n, i+1, f*(i+1));
}

int factorial(int n){
    return i_factorial(n,0,1);
}
```

```
// Pre: n >= 0
// Post: devuelve el factorial de n

int i_factorial(int n){//solución iterativa
    int i = 0; int f = 1; //de la llamada inicial
    // Invariante: f = i! and i ≤ n
    while (i != n) {
        f = f*(i + 1); //de la llamada recursiva
        i = i+1;
    }
    return f;
}
```

## Transformación a iterativo de recursividad de cola

```
T2 f(T1 x){  
    T2 s;  
    if c(x) s = d(x);  
    else s = f(g(x));  
    return s;  
}
```

## Transformación a iterativo de recursividad de cola

```
T2 f(T1 x){  
    T2 s;  
    if c(x) s = d(x);  
    else s = f(g(x));  
    return s;  
}
```

```
T2 f_iter(T1 x){  
    T2 s;  
    while (not c(x)) x = g(x);  
    s = d(x);  
    return s;  
}
```