



Programación 2

Diseño recursivo

Fernando Orejas

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.

Inmersión

Si directamente no se puede definir una función recursiva, se puede intentar añadir más parámetros

Si se repiten los cálculos, se pueden añadir más parámetros para recordarlos

Definiciones inductivas

La idea básica de una definición inductiva es que:

1. Para algunos casos básicos definimos la función directamente
2. Para el resto de los casos definimos la función en base a elementos *más pequeños*.
3. Tenemos funciones de descomposición que nos permiten obtener esos elementos más pequeños.

Factorial

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

1. Caso base $n = 0$
2. El factorial de n lo podemos definir a partir del factorial de $n-1$
3. La función de descomposición es $n-1$

Definición inductiva

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

```
int factorial(int n){  
  
    // 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 $P = e_1 \ e_2 \ \dots \ e_n$

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

1. Caso base: la pila está vacía
2. $\text{Suma}(p)$ se puede definir a partir de la suma del elemento que está en la cumbre de p y del resto de la pila
3. Las funciones de descomposición son `top` y `pop`

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());
}
```

Búsqueda en una pila

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

1. Caso base: la pila está vacía
2. $\text{busq}(p, x)$ se puede definir a partir del elemento que está en la cumbre de p y del resto de la pila
3. Las funciones de descomposición son top y pop

Búsqueda en una pila

```
// Pre: true  
// Post: Nos dice si x está en P
```

```
bool busq(Stack <int> P, int x) {  
    if (P.empty()) return false;  
    else  
        if (P.top() == x) return true  
        else {  
            P.pop();  
            return busq(P, x);  
        }  
}
```

Igualdad de árboles binarios

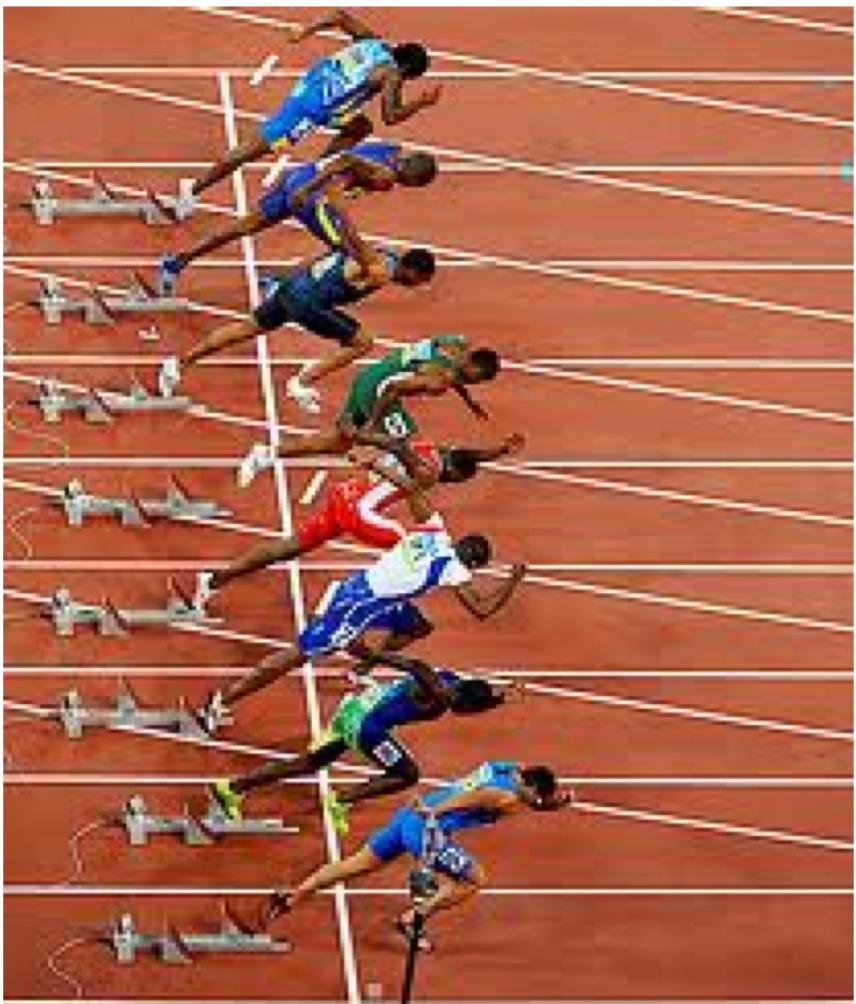
1. Casos base: Alguno de los árboles está vacío
2. $\text{eq}(a_1, a_2)$ se puede definir a partir de los valores en las raíces de a_1 y a_2 y de la igualdad de sus subárboles izquierdos y derechos.
3. Las funciones de descomposición son `value`, `left` y `right`

Igualdad de árboles binarios

```
// Pre: true
// Post: Nos dice si a1 y a2 son iguales

bool eq(BinTree <int> a1, BinTree <int> a2) {
    if (a1.empty() or a2.empty())
        return a1.empty() and a2.empty();
    else
        return (a1.value() == a2.value()) and
                (eq(a1.left(), a2.left())) and
                (eq(a1.right(), a2.right()));
}
```

Diseño recursivo



1. Casos básicos



2. Caso general

3. Análisis de terminación



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 (corrección parcial).

Terminación de una función recursiva

Para estar seguros de que una función recursiva termina:

- Hay que estar seguros de que las funciones de descomposición realmente nos dan elementos *más pequeños*,
- Los casos base son los más pequeños de todos.

Terminación de una función recursiva

La terminación se puede garantizar usando una función $|x|$ de medida o tamaño que cumple:

- $|x|$ nos devuelve un entero
- Si $|x| \leq 0$ entonces x es un caso base
- Si f es una función de descomposición, entonces $|f(x)| < |x|$. Es decir, si y es un parámetro de una llamada recursiva, entonces $|y| < |x|$.

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

- $|n| = n$

```
// 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());
}
```

- $|P| = P.size()$

Igualdad de árboles binarios

```
// Pre: true
// Post: Nos dice si a1 y a2 son iguales

bool eq(BinTree <int> a1, BinTree <int> a2) {
    if (a1.empty() or a2.empty())
        return (a1.empty() and a2.empty());
    else
        return (a1.value() == a2.value()) and
                (eq(a1.left(), a2.left())) and
                (eq(a1.right(), a2.right()));
}
```

$$|a1, a2| = \min(\text{size}(a1), \text{size}(a2))$$

Corrección parcial de un algoritmo recursivo

Hemos de demostrar:

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

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

- **Caso base.** $0 = 1!$
- **Caso general.**
 - Si $n > 0$ entonces $n-1 \geq 0$ ($n-1$ cumple la pre)
 - Si $\text{factorial}(n-1) = (n-1)!$ entonces $\text{factorial}(n) = n * ((n-1)!) = n!$

```

// 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());
}

```

- **Caso base.** Si P está vacía la suma es 0.
- **Caso general.**

Supongamos que $P = e_1 \ e_2 \ \dots \ e_n$

- Si P no está vacía entonces $|P.pop()| \geq |P|$ ($P.pop()$ cumple la pre)
- Si $\text{Suma}(n-1) = e_2 + \dots + e_n$ entonces
 $\text{Suma}(P) = P.top() + e_2 + \dots + e_n = e_1 + e_2 + \dots + e_n$

Igualdad de árboles binarios

```
// Pre: true
// Post: Nos dice si a1 y a2 son iguales

bool eq(BinTree <int> a1, BinTree <int> a2) {
    if (a1.empty() or a2.empty())
        return (a1.empty() and a2.empty());
    else
        return (a1.value() == a2.value()) and
                (eq(a1.left(), a2.left())) and
                (eq(a1.right(), a2.right()));
}
```

- **Caso base.** Si a1 o a2 están vacíos, entonces son iguales si y solo si ambos están vacíos.
 - **Caso general.**
 - Los parámetros de las llamadas recursivas cumplen trivialmente la precondición
 - Si ni a1 ni a2 están vacíos y sabemos que:
 $\text{eq}(\text{a1.left}(), \text{a2.left}()) \equiv (\text{a1.left}() == \text{a2.left}())$
 $\text{eq}(\text{a1.right}(), \text{a2.right}()) \equiv (\text{a1.right}() == \text{a2.right}())$
 $\text{a1.value}() == \text{a2.value}()$
- entonces
- $$\text{eq}(\text{a1}, \text{a2}) \equiv (\text{a1} == \text{a2})$$

// 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);  
}
```

|x,y| = y

- **Caso base.** Si $y = 0$, entonces $\text{exp}(x,y) = 1 = x^y$
- **Caso general.**
 - Si $y > 0$, $y\%2$ e $y - 1$ son mayores o iguales que 0. Por tanto, los parámetros de las llamadas recursivas cumplen la precondition
 - Si y es par entonces podemos asumir que $\text{potencia}(x*x, y/2) = (x*x)^{(y/2)} = x^{2*(y/2)} = x^y$. Es decir que $\text{potencia}(x, y) = x^y$.
 - Si y es impar entonces podemos asumir que $\text{potencia}(x, y-1) = x^{(y-1)}$. Es decir que $\text{potencia}(x, y) = x * x^{(y-1)} = x^y$.

Suma k a todos los valores de un árbol

```
/* Pre: true */
/* Post: retorna un arbol t' con la misma forma que t,
tal que el valor de cada nodo de t' es igual a k + el
valor del nodo correspondiente de t */
BinTree sumak(const BinTree <int>& t, int k){
    if (t.empty()) return t;
    else return BinTree(t.value() + k,
                        sumak(t.left(), k),
                        sumak(t.right(), k));
}
```

$$|t| = \text{size}(t)$$

- **Caso base.** Si a está vacío, entonces el resultado es el propio a, luego no hace falta hacer nada.
- **Caso general.**
 - Los parámetros de las llamadas recursivas cumplen trivialmente la precondición
 - Podemos asumir que $\text{sumak}(a.\text{left}())$ es el resultado de sumar k a todos los valores en el hijo izquierdo de a.
 - También podemos asumir que $\text{sumak}(a.\text{right}())$ es el resultado de sumar k a todos los valores en el hijo derecho de a.
 - Por tanto $\text{BinTree}(a.\text{value}() + k, \text{sumak}(a.\text{left}()), \text{sumak}(a.\text{right}()))$ es el árbol resultante de sumar k a todos los valores en a.

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);
}
```