Programación 2
Diseño recursivo 2

Fernando Orejas

1. Inmersiones
2. Inmersión por debilitamiento de la postcondición
3. Inmersión por fortalecimiento de la precondición
4. Recursividad lineal final y algoritmos iterativos

Inmersiones

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

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

```
// Pre: true
/* Post: devuelve la suma de los valores de un
   vector v */

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

¿Cómo hacemos una definición recursiva?

No podemos descomponer v

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

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 debilitamiento de la Post
- Inmersión con fortalecimiento de la Pre

Suma de un vector con inmersión

```
/* Pre:  -1 <= n < v.size() */
/* Post: devuelve la suma de los valores de
   v[0..n] */

int i_Suma(const vector <int> &v, int n);
```

Suma de un vector con inmersión

```
/* Pre:  -1 <= 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 0;
    else return v[n]+i_Suma(v,n-1);
}
```

Suma de un vector con inmersión

```
/* Pre:  -1 <= 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 0;
    else return v[n]+i_Suma(v,n-1);
}

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

Suma de un vector con otra inmersión

```
/* Pre:  0 <= n <= v.size() */
/* Post: devuelve la suma de los valores de
   v[n..v.size()-1] */

int i_Suma(const vector <int> &v, int n);
```

Suma de un vector con otra inmersión

```
/* Pre:  0 <= n <= v.size() */
/* Post: devuelve la suma de los valores de
   v[n..v.size()-1] */

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

Suma de un vector con otra inmersión

```
/* Pre:  0 <= n <= v.size() */
/* Post: devuelve la suma de los valores de
   v[n..v.size()-1] */

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

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

Funciones de inmersión

Hay que

- Decidir qué inmersión se hará y qué parámetros añadir
- Especificar la función de inmersión (incluye la especificación de los nuevos parámetros y de los resultados)
- Definir la llamada inicial a la función de inmersión

Inmersión por debilitamiento de la Post

Búsqueda en un vector ordenado

```
/* Pre: v.size()>0, v está ordenado crecientemente*/
/* Post: retorna una posición en que esté x, si x
está en v, si no retorna -1*/

int busq(const vector <int> &v, int x);
```

Debilitamiento de la Post

Dos casos posibles:

- cambiamos v por v[0..i]
- o cambiamos v por v[j..v.size()-1]
- o las dos cosas, cambiamos v por v[i,j]

Debilitamiento de la Post

```
/* Pre:  0<=i<=v.size(), -1<=j<v.size(),
i<=j+1, v está ordenado */
/* Post: retorna una posición de v[i..j] en
que esté x, si x está en v[i..j], si no
retorna -1*/

int i_busq(const vector <int> &v, int x, int
i, int j);
```

Búsqueda en un vector ordenado

```
/* Pre: v.size()>0, v está ordenado crecientemente*/
/* Post: retorna una posición en que esté x, si x
está en v, si no retorna -1*/

int busq (const vector <int> &v, int x){
    return i_busq (v,x,0,v.size-1);
}
```

Búsqueda dicotómica

```
int i_busq(const vector <int> &v, int x, int
i, int j){
    int pos;
    if (j < i) pos = -1;
    else {
        mig = (i+j)/2;
        if (v[mig] = x) pos = mig;
        else {
            if (v[mig] < x)
                pos = i_busq(v,x, mig+1,j);
            else = i_busq(v,x, i,mig-1);
        }
    } return pos;
}
```

- **Función de medida**. $|v, x, i, j| = |v[i..j]|$

- **casos base**. Si $|v, i, j| = 0$ quiere decir que $|v[i..j]| = 0$, es decir que $j < i$.
- **llamadas recursivas**. Hay dos llamadas recursivas:
  - **`i_busq(v,x, mig+1,j)`**. Claramente,
    $|v[(i+j)/2+1..j]| < |v[i..j]|$
  - **`i_busq(v,x, i,mig-1)`**. Analogamente,
    $|v[i..(i+j)/2-1]| < |v[i..j]|$

- **Caso base**. Si j<i, el fragmento de vector está vacío, por tanto el resultado es -1.

- **Caso general**.
  - Los parámetros de las llamadas recursivas cumplen la precondición. En el caso de la 1a llamada, **`i_busq(v,x, mig+1,j)`** y sabemos que mig+1 = (i+j)/2+1. Ahora bien, si 0 ≤ i ≤ v.size(), entonces 0 ≤ (i+j)/2+1. Si además sabemos que j<v.size(), entonces (i+j)/2+1 ≤ v.size(). La condición -1≤j≤v.size() tambien se cumple, porque el parámetro j es el mismo. La condición (i+j)/2+1 ≤ j+1 es equivalente a (i+j)/2≤j, que es equivalente a i+j≤2*j, pero sabemos)por la condición del if) que i≤j, por tanto i+j ≤ j+j = 2*j. Finalmente, v está ordenado, porque no lo hemos modificado. El caso de la otra llamada recursiva es similar.

- **Caso general (cont.)**.
  - Por tanto, podemos asumir que **`i_busq(v,x,mig+1,j)`** nos devuelve la posición de x en ese fragmento del vector y la llamada **`i_busq(v,x, i,mig-1)`** nos devuelve la posición de x en el otro fragmento del vector

  - Ahora, x está en el fragmento v[i..j] si x está en v[i..mig-1], x == v[mig], o x está en v[mig+1..j]. Ahora, si x == v[mig], la respuesta mig, que obtenemos en el if, es correcta. Si no, si v[mig]<x, la respuesta de **`i_busq(v,x, mig+1,j)`** es correcta. por la hipótesis de inducción y, en caso contrario, la respuesta de **`i_busq(v,x, i,mig-1)`** también es correcta por el mismo motivo.

Otro ejemplo: Mergesort

Queremos ordenar recursivamente un vector

```
/* Pre: v = V, v.size()>0*/
/* Post: v está ordenado crecientemente y v es
una permutación de V */

void ordenar (vector <int> &v);
```

Mergesort

Llamemos n a v.size().

```
/* Pre: v = V, 0 ≤ i ≤ d ≤ n, n > 0 */
/* Post: v[i..d] está ordenado crecientemente,
v[0,i-1]=V[0,i-1], v[d+1,n-1]=V[d+1,n-1] y v es
una permutación de V */

void mergesort (vector <int> &v, int i, int d);
```

Mergesort

```
void mergesort (vector <int> &v, int i, int d){
    if (i<d){
        int m = (i+d)/2;
        mergesort(v,i,m);
        mergesort(v,m+1,d);
        fusiona(v,i,m,d);
    }
}
```

Mergesort

Habría que implementar la función fusiona (seguimos llamando n a v.size()):

```
/* Pre: v = V, 0 ≤ i ≤ m ≤ d ≤ n, v[i,m] y
v[m+1,d] están ordenados crecientemente */
/* Post: v[i..d] está ordenado crecientemente,
v[0,i-1]=V[0,i-1], v[d+1,n-1]=V[d+1,n-1] y v es
una permutación de V */

void fusiona (vector <int> &v, int i, int m, int
d);
```

- **Función de medida**. $|v, i, d| = d - i$

- **casos base**. Si $|v, i, d| = 0$ quiere decir que $i = d$, ya que $d \geq i$.
- **llamadas recursivas**. Hay dos llamadas recursivas:
  - **`mergesort(v,i,m)`**, siendo $m = (i+d)/2$. Claramente,

    $(i+d)/2 - i < d - i$

    ya que $i < d$ y por tanto $(i+d)/2 < d$.
  - **`mergesort(v,m+1,d)`**. Analogamente,

    $d - (i+d)/2 < d - i$

    ya que $i < d$ y por tanto $(i+d)/2 > i$.

- **Caso base**. Si d ≤ i, es decir d=i, no hay que hacer nada, porque los fragmentos de tamaño 1 ya están ordenados.

- **Caso general**.
  - Los parámetros de las llamadas recursivas claramente cumplen la precondición. En el caso de la 1a llamada, **`mergesort(v,i,m)`**, tenemos obviamente que 0 ≤ i≤ m, ya que i≤d y m = (i+d)/2. El caso de la otra llamada recursiva es similar.
  - Por tanto, podemos asumir que **`mergesort(v,i,m)`** nos ordena v[i..m] y que **`mergesort(v,m+1,d)`** nos ordena v[m+1,d], dejando el resto del vector intacto.
  - Como consecuencia, si después de esas dos llamadas tenemos los dos fragmentos ordenados, después de **`fusiona(v,i,m,d)`** , v[i..d] estará ordenado y el resto del vector estará intacto.

Inmersión por fortalecimiento de la Pre

Inmersión por fortalecimiento de la Pre

- Cuando se debilita la postcondición, cada llamada recursiva hacen parte del trabajo.
- Cuando se fortalece la precondición, cada llamada recursiva recibe hecho parte del trabajo.

La primera suele ser más natural. La segunda tiene la ventaja de producir soluciones que son más fáciles de transformar en iterativas.

```
// Pre: true
/* Post: devuelve la suma de los valores de un
   vector v */

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

Fortalecimiento de la Pre:

```
/* Pre: 0 <= i < v.size(), s es  la suma de
v[0..i-1]  */
/* Post: devuelve la suma de los valores de v */

int i_Suma(const vector <int> &v, int i, int s);
```

```
// Pre: true
/* Post: devuelve la suma de los valores de un
   vector v */
int Suma(const vector <int> &v){
   return i_suma(v,0,0);
}
```

```
// Pre: true
/* Post: devuelve la suma de los valores de un
   vector v */
int Suma(const vector <int> &v){
    return i_suma(v,0,0);
}

/* Pre: 0 <= i < v.size(), s es  la suma de
v[0..i-1]  */
/* Post: devuelve la suma de los valores de v */

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

Conclusión: Inmersión de una función en otra

Para hacer una inmersión, hemos de:

- Definir una función auxiliar incluyendo los parámetros adicionales, con sus nuevas pre y post.

- Definir la función en términos de la función auxiliar, quizá fijando los nuevos argumentos e ignorando algunos resultados.

Conclusión: Inmersión de una función en otra

En la especificación de la función auxiliar podemos:

- Debilitar la post, ya que la función original, en general, solo usaría parte de los resultados de la función auxiliar.

o bien

- Hacemos la pre más fuerte, suponiendo que las llamadas recursivas harán parte del trabajo

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

Recursividad lineal final (tail recursion)

- Las funciones recursivas lineales finales son fácilmente transformables en funciones iterativas.

Recursividad lineal final: el factorial

```
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);
}
```

Transformación a iterativo: factorial

```
int fact_iter(int n){
    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;
}
```

Recursividad lineal final: suma de un vector

```
/* Pre: 0 <= i < v.size(), s es  la suma de
v[0..i-1]  */
/* Post: devuelve la suma de los valores de v */

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

Llamada inicial: `Suma(v)` ≡ `i_Suma(v,0,0)`

Transformación a iterativo: suma de un vector

```
int suma_iter(int n){
     int i = 0; int sum = 0; //de la llamada inicial
    // Invariante: f = i! and i ≤ n
     while (i != v.size()) {
          sum = sum + v[i]; //de la llamada recursiva
          i = i+1;
     }
     return sum;
}
```

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;
}
```

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);
   }
}
```

Igualdad de pilas

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

bool ig_iter(Stack <int>& p1, Stack <int>& p2) {
   while (not p1.empty() and not p2.empty())
      if (p1.top() != p2.top()) return false;
      else {
         p1.pop();
         p2.pop();
      }
}
   return p1.empty() and p2.empty();
}
```