



Programación 2

Corrección de programas iterativos

Fernando Orejas

1. Corrección de programas
2. Corrección de programas iterativos
3. Diseño inductivo de programas iterativos

Corrección de programas

Definición

Un programa es correcto si para todos los valores posibles que cumplen la Precondición, el programa termina y los resultados cumplen la Postcondición.

// Exponenciación

// Pre: $x = A$, $y = B \geq 0$

// Post: El resultado p es A^B

```
int p = 1;
while (y > 0) {
    y = y - 1;
    p = p*x;
}
```

¿Cómo podemos probar que un programa es correcto?

No podemos probar que funciona para cada valor

¿Cómo podemos probar que un programa es correcto?

No podemos probar que funciona para cada valor

No basta explicar cómo se ejecuta el programa

// Exponenciación (versión 2)

// Pre: $x == A$, $y == B \geq 0$

// Post: El resultado p es A^B

```
int p = 1;
while (y > 0) {
    if (y % 2 == 0) {
        x = x * x;
        y = y / 2;
    }
    else {
        p = p * x;
        y = y - 1;
    }
}
```


¿Cómo podemos probar que un programa es correcto?

No podemos probar que funciona para cada valor

No basta explicar cómo se ejecuta el programa

Razonamiento genérico sobre los valores + Inducción

Razonamiento sobre programas

Un estado de un programa es la n -upla de valores que tienen las variables en un momento dado, más la entrada y la salida

Una aserción es la descripción lógica de un conjunto de estados

Razonamiento sobre programas

Método: anotamos el programa con aserciones que describen los estados en distintos puntos y argumentamos que la anotación es correcta

Un programa es (parcialmente) correcto si es cierto que:

// Pre

Programa

//Post

```
// Intercambio de dos variables enteras
```

```
// Pre: x == A, y == B.
```

```
// Post: x == B, y == A
```

```
// Intercambio de dos variables enteras
```

```
// Pre: x == A, y == B.
```

```
    x = x + y;
```

```
    y = x - y;
```

```
    x = x - y;
```

```
// Post: x == B, y == A
```

// Intercambio de dos variables enteras

// Pre: $x = A$, $y = B$.

$x = x + y;$

// $x = A+B$, $y = B$

$y = x - y;$

// $x = A+B$, $y = A$

$x = x - y;$

// Post: $x = B$, $y = A$

// Exponenciación

// Pre: $x = A$, $y = B \geq 0$

int p = 1;

// $x = A$, $y = B \geq 0$, $p = 1$

while (y > 0) {

 y = y - 1;

 p = p*x;

}

// Post: El resultado p es A^B

Corrección de un programa con un bucle

Esquema básico:

// Pre: ...

Inicialización

```
    while (C) {  
        B  
    }
```

Tratamiento final

// Post: ...

Corrección de un programa con un bucle

Esquema básico:

```
// Pre: ...  
Inicialización  
// Pre del bucle: ...  
    while (C) {  
        B  
    }  
// Post del bucle: ...  
Tratamiento final  
// Post: ...
```

// Exponenciación

// Pre: $x = A, y = B \geq 0$

```
int p = 1;
```

// Pre del bucle: $x = A, y = B \geq 0, p = 1$

```
while (y > 0) {  
    y = y - 1;  
    p = p*x;  
}
```

// Post del bucle: $p = A^B$

// Post: El resultado p es A^B

Invariantes

Un invariante de un bucle es una aserción que siempre se cumple, independientemente del número de iteraciones

Típicamente, se coloca a la entrada del bucle

Describe la relación que existe entre las variables que intervienen en el bucle

Los invariantes se demuestran por inducción

Los invariantes son una buena documentación

// Exponenciación

// Pre: $x == A, y == B \geq 0$

int p = 1;

// Inv: $y \geq 0, A^B = x^y * p$

```
while (y > 0) {  
    y = y - 1;  
    p = p*x;  
}
```

// Post: El resultado p es A^B

Verificación de bucles

Para razonar sobre un bucle hemos de:

1. Probar que la Pre del bucle implica el Invariante
2. Si se cumple Inv y se cumple C, después de ejecutar B, se vuelve a cumplir Invariante
3. Si se cumple Inv y no se cumple C, entonces se cumple la Post del bucle
4. El bucle termina

```
// Pre del bucle  $x = A$ ,  $y = B \geq 0$ ,  $p = 1$ 
```

```
// Inv:  $y \geq 0$ ,  $A^B = x^y * p$ 
```

```
while ( $y > 0$ ) {
```

```
     $y = y - 1$ ;
```

```
     $p = p * x$ ;
```

```
//Pero  $A^B = x^C * D = x^{C-1} * x * D = x^y * p$   
}
```

Suponemos que $y = C$, $p = D$

// Inv: $y \geq 0$, $A^B = x^y * p$

while ($y > 0$) {

$y = y - 1$;

$p = p * x$;

$y = C > 0$, $A^B = x^C * D$, $p = D$

$y = C - 1 \geq 0$, $A^B = x^C * D$, $p = D$

$y = C - 1 \geq 0$, $A^B = x^C * D$, $p = D * x$

//Pero $A^B = x^C * D = x^{C-1} * x * D = x^y * p$
}

// Inv: $y \geq 0$, $A^B = x^y * p$

```
while (y > 0) {  
    y = y - 1;  
    p = p*x;  
}
```

$y \geq 0$, $A^B = x^y * p$, $y \leq 0$

// Si $y > 0$ es falso:

// $y \geq 0$, $y \leq 0$ entonces $y = 0$

// Como consecuencia $A^B = x^0 * p = p$

// Post: $p = A^B$

Terminación de bucles

Para demostrar que un bucle termina hemos de definir una función F sobre las variables del bucle que cumpla (función de cota):

1. Si se cumple el invariante $F(\dots) \geq 0$
2. Si entramos en el bucle $F(\dots) > 0$
3. Si en un punto de la ejecución del bucle $F(\dots) = N$, después de ejecutar una iteración el $F(\dots) < N$.

La idea es que la función de cota nos diga cuántas iteraciones nos quedan como máximo.

// Exponenciación

// Pre: $x == A$, $y == B \geq 0$

int p = 1;

// Inv: $y \geq 0$, $A^B = x^y * p$

while (y > 0) {

 y = y - 1;

 p = p*x;

}

// Post: El resultado p es A^B

Definimos $F(\dots) = y$

Si se cumple el invariante $F(\dots) = y \geq 0$

Si entramos en el bucle $y > 0$

Si antes de ejecutar el bucle $F(\dots) = y = C$

después de ejecutar el bucle $F(\dots) = y = C-1$

Para comprobar que una función es correcta hay que:

1. Inventar un invariante I y una función de cota F .
2. Comprobar que después de la inicialización se cumple I .
3. Si se cumple I y la condición del bucle es cierta, después de ejecutar el cuerpo del bucle se vuelve a cumplir I .
4. I y la negación de la condición del bucle implican la post.
5. Si se cumple el invariante, la función de cota es mayor o igual que cero.
6. Si entramos en el bucle, la función de cota es estrictamente positiva.
7. La función de cota decrece a cada iteración

Número de elementos diferentes en una lista

```
// Pre: cert
```

```
/* Post: Si la lista está vacía retorna 0, en caso  
contrario, retorna el número de elementos diferentes  
que hay en L */
```

```
int ndif (List <int>& L);
```

Número de elementos diferentes en una lista

// Pre: cert

/* Post: Si la lista está vacía retorna 0, en caso contrario, retorna el número de elementos diferentes que hay en L */

```
int ndif (List <int>& L) {  
    int n = 0;  
    List <int>::iterator it = L.begin();  
    while (it != L.end()) {  
        List <int>::iterator it1 = L.begin();  
        while (it1 != it and *it1 != *it) ++it1;  
        if (it1 == it) ++n;  
        ++it;  
    }  
    return n;  
}
```

Bucle interno

- **Función de cota.** F = número de elementos en el intervalo de L entre $it1$ e it (excluido el apuntado por it).
- F siempre es mayor o igual que 0.
- Cuando entramos en el bucle $F > 0$;
- A cada iteración, se reduce F , ya que incrementamos $it1$

Bucle interno

- **Invariante:**

```
/* Inv: el valor de todos los valores que estan en  
nodos anteriores a it1 son diferentes que *it, it1  
apunta a un elemento de L en el intervalo  
L.begin():it. */
```

```
/*Post del bucle interno: *it es la primera vez que  
aparece si y solamente si it == it1 */
```

1. La primera vez, el invariante se cumple trivialmente.
2. Si se cumple Inv e **it != it1 and *it1 != *it**, claramente se vuelve a cumplir el invariante
3. Si no se cumple la condición del bucle, claramente se cumple la Post del bucle.

Bucle externo

- **Función de cota.** F = número de elementos en el intervalo de L entre it y $L.end()$.
- F siempre es positivo.
- Cuando entramos en el bucle $F > 0$;
- A cada iteración, se reduce F , ya que incrementamos it

Bucle externo

- Invariante:

```
/* Inv: n = num de elementos diferentes en L hasta la  
posición apuntada por it (excluida), it apunta a un  
elemento de L */
```

```
//Post del bucle externo: n es el num de elementos  
diferentes en L */
```

1. La primera vez, el invariante se cumple trivialmente.
2. Si se cumple Inv e **it** **!=** **L.end()** , como despues del bucle interno sabemos que el elemento apuntado por it es nuevo si y solo si **it1** **==** **it**, si it apunta a un elemento nuevo se incremenra n en 1, y si no, se deja igual. con lo que al hacer ++it, se vuelve a cumplir el invariante
3. Si no se cumple la condición del bucle y se cumple el invariante, claramente se cumple la Post del bucle.

Invariantes gráficos

Se pide diseñar una función que, dado un vector que solo contiene unos, doses y treses, lo ordene. Es decir:

```
/* Pre: para todo  $i$ ,  $0 \leq i < v.size()$ :  $0 < v[i] < 4$  */
```

```
/* Post  $v$  está ordenado y es una permutación del vector original*/
```

```
void bandera_holandesa(vector<int> v);
```

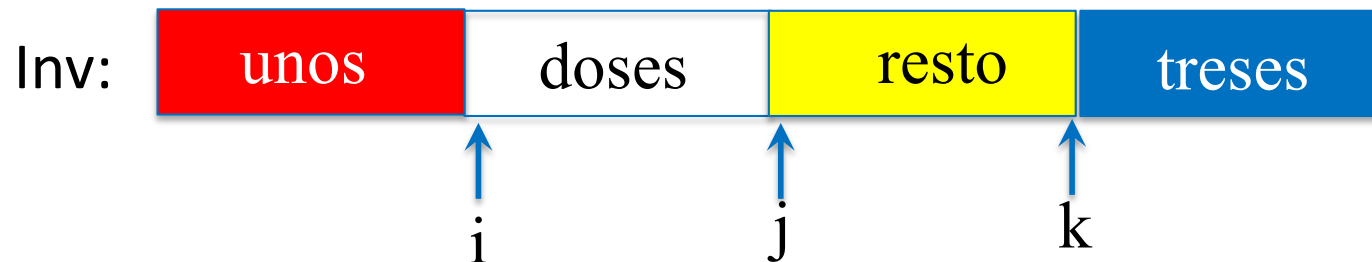
Invariantes gráficos

```
void bandera_holandesa(vector <int> v){  
    int i = 0; int j = 0; int k = v.size()-1;  
    while (j <= k){  
        if (v[j] == 1){  
            swap(v[i],v[j]); ++j; ++i;  
        }  
        else if (v[j] == 2) ++j  
        else {swap(v[k],v[j]); --k;  
        }  
    }  
}
```

Invariantes gráficos

```
void bandera_holandesa(vector <int> v){  
    int i = 0; int j = 0; int k = v.size()-1;  
    while (j <= k){  
        if (v[j] == 1){  
            swap(v[i],v[j]); ++j; ++i;  
        }  
        else if (v[j] == 2) ++j  
        else {swap(v[k],v[j]); --k;  
        }  
    }  
}
```

$$F = k - j + 1$$



Diseño Inductivo

Diseño inductivo

Invertimos el proceso. A partir de una Pre y una Post:

1. Diseñamos un invariante adecuado
2. Definimos una inicialización para que se cumpla el invariante
3. Definimos una condición del bucle que, negada, garantice la postcondición del bucle
4. Diseñamos un cuerpo del bucle que mantenga el invariante.
5. Definimos una finalización para que se cumpla la Post

```
// contar a's
```

```
/* Pre: En la entrada tenemos una secuencia S de  
caracteres acabada en un punto*/
```

```
/* Post: Escribe el número de a's que hay en S */
```

```
// contar a's
```

```
/* Pre: En la entrada tenemos una secuencia S de  
caracteres acabada en un punto*/
```

```
/* Inv: cont es el número de a's en la parte tratada  
de S y c contiene el primer carácter no tratado de S  
*/
```

```
/* Post: Escribe el número de a's que hay en S */
```



```
/* Pre: En la entrada tenemos una secuencia S de  
caracteres acabada en un punto*/
```

```
int main() {
```

```
    // Inv: cont es el número de a's en la parte tratada  
    //      de S y c contiene el primer carácter no  
    //      tratado de S
```

```
while (      ) {
```

```
    }
```

```
}
```

```
/* Post: Escribe el número de a's que hay en S */
```

```
/* Pre: En la entrada tenemos una secuencia S de
caracteres acabada en un punto*/
int main() {
    char c;
    cin >> c;
    int cont = 0;
    // Inv: cont es el número de a's en la parte tratada
    //       de S y c contiene el primer carácter no
    //       tratado de S
    while (      ) {

    }

}

/* Post: Escribe el número de a's que hay en S */
```

```
/* Pre: En la entrada tenemos una secuencia S de
caracteres acabada en un punto*/
int main() {
    char c;
    cin >> c;
    int cont = 0;
    // Inv: cont es el número de a's en la parte tratada
    //       de S y c contiene el primer carácter no
    //       tratado de S
    while (c != '.') {

    }

}

/* Post: Escribe el número de a's que hay en S */
```

```
/* Pre: En la entrada tenemos una secuencia S de
caracteres acabada en un punto*/
int main() {
    char c;
    cin >> c;
    int cont = 0;
    // Inv: cont es el número de a's en la parte tratada
    //       de S y c contiene el primer carácter no
    //       tratado de S
    while (c != '.') {
        if (c == 'a') cont = cont + 1;
        cin >> c;
    }

}

/* Post: Escribe el número de a's que hay en S */
```

```
/* Pre: En la entrada tenemos una secuencia S de
caracteres acabada en un punto*/
int main() {
    char c;
    cin >> c;
    int cont = 0;
    // Inv: cont es el número de a's en la parte tratada
    //       de S y c contiene el primer carácter no
    //       tratado de S
    while (c != '.') {
        if (c == 'a') cont = cont + 1;
        cin >> c;
    }
    // cont es el número de a's en S

}
/* Post: Escribe el número de a's que hay en S */
```

```
/* Pre: En la entrada tenemos una secuencia S de
caracteres acabada en un punto*/
int main() {
    char c;
    cin >> c;
    int cont = 0;
    // Inv: cont es el número de a's en la parte tratada
    //       de S y c contiene el primer carácter no
    //       tratado de S
    while (c != '.') {
        if (c == 'a') cont = cont + 1;
        cin >> c;
    }
    // cont es el número de a's en S
    cout << cont << endl;
}
/* Post: Escribe el número de a's que hay en S */
```

```

/* Pre: En la entrada tenemos una secuencia S de
caracteres acabada en un punto*/
int main() {
    char c;
    cin >> c;
    int cont = 0;
    // Inv: cont es el número de a's en la parte tratada
    //       de S y c contiene el primer carácter no
    //       tratado de S
    while (c != '.') {
        if (c == 'a') cont = cont + 1;
        cin >> c;
    }
    // cont es el número de a's en S
    cout << cont << endl;
}
/* Post: Escribe el número de a's que hay en S */
F(...) = longitud de S

```

// Exponenciación (versión 1)

// Pre: $x == A$, $y == B \geq 0$

// Post: El resultado p es A^B

```
int p = 1;
```

```
// Inv:  $y \geq 0$ ,  $A^B = x^y * p$ 
```

```
while (y > 0) {
```

```
    y = y - 1;
```

```
    p = p*x;
```

```
}
```


Si: $A^B = x^c * d$ y c es par

entonces: $A^B = (x*x)^{c/2} * d$

ya que $(x^2)^{c/2} = x^c$

Como consecuencia el siguiente programa podría calcular el exponencial:

```
int p = 1;
// Inv:  $y \geq 0, A^B = x^y * p$ 
while (y > 0) {
    if (y % 2 == 0) {
        x = x * x;
        y = y / 2;
    }
    else {
        p = p * x;
        y = y - 1;
    }
}
// Post:  $A^B = p$ 
```

Inserción en una lista ordenada L1 de los elementos de otra lista ordenada L2

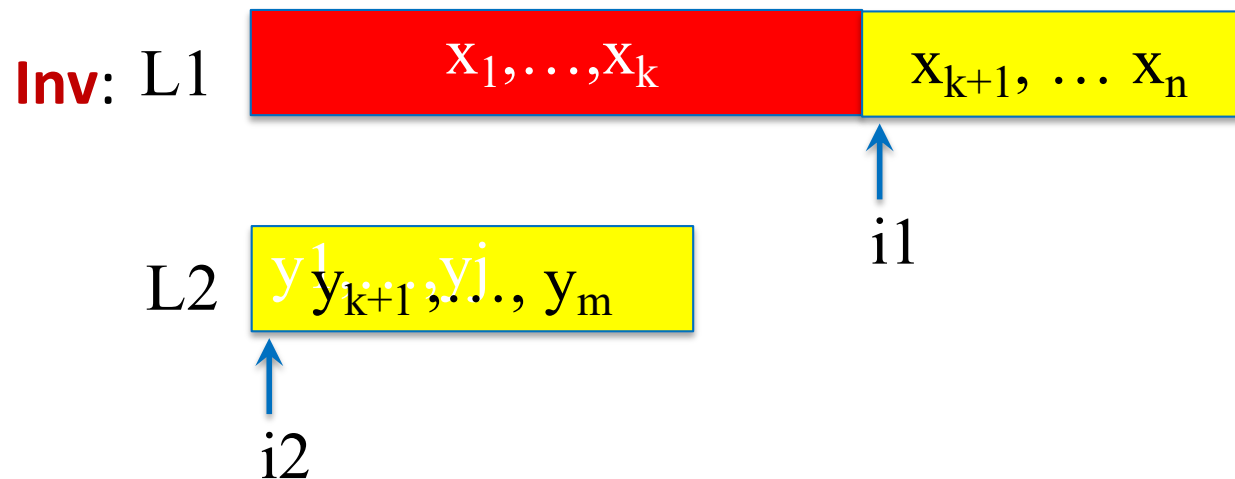
/* Pre: $L1=[x_1, \dots, x_n]$, $L2=[y_1, \dots, y_m]$ y las dos listas están ordenadas */

/* Post: L1 contiene $x_1, \dots, x_n, y_1, \dots, y_m$ y está ordenada, L2 esta vacia */

Inserción en una lista ordenada L1 de los elementos de otra lista ordenada L2

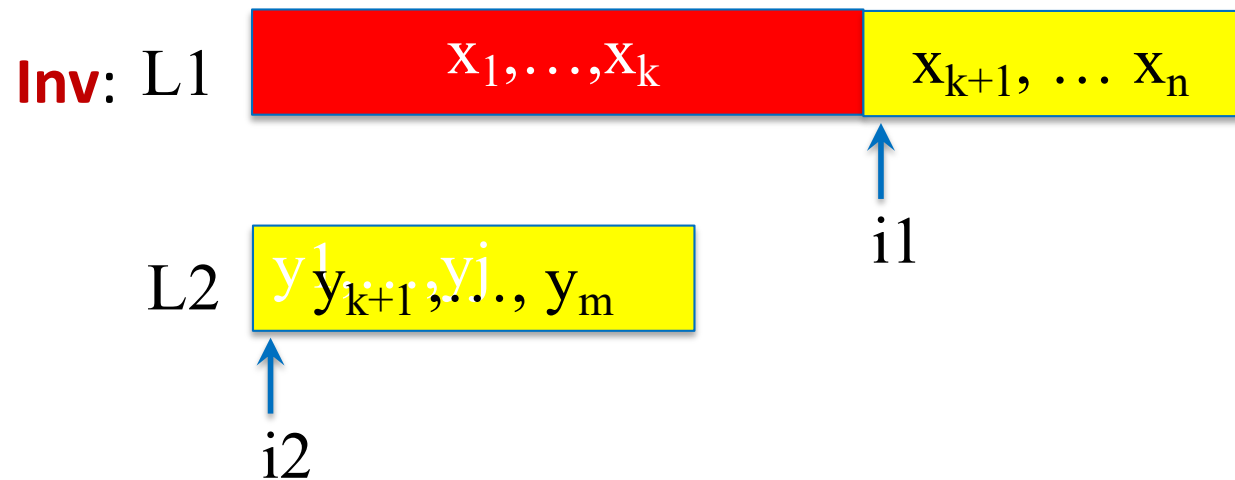
/* Pre: $L1=[x_1, \dots, x_n]$, $L2=[y_1, \dots, y_m]$ y las dos listas están ordenadas */

/* Post: L1 contiene $x_1, \dots, x_n, y_1, \dots, y_m$ y está ordenada, L2 esta vacia */



L1 y L2 están ordenadas, todos los elementos de L2 son mayores que x_k y L1 contiene $x_1, \dots, x_n, y_1, \dots, y_k$

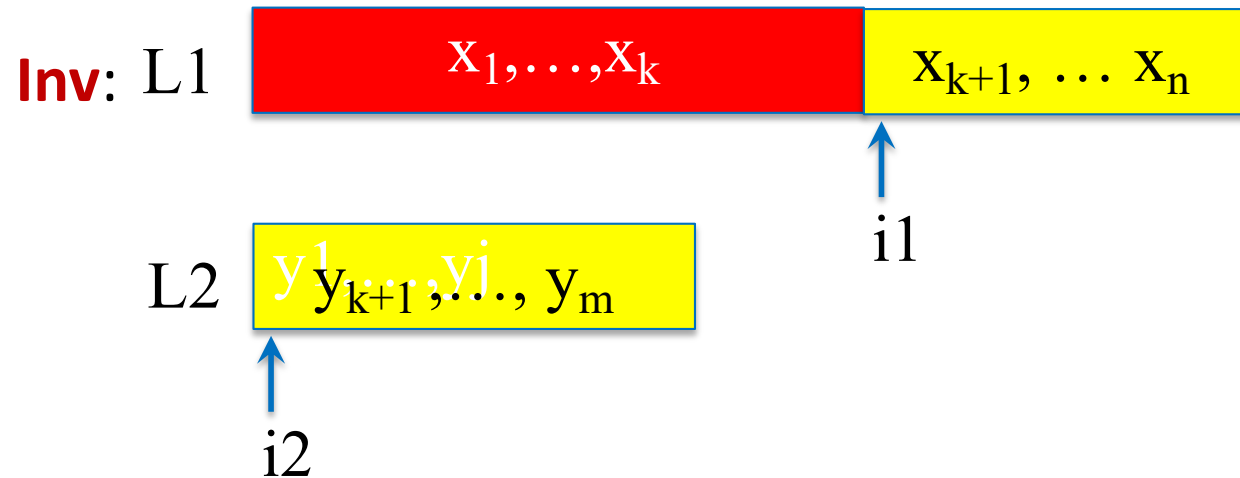
/* Pre: $L1=[x_1, \dots, x_n]$, $L2=[y_1, \dots, y_m]$ y las dos listas están ordenadas */



L1 y L2 están ordenadas, todos los elementos de L2 son mayores que x_k y L1 contiene x_1, \dots, x_n , y_1, \dots, y_k

//Iniciación:

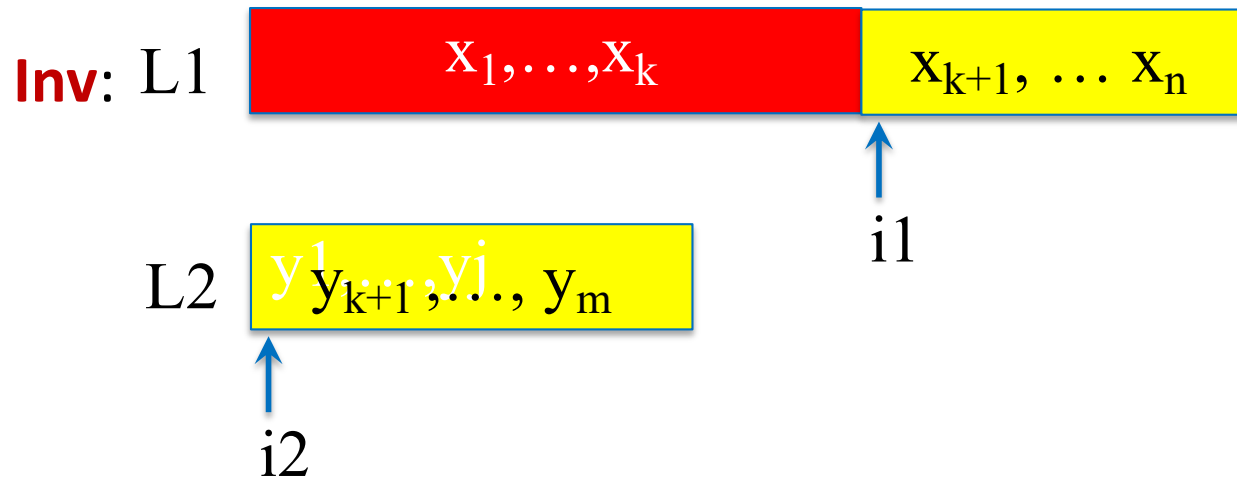
```
list<int>::iterator i1 = L1.begin();  
list<int>::iterator i2 = L2.begin();
```



L1 y L2 están ordenadas, todos los elementos de L2 son mayores que x_k y L1 contiene $x_1, \dots, x_n, y_1, \dots, y_k$

//cuerpo del bucle:

```
if (*i1 < *i2) ++i1;
else {
    L1.insert(i1,*i2); i2 = L2.erase(i2);
}
```



L1 y L2 están ordenadas, todos los elementos de L2 son mayores que x_k y L1 contiene x_1, \dots, x_n , y_1, \dots, y_k

//condición del bucle:

($i1 \neq L1.end()$ and $i2 \neq L2.end()$)

Es decir, a la salida del bucle se cumpliría:

/* L1 y L2 están ordenadas, todos los elementos de L2 son mayores que x_k , L1 contiene x_1, \dots, x_n , y_1, \dots, y_k e ($i1 == L1.end()$ o $i2 == L2.end()$) */

```
/* L1 y L2 están ordenadas, todos los elementos de L2 son  
mayores que  $x_k$ , L1 contiene  $x_1, \dots, x_n$ ,  $y_1, \dots, y_k$  e  $(i1 == L1.end()$   
o  $i2 = L2.end())$  */
```

```
L1.splice(i1,L2);
```

```
/* L1 está ordenada, L2 está vacía, todos los elementos de  
L2, mayores que  $x_k$  y L1 contiene  $x_1, \dots, x_n$ ,  $y_1, \dots, y_m$  e */
```

Es decir, la función sería:

```
void fusion(list<int> L1, list<int> L2){  
    list<int>::iterator i1 = L1.begin();  
    list<int>::iterator i2 = L2.begin();  
    while (i1 != L1.end() and i2 != L2.end() ) {  
        if (*i1 < *i2) ++i1;  
        else {  
            L1.insert(i1,*i2); i2 = L2.erase(i2);  
        }  
    }  
    L1.splice(i1,L2);  
}
```

F(...) = dist. de i1 a L1.end()+ dist. de i2 a L2.end()