

Algoritmos sobre secuencias ya ordenadas

Algoritmos y Estructuras de Datos I

Apareo (merge) de secuencias ordenadas

- **Problema:** Dadas dos secuencias ordenadas, **unir** ambas secuencias en un única secuencia ordenada.
- Especificación:

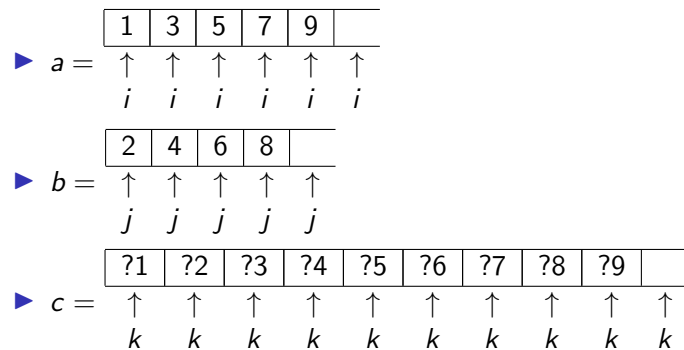

```

proc merge(in a, b : seq<ℤ>, out result : seq<ℤ>){
  Pre {ordenado(a) ∧ ordenado(b)}
  Post {ordenado(result) ∧ mismos(result, a ++ b)}
}

pred mismos(s, t : seq<ℤ>){
  (∀x : ℤ)(#apariciones(s, x) = #apariciones(t, x))
}
      
```
- ¿Cómo lo podemos implementar?
 - Podemos copiar los elementos de a y b a la secuencia c , y después ordenar la secuencia c .
 - Pero selection sort e insertion sort iteran aproximadamente $|c|^2$
 - ¿Se podrá aparear ambas secuencias **en una única pasada?**

Apareo de secuencias ordenadas

Ejemplo:



Apareo de secuencias

- ¿Qué invariante de ciclo tiene esta implementación?
- $$I \equiv \text{ordenado}(a) \wedge \text{ordenado}(b)$$

$$\wedge ((0 \leq i \leq |a| \wedge 0 \leq j \leq |b| \wedge k = i + j)$$

$$\wedge_L (\text{mismos}(a[0, i] ++ b[0, j], c[0, k]) \wedge \text{ordenado}(c[0, k])))$$
- ¿Qué función variante debería tener esta implementación?

$$fv = |a| + |b| - k$$

Apareo de secuencias

```
1 vector<int> merge(vector<int> &a, vector<int> &b) {
2     vector<int> c(a.size()+b.size());
3     int i = 0; // Para recorrer a
4     int j = 0; // Para recorrer b
5     int k = 0; // Para recorrer c
6
7     while( k < c.size() ) {
8         if( /*Si tengo que avanzar i */ ) {
9             c[k++] = a[i++];
10        } else if( /* Si tengo que avanzar j */ ) {
11            c[k++] = b[j++];
12        }
13    }
14    return c;
15 }
```

- ▶ ¿Cuándo tengo que avanzar i ? Cuando j está fuera de rango ó cuando i y j están en rango y $a[i] < b[j]$
- ▶ ¿Cuándo tengo que avanzar j ? Cuando no tengo que avanzar i

Apareo de secuencias

```
1 vector<int> merge(vector<int> &a, vector<int> &b) {
2     vector<int> c(a.size()+b.size());
3     int i = 0; // Para recorrer a
4     int j = 0; // Para recorrer b
5     int k = 0; // Para recorrer c
6
7     while( k < c.size() ) {
8         if( j>=b.size() || && a[i] < b[j] ) {
9             c[k++] = a[i++];
10        } else {
11            c[k++] = b[j++];
12        }
13    }
14    return c;
15 }
```

- ▶ Al terminar el ciclo, ¿ya está la secuencia c con los valores finales?

Apareo de secuencias

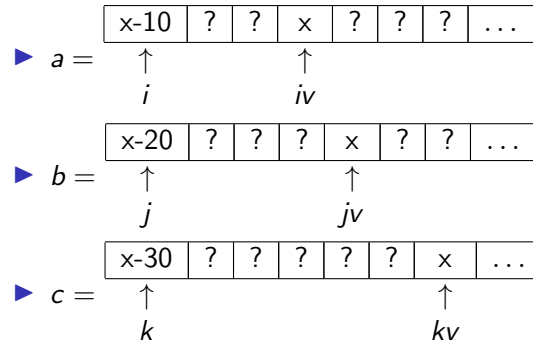
```
1 vector<int> merge(vector<int> &a, vector<int> &b) {
2     vector<int> c(a.size()+b.size());
3     int i = 0; // Para recorrer a
4     int j = 0; // Para recorrer b
5     int k = 0; // Para recorrer c
6
7     while( k < c.size() ) {
8         if( j>=b.size() || a[i] < b[j] ) {
9             c[k++] = a[i++];
10        } else {
11            c[k++] = b[j++];
12        }
13    }
14    return c;
15 }
```

- ▶ ¿Cuántas iteraciones realiza este programa en **peor caso** (como máximo)?
 - ▶ Realiza a lo sumo $|a| + |b|$ iteraciones

The welfare crook

- ▶ **Problema:** Dadas tres secuencias ordenadas, sabemos que hay al menos un elemento en común entre ellos. Encontrar los índices donde está al menos uno de estos elementos repetidos.
- ▶ Usamos las **metavariables** iv , jv y k_v para denotar las posiciones en las que las secuencias coinciden por primera vez.
- ▶ $\text{proc crook}(\text{in } a, b, c : \text{seq}(\mathbb{Z}), \text{out } i, j, k : \mathbb{Z})\{\$
 - Pre $\{ \text{ordenado}(a) \wedge \text{ordenado}(b) \wedge \text{ordenado}(c) \wedge ((0 \leq iv < |a| \wedge 0 \leq jv < |b| \wedge 0 \leq kv < |c|) \wedge a[iv] = b[jv] = c[kv]) \}$
 - Post $\{ i = iv \wedge j = jv \wedge k = kv \}$ $\}$

The welfare crook



- ▶ ¿Cuál es el invariante de esta implementación?

$$I \equiv 0 \leq i \leq iv \wedge 0 \leq j \leq jv \wedge 0 \leq k \leq kv$$

- ▶ ¿Cuál es una función variante para esta implementación?

$$fv = (iv - i) + (jv - j) + (kv - k)$$

The welfare crook

- ▶ Comenzamos con $i = j = k = 0$, y vamos subiendo el valor de estas variables.

```

1 void welfareCrook(vector<int> &a, vector<int> &b, vector<int> &c,
2   int &i, int &j, int &k) {
3   i = 0, j = 0, k = 0;
4   while( a[i] != b[j] || b[j] != c[k] ) {
5       // Incrementar i, j o k!
6   }
7   // i=iv, j=jv, k=kv
8 }
```

The welfare crook

- ▶ ¿A cuál de los índices podemos incrementar?
- ▶ Alcanza con avanzar cualquier índice que no contenga al máximo entre $a[i]$, $b[j]$ y $c[k]$
- ▶ En ese caso, el elemento que no es el máximo no es el elemento buscado

```

1 i = 0, j = 0, k = 0;
2 while( a[i] != b[j] || b[j] != c[k] ) {
3     if( a[i] < b[j] ) {
4         i++;
5     } else if( b[j] < c[k] ) {
6         j++;
7     } else {
8         k++;
9     }
10 }
```

The welfare crook

```

1 i = 0, j = 0, k = 0;
2 while( a[i] != b[j] || b[j] != c[k] ) {
3     if( a[i] < b[j] ) {
4         i++;
5     } else if( b[j] < c[k] ) {
6         j++;
7     } else {
8         k++;
9     }
10 }
```

- ▶ ¿Por qué se preserva el invariante?

1. $I \wedge B \wedge a[i] < b[j]$ implica $i < iv$, entonces es seguro avanzar i .
2. $I \wedge B \wedge b[j] < c[k]$ implica $j < jv$, entonces es seguro avanzar j .
3. $I \wedge B \wedge a[i] \geq b[j] \wedge b[j] \geq c[k]$ implica $k < kv$, por lo tanto es seguro avanzar k .

The welfare crook

```
1 void welfareCrook(vector<int> &a, vector<int> &b, vector<int> &c,  
2   int &i, int &j, int &k) {  
3   i = 0, j = 0, k = 0;  
4   while( a[i] != b[j] || b[j] != c[k] ) {  
5       if( a[i] < b[j] ) {  
6           i++;  
7       } else if( b[j] < c[k] ) {  
8           j++;  
9       } else {  
10          k++;  
11      }  
12  }  
13 }
```

- ▶ ¿Cuántas iteraciones realiza este programa en **peor caso** (i.e. como máximo)?
 - ▶ Como máximo tiene que realizar $|a| + |b| + |c|$ iteraciones

Bibliografía

- ▶ Vickers et al. - Reasoned Programming
 - ▶ 6.6 - Sorted Merge (apareo)
- ▶ David Gries - The Science of Programming
 - ▶ Chapter 16 - Developing Invariants (Welfare Crook)