

Algoritmos sobre secuencias ya ordenadas

Algoritmos y Estructuras de Datos I

1

Apareo (merge) de secuencias ordenadas

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


```

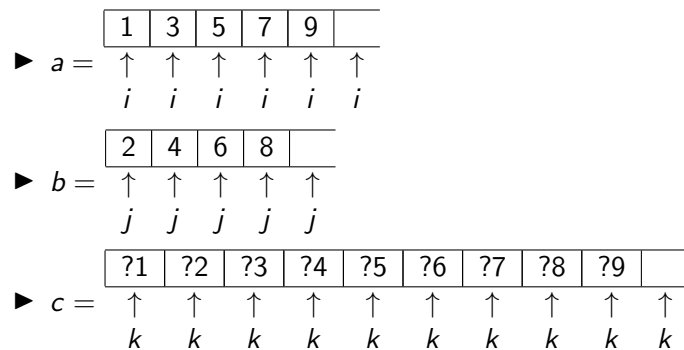
proc merge(in a, b : seq<Z>, out result : seq<Z>){
  Pre {ordenado(a) ∧ ordenado(b)}
  Post {ordenado(result) ∧ mismos(result, a ++ b)}
}

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**?

2

Apareo de secuencias ordenadas

Ejemplo:



3

Apareo de secuencias

- ¿Qué invariante de ciclo tiene esta implementación?
- $$\begin{aligned}
 I \equiv & \text{ordenado}(a) \wedge \text{ordenado}(b) \wedge |c| = |a| + |b| \\
 & \wedge ((0 \leq i \leq |a| \wedge 0 \leq j \leq |b| \wedge k = i + j) \\
 & \wedge_L (\text{mismos}(\text{subseq}(a, 0, i) ++ \text{subseq}(b, 0, j), \text{subseq}(c, 0, k)) \\
 & \wedge \text{ordenado}(\text{subseq}(c, 0, k))) \\
 & \wedge i < |a| \rightarrow_L (\forall t : \mathbb{Z})(0 \leq t < j \rightarrow_L b[t] \leq a[i]) \\
 & \wedge j < |b| \rightarrow_L (\forall t : \mathbb{Z})(0 \leq t < i \rightarrow_L a[t] \leq b[j])
 \end{aligned}$$
- ¿Qué función variante debería tener esta implementación?
- $$fv = |a| + |b| - k$$

4

Apareo de secuencias

```
vector<int> merge(vector<int> &a, vector<int> &b) {
    vector<int> c(a.size()+b.size());
    int i = 0; // Para recorrer a
    int j = 0; // Para recorrer b
    int k = 0; // Para recorrer c

    while( k < c.size() ) {
        if( /*Si tengo que avanzar i */ ) {
            c[k++] = a[i++];
        } else if( /* Si tengo que avanzar j */ ) {
            c[k++] = b[j++];
        }
    }
    return c;
}
```

- ¿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

5

Apareo de secuencias

```
vector<int> merge(vector<int> &a, vector<int> &b) {
    vector<int> c(a.size()+b.size(),0);
    int i = 0; // Para recorrer a
    int j = 0; // Para recorrer b
    for(int k=0; k < c.size(); k++) {
        if( j>=b.size() || ( i<a.size() && a[i] < b[j] ) ) {
            c[k] = a[i];
            i++;
        } else {
            c[k] = b[j];
            j++;
        }
    }
    return c;
}
```

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

6

Apareo de secuencias

```
vector<int> merge(vector<int> &a, vector<int> &b) {
    vector<int> c(a.size()+b.size(),0);
    int i = 0; // Para recorrer a
    int j = 0; // Para recorrer b
    for(int k=0; k < c.size(); k++) {
        if( j>=b.size() || ( i<a.size() && a[i] < b[j] ) ) {
            c[k] = a[i];
            i++;
        } else {
            c[k] = b[j];
            j++;
        }
    }
    return c;
}
```

- ¿Cuál es el tiempo de ejecución de peor caso de merge?

7

Tiempo de ejecución de peor caso

```
vector<int> merge(vector<int> &a, vector<int> &b) {
    vector<int> c(a.size()+b.size(),0); // inicializa  $O(|a|+|b|)$ 
    int i = 0; //  $O(1)$ 
    int j = 0; //  $O(1)$ 
    for(int k=0; k < c.size(); k++) { //  $O(1)$ 
        if( j>=b.size() || ( i<a.size() && a[i] < b[j] ) ) { //  $O(1)$ 
            c[k] = a[i]; //  $O(1)$ 
            i++; //  $O(1)$ 
        } else {
            c[k] = b[j]; //  $O(1)$ 
            j++; //  $O(1)$ 
        }
    }
    return c; // copia secuencia  $O(|a|+|b|)$ 
}
```

- Sea $n = |c| = |a| + |b|$
- El while se ejecuta $n + 1$ veces.
- Por lo tanto, $T_{merge}(n) \in O(n)$

8

The welfare crook

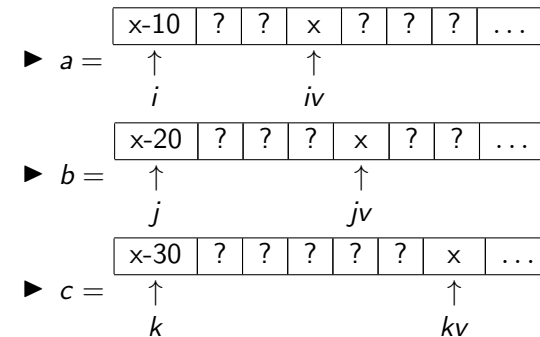
- **Problema:** Dadas tres secuencias ordenadas, sabemos que hay al menos un elemento en común entre ellos. Encontrar los índices donde está al menos uno de estos elementos repetidos.
- Usamos iv , jv y k_v para denotar las posiciones en las que las secuencias coinciden.

```

► proc crook(in a, b, c : seq⟨ℤ⟩, out i, j, k : ℤ){
    Pre {ordenado(a) ∧ ordenado(b) ∧ ordenado(c)
        ∧ (∃iv, jv, kv : ℤ)
        ((0 ≤ iv < |a| ∧ 0 ≤ jv < |b| ∧ 0 ≤ kv < |c|)
        ∧L a[iv] = b[jv] = c[kv])}
    Post {(0 ≤ i < |a| ∧ 0 ≤ j < |b| ∧ 0 ≤ k < |c|) ∧L
        a[i] = b[j] = c[k]}
}

```

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?

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

The welfare crook

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

```
void crook(vector<int> &a, vector<int> &b, vector<int> &c,
          int &i, int &j, int &k) {
    i = 0, j = 0, k = 0;
    while( a[i] != b[j] || b[j] != c[k] ) {
        // Incrementar i, j o k!
    }
    // i=iv, j=jv, k=kv
}
```

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

```
i = 0, j = 0, k = 0;
while( a[i] != b[j] || b[j] != c[k] ) {
    if( a[i] < b[j] ) {
        i++;
    } else if( b[j] < c[k] ) {
        j++;
    } else {
        k++;
    }
}
```

The welfare crook

```
i = 0, j = 0, k = 0;
while( a[i] != b[j] || b[j] != c[k] ) {
    if( a[i] < b[j] ) {
        i++;
    } else if( b[j] < c[k] ) {
        j++;
    } else {
        k++;
    }
}
```

► ¿Por qué se preserva el invariante?

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

13

The welfare crook

```
void crook(vector<int> &a, vector<int> &b, vector<int> &c,
           int &i, int &j, int &k) {
    i = 0, j = 0, k = 0;
    while( a[i] != b[j] || b[j] != c[k] ) {
        if( a[i] < b[j] ) {
            i++;
        } else if( b[j] < c[k] ) {
            j++;
        } else {
            k++;
        }
    }
}
```

► ¿Cuántas iteraciones realiza este programa en **peor caso** (i.e. como máximo)?

14

Tiempo de ejecución de peor caso

```
void crook(vector<int> &a, vector<int> &b, vector<int> &c,
           int &i, int &j, int &k) {
    i = 0, j = 0, k = 0; //  $O(1)$ 
    while( a[i] != b[j] || b[j] != c[k] ) { //  $O(1)$ 
        if( a[i] < b[j] ) { //  $O(1)$ 
            i++; //  $O(1)$ 
        } else if( b[j] < c[k] ) { //  $O(1)$ 
            j++; //  $O(1)$ 
        } else {
            k++; //  $O(1)$ 
        }
    }
}
```

- El while se ejecuta como mucho $|a| + |b| + |c|$ veces
- Sea $n = |a|$, $m = |b|$, $l = |c|$,
- $T_{crook}(n, m, l) \in O(n + m + l)$

15

Bibliografía

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

16