Algoritmos y Estructuras de Datos I

Segundo cuatrimestre de 2019

Departamento de Computación - FCEyN - UBA

Algoritmos sobre secuencias ya ordenadas

Apareo (merge) de secuencias ordenadas

- **Problema:** Dadas dos secuencias ordenadas, unir ambas secuencias en un única secuencia ordenada.
- Especificación: proc $merge(\text{in } a, b : seq\langle\mathbb{Z}\rangle, \text{c } result : seq\langle\mathbb{Z}\rangle)\{$
  Pre $\{ordenado(a) \ \wedge \ ordenado(b)\}$
  Post $\{ordenado(result) \ \wedge \ mismos(result, a++b)\}$
  $\}$

  pred $mismos(s, t : seq\langle\mathbb{Z}\rangle)\{$
  $(\forall x : \mathbb{Z})(\#apariciones(s, x) = \#apariciones(t, x))$
  $\}$
- ¿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 apróximadamente $|c|^2$
  - ¿Se podrá aparear ambas secuencias en una única pasada?

Apareo de secuencias ordenadas

Ejemplo:

- $a =$

1	3	9	11	15	
↑	↑	↑	↑	↑	↑
i	i	i	i	i	i

- $b =$

2	4	6	8	
↑	↑	↑	↑	↑
j	j	j	j	j

- $c =$

?	?	?	?	?	?	?	?	?	
↑	↑	↑	↑	↑	↑	↑	↑	↑	↑
k	k	k	k	k	k	k	k	k	k

Apareo de secuencias

- ¿Qué tiene que decir el invariante de ciclo tiene esta implementación?
  - a y b están ordenadas
  - la longitud de c es igual a la de a más la de b
  - i es índice de a y j de b y juntos suman k
  - la subsecuencia de c hasta k tiene los mismo elementos que a hasta i agregados a los de b hasta j.
  - además está ordenada
  - si $i < |a|$ todos los elementos de b hasta j son menores o iguales a $a[i]$
  - si $j < |b|$ todos los elementos de a hasta i son menores o iguales a $b[i]$
- ¿Qué función variante debería tener esta implementación?

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

Apareo de secuencias

- ¿Qué invariante de ciclo tiene esta implementación?

$$
\begin{aligned}
\mathbb{I} \quad \equiv \quad & ordenado(a) \wedge ordenado(b) \wedge |c| = |a| + |b| \\
\wedge \quad & ((0 \leq i \leq |a| \ \wedge \ 0 \leq j \leq |b| \ \wedge \ k = i + j) \\
\wedge_L \quad & \Big(mismos(subseq(a,0,i) + +subseq(b,0,j), subseq(c,0,k)\Big) \\
\wedge \quad & ordenado(subseq(c,0,k)))) \\
\wedge \quad & i < |a| \ \rightarrow_L \ (\forall t : \mathbb{Z})(0 \leq t < j \rightarrow_L b[t] \leq a[i]) \\
\wedge \quad & j < |b| \ \rightarrow_L \ (\forall t : \mathbb{Z})(0 \leq t < i \rightarrow_L a[t] \leq b[j])
\end{aligned}
$$

- Función variante:

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

Apareo de secuencias

```
vector<int> merge(vector<int> &a, vector<int> &b) {
  /* inicializacion de variables */
  while( k < c.size() ) {
    if( /*Si tengo que avanzar i */ ) {
      c[k] = a[i];
      i++;
    } else if(/* Si tengo que avanzar j */)  {
      c[k] = b[j];
      j++;
    }
    k++
  }
  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

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?

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`?

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 `for` se ejecuta $n + 1$ veces.
- Por lo tanto, $T_{merge}(n) \in O(n)$

The welfare crook (Gries'1981)

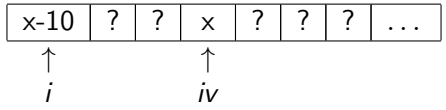
- **Problema:** Dadas tres secuencias ordenadas, sabemos que hay al menos un elemento en común entre ellos. Encontrar los índices donde está al menos uno de estos elementos repetidos.
- Usamos iv, jv y kv para denotar las posiciones en las que las secuencias coinciden.
- proc $crook$(in $a, b, c : seq\langle\mathbb{Z}\rangle$, out $i, j, k : \mathbb{Z}$){

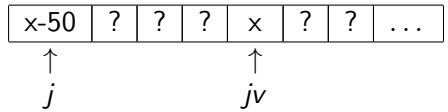
  Pre $\{ordenado(a) \ \wedge \ ordenado(b) \ \wedge \ ordenado(c)$
  $\wedge(\exists iv, jv, kv : \mathbb{Z})$
  $((0 \leq iv < |a| \wedge \ 0 \leq jv < |b| \wedge \ 0 \leq kv < |c|)$
  $\wedge_L \ a[iv] = b[jv] = c[kv])\}$

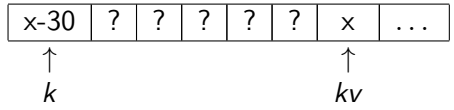
  Post $\{(0 \leq i < |a| \wedge \ 0 \leq j < |b| \wedge \ 0 \leq k < |c|) \wedge_L$
  $a[i] = b[j] = c[k]\}$

  }

The welfare crook

- $a =$ 

- $b =$ 

- $c =$ 

- ¿Cuál es el invariante de esta implementación?
- ¿Cuál es una función variante para esta implementación?

The welfare crook

- ¿Cuál es el invariante de esta implementación?
  $\mathbb{I} \equiv ordenado(a) \ \wedge \ ordenado(b) \ \wedge \ ordenado(c) \wedge$
  $(\exists iv, jv, kv : \mathbb{Z}) \ \ ((0 \leq iv < |a| \ \wedge 0 \leq jv < |b| \wedge$
  $0 \leq kv < |c|) \ \wedge_L \ a[iv] = b[jv] = c[kv]) \wedge$
  $0 \leq i \leq iv \ \wedge \ 0 \leq j \leq jv \ \wedge \ 0 \leq k \leq kv$
- ¿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.

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

The welfare crook

```
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)?

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 $n = |a|$, $m = |b|$, $l = |c|$,
- $T_{crook}(n, m, l) \in O(n + m + l)$

Intervalo

Break!

Incrementar una secuencia binaria

- Una secuencia *binaria* contiene solo 0's y 1's.
- Escribir un programa que incremente en 1 la secuencia binaria. Ejemplos:
  - $\langle 0, 0, 1, 0 \rangle \to \langle 0, 0, 1, 1 \rangle$
  - $\langle 0, 0, 1, 1 \rangle \to \langle 0, 1, 0, 0 \rangle$
  - $\langle 1, 1, 1, 1 \rangle \to \langle 0, 0, 0, 0 \rangle$

Incrementar una secuencia binaria

```
void incrementar(vector<int> &s) {
  int carry = 1;
  for (int i=s.size()-1; i>=0; i--) {
     if (s[i]==1 && carry==1) {
        s[i]=0;
        carry=1;
     } else {
        s[i]=s[i]+carry;
        carry=0;
     }
  }
}
```

- ¿Cuál es el tiempo de ejecución de peor caso de `incrementar`?

Incrementar una secuencia binaria

```
void incrementar(vector<int> &s) {
  int carry = 1; // O(1)
  for (int i=s.size()-1; i>=0; i--) { // O(1)
    if (s[i]==1 && carry==1) { // O(1)
      s[i]=0; // O(1)
      carry=1; // O(1)
    } else {
      s[i]=s[i]+carry; // O(1)
      carry=0; // O(1)
    }
  }
}
```

- Sea $n = |s|$, el cuerpo del `while` se ejecuta n veces
- Por lo tanto, $T_{incrementar}(n) \in O(n)$

Listar secuencias binarias

- Escribir un programa que retorne *todas* las secuencias binarias para una longitud $n \geq 0$
- Observación: usar el programa `incrementar`

Listar secuencias binarias

```
#include <math.h>

vector<vector<int>> listarSecuenciasBinarias(int n) {
  vector<vector<int>> rv(pow(2,n),vector<int>());
  vector<int> s(n,0);
  for (int i=0;i<pow(2,n);i++) {
     rv[i] = s;
     incrementar(s);
  }
  return rv;
}
```


g++

Demo!

Listar secuencias binarias

```
vector<vector<int>> listarSecuenciasBinarias(int n) {
  vector<vector<int>> rv(pow(2,n),vector<int>());
  vector<int> s(n,0);
  for (int i=0;i<pow(2,n);i++) {
     rv[i] = s;
     incrementar(s);
  }
  return rv;
}
```

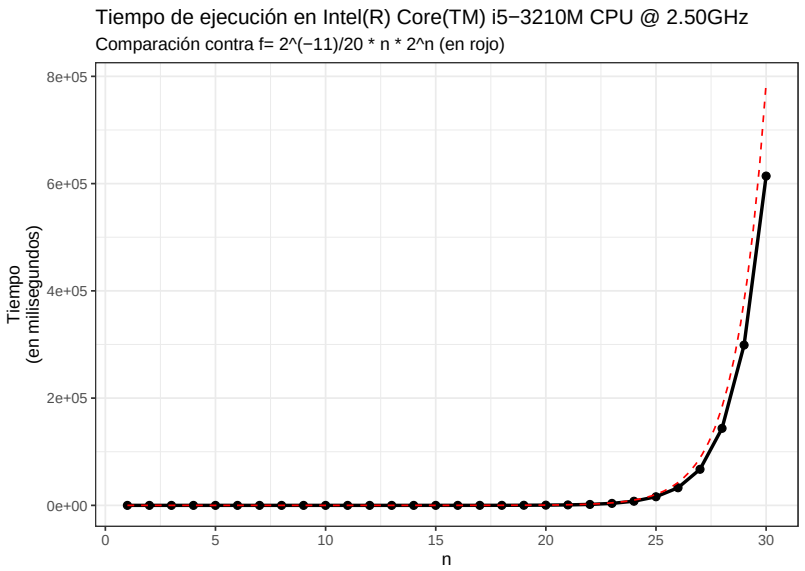
- ¿Cuál es el tiempo de ejecución de peor caso del programa?

Listar secuencias binarias

```
vector<vector<int>> listarSecuenciasBinarias(int n) {
  vector<vector<int>> rv(pow(2,n),vector<int>()); // O(2^n)
  vector<int> s(n,0); // O(n)
  for (int i=0;i<pow(2,n);i++) { //O(1)
    rv[i] = s; // O(n) porque copia toda la secuencia s
    incrementar(s);  // O(n)
  }
  return rv; // O(n*2^n) porque copia la secuencia
             // de secuencias rv
}
```

- El cuerpo del `for` se ejecuta (2^n) veces
- $T_{listarSecuenciasBinarias}(n) \notin O(n)$
- $T_{listarSecuenciasBinarias}(n) \notin O(2^n)$
- $T_{listarSecuenciasBinarias}(n) \in O(2^n + n + n*2^n + n*2^n) = O(n*2^n)$

g++



Bibliografía

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