Algoritmos y Estructuras de Datos I

Segundo cuatrimestre de 2011

Departamento de Computación - FCEyN - UBA

Programación imperativa - clase 5

Otros Algoritmos de sobre secuencias

Problema merge: intercalar dos secuencias ordenadas

Dados dos arreglos ordenados, a y b,
dar un tercer arreglo de salida c
que contenga a los elementos de ambos arreglos de entrada, y
esté ordenado.

Nota: merge(a,b) == sort(a++b)

Problema merge

problema merge(a : [Int], b : [Int], c : [Int], n : Int, m : Int){
  requiere $n == |a| \land m == |b| \land n + m == |c|$;
  requiere $ordenado(a) \land\ ordenado(b)$;
  modifica c;
  asegura $ordenado(c) \land mismos(c, a++b)$;
}

Algoritmo de merge

Mantener un indice para recorrer a otro para b .

Recorrer linealmente los arreglos a y b, asignando de a un elemento por vez en el arreglo de salida c.

El elemento a asignar es el menor entre el elemento actual de a y el actual de b.

Incrementar en 1 el índice del arreglo del que provino el elemento.

Cuando uno de los arreglos de entrada ya esté completamente recorrido, asignar la cola sin recorrer del otro arreglo, desde la posicion actual del arreglo de salida.

Especificación del ciclo de merge

Pc: $i == 0 \wedge j == 0$

invariante: $0 \leq i \leq n \ \wedge \ 0 \leq j \leq m \ \wedge$
$ordenado(c[0..i+j]) \wedge$
$mismos(c[0..i+j), a[0..i) + +b[0..j))$;

variante: $n + m - (i + j)$;

guarda: $i + j < n + m$;

Qc: $i == n \wedge j == m \wedge ordenado(c[0..n+m)) \wedge$

$mismos(c[0..n+m), a[0..n) + +b[0..m))$

Programa merge

```
void merge(int a[], int b[], int c[], int n, int m) {
int i,j = 0;
// Pc: i == 0 ∧ j == 0
while (i+j < n+m) {
// invariante 0 ≤ i ≤ n ∧ 0 ≤ j ≤ m ∧ ordenado(c[0..i + j))∧
    // mismos(c[0..i + j), a[0..i) + +b[0..j));
// variante n + m − (i + j);
    if (i<n && j<m)
            if (a[i]≤ b[j]) { c[i+j]= a[i]; i++; }
            else             { c[i+j]= b[j]; j++; }
    else
            if (i==n) { c[i+j]= b[j]; j++; }
            else      { c[i+j]= a[i]; i++; }
}
// Qc: i == n ∧ j == m ∧ ordenado(c[0..n + m))∧
//       mismos(c[0..n + m), a[0..n) + +b[0..m))
}
```


Variantes de merge

merge entre n elementos

merge y filtrado

merge en paralelo

Problema: cuenta cantidad de ocurrencias

Dado un arreglo a de enteros de dimensión n, cuyos valores están entre 0 y $n-1$, dar un arreglo de salida b,tal que en la posición $b[i]$ indique la cantidad de ocurrencias del entero i en a.

Problema: cuenta cantidad de ocurrencias

problema *ocurrencias*$(a : [Z], n, b : [Z])\{$
requiere $|a| == n$;
requiere $|b| == n$;
requiere todos$([0 \leq a[i] < n \mid i \in [0..n)]$;
modifica b;
asegura $b == [cuenta(i, a) \mid i \in [0..n)]$, where
$cuenta(x, c) = \sum[x == c[j] | j \in [0..|c|)]$
$\}$

Algoritmo fuerza bruta para contar cantidad de ocurrencias

Inicializar b con 0 en todas las posiciones
Para cada entero $i : 0..n-1$
Contar cuantas ocurrencias de i hay en a.
Asignar este valor en $b[i]$.

Cantidad cuadrática de iteraciones.

Algoritmo más eficiente para contar cantidad de ocurrencias, usando sort

Requiere todos$([0 \leq a[i] < n \mid i \in [0..n)]$;

Inicializar b con 0 en todas las posiciones
Ordenar a de menor a mayor
(si no queremos modificar a, usar un arreglo auxiliar)
Usaremos un índice j para recorrer a linealmente
Inicializar $j = 0$
Para cada entero $i : 0..n-1$
Mientras $a[j]$ sea igual a i,
Incrementar $b[i]$ en uno, e incrementar j en uno.
Incrementar i en uno.

Cantidad $O(n \log n)$ de iteraciones.

Notar que este algoritmo también puede usarse en caso de que los elementos de a no sean positivos menores que n: hacemos que la salida sea una lista de pares (i, c), tal que $i \in a$, y c es la cantidad de ocurrencias de i en a.

Algoritmo lineal que cuenta ocurrencias

Este es el más eficiente de los tres, y no usa sort!
Requiere todos$([0 \leq a[i] < n \mid i \in [0..n)]$;

Inicializar el arreglo b de salida con 0 en todas las posiciones

Recorrer linealmente el arreglo de entrada a con un índice i

Incrementar $b[a[i]]$ en uno.
Incrementar i en uno

Cantidad lineal de operaciones.

Especificación del ciclo de 'contar ocurrencias'

Entrada: arreglo a de dimensión n.

Variable local: arreglo b de dimensión n.

Pc: $j == 0 \wedge todos(b[k] == 0 \mid k \in [0..|n)]$

invariante: $0 \leq j \leq n \wedge b == [cuenta[i, a[0..j)] \mid i \in [0..n)]$

variante: $n - j$;

Qc: $j == n \;\wedge\; b == [cuenta(x, a[0..n)) \mid x \in [0..n)]$;

Programa ocurrencias

```
void ocurrencias(int a[], int b[], int n) {
// modifica b
    int j = 0;
    while (j<n)  b[j]==0; j++;
    j=0;
    // Pc: j == 0 ∧ todos(b[k] == 0 | k ∈ [0..|n)]
    while (j < n) {
    // invariante: 0 ≤ j ≤ n ∧ b == [cuenta[i, a[0..j)] | i ∈ [0..n)]
    // variante n − j;
        b[a[j]]++;
        j++;
    }
// Qc: j == n ∧ b == [cuenta(k, a[0..n)) | k ∈ [0..n)]
}
```


Problema: Distancia Hamming

Métrica de la diferencia entre una palabra válida y otra. (Teoría de la información, Richard Hamming, 1950)

La distancia Hamming entre dos palabras se define como el número de símbolos que tienen que cambiarse para transformar una palabra de código válida en otra palabra de código válida.

Ejemplos:
$hamming(1011101, 1001001) = 2$.
$hamming(123, 321) = 2$.

Problema: distancia Hamming

problema $hamming(a[char], b[char], n : Z, m : Z) = res : Z\{$
requiere $|a| == n$;
requiere $|b| == m$;
asegura $res == \sum[\beta(a[i]! = b[i]) \mid i \in [0..\min(n, m))] + abs(n - m)$

Algoritmo de distancia Hamming

Inicializar contador en 0

Utilizar un único indice i para recorrer a y b linealmente
hasta alcanzar la última posición de la más corta

Comparar $a[i]$ y $b[i]$
Si difieren incrementar el contador en 1
Incrementar el índice en 1

Sumar al contador la diferencia entre las longitudes de a y b
Retornar el valor del contador.

Ese algoritmo no tiene precondiciones.

Realiza una cantidad lineal de iteraciones.

Especificación del ciclo de distancia Hamming

Pc: $i == 0 \wedge c == 0$

invariante:$0 \leq i \leq \text{mín}(n, m) \wedge c == \sum([\beta(a[j]! = b[j])| j \in [0..i)]$

variante: $\text{mín}(n, m) - i$;

Qc: $i == \text{mín}(m, n) \wedge c == \sum([\beta(a[j]! = b[j])| \; j \in [0.. \text{mín}(n, m))]$

Programa hamming

```
int hamming(char a[], char b[], int n, int m) {
    int i = 0;
    int c=0;
    // Pc: i == 0 ∧ c == 0
    while (i < n &&  i<m) {
    invariante: 0 ≤ i ≤ mín(n, m)∧
                c == ∑([β(a[j]! = b[j])| j ∈ [0..i)]
    variante: mín(n, m) − i ;
        if ( a[i]!=b[i] ) c++;
        i++;
    }
```

Qc:$i == \text{mín}(m, n) \wedge c == \sum([\beta(a[j]! = b[j])| j \in [0.. \text{mín}(n, m))]$

```
    return c + abs(n-m);
```

$res == \sum[\beta(a[i]! = b[i]) \;|\; i \in [0.. \text{mín}(n, m))] + abs(n - m)$

}