Algoritmos y Estructuras de Datos I

Primer cuatrimestre de 2013

Departamento de Computación - FCEyN - UBA

Programación imperativa - clase 5

Otros algoritmos sobre secuencias

Otros algoritmos sobre secuencias

1. Problema **distancia de Hamming**: Medir la diferencia entre dos secuencias del mismo largo.
2. Problema **cantidad de ocurrencias**: Contar la cantidad de veces que aparece cada entero entre 0 y n dentro de una secuencia.
3. Problema **merge**: Dadas dos secuencias ordenadas, obtener una tercera secuencia intercalando ordenadamente sus elementos.

Problema: Distancia Hamming

- **Definición.** (Richard Hamming, 1950) La distancia de Hamming entre dos palabras es el número de caracteres que tienen que cambiarse para transformar una en la otra.
- Es una métrica de la diferencia entre las dos palabras, que sólo tiene en cuenta la cantidad de diferencias entre ellas.
- Ejemplos:
  1. $hamming(1011101, 1001001) = 2$.
  2. $hamming(123, 321) = 3$.
  3. $hamming(123, 133) = 1$.

Problema: distancia Hamming

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

Algoritmo de distancia Hamming

1. Inicializar contador en 0.
2. 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.
3. Sumar al contador la diferencia entre las longitudes de a y b.
4. Retornar el valor del contador.

- Este algoritmo no tiene precondiciones, y realiza una cantidad lineal de operaciones.

Especificación del ciclo de distancia Hamming

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

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

variante: $\text{mín}(n, m) - i$;

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

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 == mín(m,n) ∧ c == ∑([β(a[j]! = b[j]) | j ∈ [0..mín(n,m))]
    return c + abs(n-m);
    //
res == ∑[β(a[i]! = b[i]) | i ∈ [0..mín(n,m))] + abs(n − m)
}
```


Problema: cuenta cantidad de ocurrencias

Problema. 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 *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 de fuerza bruta para contar cantidad de ocurrencias

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

- Observar que este algoritmo no tiene precondiciones.
- ¿Qué complejidad tiene este algoritmo? Realiza una cantidad cuadrática de operaciones!

Algoritmo lineal que cuenta ocurrencias

1. Inicializar el arreglo b de salida con 0 en todas las posiciones.
2. Para $i = 0, \ldots, |a| - 1$, incrementar $b[a[i]]$ en uno.

- Requiere todos($[0 \leq a[i] < n \mid i \in [0..n)]$;
- Este algoritmo realiza una cantidad lineal de operaciones!

Especificación del ciclo de 'contar ocurrencias'

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; i++;
    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 merge: intercalar dos secuencias ordenadas

Problema merge. 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.

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

1. Mantener un indice para recorrer a y otro para b.
2. 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.
3. Incrementar en 1 el índice del arreglo del que provino el elemento.
4. 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: $i == 0 \land j == 0$

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

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

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

Qc: $i == n \land j == m \land ordenado(c[0..n+m)) \land$
$mismos(c[0..n+m), a[0..n) + +b[0..m))$

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))
}
```

Bibliografía sobre corrección en programación imperativa

1. D. Gries, *The science of programming*, Springer-Verlag, 1987.
2. E. Dijkstra y W. Feijen, *A method of programming*, Addison-Wesley, 1988.
3. J. Balcázar, *Programación metódica*, McGraw-Hill, 1993.
4. E. Dijkstra, *The manuscripts of Edsger W.Dijkstra*, `www.cs.utexas.edu/~EWD`.