Prácticas de Programación 2

Ingeniería Técnica de Informática de Sistemas

UNED

Curso 2000−2001

Clase presencial el 28 de Abril en Las Rozas, Madrid

- **Enunciado del problema**

Hay elecciones en la República de Tranfuyastán y se trata de diseñar una función recursiva, completamente verificada, que permita comprobar si hay predominio de algún partido político en una circunscripción determinada, compuesta por un número variable de distritos electorales. Así, esa función deberá poder ser utilizada para poder comprobar los datos de cada curcunscripción que en la noche electoral llegarán a la sede central del servicio informático del PINTA. En los casos en que no haya predominancia, será necesaria una segunda vuelta.

- **Especificación del algoritmo**

{Q " n 0 }

fun *kpred* (a: vector [1..n] de etiquetas, p: etiqueta, u: natural dev b: booleano)

{R " b = (N {1..n}.p = a[ ])>u}

Dado un vector a de etiquetas de partidos políticos (vencedores de los distritos de una circunscripción), *kpred* devolverá 1 o verdadero si la etiqueta p (el partido p) está presente en ese vector en número superior al umbral u. Si es inferior, *kpred* devolverá 0. Veamos qué ocurre con 3 valores para el valor umbral u:

u=0 *kpred* devolverá verdadero siempre que el partido p haya sido votado

u=n−1 *kpred* devuelve verdadero sólo si p ha ganado por unanimidad

nMOD2 *kpred* devolverá verdadero si p tiene mayoría absoluta

- **Algoritmo recursivo no final**
- **Función auxiliar**.

Nuestro problema no permite un diseño recursivo de forma directa. Para eso dividiremos el problema en dos partes: conteo y decisión sobre la predominancia. Para la primera parte del problema definimos una función más sencilla que *kpred*, que llamaremos *kcont*, que solamente cuenta las veces que una etiqueta p aparece en un vector a. Es nuestra función auxiliar. Dejamos para después la decisión sobre u.

- **Especificación de *kcont***.

{Q " n 0 }

fun *kcont* (a: vector [1..n] de etiquetas, p: etiqueta dev k: natural)

{R " k = (N {1..n}.p = a[ ])}

Como decía en 3.1, *kcont* devuelve el número k de veces que p aparece en el vector de etiquetas a.

- **Expresión de *kpred* respecto a *kcont*.**

kpred(a, p, u) = (*kcont*(a, p) > u)

- **Especificación de la función inmersora y llamada inicial**

A partir de *kcont* construiremos una función inmersora *ikcont* que permitirá una solución recursiva del problema. Para ello debilitaremos la poscondición de *kcont*, sustituyendo la constante n por una variable i:

{R " k = (N {1..i}.p = a[ ]) i = n}

pasa a ser

{R " k = (N {1..i}.p = a[ ])}

y por lo tanto diseñaremos *ikcont* para que se cumpla *kcont*(a, p) = *ikcont*(a, p, i) i =n, que sería la llamada inicial.

La especificación de la función inmersora queda entonces:

{Q " n 0 0 " i " n }

fun i*kcont* (a: vector [1..n] de etiquetas, p: etiqueta, i: natural dev k: natural)

{R " k = (N {1..i}.p = a[ ])}

- **Análisis por casos de *ikcont*.**

Si n 0, como dice la precondición, entonces se pueden dar tres casos:

i = 0 {1..0} = conjunto vacío, luego *ikcont* devolverá k = 0

Este es el caso trivial: cuando i=0 el vector está vacío. Este será el caso donde deberá terminar la descomposición recursiva.

i > 0 si p = a[i] k = *ikcont*(a, p, i−1) + 1

si p " a[i] k = *ikcont*(a, p, i−1) + 0

Este sería el caso no trivial, que tiene además un predicado lógico que será necesario evaluar para enlazar con la llamada recursiva posterior. En concreto, se comprueba si el elemento a[i] es igual o no a la etiqueta de referencia p. Cada vez que el algoritmo pase por este punto, la recursión avanzará y la fracción del vector original que queda por analizar será 1 posición menor: i−1.

- **Composición algorítmica**

{Q " n 0 0 " i " n }

fun i*kcont* (a: vector [1..n] de etiquetas, p: etiqueta, i: natural dev k: natural)

caso i = 0 k = 0

i > 0 p = a[i] k = *ikcont*(a, p, i−1) + 1

p " a[i] k = *ikcont*(a, p, i−1) + 0

fcaso

ffun

{R " k = (N {1..i}.p = a[ ])}

Como vemos, es una función recursiva no final, puesto que hay que combinar (sumar 1 ó 0 en este caso) cada llamada recursiva con un valor acumulado anterior.

- **Verificación formal de la corrección**
- **Completitud de la alternativa**

Q(a, p, i) ! Bt(a, p, i) " Bnt(a, p, i)

n 0 0 " i " n ! i = 0 " i > 0

n 0 0 " i i " n ! 0" i

- **Satisfacción de la precondición para la llamada interna**

Q(a, p, i) Bnt(a, p, i) ! Q(s (a, p, i))

n 0 0 " i " n i > 0 ! n 0 0 " i−1 " n

- **Base de la inducción noetheriana**

Q(a, p, i) Bt(a, p, i) ! R((a, p, i) , Bnt(a, p, i))

n 0 0 " i " n i = 0 ! 0 = (N {1..0}.p = a[ ])

n 0 i = 0 ! 0 = 0

- **Paso de inducción**

Hipótesis de inducción: R(s ((a, p, i) , y'))

Q(a, p, i) Bnt(a, p, i) R(s ((a, p, i) , y')) ! R((a, p, i) , c(y',(a, p, i)))

En este caso particular, la demostración del paso de inducción la haré para los dos posibles casos no triviales:

a) n 0 0 " i " n i >0 (N {1..i−1}.p = a[ ]) !

(N {1..i−1}.p = a[ ]) + 1 = (N {1..i}.p = a[ ]) *si a[i] es igual a p*

b) n 0 0 " i " n i >0 (N {1..i−1}.p = a[ ]) !

(N {1..i−1}.p = a[ ]) + 0 = (N {1..i}.p = a[ ]) *si a[i] no es igual a p*

Hasta quedaría demostrado que ikcont es una función parcialmente correcta.Ahora terminaremos de comprobar su corrección estudiando si termina.

- **Elección de un preorden bien fundado**

Observando el dominio de los datos de entrada (a, p, i) en la precondición de la función, elegimos una función *t* que sobre ellos nos devuelva un natural, puesto que los números naturales forman un preorden bien formado, cuyo minimal es el 0. En tal caso, *t* (vector, etiqueta, natural) natural, o *t* (a, p, i) i .

- **Decrecimiento de los datos con las llamadas recursivas**

Q(a, p, i) Bnt(a, p, i) ! *t* (s(a, p, i)) < *t* (a, p, i)

n 0 0 " i " n i >0 ! i−1 < i

- **Estudio del coste**

Nuestra función *ikcont* es recursiva y reduce el problema por sustracción.

Para el caso trivial, el coste es el coste de una asignación, k=0, por lo tanto es constante, c0. Para el caso no trivial, la función de coste es recursiva: coste(i) = c1 + coste(i−1) . En general, el coste del algoritmo es coste(i) = c0 + i c1 , lineal en i.

- **Transformación de *ikcont* a recursiva final: desplegado−plegado**
- **Árbol sintáctico de la función recursiva**

Basándonos en la rama no trivial de *ikcont*, con la función de combinación = suma, dibujamos el árbol sintáctico de la función:

Suma +

Resultado de la comparación *ikcont*(a, p, i−1)

con p: 1 ó 0

- !

introduzco un nuevo parámetro pongo los parámetros originales

u *ikcont*(a, p, i)

Y la función generalizadora queda como: *iikcont*(a, p, i, u) = u + *ikcont*(a, p, i)

La llamada inicial sería entonces *iikcont*(a, p, i, 0) = *ikcont*(a, p, i). El desplegado de la función sería:

iikcont(a, p, i, u) = u + 0 si i = 0

u + *ikcont*(a, p, i−1) + 1 si i > 0 y además p = a[i]

u + *ikcont*(a, p, i−1) + 0 si i > 0 y además p " a[i]

Por lo tanto,

iikcont(a, p, i, u) = *iikcont*(a, p, i−1, u + 1) si a[i] = p y

iikcont(a, p, i, u) = *iikcont*(a, p, i−1, u + 0) si a[i] " p

- **Código de la función *iikcont,* recursiva final**

{Q " n 0 0 " i " n u = (N {i+1..n}.p = a[ ])}

fun ii*kcont* (a: vector [1..n] de etiquetas, p: etiqueta, i: natural, u: natural dev k: natural)

caso i = 0 k = u

i > 0 p = a[i] k = *iikcont*(a, p, i−1, u + 1)

p " a[i] k = *iikcont*(a, p, i−1, u + 0)

fcaso

ffun

{R " k = (N {1..i}.p = a[ ])}

El código de la función recursiva final es muy similar al de la no final, salvo que aquí reforzamos la precondición con el valor previo del parámetro u, el contenido del vector desde i+1 hasta n.Podríamos haber modificado la poscondición, pero al hacerlo de esta manera obtenemos el invariante del bucle iterativo equivalente a esta recursión, que es lo que haremos ahora.

- **Diseño iterativo de un algoritmo que resuelva el mismo problema**

Una función recursiva para este problema tendrá un aspecto similar a:

{Q }

<condiciones iniciales>

mientras B hacer {P}

{P B}

reestablecer

avanzar

{P}

fmientras

{R " k = (N {1..n}.p = a[ ])}

Sabemos que el invariante P de un bucle se deberá conservar siempre antes de la primera iteración y después de todas las demás, en particular al terminar el bucle. Derivemos ese invariante.

- **Derivación del invariante partiendo de la poscondición de *kcont***

{Q " n 0 }

fun *kcont* (a: vector [1..n] de etiquetas, p: etiqueta dev k: natural)

{R " k = (N {1..n}.p = a[ ])}

Esta poscondición podría reescribirse como: (N {1..n}.p = a[ ]) i=n . Si eliminamos el segundo término de la conjunción, la debilitamos, quedando entonces

N {1..i}.p= a[ ] . Como el invariante es un predicado más débil que la poscondición, podemos asumir que éste es el invariante P. Y la expresión i=n es la condición de terminación del bucle, ¬B.

- **Instrucción avanzar**

Sabiendo que ¬B es i = n, y que por tanto B es i " n, avanzaremos en el bucle incrementando la variable i, con un contador como i := i+1 .

- **Reestablecimiento del invariante**

Al avanzar el bucle generalmente es necesario incorporar una nueva instrucción −reestablecer− que garantice la conservación del invariante al avanzar el bucle. Nuestro predicado de reestablecer será:

k := k + 1 si a[i+1] = p y

k := k + 0 si a[I+1] " p

- **Código completo del algoritmo iterativo**

{ Q " n i 0}

<i,k> := <0,0>

mientras i " n hacer /* { P " N {1..i}.p= a[ ] } */

caso a[i+1] = p k := k + 1

a[i+1] " p k := k + 0

fcaso

i := i+1

fmientras

{R " k = (N {1..n}.p = a[ ])}

- **Estudio del coste**

Primero proponemos una función de cota para este algoritmo. Debe ser una función descendente, cuyos valores sean menores según avance el bucle. El valor de i aumenta, por lo que no nos vale. Si pensamos en −i, entonces sí es descendente, pero para asegurarnos mayores o iguales a 0, proponemos cota(i): n−i .

En cuanto al coste, este algoritmo tiene un coste citer por iteración constante, puesto que siempre se ejecutan una comparación y dos sumas. En tal caso, el coste total será lineal respecto al número de iteraciones, n. Por lo tanto, en total queda una expresión como:

Coste = n citer .

Podríamos haber puesto el cuerpo del bucle así:

mientras i " n hacer /* { P " N {1..i}.p= a[ ] } */

caso a[i+1] = p k := k + 1

fcaso

i := i+1

fmientras

En tal caso, el coste por iteración sería dependiente de la operación de comparación de p. En caso de desigualdad sólo se haría una suma, ganándose en eficiencia. Estas iteraciones tendrían un coste cdesig < citer . El coste total estaría entonces comprendido entre n cdesig y n citer , dependiendo del contenido del vector a.