8. Soluciones recursiva de problemas

8.1. Planteamiento de soluciones recursivas de problemas

En este capítulo hablaremos de soluciones recursivas de problemas, es decir, soluciones que vienen expresadas en función de soluciones a problemas mas sencillos del mismo tipo que el problema original. Esta estrategia de solución de problemas se conoce como *divide y conquistarás*.

Se trata de resolver un problema mediante la solución de varios problemas similares (o del mismo tipo) mas pequeños. Por pequeño entendemos que el problema posee menor cantidad datos. Si la división puede ser llevada a cabo, entonces el mismo proceso puede ser aplicado a los problemas más pequeños.

Frecuentemente la estrategia "divide y conquistarás" consiste en dividir el problema original en dos o más problemas similares. Sin embargo, el mismo principio aplica cuando reducimos el problema original a sólo un problema similar más pequeño. Por ejemplo, para calcular el máximo elemento de un arreglo b de enteros de largo N, basta con calcular el máximo m_1 del segmento b[0..N-1) y y tomar como máximo m de b[0..N) al máximo entre m_1 y b[N-1]. Cuando b posee un solo elemento no es necesario, pues el máximo de b[0..1) es b[0]. La solución del problema queda de la forma:

"máximo de b[0..N) = max (máximo de b[0..N-1), b[N-1]) $\wedge$ máximo de b[0..1) = b[0]"

Este tipo de solución, en donde la solución del problema queda expresada en términos de la solución de un problema más pequeño del mismo tipo, la llamamos "solución recursiva" del problema.

Note que si el lenguaje de programación permitiese que un procedimiento o una función se llame a si mismo, entonces podríamos obtener un programa recursivo que resuelve nuestro problema. Los programas que haremos en este capítulo serán iterativos, pero partiendo de una primera versión recursiva de la solución del problema. Por ejemplo, ya hemos hecho un programa iterativo para el cálculo del n-ésimo número de Fibonacci, aún cuando la definición de éstos es recursiva: Fib(n)= Fib(n-1) + Fib(n-2), n ≥ 2, Fib(0)=0, Fib(1)=1.

La diferencia entre la estrategia discutida en la sección 7.2.1 y la estrategia "divide y conquistarás" es muy sutil. En la primera estrategia, primero reconocemos un problema más simple y luego nos preguntamos cómo éste puede ser utilizado de manera efectiva. Este fue el caso del ejemplo que intercambiaba dos segmentos de un arreglo. En la estrategia "divide y conquistarás" primero nos preguntamos lo que significa dividir el problema en piezas más pequeñas y luego buscar la manera de resolver el problema original en términos de las piezas, y esto puede conllevar la solución de problemas más simples como la estrategia de la sección 7.2.1.

Veamos otros ejemplos donde se aplica la estrategia "divide y conquistarás".

Problema 1: Calculo del máximo de un arreglo de enteros.

Otra forma de formular la solución de este problema en términos de problemas mas pequeños del mismo tipo es:

Suponiendo que m-n > 1, el máximo de b[n..m) es el máximo de b[n..m-1) si b[n] ≥ b[m-1], si no es el máximo de b[n+1..m). Cuando m=n+1 el máximo es b[n].

Problema 2: Determinar la suma de los dígitos de un número natural n.

La solución se puede plantear como:

Suma de los dígitos de n = n mod 10 + (suma de los dígitos de (n div 10))

Problema 3: Cálculo de A^B, para A, B enteros y B entero no negativo.

$$A^B = \begin{cases} 1 & si \quad B = 0 \\ (A * A)^{B \text{ div } 2} & si \quad B \bmod 2 = 0 \\ A * A^{B-1} & si \quad B \bmod 2 = 1 \end{cases}$$

De esta solución recursiva podemos deducir un programa iterativo eficiente para el cálculo de $r = A^B$ utilizando el invariante "$r*x^y = A^B \wedge 0 \leq y$". Más adelante explicaremos cómo obtener a partir de un esquema más general, el programa que a continuación se presenta:

```
[  r, x, y := 1, A, B;
   { Invariante: r*x^y = A^B ∧ 0≤y; cota: y }
   do  y ≠ 0
         →  if     y mod 2 = 0  →  x, y := x*x, y div 2
            []     y mod 2 = 1  →  r,y := r*x, y-1
            fi
   od
]
```

Problema 4: (búsqueda binaria)

Dado un arreglo de enteros b[0..N) ordenado en forma creciente, determinar si un número x dado se encuentra en b.

Podemos utilizar el esquema de búsqueda secuencial visto en la sección 7.1. Sin embargo podemos aprovechar el acceso directo a los elementos de un arreglo y al hecho que éste está ordenado con el fin de conseguir un programa más eficiente.

La solución a este problema es similar a la que usamos cuando deseamos buscar un número de teléfono en una guía telefónica: no hacemos una lectura secuencial de la guía partiendo de la primera página hasta encontrar el número buscado (pasaríamos todo un día hasta

llegar al apellido Rodríquez). Lo que hacemos es abrir la guía en un lugar que nos parezca conveniente, y si el apellido de la persona está en las dos páginas donde abrimos, entonces paramos, si no determinamos si está mas allá de la página de la derecha o antes de la página de la izquierda dependiendo de si el apellido está antes o después.

La solución entonces es: comparar el elemento del medio b[N div 2] de b[0..N) con x, si son iguales entonces hemos encontrado el elemento en b, si no son iguales entonces se busca x en el segmento b[0..N div 2) o en el segmento b[N div 2 + 1 .. N) dependiendo respectivamente de si x es menor o mayor que b[N div 2] . Si el segmento es vacío, x no está en el segmento. La solución como vemos depende del largo del segmento, por lo que la podemos generalizar como sigue:

x está en b[n..m) si y sólo si ((m-n > 0) $\wedge$ (x = b[(m+n) div 2]
$\vee$ ((x < b [(m+n) div 2]) $\wedge$ (x está en b[n..(m+n) div 2)))
$\vee$ ((x > b [(m+n) div 2]) $\wedge$ (x está en b[(m+n) div 2 +1 .. m))))

Podemos entonces deducir el invariante siguiente (en forma parecida al ejemplo de intercambio de segmentos de las sección 7.1), reemplazando las constantes n y m por dos variables i y j:

P: x está en b[n..m) $\equiv$ x está en b[i,j) $\wedge$ $n \leq i \leq j \leq m$

inicialmente el invariante se establece con i, j := n, m y la guardia será i-j>1 pues cuando el arreglo b tiene al lo sumo un elemento es sencillo comprobar si el elemento x está en b. La función de cota será j-i y el programa sería:

```
i,j := n,m;
do j-i>1
      → k := (i+j) div 2;
         if   b[k] = x →  i :=k; j := k+1
         []   b[k] > x →  j := k
         []   b[k] < x →  i := k + 1
         fi
od;
if  i=j →  esta=falso
[]  i≠j →  esta= (x = b[i])
fi
{ esta ≡ x está en b[0..N)  ∧ (esta ⇒ x = b[i]) }
```

Problema 5: Torres de Hanoi.

8.2. Diseño iterativo de soluciones recursivas de problemas: Invariante de Cola.

En esta sección analizamos en un contexto general las soluciones iterativas de problemas que admiten una **solución recursiva de cola**. Por solución recursiva de cola entendemos una solución "divide y conquistarás" donde la solución del problema original depende de un solo problema más pequeño del mismo tipo, como los problemas 1 a 4 de la sección 8.1.

Supongamos que tenemos una función F que cumple:

$$F(x) = \begin{cases} h(x) & si\ b(x)\ es\ verdadero \\ F(g(x)) & si\ b(x)\ es\ falso \end{cases}$$

Vemos que el valor de la función en un punto x viene expresado en términos de la misma función para un punto g(x), por lo que este es un caso particular de una definición recursiva de cola de la función F. Queremos desarrollar un programa que calcule F(X) para X dado, es decir, que satisfaga la especificación siguiente:

```
[   const X: ...;
    var r: ...;
   { verdad }
    cálculo de F(X)
    {  r = F(X)  }
]
```

Si tomamos como invariante al predicado siguiente (llamado **invariante de cola**):

$$F(x) = F(X)$$

Obtenemos el siguiente programa iterativo:

```
[  var x;
   x := X;
   { Invariante:  F(x) = F(X)  }
   do  ¬ b(x)  →   x  :=  g(x)  od;
   r  :=  h(x)
]
```

Por lo tanto, resolver un problema por recursión de cola significa encontrar una función F con las características anteriores.

Ejemplos:

Problema 1: Cálculo del máximo de un arreglo de enteros con dominio [m.n) y n>m.

La especificación formal sería:

```
[  const N: entero;
   const b: arreglo [0..N) de entero;
   var r: entero;
   { N > 0 }
   máximo
   {  r = (max i : 0 ≤ i < N : b[i])  }
]
```

Una forma de formular la solución de este problema en términos de la solución de problemas mas pequeños del mismo tipo es:

Suponiendo que m-n > 1, el máximo de b[n..m) es el máximo de b[n..m-1) si b[n] ≥ b[m-1], si no es el máximo de b[n+1..m). Cuando m=n+1 el máximo es b[n]. En general, si definimos F(x,y) = (max i : x ≤ i < y : b[i]), con x < y, entonces tenemos que F tiene la siguiente definición recursiva de cola:

$$F(x,y) = \begin{cases} b[x] & si \quad y - x = 1 \\ F(x+1, y) & si \quad b[x] \le b[y-1] \wedge (y - x \ne 1) \\ F(x, y-1) & si \quad b[y-1] \le b[x] \wedge (y - x \ne 1) \end{cases}$$

Por lo tanto la especificación de la postcondición puede ser escrita como:

$$r = F(0,N)$$

Por lo tanto podemos proponer como invariante:

Inv: $F(x,y) = F(0,N) \wedge 0 \le x \le y \le N$

Y el programa *máximo* sería:

```
[ var x, y : entero
   x, y := 0, N;   {  N > 0  }
   { Inv: F(x,y) = F(0,N) ∧  0 ≤ x ≤ y ≤ N; cota: y-x }
   do y-x ≠1 →
         if b[x] ≤ b[y-1] → x := x+1
         [] b[y-1] ≤ b[x] → y := y-1
         fi
   od;
   { Inv ∧ y-x=1, por lo que se tiene b[x] = F(0,N)  }
   r := b[x]
]
```

Note que el programa anterior es un bloque interno del programa donde se definen las variables N y b; es decir, el programa completo sería:

```
[  const N: entero;
   const b: arreglo [0..N) de entero;
   var r: entero;
   { N > 0 }
    [ var x, y : entero
      x, y := 0, N;   {  N > 0  }
      { Inv: F(x,y) = F(0,N) ∧  0 ≤ x ≤ y ≤ N; cota: y-x }
      do y-x ≠1 →
         if b[x] ≤ b[y-1] → x := x+1
         [] b[y-1] ≤ b[x] → y := y-1
         fi
      od;
      { Inv ∧ y-x=1, por lo que se tiene b[x] = F(0,N)  }
      r := b[x]
    ]
   {  r = (max i : 0 ≤ i < N : b[i])  }
]
```

Problema 2: Hacer un programa que determine el máximo común divisor (MCD) de dos números enteros positivos A y B .

La especificación formal sería:

```
[  const A, B: entero;
   var r : entero;
   { A > 0 ∧  B > 0  }
   máximo común divisor
   {  r  =  MCD(A,B)  }
]
```

Por propiedades matemáticas se sabe que si x e y son enteros positivos:

$$MCD(x,y) = \begin{cases} x \quad si \quad x = y \\ MCD(x-y,y) \quad si \quad x > y \\ MCD(x,y-x) \quad si \quad y > x \end{cases}$$

Por lo tanto tenemos una definición recursiva de cola del Máximo Común Divisor, lo cual nos dice que podemos proponer como invariante a:

$$x > 0 \wedge \ y > 0 \ \wedge \ MCD(x,y) = MCD(A,B)$$

Y el programa sería:

```
[  var x,y: entero;
```

```
  x, y := A, B;
  do x > y →  x :=  x-y
  []  y > x →  y :=  y-x
  od;
  r := x
]
```

Un caso más general de recursión de cola viene dado cuando queremos calcular r = G(X), para X dado, donde la función G viene definida como sigue:

$$G(x) = \begin{cases} a \quad si \quad b(x) \\ h(x) \oplus G(g(x)) \quad si \quad \neg b(x) \end{cases}$$

donde ⊕ es una operación asociativa con elemento neutro e.

Este problema puede ser resuelto con un invariante de cola de la forma:

P: G(X) = r ⊕ G(x)

El cual puede ser interpretado como:

"el resultado" = "lo que ha sido calculado hasta el momento" ⊕ "lo que resta por calcular"

P puede ser establecido inicialmente con r, x := e, X. Además , si b(x) es verdad, entonces:

```
        G(X) = r ⊕ G(x)
≡       definición de G y B(x) verdadero
        G(X) = r ⊕ a
```

Y cuando ¬b(x) es verdadero:

```
        G(X) = r ⊕ G(x)
≡  definición de G y que se cumple ¬b(x)
        G(X) = r ⊕ (h(x) ⊕ G(g(x)))
≡  ⊕  es asociativa
        G(X) = (r ⊕ h(x) ) ⊕ G(g(x))
≡  definición de P
        P(  r, x := r ⊕  h(x), g(x)  )
```

Esto lleva al siguiente esquema de programa:

```
{  verdad  }
[  var x;
   x, r := X, e;
   { Invariante: G(X) = r ⊕ G(x)  }
```

  do $\neg b(x) \rightarrow x, r := g(x) , r \oplus h(x)$ od
  { $G(X) = r \oplus a$ }
]
{ $r = G(X)$ }

Ejemplo:

Problema 3: Dado un número natural X, hacer un programa que calcule la suma de los dígitos decimales de X. Por ejemplo, la suma de los dígitos de 4329 es 4+3+2+9=18.

Si G(X) representa la suma de los dígitos de X, se tiene que G puede ser definida de la siguiente forma:

$$G(x) = \begin{cases} 0 & si \quad x = 0 \\ x \bmod 10 + G(x div 10) & si \quad x > 0 \end{cases}$$

Y la especificación formal del programa sería:

[const X: entero;
  var r: entero;
  { $X \geq 0$ }
  suma dígitos
  { $r = G(X)$ }
]
Aplicando los resultados anteriores, obtenemos como invariante:

$$G(X) = r + G(x) \ \wedge \ x \geq 0$$

Y obtenemos el programa *suma dígitos* es:

{ $X \geq 0$ }
[var x: entero;
  x, r := X, 0;
  { Invariante: $G(X) = r + G(x) \ \wedge \ x \geq 0$; cota: x }
  do $x \neq 0 \rightarrow$ x, r := x div 10, r + x mod 10 od
]
{ $r = G(X)$ }

Problema 4: Calcular A^B con A, B enteros y $B \geq 0$.

Si G(x,y) es la función x^y, para $y \geq 0$, tenemos:

$$G(x,y)=\begin{cases}1 & si \quad y=0\\ 1*G((x*x),y \text{ div } 2) & si \quad y \bmod 2=0\\ x*G(x,y-1) & si \quad y \bmod 2=1\end{cases}$$

El invariante de cola correspondiente a G es:

$$r*x^y = A^B \ \wedge \ y \geq 0$$

Y el programa es el que ya vimos en la sección 7.2.2. problema 3:

```
{  B ≥ 0 }
[  var x, y: entero;
   r, x, y := 1, A, B;
   { Invariante: r*x^y = A^B ∧ 0≤y; cota: y }
   do  y ≠ 0
         →  if     y mod 2 = 0  →  x, y := x*x, y div 2
            []     y mod 2 = 1  →  r,y := r*x, y-1
            fi
   od
]
{  r = A^B  }
```

Otros ejemplo: Quicksort pag. 226 de Gries.

Ejercicios:

1) Defina una función recursiva de cola para el problema de búsqueda binaria.
2) Diseñe una solución recursiva para ordenar en forma creciente los elementos de un arreglo de enteros.
3) Página 78 Kaldewaij