Diseño de algoritmos recursivos

Isabel Pita (2017/18)

Facultad de Informática - UCM

15 de noviembre de 2017

Objetivos

1. Tipos de recursión.
2. Análisis de la complejidad de algoritmos recursivos. Despliegue de recurrencias.
3. Reglas de verificación de programas recursivos
4. Derivación de programas recursivos
5. Uso de parámetros para mejorar el coste de las soluciones recursivas

Problemas propuestos de acepta el reto

Programas recursivos:

- 140. Suma de dígitos
- 183. Anélidos.
- 252. ¿Acaso hubo búhos acá?
- 272. Tres dedos en cada mano.
- 315. Jugando al buscaminas.

Solución recursiva de un problema:

- Se conoce la solución del problema para un conjunto de datos *simple*. (Caso base o directo)
- Se conoce como resolver el problema para el resto de los datos a partir de la solución del problema para casos más sencillos.(Caso recursivo)

Ejemplos:

- Cálculo de la potencia: $x^0 = 1; x^n = x^{n-1} * x;$
- Cálculo del factorial: $0! = 1; n! = (n-1)! * n;$
- Suma de las componentes de un vector.
- Problema de las torres de Hanoi

Tipos de recursión

Recursión simple (o lineal): cada caso recursivo realiza exactamente una llamada recursiva.

```
int factorial ( int n ){
  if ( n == 0 ) return 1;
  else return = n * factorial(n-1);   // (n > 0)
}
```

1. Condición del caso directo: $n == 0$
2. Cálculo del caso directo: `return` 1
3. Función sucesor: $s(n) = n - 1$
4. Función de combinación: $n * fact(s(n))$

- Se pueden tener varios casos directos o varias descomposiciones para el caso recursivo. Las alternativas deben ser **exhaustivas** y **excluyentes**.
- Ejemplo: multiplicación de dos naturales por el método del *campesino egipcio*.

```
method prod (a : int, b : int) returns (p : int)
requires a >= 0 && b >= 0
ensures p == a * b
decreases b
{
  if (b == 0) {p := 0;}
  else if (b == 1) {p := a;}
  else if (b % 2 == 0) {p := prod(2*a, b/2);}
  else {
    assume b%2!=0 ==> (a*b) == ((2*a)*(b/2)+a);
    p := prod(2*a, b/2);
    p := p + a;
  }
}
```

Tipos de recursión simple o lineal.

La recursión simple o lineal puede ser:

- Recursión final o de cola (*tail recursion*). La función de combinación se limita a transmitir el resultado de la llamada recursiva. El resultado será siempre el obtenido en uno de los casos base.
  Ejemplo: Cálculo del máximo común divisor por el método de Euclides.
- Recursión no final. La función de combinación modifica el resultado de la llamada.
  Ejemplo: función factorial.

Función recursiva final:

```
void nomPro(τ1 x1 , ... , δ1 & y1 , ... ) {
  τ1 x11 ; ... ; δ1 y11 ; ... ;
  if ( d(x⃗) ) y⃗ = g(x⃗);
  else if ( ¬ d(x⃗) ) {
    x⃗1 = s1( x⃗ );
    nombreProc(x⃗1 , y⃗1 );
  }
}
```

```
void nomPro(τ1 x1 , ... , δ1 & y1 , ... ) {
  τ1 x11 ; ... ; δ1 y11 ; ... ;
  while( !d(x⃗) ) x⃗1 = s1( x⃗ );
  y⃗ = g(x⃗);
}
```

Tipos de recursión. Transformación recursivo-iterativo

Ejemplo: cálculo del mcd por el algoritmo de Euclides.

```
method Calculomcd(a : int, b : int) returns (n : int)
requires a >= 0 && b >= 0
ensures n == mcd(a,b)
```

Implementación recursiva:

```
if (b == 0) {n := a; return;}
else {n := Calculomcd(b,a%b); return;}
```

Implementación iterativa:

```
var a1 := a; var b1 := b;
while (b1 != 0)
invariant mcd(a,b) == mcd(a1,b1)
{   var aux := a1; a1 := b1;  b1 := aux%b1; }
n:= a1;
```

Tipos de recursión. Transformación recursivo-iterativo

Función recursiva no final (cuando existe inversa de la función sucesor):

```
void nomPro(τ1 x1 , ... , δ1 & y1 , ... ) {
  τ1 x11 ; ... ; δ1 y11 ; ... ;
  if ( d(x⃗) ) y⃗ = g(x⃗);
  else if ( ¬ d(x⃗) ) {
    x⃗1 = s1( x⃗ );
    nombreProc(x⃗1 , y⃗1 );
    y⃗ = c(x⃗ , y⃗1 , ... , y⃗k ); }
}
```

```
void nomPro(τ1 x1 , ... , δ1 & y1 , ... ) {
  τ1 x11 ; ... ; δ1 y11 ; ... ; τ1 x'1 = τ1 x1 ...
  while( !d(x⃗') ) x⃗' = s(x⃗') ;
  y⃗ = g(x⃗');
  while( x⃗'!= x⃗ ) {
    x⃗' = s^-1( x⃗' ); y⃗ = c(x⃗' , y⃗1 , ... , y⃗k );
} }
```

Tipos de recursión. Transformación recursivo-iterativo

Ejemplo: cálculo del factorial.

```
int factorialRec(int n) {
  if  (n == 0 ) return 1;
  else return  n * factorial(n-1);
}
int factorialIt (int n) {
  int n1 = n;
  while (n1 != 0) n1 = n1-1;
  r = 1;
  while (n1 != n) {n1 = n1+1; r = r*n1;}
  return r;
}
int factorialItOpt (int n) {
  int n1 = 0; r = 1;
  while (n1 != n) {n1 = n1+1; r = r*n1;}
  return r;
}
```

Tipos de recursión

Recursión múltiple: al menos en un caso recursivo, se realizan varias llamadas recursivas.

```
int fib( int n )
{
  if ( n == 0 ) return 0;
  else if ( n == 1 ) return 1;
  else  return fib(n-1) + fib(n-2);
}
```

Análisis de la complejidad de algoritmos recursivos

- Para analizar la complejidad de un algoritmo recursivo debemos analizar la complejidad de lo que se hace en una llamada recursiva, y estimar el número de llamadas recursivas que se realizarán.
- Introducimos el concepto de *ecuaciones de recurrencia*
- Cálculo del factorial.

```
int factorial ( int n ){
  if (n == 0) return 1;
  else return n * factorial(n-1);   // (n > 0)
}
```

- Ecuaciones de recurrencia:

$$T(n) = \begin{cases} c_0 & \text{si } n = 0 \\ c_1 + T(n-1) & \text{si } n > 0 \end{cases}$$

- Multiplicación por el método del campesino egipcio.

```
int prod ( int a, int b )
{
  if ( b == 0 ) return 0;
  else if ( b == 1 ) return a;
  else if (b %2 == 0) return prod(2*a, b/2);
  else if (b %2 == 1) return prod(2*a, b/2) + a;
}
```

- Ecuaciones de recurrencia: ($n = b$ tamaño del problema)

$$T(n) = \begin{cases} c_0 & \text{si } n = 0, 1 \\ c_1 + T(n/2) & \text{si } n > 1 \end{cases}$$

- Números de fibonacci.

```
int fib( int n )
{
  if   ( n == 0 ) return 0;
  else if ( n == 1 ) return 1;
  else return fib(n-1) + fib(n-2);
}
```

- Ecuaciones de recurrencia:

$$T(n) = \begin{cases} c_0 & \text{si } n = 0, 1 \\ T(n-1) + T(n-2) + c_1, & \text{si } n > 1 \end{cases}$$

Despliegue de recurrencias

Método para obtener una *fórmula explícita* del orden de complejidad de una recurrencia.

1. **Despliegue**. Sustituimos las apariciones de T en la recurrencia tantas veces como sea necesario hasta encontrar una fórmula que dependa del número de llamadas recursivas k.
2. **Postulado**. Obtenemos el valor de k que nos permite alcanzar un caso directo y, en la fórmula paramétrica, sustituimos k por ese valor y la referencia recursiva T por la complejidad del caso directo. La fórmula obtenida es la expresión explícita del orden de complejidad.
3. **Demostración**. La fórmula explícita así obtenida sólo es correcta si la recurrencia para el caso recursivo también es válida para el caso directo. Se comprueba demostrando por inducción que la fórmula obtenida cumple las ecuaciones de recurrencia.

- Factorial
  - Ecuaciones

$$T(n) = \begin{cases} c_0 & \text{si } n = 0 \\ c_1 + T(n-1) & \text{si } n > 0 \end{cases}$$

  - Despliegue

$$\begin{aligned} T(n) &= c_1 + T(n-1) \\ &= c_1 + c_1 + T(n-2) \\ &= c_1 + c_1 + c_1 + T(n-3) \\ &\dots \\ &= c_1 * k + T(n-k) \end{aligned}$$

  - Postulado
    El caso directo se tiene para $n = 0$

$$n - k = 0 \Leftrightarrow k = n$$

$$T(n) = c_1 n + T(n-n) = c_1 n + T(0) = c_1 n + c_0$$

    Por lo tanto $T(n) \in O(n)$

Disminución del tamaño del problema por sustracción.

- Cuando:
  1. la descomposición recursiva se obtiene restando una cierta cantidad constante
  2. el caso directo tiene coste constante
  3. la preparación de las llamadas y la combinación de los resultados tiene coste polinómico, entonces la ecuación de recurrencias será de la forma:

$$T(n) = \begin{cases} c_0 & \text{si } 0 \leq n < n_0 \\ a * T(n-b) + c * n^k & \text{si } n \geq n_0 \end{cases}$$

- Se puede demostrar (libro Diseño de programas. Formalismo y Abstracción. R. Peña. Pag: 17):

$$T(n) \in \begin{cases} \Theta(n^{k+1}) & \text{si } a = 1 \\ \Theta(a^{n/b}) & \text{si } a > 1 \end{cases}$$

- Vemos que, cuando el tamaño del problema disminuye por sustracción,
  - En recursión simple ($a = 1$) el coste es polinómico y viene dado por el producto del coste de cada llamada ($c * n^k$) y el coste lineal de la recursión (n).
  - En recursión múltiple ($a > 1$), por muy grande que sea b, el coste siempre es exponencial.

Disminución del tamaño del problema por división.

- Cuando:
  1. la descomposición recursiva se obtiene dividiendo por una cierta cantidad constante
  2. el caso directo tiene coste constante
  3. la preparación de las llamadas y la combinación de los resultados tiene coste polinómico, entonces la ecuación de recurrencias será de la forma:

$$T(n) = \begin{cases} cn^k & \text{si } 1 \leq n < b \\ a * T(n/b) + c * n^k & \text{si } n \geq b \end{cases}$$

- Se puede demostrar (libro Diseño de programas. Formalismo y Abstracción. R. Peña. Pag: 20):

$$T(n) = \begin{cases} \Theta(n^k) & \text{si } a < b^k \\ \Theta(n^k * \log n) & \text{si } a = b^k \\ \Theta(n^{\log_b a}) & \text{si } a > b^k \end{cases}$$

- Si $a \leq b^k$ la complejidad depende de n^k que es el término que proviene de $c * n^k$ en la ecuación de recurrencias, y, por lo tanto, la complejidad de un algoritmo de este tipo se puede mejorar disminuyendo la complejidad de la preparación de las llamadas y la combinación de los resultados.
- Si $a > b^k$ las mejoras en la eficiencia se pueden conseguir
  - disminuyendo el número de llamadas recursivas a o aumentando el factor de disminución del tamaño de los datos b, o bien
  - optimizando la preparación de las llamadas y combinación de los resultados, pues, si esto hace disminuir k suficientemente, podemos pasar a uno de los otros casos: $a = b^k$ o incluso $a < b^k$.

Verificación de algoritmos recursivos

- Para verificar que un algoritmo recursivo es correcto debemos probar que: ($d(\vec{x})$ es la condición del caso base, $s(\vec{x})$ es la función sucesor, y $c(\vec{x}, \vec{y}')$ es la función de combinación).
  1. Se cubren todos los casos:
  $$P(\vec{x}) \Rightarrow d(\vec{x}) \vee \neg d(\vec{x})$$
  2. El caso base es correcto:
  $$P(\vec{x}) \wedge d(\vec{x}) \Rightarrow Q(\vec{x}, \vec{y})$$
  3. Los argumentos de la llamada recursiva deben satisfacer su precondición:
  $$P(\vec{x}) \wedge \neg d(\vec{x}) \Rightarrow P(s(\vec{x}))$$
  4. El paso de inducción es correcto ($\Leftrightarrow$ la etapa de combinación es correcta):
  $$P(\vec{x}) \wedge \neg d(\vec{x}) \wedge Q(s(\vec{x}), \vec{y}') \Rightarrow Q(\vec{x}, c(\vec{x}, \vec{y}'))$$

- Para garantizar que termine la secuencia de llamadas a la función debemos exigir además:
  1. Existe una función de cota $t(\vec{x})$ tal que:

$$P(\vec{x}) \Rightarrow t(\vec{x}) \geq 0$$

  2. La función de cota debe decrecer en cada iteración:

$$P(\vec{x}) \wedge \neg d(\vec{x}) \Rightarrow t(s(\vec{x})) < t(\vec{x})$$

- **Ejemplo**. Verifica el siguiente algoritmo:

```
method potencia (a : int, n : int) returns (p : int)
requires n >= 0
ensures p == pot(a,n)
{
  if (n == 0) {p := 1;}
  else {p := potencia(a,n-1); p := p*a;}
}
```

- Solución:
  - Se cubren todos los casos: Como $n \geq 0$: $n = 0 \vee n > 0$ es cierto.
  - El caso base es correcto: Verifica la postcondición:

$$\{n = 0\}p = 1\{p = a^n\}$$

    ya que $1 = a^n \Leftarrow n = 0$.
  - En cada iteración los argumentos de la llamada recursiva verifican la precondición. El único problema consistiría en que n dejase de ser ≥ 0. Como en el caso recursivo $n > 0$, $n - 1$ sigue siendo ≥ 0.
  - El paso de inducción es correcto:

$$n \geq 0 \wedge n > 0 \wedge p = a^{n-1} \Rightarrow (p * a = a^n)$$

  - Existe una función cota:

$$t = n(\geq 0)$$

  - La función cote decrece:

$$n \geq 0 \wedge n > 0 \Rightarrow n - 1 < n$$

Diseño de algoritmos recursivos

1. Planteamiento recursivo. Realizar una descomposición recursiva de la postcondición. Definir la estrategia recursiva para alcanzar la postcondición.
2. Análisis de casos. Identificar las condiciones que permiten discriminar los casos directos de los recursivos. Deben tratarse de forma exhaustiva y mutuamente excluyente todos los casos contemplados en la precondición.
    - Caso directo: Encontrar las acciones que resuelven los casos directos.
    - Obtener las funciones sucesor que nos proporcionan los datos que empleamos para realizar las llamadas recursivas.
    - Cada una de las descomposiciones recursivas ha de permitir realizar la(s) llamada(s) recursiva(s), es decir, la función sucesor debe proporcionar unos datos que cumplan la precondición de la función recursiva.
    - Obtener las funciones de combinación de los casos recursivos.

3. Función de acotación y terminación. Determinamos si la función sucesor escogida garantiza la terminación de las llamadas. Definir una función cota que estime el número de llamadas restantes hasta alcanzar un caso base. Justificar que se decrementa en cada llamada.
   Si hay más de un caso recursivo, se ha de garantizar la terminación para cada uno de ellos.
4. Escritura del caso recursivo. Eliminar las variables auxiliares que no sean necesarias.

- Ejemplo: suma de las n primeras componentes de un vector de enteros.

```
function Sum(s : seq<int>) : int
ensures s == [] ==> Sum(s) == 0
ensures s != [] ==> Sum(s) == s[0] + Sum(s[1..])
{ if s == [] then 0 else s[0] + Sum(s[1..]) }

method sumaVector (v : array<int>, n : int)
     returns (s : int)
requires v != null
ensures s == Sum(v[..n])
{
   // cuerpo de la función
}
```

1. Planteamiento recursivo. Descomposición de la postcondición:

```
Sum(v[..n]) == Sum(v[..n-1]) + v[n-1]
```

Estrategia recursiva:

```
sumaVec(v,n) = sumaVec(v, n-1) + v[n-1]
```

2. Análisis de casos.

Solución directa: $n = 0$, donde el resultado es 0.

Solución recursiva: $n > 0$

- Función sucesor : `s(v, n) = (v, n-1)`. Cumple la precondición del algoritmo, ya que en la solución recursiva $n > 0$.
- Función de combinación: sumar al resultado de la llamada recursiva `v[n-1]`

1. Función de acotación y terminación.

   Al avanzar la recursión, n se va acercando a 0

$$t(v, n) = n$$

   Se cumple

$$n - 1 < n$$

2. La función obtenida es:

```
method sumaVector(v:array<int>,fin:int) returns(s:int)
requires v != null
requires 0 <= fin <= v.Length
ensures s == Sum(v[..fin])
{
  if (fin == 0) {s:= 0;}
  else {
    assume Sum(v[..fin-1]) + v[fin-1] == Sum(v[..fin]);
    s := sumaVector(v,fin-1); s := s + v[fin-1];
  }
}
```

Técnicas de generalización o técnicas de inmersión

Decimos que una acción parametrizada (procedimiento o función) F es una generalización de otra acción f cuando:

- F tiene más parámetros de entrada y/o devuelve más resultados que f.
- Particularizando los parámetros de entrada adicionales de F a valores adecuados y/o ignorando los resultados adicionales de F se obtiene el comportamiento de f.

Técnicas de generalización o técnicas de inmersión

Conseguir planteamientos recursivos.

Ejemplo: Búsqueda de un elemento en un vector(búsqueda binaria):

```
method searchGen(v:array<int>,ini:int,fin:int,x:int)
    returns (b : bool)
requires v != null
requires 0 <= ini <= fin <= v.Length
requires forall u,w::ini<=u<w<fin==> v[u] <= v[w]
ensures b == (x in v[ini..fin])

method search(v:array<int>,x:int)
     returns (b : bool)
requires v != null
requires forall u,w::0<=u<w<v.Length==> v[u] <= v[w]
ensures b == (x in v[..])
{
  b := searchGen(v,0,v.Length,x);
}
```

Técnicas de generalización o técnicas de inmersión

Transformar algoritmos recursivos no finales en algoritmos finales.

```
function escalar(s1:seq<int>,s2:seq<int>):int
requires |s1| == |s2|
ensures s1==[] ==> escalar(s1,s2)==0
ensures s1!=[] ==>
   escalar(s1,s2)== escalar(s1[1..],s2[1..])+s1[0]*s2[0]

method prodEscalar(v1:array<int>,v2:array<int>)
      returns(r:int)
requires v1 != null && v2 != null
requires v1.Length == v2.Length
ensures r == escalar(v1[..v1.Length],v2[..v2.Length])
{
  r := prodEscalarGen(v1,v2,v1.Length);
}
```

Técnicas de generalización o técnicas de inmersión

```
method escalarGen(v1:array<int>,v2:array<int>,n:int)
    returns(r:int)
requires v1 != null && v2 != null
requires v1.Length == v2.Length
requires 0 <= n <= v1.Length
ensures r == escalar(v1[..n],v2[..n])
{
  if (n == 0) {r:= 0;}
  else {
    r := escalarGen(v1,v2,n-1);
    r := r + v1[n-1]*v2[n-1];
  }
}
```

Técnicas de generalización o técnicas de inmersión

```
method escalarFinal(v1:array<int>,v2:array<int>,n:int,
    j:int, sol:int) returns(r:int)
requires v1 != null && v2 != null
requires v1.Length == v2.Length
requires 0 <= j <= n <= v1.Length
requires sol == escalar(v1[j..n],v2[j..n])
ensures r == escalar(v1[..n],v2[..n])
{
  if (j == 0) {r:= sol;}
  else {
    r := escalarFinal(v1,v2,n,j-1,sol+v1[j-1]*v2[j-1]);
  }
}
```

Técnicas de generalización o técnicas de inmersión

Mejorar la eficiencia añadiendo parámetros acumuladores. Se aplica esta técnica cuando en un algoritmo recursivo f se detecta una expresión $e(\vec{x})$, que sólo depende de los parámetros de entrada, cuyo cálculo se puede simplificar utilizando el valor de esa expresión en anteriores llamadas recursivas.

Ejemplo: función que calcula cuántas componentes de un vector son iguales a la suma de las componentes situadas a su derecha:

```
function Count(s:seq<int>):nat
  ensures s == [] ==> Count(s) == 0
  ensures s != [] && s[0] == Sum(s[1..]) ==>
        Count(s) == 1 + Count(s[1..])
  ensures s != [] && !(s[0] == Sum(s[1..])) ==>
        Count(s) == Count(s[1..])

method numCortesDr(v : array<int>) returns (s : int)
requires v != null
ensures s == Count(v[..v.Length])
```

Técnicas de generalización o técnicas de inmersión

Primera generalización: flexibilizar límites del vector.

```
method numCortesDrGen (v : array<int>, fin : int)
    returns (s : int)
requires v != null
requires 0 <= fin <= v.Length
ensures s == Count(v[..fin])
{
    if (fin == 0) {s:= 0;}
    else {
        var suma := 0; var i := fin;
        while (i < v.Length)
          {  suma := suma+v[i]; i := i+1;}
        if (suma == v[fin-1])
          { s := numCortesDrGen(v,fin-1); s := s+1;}
        else {s := numCortesDrGen(v,fin-1); }
    }
}
```

Técnicas de generalización o técnicas de inmersión

Cálculo de la suma en un parámetro acumulador.

```
method numCortesDrGen (v:array<int>, fin:int, suma:int)
    returns (s : int)
requires v != null
requires 0 <= fin <= v.Length
requires suma == Sum(v[fin..])
ensures s == Count(v[..fin])
{
  if (fin == 0) {s:= 0;}
  else if (suma == v[fin-1])  {
       s := numCortesDrGen(v,fin-1,suma+v[fin-1]);
       s := s + 1;
  }
  else {
       s := numCortesDrGen(v,fin-1,suma+v[fin-1]);
  }
}
```

Técnicas de generalización o técnicas de inmersión

Mejorar la eficiencia añadiendo resultados acumuladores. Se aplica cuando en un algoritmo recursivo f se detecta una expresión, que puede depender de los parámetros de entrada y los resultados de la llamada recursiva y cuyo cálculo se puede simplificar utilizando el valor de esa expresión en posteriores llamadas recursivas.

Ejemplo: cuántas componentes de un vector son iguales a la suma de las componentes que la preceden.

```
function Count(s:seq<int>):nat
  ensures s == [] ==> Count(s) == 0
  ensures s != [] && s[|s|-1] == Sum(s[..|s|-1]) ==>
       Count(s) == 1 + Count(s[..|s|-1])
  ensures s != [] && !(s[|s|-1] == Sum(s[..|s|-1])) ==>
       Count(s) == Count(s[..|s|-1])

method numCortesIz(v : array<int>) returns (s : int)
requires v != null
ensures s == Count(v[..v.Length])
```

Implementación:

```
method numCortesIzGen (v : array<int>, fin : int)
    returns (s : int)
requires v != null
requires 0 <= fin <= v.Length
ensures s == Count(v[..fin])
{
  if (fin == 0) {s:= 0;}
  else {
    var suma := 0; var i := 0;
    while (i < fin) { suma := suma + v[i]; i := i +1; }
    if (suma == v[fin-1]) {
        s := numCortesIzGen(v,fin-1); s := s + 1;
    }
    else {s := numCortesIzGen(v,fin-1); }
  }
}
```

Generalización con resultados acumuladores:

```
method numCortesIzGen (v : array<int>, fin : int)
    returns (s : int, suma : int)
requires v != null && 0 <= fin <= v.Length
ensures suma == Sum(v[..fin])
ensures s == Count(v[..fin])
{
  if (fin == 0) {s:= 0;suma := 0;}
  else {
    s,suma := numCortesIzGen(v,fin-1);
    if (suma == v[fin-1]) { s := s + 1; }
    suma := suma + v[fin-1];
  }
}

method numCortesIz(v : array<int>) returns (s : int)
requires v != null
ensures s == Count(v[..v.Length])
{ var suma;
  s,suma := numCortesIzGen(v,v.Length);
}
```