Diseño de algoritmos iterativos

Isabel Pita

Facultad de Informática - UCM

17 de octubre de 2017

Bibliografía Recomendada

- Algoritmos correctos y eficientes: Diseño razonado ilustrado con ejercicios. *Matí-Oliet, N.; Segura Diaz, C. M., Verdejo Lopez, A.*. Ibergarceta Publicaciones, 2012.
- Especificación, Derivación y Análisis de Algoritmos: ejercicios resueltos. *Narciso Martí Oliet, Clara María Segura Díaz y Jose Alberto Verdejo López*. Colección Prentice Práctica, Pearson Prentice-Hall, 2006

Capítulos 2, 3, y 4 de ambos libros.

- Complejidad de algoritmos iterativos:
  - Ejercicios resueltos: 3.21 a), 3.23 a), 3.25
  - Ejercicios propuestos: 3.12, 3.13
- Verificación de algoritmos iterativos
  - Ejercicios resueltos: 2.7, 2.8, 2.9, 2.13, 2.15
- Derivación de algoritmos iterativos
  - Ejercicios resueltos: 4.3, 4.4, 4.5, 4.6, 4.7, 4.8, 4.9, 4.10, 4.11, 4.12, 4.13, 4.14, 4.15, 4.22, 4.23, 4.24, 4.25, 4.27

Problemas de acepta el reto relacionados con este tema:

- Recorridos de secuencias
    1. 171 . Abadías pirenaicas
    2. 179. Molinos de viento
    3. 234 Carreras de coches
    4. 248. Los premios de las tragaperras
    5. 249 El burro y las alforjas
    6. 314. Temperaturas extremas
    7. 316. Racha afortunada
    8. 320. Palmeras en la nieve
    9. 345. Sudokus correctos
    10. 346. El hombre de los 6 dedos
- Búsquedas
    1. 168. La pieza perdida
    2. 209. Pánico en el túnel
    3. 247. Saliendo de la crisis
    4. 300. Palabras pentavocálicas

Objetivos

- Reglas para calcular el coste asintótico de programas iterativos.
- Reglas para comprobar que un programa es correcto respecto a su especificación (verificación).
- Cómo implementar un programa correcto por construcción (derivación).
- Ver soluciones de problemas iterativos típicos para conocer diversas formas de solución.

Reglas prácticas para el cálculo de la eficiencia de algoritmos iterativos

- Las instrucciones de asignación, de entrada-salida, los accesos a elementos de un vector y las expresiones aritméticas y lógicas, (siempre que no involucren variables cuyos tamaños dependan del tamaño del problema) tendrán un coste constante, $\Theta(1)$. No se cuentan los *return*.
- Para calcular el coste de una composición secuencial de instrucciones, $S_1; S_2$ se suma los costes de S_1 y S_2. Si el coste de S_1 está en $\Theta(f_1(n))$ y el de S_2 está en $\Theta(f_2(n))$, entonces: $\Theta(f_1(n)) + \Theta(f_2(n)) = \Theta(max(f_1(n), f_2(n)))$.

- Para calcular el coste de una instrucción condicional:

$$\textbf{if } (B)\ \{S_1\}\ \textbf{else}\ \{S_2\}$$

  Si el coste de S_1 está en $\mathcal{O}(f_1(n))$, el de S_2 está en $\mathcal{O}(f_2(n))$ y el de B en $\mathcal{O}(f_B(n))$, podemos señalar dos casos para el condicional:

  - *Caso peor*: $\mathcal{O}(max(f_B(n), f_1(n), f_2(n))$.
  - *Caso promedio*: $\mathcal{O}(max(f_B(n), f(n))$ donde $f(n)$ es el promedio de $f_1(n)$ y $f_2(n)$.

  Se pueden encontrar expresiones análogas a éstas para la clase omega.

- Para calcular el coste de un bucle:

$$\textbf{while } (B)\ \{S\}$$

  Hay que calcular primero el coste de cada vuelta del bucle y después sumar los costes de todas las vueltas. El número de iteraciones dependerá de lo que tarde en hacerse falso *B*, teniendo en cuenta los datos concretos sobre los que se ejecute el programa.

- Búsqueda secuencial

```cpp
template <class T>
bool search(std::vector<T> const& v, T const& x ) {
  size_t j = 0;
  while (j < v.size() && v[j] != x ) ++j;
  return j < v.size();
}
```

- Búsqueda binaria

```cpp
template <class T>
bool BinSearch(std::vector<T> const& v,T const& x) {
    int ini = 0,  fin = v.size(), mitad;
    bool enc = false;
    while ( ini < fin && ! enc) {
        mitad = (ini+fin-1) / 2;
        if ( v[mitad] == x ) enc = true;
        else if ( v[mitad] < x ) ini = mitad + 1;
        else fin = mitad;
    }
    return enc;
}
```

Verificación

- Verificar es demostrar que las instrucciones de un algoritmo son correctas, es decir, para una entrada válida (precondición) producen el resultado esperado (postcondición).
- Ejemplo: Intercambiar el valor de dos componentes de un vector
  Especificación:

```
method swap (a:array<int>, i:int, j:int)
requires a != null
requires 0<=i<a.Length && 0<=j<a.Length && i!=j
ensures a[i] == old(a[j]) && a[j] == old(a[i])
modifies a
```

- ¿ Cuales de los siguientes algoritmos son correctos?

aux=v[i];
v[i]=v[j];
v[j]=aux;
	v[i]=v[j];
	v[j]=v[i];
		v[i]=v[i]-v[j];
		v[j]=v[i]+v[j];
		v[i]=v[j]-v[i];
			v[i]=v[i]+v[j];
			v[j]=v[i]-v[j];
			v[i]=v[j]-v[i];

- Para verificar un algoritmo $A \equiv A_1; A_2; \ldots, A_n$ se definen predicados intermedios, $R_0, \ldots R_n$, entre cada instrucción elemental, llamados *aserciones* o *asertos*:

$$\{R_0\}A_1\{R_1\}; ...; \{R_{n-1}\}A_n\{R_n\}$$

- Si para cada instrucción A_k se satisface $\{R_{k-1}\}A_k\{R_k\}$ y $P \Rightarrow R_0$ y $R_n \Rightarrow Q$ entonces se satisface $\{P\}A\{Q\}$.
- La semántica del lenguaje define para cada tipo de instrucción del lenguaje las reglas que determinan si se satisface $\{R_{k-1}\}A_k\{R_k\}$ (reglas de verificación).

Verificación

Axioma de asignación Para toda variable x, toda expresión válida del mismo tipo E y todo predicado R, la especificación:

$$\boxed{\begin{array}{l} P : \{Dom(E) \wedge \{R_x^E\} \\ \quad\quad \mathrm{x = E} \\ Q : \{R\} \end{array}}$$

es correcta.

P se denomina precondición más débil (*pmd*).

- $Dom(E)$ el conjunto de todos los estados en los que la expresión E está definida.
- R_x^E el predicado resultante de sustituir toda aparición de x por E en el predicado R.

Ejemplo: Suponiendo x entero determina la precondición más débil que safisfaga la especificación:

$pmd(Q) :$ $\mathrm{x = x + 2}$ $Q : \{x \geq 0\}$	$pmd(Q) : \{x + 2 \geq 0\} \equiv \{x \geq -2\}$ $\mathrm{x = x + 2} \quad \uparrow$ $Q : \{x \geq 0\}$

Regla de inferencia de la asignación

La especificación $\{P\}\ \mathrm{x} = \mathrm{E}\ \{Q\}$ es correcta si $P \Rightarrow pmd(Q)$.

Indica que especificaciones son correctas:

$P : \{x \geq 10\}$ $\mathrm{x} = \mathrm{x} + 2$ $Q : \{x \geq 0\}$	$P : \{x \geq -5\}$ $\mathrm{x} = \mathrm{x} + 2$ $Q : \{x \geq 0\}$	$P : \{x < 10\}$ $\mathrm{x} = \mathrm{x} + 2$ $Q : \{x \geq 0\}$

Regla de inferencia de la composición secuencial

La especificación $\{P\}\ A_1; A_2\ \{Q\}$ es correcta si existe un predicado R tal que las especificaciones $\{P\}\ A_1\ \{R\}$ y $\{R\}\ A_2\ \{Q\}$ son correctas.

Ejemplo:

```
method swap (a : array<int>, i : int, j : int)
 requires a != null
 requires 0<=i<a.Length && 0<=j<a.Length && i!=j
 ensures a[i] == old(a[j]) && a[j] == old(a[i])
 modifies a
 {
  a[i] := a[i] - a[j];
  a[j] := a[i] + a[j];
  a[i] := a[j] - a[i];
  assert a[i]==old(a[j]) && a[j]==old(a[i]);
 }
```

Verificación

```
method swap (a : array<int>, i : int, j : int)
 requires a != null
 requires 0<=i<a.Length && 0<=j<a.Length && i!=j
 ensures a[i] == old(a[j]) && a[j] == old(a[i])
 modifies a
 {
  a[i] := a[i] - a[j];
  a[j] := a[i] + a[j];
  assert a[j]-a[i]==old(a[j]) && a[j]==old(a[i]);
  a[i] := a[j] - a[i];
  assert a[i]==old(a[j]) && a[j]==old(a[i]);
 }
```

Verificación

```
method swap (a : array<int>, i : int, j : int)
 requires a != null
 requires 0<=i<a.Length && 0<=j<a.Length && i!=j
 ensures a[i] == old(a[j]) && a[j] == old(a[i])
 modifies a
 {
  assert a[j]==old(a[j]) && a[i]==old(a[i]);
  assert a[j]==old(a[j]) && a[i]-a[j]+a[j]==old(a[i]);
  a[i] := a[i] - a[j];
  assert a[j]==old(a[j]) && a[i]+a[j]==old(a[i]);
  assert a[i]+a[j]-a[i]==old(a[j])&&a[i]+a[j]==old(a[i])
  a[j] := a[i] + a[j];
  assert a[j]-a[i]==old(a[j]) && a[j]==old(a[i]);
  a[i] := a[j] - a[i];
  assert a[i]==old(a[j]) && a[j]==old(a[i]);
 }
```

Regla de inferencia de la composición alternativa

La especificación $\boxed{\begin{array}{l} \{P\} \\ \quad \texttt{if}\ (B)\ A_1\ \texttt{else}\ A_2; \\ \{Q\} \end{array}}$ es correcta si

las especificaciones $\boxed{\begin{array}{l} \{P \wedge B\} \\ \quad A_1 \\ \{Q\} \end{array}}$ y $\boxed{\begin{array}{l} \{P \wedge \neg B\} \\ \quad A_2 \\ \{Q\} \end{array}}$

son correctas.

La *pmd* es $B \wedge pmd(A_1, Q)) \vee (\neg B \wedge pmd(A_2, Q))$

Regla de inferencia de la composición iterativa

La especificación $\boxed{\{P\}\ \texttt{while}\ (B)\ \texttt{do}\ A\ \{Q\}}$ es correcta si existe un predicado I que llamaremos invariante y una función t dependiente de las variables que intervienen en el proceso y que toma valores enteros, que llamaremos función cota, de forma que:

1. $P \Rightarrow I$
2. $\{I \wedge B\}\ A\ \{I\}$
3. $I \wedge \neg B \Rightarrow Q$
4. $I \wedge B \Rightarrow t > 0$
5. $\{I \wedge B \wedge t = T\}\ A\ \{t < T\}$

Existen muchos predicados invariantes, se ha de elegir uno que nos permita verificar las 5 condiciones de corrección del bucle, esto es

- Lo suficientemente fuerte para $I \wedge \neg B \Rightarrow Q$
- Lo suficientemente débil para $P \Rightarrow I$.

Verificación

El invariante es un predicado que describe todos los estados por los que atraviesa el cómputo realizado por el bucle, observados justo antes de evaluar la condición B de terminación.

Cálculo de la posición del máximo de un vector.

```
v := [5,3,8,2,9,6]; i := 1; p := 0;
assert i==1 && p==0 && v== [5,3..]
while (i < v.Length) {
  if (v[i] > v[p]) {p := i;}
  i := i+1;
}
assert
  forall k::0<=k<v.Length==>v[k]<v[p]
```

valores antes cond.

estado	i	p
P_0	1	0
P_1	2	0
P_2	3	2
P_3	4	2
...		

¿ Que relaciones entre las variables se mantienen durante la ejecución del bucle?
Un invariante del bucle es:

```
invariant 0 <= i <= v.Length && 0 <= p < i
invariant forall k::0<=k<i==>v[k]<v[p]
```

Verificación

- La *función cota o función limitadora* es una función $t : estado \rightarrow \mathcal{Z}$ positiva que decrece cada vez que se ejecuta el cuerpo del bucle.
- La función cota garantiza la terminación del bucle
- Para encontrar una función cota se observan las variables que son modificadas por el cuerpo A del bucle, y se construye con ellas una expresión entera t que decrezca
- Ejemplo: cálculo del máximo del vector.
  - Las variables `i`, y `p` crecen, `v.Length` se mantiene inalterable, por lo tanto `v.Length` $-$ `i` decrece.
  - La condición del bucle `i <= v.Length` garantiza que la función es positiva.
  - La función cota es : `v.Length - i`.
- En Dafny la función cota se indica con la clausula `decreases`.
- Normalmente Dafny es capaz de encontrar la función cota sin necesidad de hacerla explícita. En los problemas de clase lo indicaremos para que Dafny la compruebe.

Ejemplos

```
method positivo (v : array<int>) returns (r : bool)
   requires v != null
   ensures r==forall k::0<=k<v.Length==>v[k]>0
   {
     var i := 0; r := true;
     while (i < v.Length && r)
       invariant 0<=i<=v.Length
       invariant r==forall k::0<=k<i==>v[k]>0
       decreases v.Length - i
       {
         if (v[i] <= 0) {r := false;}
         i := i+1;
       }
   }
```

El sistema verifica automáticamente la corrección del algoritmo dado.

Ejemplos

Otra implementación del mismo problema

```
method positivo2 (v : array<int>) returns (r : bool)
  requires v != null
  ensures r==forall k::0<=k<v.Length==>v[k]>0
  { var i := 0;
    while (i < v.Length && v[i]>0)
      invariant 0<=i<=v.Length
      invariant forall k::0<=k<i==>v[k]>0
      decreases v.Length - i
      {
           i := i+1;
      }
    r := i == v.Length;
  }
```

- La precondición y poscondición son las mismas
- Al cambiar la implementación del bucle cambia el invariante.
- Observa que en este invariante no aparece la variable r.
- El invariante se cumple si el predicado es cierto.

Comprobamos que el sistema verifica correctamente haciendo explícitos los 5 puntos de la verificación del bucle.

```
method positivo2 (v : array<int>) returns (r : bool)
  requires v != null
  ensures r==forall k::0<=k<v.Length==>v[k]>0
  {
    var i := 0;
  // Propiedad 1. P==>I
  assert i==0 ==>
        0<=i<=v.Length && forall k::0<=k<i==>v[k]>0;
    while (i < v.Length && v[i]>0)
      invariant 0<=i<=v.Length
      invariant forall k::0<=k<i==>v[k]>0
      {
        i := i+1;
      }
    r := i == v.Length;
  }
```

Ejemplos

```
method positivo2 (v : array<int>) returns (r : bool)
  requires v != null
  ensures r==forall k::0<=k<v.Length==>v[k]>0
  {
    var i := 0;
    while (i < v.Length && v[i]>0)
      invariant 0<=i<=v.Length
      invariant forall k::0<=k<i==>v[k]>0
      {  // Propiedad 2. {I && B} A {I}
  // I && B ==> pmd(I)
  assert 0<=i<=v.Length&&forall k::0<=k<i==>v[k]>0 ==>
       0<=i+1<=v.Length && forall k::0<=k<i+1==>v[k]>0;
  // pmd(I)
  assert 0<=i+1<=v.Length&&forall k::0<=k<i+1==>v[k]>0;
        i := i+1;
  assert 0<=i<=v.Length && forall k::0<=k<i==>v[k]>0;
      }
    r := i == v.Length;
  }
```

Ejemplos

```
method positivo2 (v : array<int>) returns (r : bool)
  requires v != null
  ensures r==forall k::0<=k<v.Length==>v[k]>0
  { var i := 0;
    while (i < v.Length && v[i]>0)
      invariant 0<=i<=v.Length
      invariant forall k::0<=k<i==>v[k]>0
      { i := i+1; }
// Propiedad 3. I && !B ==> Q
assert forall k::0<=k<i==>v[k]>0 && 0<=i<=v.Length &&
    (i>=v.Length ||v[i]<=0) ==>
    (i==v.Length)==forall k::0<=k<v.Length==>v[k]>0;
// Poscondicion del bucle
assert (i==v.Length)==forall k::0<=k<v.Length==>v[k]>0;
    r := i == v.Length;
//Poscondición del algoritmo
assert r==forall k::0<=k<v.Length==>v[k]>0;
  }
```

Ejemplos

Proponemos una función cota y probamos los puntos 4 y 5

```
method positivo2 (v : array<int>) returns (r : bool)
 ...while (i < v.Length && v[i]>0)
      invariant 0<=i<=v.Length
      invariant forall k::0<=k<i==>v[k]>0
      decreases v.Length - i
      {  // Propiedad 4. t > 0
         assert v.Length - i > 0;
// Propiedad 5. {I && B && t=T} A {t<T}
 ghost var T;
assume v.Length - i == T;
 // I && B && t =T ==> pmd(t < T)
assert 0<=i<=v.Length&&forall k::0<=k<i==>v[k]>0 &&
  v.Length-i < T==>v.Length - (i+1) < T;
assert v.Length - (i+1) < T;  // pmd(I)
        i := i+1;
  assert v.Length - i < T;
      }
...
```

Ejemplos

Verifica el siguiente algoritmo respecto a su especificación.

Especificación:

```
method suma (v : array<int>) returns (r : int)
requires v != null
ensures r == Sum(v[..])
```

Algoritmo:

```
var i := v.Length; r := 0;
while (i > 0)
{
  r := r + v[i-1];
  i := i-1;
}
```

La siguiente verificación la realiza automáticamente Dafny

```
method suma (v : array<int>) returns (r : int)
requires v != null
ensures r == Sum(v[..])
{
  var i := v.Length; r := 0;
  while (i > 0)
  invariant 0<=i<=v.Length && r==Sum(v[i..])
  {
    r := r + v[i-1];
    i := i-1;
  }
}
```

Derivación

Derivar: construir las instrucciones de un algoritmo a partir de su especificación asegurando su corrección.

Los algoritmos se ajustan al esquema:

$\{P\}$
A_0 (Inicialización)
$\{I, Cota\}$
while (B) {
  $\{I \wedge B\}$
  A_1 (Restablecer)
  $\{R\}$
  A_2 (Avanzar)
  $\{I\}$
}
$\{Q\}$

- A_0 Hace que el invariante se cumpla inicialmente.
- A_2 hace que la cota decrezca.
- A_1 mantiene el invariante a cierto.

- **Pasos** para construir un algoritmo con bucle:
  1. Diseñar el invariante y la condición del bucle sabiendo que se tiene que cumplir:
     $I \wedge \neg B \Rightarrow Q$
  2. Diseñar A_0 para hacer el invariante cierto: $\{P\}A_0\{I\}$
  3. Diseñar la función cota, *C*, de tal forma que: $I \wedge B \Rightarrow C \geq 0$.
  4. Diseñar A_2 y el predicado $R \equiv pmd(A_2, I)$.
  5. Diseñar A_1 para que se cumpla: $\{I \wedge B\}A_1\{R\}$.
  6. Comprobar que la cota realmente decrece:
     $$\{I \wedge B \wedge C = T\}A_1, A_2\{C < T\}$$

Ejemplos de derivación

Derivar un algoritmo para calcular la suma de los valores pares de un vector:

```
method sumaPares(v : array<int>) returns (r : int)
  requires v != null
  ensures r == SumP1Elem(v[..],par)
```

1. Obtenemos un invariante del bucle debilitando la postcondición.

```
    invariant 0 <= k <= v.Length
    invariant r == SumP1Elem(v[k..],par)
```

   Para que se cumpla $I \wedge \neg B \Rightarrow Q$, la condición de parada debe ser: $k == 0$ y por lo tanto la condición del bucle `k!=0` o `k>0`.

1. Instrucciones de inicialización. Para que $\{P\}A_0\{I\}$ sea correcto, `k = v.Length` y `r = 0`.
2. Se puede comprobar que Dafny considera correcto el siguiente aserto

```
method sumaPares(v : array<int>) returns (r : int)
  requires v != null
  ensures r == SumP1Elem(v[..],par)
{
  var k := v.Length;
  r := 0;
  assert 0 <= k <= v.Length&&r==SumP1Elem(v[k..],par);
  while (k != 0)
    invariant 0 <= k <= v.Length
    invariant r == SumP1Elem(v[k..],par)
  {
  }
}
```

1. La función cota: `decreases k`. Decrece en cada vuelta del bucle.
2. La función de avance `k = k-1`.

```
var k := v.Length;  r := 0;
while (k != 0)
  invariant 0 <= k <= v.Length
  invariant r == SumP1Elem(v[k..],par)
  decreases k
{
  .........
  k := k - 1;
}
```

- Dafny nos informa de que el invariante no se mantiene en el bucle.
- Nos falta la instrucción de recuperar el invariante. Esta instrucción debe dar valor a la variable `r`.

1. Para restaurar el invariante y que $\{I \wedge B\}A_1\{R\}$ sea correcto, vemos que necesitamos incrementar `s` en el valor de `v[k-1]` cuando el valor es par.
   Ahora el programa se verifica completo con Dafny

```
method sumaPares(v : array<int>) returns (r : int)
  requires v != null
  ensures r == SumP1Elem(v[..],par)
{
  var k := v.Length;     r := 0;
  while (k != 0)
    invariant 0 <= k <= v.Length
    invariant r == SumP1Elem(v[k..],par)
    decreases k
  {
    if (v[k-1] % 2) == 0 {    r := r + v[k-1]; }
    k := k - 1;
  }
}
```