Capítulo 1

Diseño de algoritmos iterativos[1]

Cada cosa tiene su belleza, pero no todos pueden verla.

Confucio

Resumen: En este tema se introduce los conceptos de verificación y derivación de algoritmos como métodos que nos permiten razonar sobre la corrección de un algoritmo y su construcción correcta respectivamente.

1. Introducción

En el capítulo anterior se ha visto la posibilidad de utilizar predicados para definir conjuntos de estados y cómo es posible especificar un algoritmo mediante dos predicados llamados precondición y postcondición. El primero describe el conjunto de estados válidos al comienzo del algoritmo y el segundo el conjunto de estados alcanzables con la ejecución del algoritmo.

En este capítulo representaremos los estados "intermedios" de un algoritmo mientras se ejecuta utilizando predicados, generalmente denominadas *aserciones* o *asertos*. Escribiendo asertos entre cada dos instrucciones tenemos un modo de razonar sobre la corrección de un programa imperativo.

2. Reglas prácticas para el cálculo de la eficiencia

1. 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*.

2. 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: $\Theta(f_1(n)) + \Theta(f_2(n)) = \Theta(max(f_1(n), f_2(n)))$.

3. Para calcular el coste de una instrucción condicional:

[1] Ramón González del Campo es el autor principal de este tema.

if (B) $\{S_1\}$ **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.

4. Bucles con la forma:

   while (B)

   $\{$
   S
   $\}$

   Para calcular el coste de tal bucle hay que calcular primero el coste de cada pasada y después sumar los costes de todas las pasadas que se hagan en el bucle. 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 y los grandes que sean.

2.1. Ejemplos de cálculo de complejidad de algoritmos iterativos

De manera práctica, se analizará el código de los algoritmos presentados de arriba hacia abajo y de dentro hacia fuera.

- Búsqueda secuencial

```
1  bool buscaSec( int v[], int num, int x ) {
2    int j;
3    bool encontrado;
4
5    j = 0;
6    encontrado = false;
7    while ( (j < num) && ! encontrado ) {
8      encontrado = ( v[j] == x );
9      j++;
10   }
11   return encontrado;
12 }
```

  - Caso peor: el elemento buscado x no está en el vector
    - Cuerpo del bucle while: 4 (3 en la línea 8 y 1 en la línea 9)
    - Bucle while: $\underbrace{n}_{j=0..n-1} * (\underbrace{4}_{cuerpo} + \underbrace{3}_{condicion}) + \underbrace{3}_{condicion} = 7n + 3$
    - Total: $7n + 3 + 2_{(inicializaciones)} = 7n + 5$
  - Caso mejor: el elemento buscado x está en la primera posición del vector; sólo se entra una vez en el bucle while
    - Cuerpo del bucle while: 4

- Bucle while: $(4+3)+3=10$
- Total: $10+2=12$

- Búsqueda binaria

```
1  int buscaBin( int v[], int num, int x ) {
2    int izq, der, centro;
3
4    izq = -1;
5    der = num;
6    while ( der != izq+1 ) {
7      centro = (izq+der) / 2;
8      if ( v[centro] <= x )
9        izq = centro;
10     else
11       der = centro;
12   }
13   return izq;
14 }
```

- Caso peor: el elemento no se encuentra en el vector
  - Cuerpo del bucle while: $\underbrace{3}_{linea\ 7} + \underbrace{3}_{if} = 6$
  - Bucle while: $\underbrace{\log(n)}_{vueltas\ while} * (\underbrace{6}_{cuerpo} + \underbrace{2}_{condicion}) + \underbrace{2}_{condicion} = 8\log(n)+2$
  - Total: $(8\log(n)+2)+2_{(inicializaciones)} = 8\log(n)+4$
- Caso mejor: el elemento buscado está en el centro del vector; sólo se entra una vez en el bucle while
  - Cuerpo del bucle while: 6
  - Bucle while:$(6+2)+2=10$
  - Total: $10+2=12$

En caso de que el elemento buscado estuviese repetido, ¿qué posición se devolvería?

- Ordenación por inserción

```
1  void ordenaIns ( int v[], int num ) {
2    int i, j, x;
3
4    for ( i = 1; i < num; i++ ) {
5      x = v[i];
6      j = i-1;
7      while ( (j >= 0 ) && (v[j] > x) ){
8        v[j+1] = v[j];
9        j = j-1;
10     }
11     v[j+1] = x;
12   }
13 }
```

- Caso peor: el vector está ordenado a la inversa; la segunda parte de la condición del while se cumple siempre

- Cuerpo del bucle while: 6 (4 de la línea 8 y 2 de la línea 9)
- Bucle while: $\underbrace{i}_{j=0..i-1} * (\underbrace{6}_{cuerpo} + \underbrace{4}_{condicion}) + \underbrace{4}_{condicion} = 10i + 4$
- Cuerpo del bucle for: $\underbrace{(10i+4)}_{while} + \underbrace{2}_{l5} + \underbrace{2}_{l6} + \underbrace{3}_{l11} + \underbrace{1}_{i++} = 10i + 12$
- Bucle for:

$$\sum_{i=1}^{n-1}(\underbrace{(10i+12)}_{cuerpo} + \underbrace{1}_{condicion}) + \underbrace{1}_{condicion} + \underbrace{1}_{ini} = \sum_{i=1}^{n-1}(10i+13) + 2 =$$

$$= 10\sum_{i=1}^{n-1} i + \sum_{i=1}^{n-1} 13 + 2 = 10\frac{n(n-1)}{2} + 13(n-1) + 2 =$$

$$= 5n^2 - 5n + 13n - 13 + 2 = 5n^2 + 8n - 11$$

- Caso mejor: el vector está ordenado; la segunda parte de la condición del while no se cumple nunca
  - Bucle while: $4_{(condicion)}$
  - Cuerpo del bucle for: $4 + 2 + 2 + 3 + 1 = 12$
  - Bucle for:

$$\sum_{i=1}^{n-1}(12+1) + 1 + 1 = \sum_{i=1}^{n-1} 13 + 2 = 13(n-1) + 2 = 13n - 11$$

- Ordenación por selección

```
1  void ordenaSel ( int v[], int num ) {
2    int i, j, menor, aux;
3
4    for ( i = 0; i < num; i++ ) {
5      menor = i;
6      for ( j = i+1; j < num; j++ )
7        if ( v[j] < v[menor] )
8          menor = j;
9      if ( i != menor ) {
10       aux = v[i];
11       v[i] = v[menor];
12       v[menor] = aux;
13     }
14   }
15 }
```

- Caso peor: vector desordenado
  - Cuerpo del for interno: $\underbrace{3}_{if} + \underbrace{1}_{l8} + \underbrace{1}_{j++} = 5$
  - Bucle for interno: $\underbrace{(n-i-1)}_{j=i+1..n-1}(\underbrace{5}_{cuerpo} + \underbrace{1}_{condicion}) + \underbrace{1}_{condicion} + \underbrace{2}_{ini} = 6n - 6i - 3$
  - Cuerpo del for externo: $\underbrace{1}_{l5} + \underbrace{(6n-6i-3)}_{for} + \underbrace{1}_{if} + \underbrace{2}_{l10} + \underbrace{3}_{l11} + \underbrace{2}_{l12} + \underbrace{1}_{i++}$
    $= 6n - 6i + 7$

- Bucle for externo:

$$\sum_{i=0}^{n-1}(\underbrace{(6n-6i+7)}_{cuerpo}+\underbrace{1}_{condicion})+\underbrace{1}_{condicion}+\underbrace{1}_{ini}=6\sum_{i=0}^{n-1}n-6\sum_{i=0}^{n-1}i+\sum_{i=0}^{n-1}8+2=$$

$$=6n^2-6\frac{n(n-1)}{2}+8n+2=6n^2-\frac{6}{2}n^2+\frac{6}{2}n+8n+2=3n^2+11n+2$$

- Caso mejor: vector ordenado; la condición del if más interno no se cumple nunca, y como consecuencia la del más externo tampoco
  - Cuerpo del for interno: 4
  - Bucle for interno: $(n-i-1)(4+1)+1+2=5n-5i-2$
  - Cuerpo del for externo: $1+(5n-5i-2)+1+1=5n-5i+1$
  - Bucle for externo:

$$\sum_{i=0}^{n-1}((5n-5i+1)+1)+1+1=5\sum_{i=0}^{n-1}n-5\sum_{i=0}^{n-1}i+\sum_{i=0}^{n-1}2+2=$$

$$=5n^2-5\frac{n(n-1)}{2}+2n+2=5n^2-\frac{5}{2}n^2+\frac{5}{2}n+2n+2=\frac{5}{2}n^2+\frac{9}{2}n+2$$

¿Qué sucede si el vector está ordenado al revés?

- Ordenación por el método de la burbuja modificado

```
1  void ordenaBur ( int v[], int num ) {
2    int i, j, aux;
3    bool modificado;
4
5    i = 0;
6    modificado = true;
7    while ( (i < num-1) && modificado ) {
8      modificado = false;
9      for ( j = num-1; j > i; j-- )
10       if ( v[j] < v[j-1] ) {
11         aux = v[j];
12         v[j] = v[j-1];
13         v[j-1] = aux;
14         modificado = true;
15       }
16     i++;
17   }
18 }
```

- Caso peor: El vector está ordenado al revés; se entra siempre en el if
  - Cuerpo del bucle for: $\underbrace{4}_{if}+\underbrace{2}_{l11}+\underbrace{4}_{l12}+\underbrace{3}_{l13}+\underbrace{1}_{l14}+\underbrace{1}_{j--}=15$
  - Bucle for: $\underbrace{(n-i-1)}_{j=i+1..n-1}(\underbrace{15}_{cuerpo}+\underbrace{1}_{condicion})+\underbrace{1}_{condicion}+\underbrace{2}_{ini}=16n-16i-13$
  - Cuerpo del bucle while: $\underbrace{1}_{l8}+\underbrace{16n-16i-13}_{for}+\underbrace{1}_{i++}=16n-16i-11$

- Bucle while:

$$\sum_{i=0}^{n-2}(\underbrace{(16n-16i-11)}_{cuerpo}+\underbrace{3}_{condicion})+\underbrace{3}_{condicion}=16\sum_{i=0}^{n-2}n-16\sum_{i=0}^{n-2}i-\sum_{i=0}^{n-2}8+3=$$

$$=16n(n-1)-16\frac{(n-2)(n-1)}{2}-8(n-1)+3=$$

$$=16n^2-16n-8n^2+24n-16-8n+8+3=8n^2-5$$

- Total: $\underbrace{8n^2-5}_{while}+\underbrace{2}_{ini}=8n^2-3$

- Caso mejor: El vector está ordenado; nunca se entra en el if y sólo se da una vuelta al while
  - Cuerpo del bucle for: $4+1=5$
  - Bucle for: $(n-1)(5+1)+1+2=6n-3$
  - Cuerpo del bucle while: $1+(6n-3)+1=6n-1$
  - Bucle while:
$$((6n-1)+3)+3=6n+5$$
  - Total: $6n+5+2=6n+7$

3. Verificación

3.1. Semántica de un lenguaje imperativo

- Especificar un algoritmo: encontrar dos predicados P y Q tales que: {P } A {Q }.
- Verificar: introducir predicados intermedios entre cada instrucción elemental:

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

- Si se satisface $\{R_{k-1}\}A_k\{R_k\}\ \forall k$ entonces se satisface $\{P\}A\{Q\}$.
- Buscaremos reglas para determinar si se satisface $\{R_{k-1}\}A_k\{R_k\}$ (reglas de verificación).
- Conocer dichas reglas para cada instrucción del lenguaje es equivalente a conocer la *semántica* de éste.

Todas las instrucciones que se definen más adelante cumplen las siguientes reglas generales:

- Si se cumple $\{P\}A\{Q\}$ podemos conocer otras especificaciones de A. Si $P'\Rightarrow P$ entonces se cumple $\{P'\}A\{Q\}$:

$$\frac{P'\Rightarrow P\quad \{P\}A\{Q\}}{\{P'\}A\{Q\}}\quad \text{(fortalecimiento de la precondición)}$$

- Si $Q\Rightarrow Q'$ entonces se cumple $\{P\}A\{Q'\}$:

$$\frac{\{P\}A\{Q\}\quad Q\Rightarrow Q'}{\{P\}A\{Q'\}}\quad \text{(debilitamiento de la postcondición)}$$

- Conjunción en la postcondición:

$$\frac{\{P\}A\{Q_1\} \quad \{P\}A\{Q_2\}}{\{P\}A\{Q_1 \wedge Q_2\}}$$

- Disyunción en la precondición:

$$\frac{\{P_1\}A\{Q\} \quad \{P_2\}A\{Q\}}{\{P_1 \vee P_2\}A\{Q\}}$$

Acabamos de ver que la precondición de una especificación correcta se puede fortalecer. Esto implica que para demostrar la corrección de $\{P\}A\{Q\}$ es suficiente con demostrar $\{R\}A\{Q\}$ (donde R es la condición más débil de A con respecto a Q, $pmd(A, Q)$, que verifica la postcondición) y ver que $P \Rightarrow R$:

$$\frac{P \Rightarrow pmd(A, Q)}{\{P\}A\{Q\}}$$

3.2. Reglas específicas para el cálculo de la precondición más débil

Denotemos por Q_x^e la sustitución en Q de las apariciones libres de la variable x por la expresión e. Entonces:

- Instrucción *nada*:

$$pmd(nada, Q) = Q$$

- Asignación simple:

$$pmd(x = e, Q) = dom(e) \wedge Q_x^e$$

  donde $dom(e)$ denota el conjunto de estados en los que la expresión e está definida.

- Composición secuencial:
  Consideremos la siguiente composición de instrucciones $A_1; A_2$.

$$pmd(A_1; A_2, Q) = pmd(A_1, pmd(A_2, Q))$$

- Sentencias condicionales:

$$pmd(\textbf{\textit{if}}\ (B)\{A_1\}\ \textbf{\textit{else}}\{A_2\}, Q) = (B \wedge pmd(A_1, Q)) \vee (\neg B \wedge pmd(A_2, Q))$$

3.3. Ejemplos

- Suponiendo x entero determina el predicado más débil, P, que safisfaga la especificación: $\{P\}x = x + 2\{x \geq 0\}$:

  $pmd(x = x + 2, x \geq 0) \Leftrightarrow (x \geq 0)_x^{x+2}$

  $\Leftrightarrow (x + 2) \geq 0$
  $\Leftrightarrow x \geq -2$

- Suponiendo x, y enteros, calcula en cada caso el predicado P más débil que satisfaga la especificación: $\{P\}x = x * x; x = x + 1\{x > 0\}$

  $pmd(x = x * x; x = x + 1, x > 0)$

  $\Leftrightarrow pmd(x = x * x; pmd(x = x + 1, x > 0))$
  $\Leftrightarrow pmd(x = x * x; (x > 0)^{x+1}_x$
  $\Leftrightarrow ((x > 0)^{x+1}_x)^{x*x}_x$
  $\Leftrightarrow (x + 1 > 0)^{x*x}_x$
  $\Leftrightarrow (x * x) + 1 > 0$
  $\Leftrightarrow cierto$

- Suponiendo x, y enteros, determina el predicado más débil, P, que safisfaga la especificación dada:

  $\{P\}$ **if** $(x \geq 0)$ $\{y = x\}$ **else** $\{y = -x\}$ $\{y = 4\}$

  pmd(**if** $(x \geq 0)$ $\{y = x\}$ **else** $\{y = -x\}$,y=4)

  $\Leftrightarrow (x \geq 0 \wedge pmd(y = x, y = 4)) \vee (x < 0 \wedge pmd(y = -x, y = 4))$
  $\Leftrightarrow (x \geq 0 \wedge x = 4) \vee (x < 0 \wedge -x = 4)$
  $\Leftrightarrow x = 4 \vee x = -4$

  Luego, este fragmento calcula el valor absoluto de un valor.

3.4. Bucles e invariantes

- Consideremos la siguiente instrucción:

  $\{P\}$
  while (B) {
  A
  }
  $\{Q\}$

  donde A es una instrucción y B es una expresión booleana.

- Si B es falso el cuerpo del bucle no se ejecuta ninguna vez → Es como si no estuviera ⇔ Es equivalente a la instrucción *nada*.

- Para la verificación del bucle necesitamos un predicado llamado *invariante*, I, que describe los distintos estados por los que pasa el bucle:

  - I se satisface antes de ejecutarse el bucle: $P \Rightarrow I$.
  - I se satisface en cada iteración del bucle: $\{I \wedge B\}A\{I\}$.
  - I se satisface cuando finaliza la ejecución del bucle: $\{I \wedge \neg B\} \Rightarrow \{Q\}$.

- El invariante representa la relación entre las variables del bucle.

- En cada iteración, el cuerpo del bucle modifica los valores de las variables pero **no** su relación ⇔ Existen conjuntos de valores para las variables incompatibles entre sí.

- Es preciso que el programador "invente" el predicado invariante → PUEDE SER MUY DIFÍCIL!!!.

- PROBLEMA:

- Consideremos una especificación de un algoritmo y un algoritmo que verifica esta especificación.
- Sustituyamos el cuerpo del bucle por la instrucción *nada*.
- ¿Qué ocurre?
  - El algoritmo no funciona, se mete en un bucle infinito.
  - ¡NUESTRO INVARIANTE SIGUE VALIENDO! $\Leftrightarrow$ Verifica las condiciones anteriores.
- Nuestra verificación es defectuosa $\rightarrow$ Es preciso exigir algo que cumpla nuestra primera verificación y **no** cumpla la segunda.

- Introducimos una función $cota : estado \rightarrow \mathbb{Z}$ que depende de las variables del cuerpo del bucle.
  - Es mayor o igual que cero cuando B se cumple: $I \wedge B \Rightarrow cota \geq 0$.
  - Decrece a ejecutarse el bucle: $\{I \wedge B \wedge cota = T\}A\{cota < T\}$, donde T es una constante.

3.5. Ejemplos

Verifica el siguiente algoritmo:

```
1   int suma(int V[], int N){
2   int n, x;
3   n = N;
4   x = 0;
5   while (n != 0)
6   {
7       x = x+V[n-1];
8       n = n-1;
9   }
10  return x;
11  }
```

donde:

$\{N \geq 0\}$
fun suma(**int** V[N]) **return int** x
$\{x = (\Sigma i : 0 \leq i < N : V[i])\}$

utilizando como invariante: $I \equiv 0 \leq n \leq N \wedge x = (\Sigma i : n \leq i < N : V[i])$.

- $0 \leq n \leq N$: representa el rango de valores válidos de n.
- $x = (\Sigma i : n \leq i < N : V[i]) \rightarrow x$ contiene la suma parcial de los elementos procesados.
- Relación entre x y n: x representa la suma parcial de los valores de V comprendidos entre n y $N - 1$.

Solución:

1. Es preciso demostrar que el invariante se cumple justo antes de comenzar el bucle $\rightarrow$ Es preciso demostrar:

$$\boxed{\{P\}n = N; x = 0\{I\}}$$

Utilizamos la regla de la asignación y la composición secuencial: $pmd(n = N; x = 0, I) \Leftrightarrow$

$\Leftrightarrow (I_x^0)_n^N$
$\Leftrightarrow 0 \leq N \leq N \wedge \underbrace{(0 = \Sigma i : N \leq i < N : V[i])}_{Cierto}$
(ya que el rango del sumatorio es vacío, por lo que es igual a 0)
$\Leftarrow P$

2. El invariante se mantiene en cada iteración de bucle:

$$\boxed{\{I \wedge B\}A\{I\} \leftrightsquigarrow \{I \wedge n \neq 0\}x = x + V[n-1]; n = n-1\{I\}}$$

$pmd(x = x + V[n-1]; n = n-1, I) \Leftrightarrow$

$\Leftrightarrow (I_n^{n-1})_x^{x+V[n-1]}$
$\Leftrightarrow 0 \leq n-1 \leq N \wedge x + V[n-1] = (\Sigma i : n-1 \leq i < N : V[i])$
$\Leftrightarrow 0 \leq n-1 \leq N \wedge x + V[n-1] = V[n-1] + (\Sigma i : n \leq i < N : V[i])$
$\Leftarrow I \wedge n \neq 0$

3. Al salir del bucle se cumple la postcondición:

$$\boxed{\{I \wedge \neg B\} \Rightarrow \{Q\}}$$

$I \wedge \neg(n \neq 0) \Leftrightarrow (0 \leq n \leq N \wedge x = (\Sigma i : n \leq i < N : V[i])) \wedge n = 0$
$\Rightarrow x = (\Sigma i : 0 \leq i < N : V[i]) \equiv Q$

4. Consideramos la siguiente función cota: $cota(x, n) = n$. Es trivial ver que es positiva:

$$I \wedge n \neq 0 \Rightarrow n \geq 0$$

5. Cuando se ejecuta el bucle y decrece con cada iteración:

$$\boxed{\{I \wedge n \neq 0 \wedge n = T\}x = x + V[n-1]; n = n-1\{n < T\}}$$

$pmd(x = x + V[n-1]; n = n-1, n < T) \Leftrightarrow$

$\Leftrightarrow ((n < T)_n^{n-1})_x^{x+V[n-1]}$
$\Leftrightarrow n - 1 < T$
$\Leftarrow I \wedge n \neq 0 \wedge n = T$

Veamos otro ejemplo. Demostrar la corrección de la siguiente especificación suponiendo x, y y n: enteros.
$\{n \geq 0\}$

```
1    int x, y;
2    x = 0;
3    y = 1;
4    while (x != n)
5    {
6        x = x+1;
7        y = y+y;
8    }
```

$\{y = 2^n\}$

Solución:

- x representa el número de iteraciones del bucle.
- En cada iteración y se duplica.
- Por lo tanto, consideramos el siguiente invariante: $I \equiv 0 \leq x \leq n \wedge y = 2^x$

1. El invariante se cumple antes de comenzar el bucle:

$$\boxed{\{n \geq 0\}x = 0; y = 1\{I\}}$$

$pmd(x = 0; y = 1, I) \Leftrightarrow$

$\Leftrightarrow 0 \leq 0 \leq n \wedge 1 = 2^0$
$\Leftarrow n \geq 0$

2. El invariante se cumple dentro del bucle:

$$\boxed{\{I \wedge x \neq n\}x = x + 1; y = y + y\{I\}}$$

$pmd(x = x + 1; y = y + y, I) \Leftrightarrow$

$\Leftrightarrow 0 \leq x + 1 \leq n \wedge y + y = 2^{x+1}$
$\Leftrightarrow 0 \leq x + 1 \leq n \wedge 2y = 2 \cdot 2^x$
$\Leftarrow I \wedge x \neq n$

3. Al salir del bucle se cumple la postcondición:

$$I \wedge x = n \Rightarrow y = 2^n$$

4. Como función cota tomamos n-x (número de iteraciones que quedan). Mientras se ejecuta el bucle es positiva:

$$\boxed{I \wedge x \neq n \Rightarrow n - x \geq 0}$$

5. n-x decrece en cada iteración:

$$\boxed{\{I \wedge x \neq n \wedge n - x = T\}x = x + 1; y = y + y\{n - x < T\}}$$

$pmd(x = x + 1; y = y + y, n - x < T) \Leftrightarrow$

$\Leftrightarrow n - (x + 1) < T$
$\Leftrightarrow (n - x) - 1 < T$
$\Leftarrow n - x = T$

4. Derivación

- **Derivar**: construir las instrucciones a partir de la especificación asegurando su corrección.
- La postcondición dirige el proceso de verificación.
- Las igualdades de la postcondición se intentan satisfacer mediante las asignaciones correspondientes:
  - Si la precondición es más fuerte que el predicado más débil de estas asignaciones y la postcondición → Proceso finalizado.
  - En caso contrario utilizaremos instrucciones iterativas. Intentaremos ceñirnos al siguiente esquema:

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

    donde:
    - A_0 es la instrucción que el invariante se cumpla inicialmente.
    - A_1 mantiene el invariante a cierto.
    - A_2 hace que la cota decrezca.

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:

$$\{I \wedge B \wedge C = T\}A_1, A_2\{C < T\}$$

4.1. Ejemplos

Deriva un algoritmo que verifique la siguiente especificación:

$\{a \geq 0 \wedge b > 0\}$
proc divide(**int** a,b; **out int** c,r)
$\{a = b * c + r \wedge 0 \leq r < b\}$

Solución:

1. La postcondición tiene forma conjuntiva:
   - $I \equiv a = b * c + r \wedge 0 \leq r$
   - Condición del bucle: $r \geq b$
2. Tenemos que asignar valores a c y r para que el invariante se cumpla:

$$\{P\}c = 0, r = a\{I\}$$

   $\Leftrightarrow ((a = b * c + r \wedge 0 \leq r)_c^0)_r^a \Leftrightarrow a = b * 0 + a \wedge 0 \leq a \Leftarrow a \geq 0$
3. Probemos cota=r: $I \wedge B \Rightarrow r \geq 0$ (ya que cota=c sería creciente pues c vale inicialmente 0).
4. Consideremos restar b a r: $R \equiv I_r^{r-b} \Leftrightarrow$

   $\Leftrightarrow a = b * c + (r - b) \wedge 0 \leq (r - b)$
   $\Leftrightarrow a = b * (c - 1) + r \wedge 0 \leq (r - b)$
   $\Leftarrow^?$ $I \wedge B$ Falso, el invariante no se mantiene!!!
5. Consideremos, además, incrementar c en 1:
   $\{I \wedge B\}c = c + 1\{R\} : R_c^{c+1} \Leftrightarrow a = b * c + r \wedge 0 \leq r - b \Leftarrow I \wedge B$
6. Veamos que la cota, r, decrece en cada vuelta del bucle:

$$\{I \wedge B \wedge r = T\}c = c + 1, r = r - b\{r < T\}$$

   $((r < T)_r^{r-b})_c^{c+1} \Leftrightarrow r - b < T \Leftarrow I \wedge B \wedge r = T$, que es cierto si usamos la propiedad $b > 0$ de la precondición.

El algoritmo resultante:

```
1    proc divide(int a,b; out int c, r){
2    c = 0;
3    r = a;
4    while (r >= b)
5    {
6        c = c+1;
7        r = r-b;
8    }
9    }
```

Veamos otro ejemplo.
Deriva un algoritmo que verifique la siguiente especificación:

$\{n \geq 0\}$
fun raiz-entera(**int** n) **return int** r
$\{r \geq 0 \wedge r^2 \leq n < (r+1)^2\}$

Solución A:

1. La postcondición tiene forma conjuntiva:
   - $I \equiv r \geq 0 \wedge r^2 \leq n$
   - Condición del bucle: $n \geq (r+1)^2$
2. Para hacer cierto el invariante r puede valer 0:
$$n \geq 0 \Rightarrow I_r^0$$
3. Consideremos la instrucción avanzar: $r = r + 1$:
$$I_r^{r+1} \Leftrightarrow r + 1 \geq 0 \wedge (r+1)^2 \leq n \Leftarrow I \wedge B = r \geq 0 \wedge r^2 \leq n \wedge n \geq (r+1)^2$$
4. Algo que decrezca y sea fácil de calcular puede ser: $n - r$ Luego, no es preciso que haya más instrucciones en el cuerpo del bucle.
5. El algoritmo resultante:

```
1    int raiz-entera(int n){
2    r = 0;
3    while (n >= (r+1)*(r+1))
4    {
5        r = r+1;
6    }
7    return r;
8    }
```

6. Número de iteraciones del bucle: $\sqrt{n} \Rightarrow \Theta(\sqrt{n})$.

Solución B:

1. La postcondición tiene forma conjuntiva:
   - $I \equiv r \geq 0 \wedge n < (r+1)^2$
   - Condición del bucle: $r^2 > n$
2. Para hacer cierto el invariante r puede valer n.
$$n \geq 0 \Rightarrow I_r^n$$
3. Consideremos la instrucción avanzar: $r = r - 1$:
$$I_r^{r-1} \Leftrightarrow r - 1 \geq 0 \wedge n < ((r-1)+1)^2 \Leftarrow^? I \wedge r^2 > n = r \geq 0 \wedge n < (r+1)^2 \wedge r^2 > n$$
$$\underbrace{r \geq 0}_{Invariante} \wedge \underbrace{n \geq 0}_{Precondicion} \wedge \underbrace{r^2 > n}_{CondBucle} \Rightarrow r > 0 \Rightarrow r - 1 \geq 0$$
4. Cota=r

5. El algoritmo resultante:

```
1     int raiz-entera(int n){
2     r = n;
3     while (r*r > n)
4     {
5         r = r-1;
6     }
7     return r;
8     }
```

6. Número de iteraciones del bucle: $n - \sqrt{n} \Rightarrow \Theta(n)$.

Notas bibliográficas

Se recomienda ampliar el contenido de estas notas estudiando los capítulos 1 y 4 de (Peña, 2005).

Para coger agilidad en la aplicación de estas técnicas se recomiendan los capítulos 2, 3 y 4 de Martí Oliet et al. (2012) en los que se pueden encontrar numerosos ejemplos resueltos.

Ejercicios

1. Dado el siguiente algoritmo, determina su complejidad.

```
1     int valor(int n){
2     int i = 1, j, aux, r = 3;
3     while (i <= n){
4         j = 1;
5         while (j <= 20){
6             aux = j * 2;
7             aux = aux*i+r;
8             j++;
9         }
10        i++;
11    }
12    return aux;
13    }
```

2. Dado el siguiente algoritmo, determina su complejidad.

```
1     int valor(int n){
2     int i = 1, j, aux;
3     while (i <= n){
4         j = 1;
5         while (j <= n){
6             if (i < n){
7                 aux = i + j;
8             }
9             else
10            {
11                aux = i - j;
12            }
13            j++;
```

```
14          }
15          i++;
16      }
17      return aux;
18      }
```

3. Dado el siguiente algoritmo, determina su complejidad.

```
1       int valor(int n){
2       int i = 1, j, aux;
3       while (i <= n){
4           j = 1;
5           while (j <= i){
6               aux = i + j;
7               if (i + 2 < j){
8                   aux = aux * 2;
9               }
10              j++;
11          }
12          i++;
13      }
14      return aux;
15      }
```

4. Suponiendo que *int x,y,n* y *bool b*, determina el predicado más débil, P, que satisfaga la especificación dada

   a) $\{P\}x = 3 * x\{x \leq 20\}$

   b) $\{P\}x = 3 * x\{\forall i : 1 \leq i \leq n : x \neq 6 * i\}$

   c) $\{P\}x = x + 1\{x = x + 1\}$

5. Suponiendo x, y enteros, calcula en cada caso el predicado P más débil que satisfaga la especificación dada:

   a) $\{P\}x = x + y; y = 2 * y - x\{y > 0\}$

   b) $\{P\}y = 4 * x; x = x * x - y; x = x + 4\{x > 0\}$

   c) $\{P\}x = y; y = x\{x = A \wedge y = B\}$

6. Suponiendo x un número entero, determina el predicado más débil, P, que safisfaga la especificación dada:

```
{P}
if (x % 2==0) x = x / 2;
else x = x / 2 + 1;
{x=X}
```

   donde X es un valor "constante".

7. Verifica el siguiente algoritmo:
   $\{N \geq 0\}$

```
1       int suma(int V[], int N){
2       int m,s;
3       s = 0;
```

```
4       m = 0;
5       while (m < N)
6       {
7           s = s+V[m];
8           m = m+1;
9       }
10      return s;
11      }
```

$\{s = (\Sigma i : 0 \leq i < N : V[i])\}$

8. Suponiendo todas las variables enteras, demostrar la corrección de los tres programas siguientes para calcular potencias, con:

   - Precondición: $P \equiv x = X \wedge y = Y \wedge Y \geq 0$.
   - Postcondición $Q \equiv p = X^Y$.

a)

```
1       p = 1;
2       while (y != 0)
3       {
4           p = p*x;
5           y = y-1;
6       }
```

b)

```
1       p = 1;
2       while (y > 0)
3       {
4           if (par(y))
5           {
6               y = y / 2;
7               x = x*x;
8           ]
9           else
10          {
11              y = y-1;
12              p = p*x;
13          }
14      }
```

c)

```
1       p = 1;
2       while (y > 0)
3          {
4           if (impar(y))
5               p = p*x;
6           y = y / 2;
7           x = x*x;
8          }
```

9. Demostrar la corrección de la siguiente especificación suponiendo x, y enteros.
   $\{x = X \wedge y = Y \wedge x > y > 0\}$

```
1     while (y != 0)
2     {
3         z = x;
4         x = y;
5         y = z % x;
6     }
```

$\{x = mcd(X, Y)\}$

10. Demuestra la corrección del siguiente programa:

    $\{x \geq 0\ \}$

```
1     r = 0;
2     while ((r+1)*(r+1) <= x  )
3     {
4         r = r+1;
5     }
```

$\{r \geq 0 \wedge r^2 \leq x < (r+1)^2\ \}$

11. Deriva un algoritmo que satisfaga la siguiente especificación:

    $\{b > 1 \wedge n > 0\}$
    fun log(**int** b, **int** n) **return int** r
    $\{b^r \leq n < b^{r+1}\}$

12. Especifica y deriva una función iterativa de coste lineal que reciba como argumento un vector de números enteros y calcule el producto de todos ellos.

13. Especifica y deriva una función que reciba como argumento un vector de números y calcule el máximo de todos ellos.

14. Deriva un algoritmo que satisfaga la siguiente especificación:

    $\{N > 0\}$
    fun pares(**int** V[N]) **return int** x
    $\{x = (\sharp i : 0 \leq i < N : V[i] mod\ 2 = 0)\}$

15. Deriva un algoritmo que satisfaga la siguiente especificación:

    $\{N > 0\}$
    fun suma-buenos(**int** X[N]) **return int** s
    $\{s = (\Sigma i : 0 \leq i < N \wedge bueno(i, X) : X[i])\}$

    donde $bueno(i, X) \equiv (X[i] = 2^i)$. No se pueden emplear operaciones que calculen potencias.

16. **Definición**: Dado V[N] de enteros, el índice i es un pico si V[i] es el mayor elemento de V[0..i].
    Especifica y deriva un algoritmo de coste lineal que reciba un vector y calcule la suma de todos los valores almacenados en los picos de V.

17. Dado un vector de enteros X[N], el índice se llama "heroico" si X[i] es estrictamente mayor que el i-ésimo número de la sucesión de Fibonacci. Especifica y deriva un algoritmo de coste lineal que reciba como argumento un vector y determine cuántos índices heroicos tiene.

18. **Definición**: Dado X[N] de enteros, el índice i es un mirador si X[i] es el mayor elemento de X[i+1..n).
Especifica y deriva un algoritmo de coste lineal que reciba un vector y calcule el número de miradores que hay en X.

19. **Definición**: Un vector A[N] de enteros es *gaspariforme* si todas sus sumas parciales (A[0]+...+A[i], $0 \leq i < n$) son no negativas y además la suma total es cero.
Especifica y deriva una función de coste lineal que decida si A[N] es gaspariforme.

20. (Martí Oliet et al. (2012)) Derivar un algoritmo de coste lineal (con respecto a la longitud del vector) que satisfaga la siguiente especificación:

    $\{ N \geq 2 \}$
    fun máx-resta(**int** A[N]) **return int** r
    $\{ r = (\text{máx}\, p, q : 0 \leq p < q < N : A[p] - A[q]) \}$

21. (Martí Oliet et al. (2012)) Derivar un algoritmo de coste lineal (con respecto a la longitud del vector) que satisfaga la siguiente especificación:

    $\{ N \geq 0 \}$
    fun crédito-seg-máx(**int** A[N]) **return int** r
    $\{ r = (\text{máx}\, p, q : 0 \leq p \leq q \leq N : \mathit{crédito}(p, q)) \}$

    donde $\mathit{crédito}(p, q) = (\#i : p \leq i < q : A[i] > 0) - (\#i : p \leq i < q : A[i] < 0)$.

22. (Martí Oliet et al. (2012))

    Derivar un algoritmo que satisfaga la siguiente especificación:

    $\{ N \geq 0 \}$
    fun buscar-cero(**int** A[N]) **return int** r
    $\{ r = (\text{máx}\, i : 0 \leq i \leq N \wedge (\forall j : 0 \leq j < i : A[j] \neq 0) : i) \}$

23. (Martí Oliet et al. (2012))

    Derivar algoritmos de coste lineal (con respecto a la longitud del vector) para resolver los siguientes problemas de segmento de longitud máxima:

    $\{ N \geq 0 \}$
    fun seg-long-máx(**int** X[N]) **return int** r
    $\{ r = (\text{máx}\, p, q : 0 \leq p \leq q \leq N \wedge \mathcal{A}(p, q) : q - p) \}$

    donde

    a) $\mathcal{A}(p, q) = (\forall i : p \leq i < q : X[i] = 0)$.
    b) $\mathcal{A}(p, q) = (\forall i, j : p \leq i \leq j < q : X[i] = X[j])$.

Bibliografía

Y así, del mucho leer y del poco dormir, se le secó el celebro de manera que vino a perder el juicio.

Miguel de Cervantes Saavedra

Brassard, G. y Bratley, P. *Fundamentos de Algoritmia*. Prentice Hall, 1997.

Cormen, T. H., Leiserson, C. E., Rivest, R. L. y Stein, C. *Introduction to Algorithms*. MIT Press, 2nd edición, 2001.

Martí Oliet, N., Segura Díaz, C. M. y Verdejo López, J. A. *Algoritmos correctos y eficientes: Diseño razonado ilustrado con ejercicios*. Ibergarceta Publicaciones, 2012.

Martí Oliet, N., Ortega Mallén, Y. y Verdejo López, J. A. *Estructuras y datos y métodos algorítmicos: 213 Ejercicios resueltos*. Ibergarceta Publicaciones, 2013.

Peña, R. *Diseño de Programas: Formalismo y Abstracción*. Tercera edición. Pearson Prentice-Hall, 2005.

Rodriguez Artalejo, M., González Calero, P. A. y Gómez Martín, M. A. *Estructuras de datos: un enfoque moderno*. Editorial Complutense, 2011.

Stroustrup, B. *The C++ Programming Language, 3rd Edition*. Addison-Wesley, 1998. ISBN 0201889544.