Programación 2

Lección 14. Diseño de algoritmos iterativos correctos

1. Derivación del código de un algoritmo
2. Diseño de algoritmos correctos con bucles
3. Diseño de bucles mediante debilitamiento de su postcondición
4. Diseño de algoritmos iterativos más complejos

1. Derivación del código de un algoritmo

Diseñar el código que falta:

```
// x = A ∧ y = B

???

// x = B ∧ y = A
```

Hagamos un intento de diseño. Para empezar vamos a modificar el valor de una de las variables, por ejemplo, modificamos el valor de la variable **x**.

```
// x = A ∧ y = B
x = x + y;
// x = A + B ∧ y = B
???
// x = B ∧ y = A
```

1º

¿Cómo debiéramos continuar el diseño? ¿Cómo debiéramos modificar ahora el valor de la variable **y** para aproximarnos a nuestro objetivo?

Debiéramos modificar el valor de la variable **y** para que su valor sea igual a **A**, es decir, el valor que exige la postcondición:

```
// x = A ∧ y = B
x = x + y;
// x = A + B ∧ y = B
y = x - y;
// x = A + B ∧ y = A
???
// x = B ∧ y = A
```

1º

2º

Ahora sólo falta ajustar el valor de la variable **x**.

Basta con modificar el valor de la variable **x** para que valga **B**, el valor exigido por la postcondición, sin alterar el valor de **y**:

```
// x = A ∧ y = B
x = x + y;
// x = A + B ∧ y = B
y = x - y;
// x = A + B ∧ y = A
x = x - y;
// x = B ∧ y = A
???
// x = B ∧ y = A
```

1º

2º

3º

¿ Es correcto el diseño realizado ?

¡ El diseño está concluido y es correcto (regla del debilitamiento de la postcondición) !

```
// x = A ∧ y = B
x = x + y;
// x = A + B ∧ y = B
y = x - y;
// x = A + B ∧ y = A
x = x - y;
// x = B ∧ y = A  ⇒ x = B ∧ y = A

// x = B ∧ y = A
```

He aquí el algoritmo deducido por **derivación** del código a partir de su especificación:

```
// x = A ∧ y = B
x = x + y;
y = x - y;
x = x - y;
// x = B ∧ y = A
```

Hagamos un segundo diseño, modificando en primer lugar el valor de la variable **y**:

```
// x = A ∧ y = B
y = x - y;
// x = A ∧ y = A - B
???
// x = B ∧ y = A
```

1º

Hagamos que el valor de la variable **x** sea igual a **B**, el exigido por la postcondición, sin modificar el valor de la variable **y**:

```
// x = A ∧ y = B
y = x - y;
// x = A ∧ y = A - B          1º
x = x - y;
// x = B ∧ y = A - B          2º
???
// x = B ∧ y = A
```

Sólo falta modificar el valor de la variable y para que valga **A**, lo exigido por la postcondición, sin alterar el valor de **x**:

```
// x = A ∧ y = B
y = x - y;
// x = A ∧ y = A - B                    1º
x = x - y;
// x = B ∧ y = A - B                    2º
y = x + y;
// x = B ∧ y = B + A - B                3º
// ⇒
// x = B ∧ y = A

// x = B ∧ y = A
```

¡Hemos sabido **derivar** un segundo diseño del algoritmo!

2. Diseño de algoritmos correctos con bucles

Debemos diseñar el siguiente algoritmo con un bucle:

```
// P

???

// Q
```

Escribimos el bucle y ponemos interrogantes donde haya que incluir código para completar el diseño:

```
// P
???
while (???) {
    ???
}
???
// Q
```

Lo primero que debemos hacer es dar con un **método iterativo** que resuelva el problema y **resumirlo en un predicado invariante**. Una alternativa, que puede ser muy interesante, es la de **escribir primero el invariante y deducir de él un método iterativo**.

Anotamos en el bucle el predicado invariante **I**:

```
// P
???
while (???) {   // I                                    (1)
   ???
}
???
// Q
```

Paso 1. El predicado invariante **I** puede deducirse, en muchos casos, aplicando técnicas de debilitamiento de la postcondición **Q**.

Paso 2. Diseñamos el código previo al bucle:

```
// P
Acción_previa          } diseñar este código          (2)
// I
while (???) {   // I                                  (1)
   ???
}
???
// Q
```

Pasos 3 y 4. Deducimos la condición [C] del bucle y el código posterior al bucle:

```
// P
Acción_previa                                          (2)
// I
while (C) {   // I                                  (1),(3)
   ???
}
// ¬C ∧ I
Acción_posterior      } diseñar este código            (4)
// Q
```

Pasos 5 y 6. Diseñamos el código interior del bucle, garantizando su terminación mediante una función de cota:

```
// P
Acción_previa                                              (2)
// I
while (C) {   // I     // f_cota                  (1),(3),(6)
   // I ∧ C
   Acción_a_iterar     } diseñar este código               (5)
   // I
}
// I ∧ ¬C
Acción_posterior                                           (4)
// Q
```

3. Diseño de bucles mediante debilitamiento de su postcondición

Derivar un diseño iterativo del siguiente algoritmo:

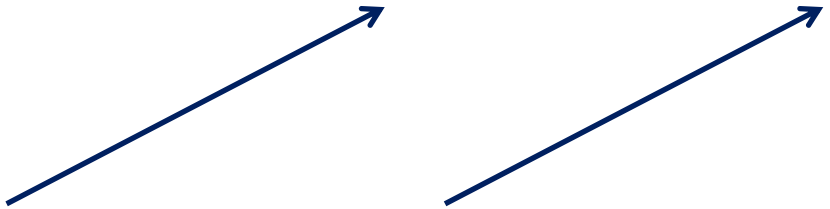
```
// n > 0  AND  n <= #v

???

// sumar(v,n) = (Σα∈[0,n-1]. v[α])
```

El esquema algorítmico iterativo a diseñar es el siguiente:

```
// n > 0  AND  n <= #v
???
while (???) {   //  ¿ I ?
   ???
}
???
// sumar(v,n) = (Σα∈[0,n-1]. v[α])
```

Paso 1. Debilitamos la postcondición sustituyendo la constante `n-1` por una variable, por ejemplo, `hasta`, y el resultado de la invocación `sumar(v,n)` por una variable, por ejemplo `suma`:

```
// n > 0  AND  n <= #v
???
while (???) {    // Inv: suma = (Σα∈[0,hasta]. v[α])
   ???
}
???
// sumar(v,n) = (Σα∈[0,n-1]. v[α])
```



Paso 2. Procedemos a diseñar las acciones previas al bucle:

```
// n > 0  AND  n <= #v
???
// suma = (Σα∈[0,hasta]. v[α])
while (???) {   // Inv: suma = (Σα∈[0,hasta]. v[α])
   ???
}
???
// sumar(v,n) = (Σα∈[0,n-1]. v[α])
```

Estas son las acciones previas al bucle que hemos diseñado:

```
// n > 0  AND  n <= #v
int hasta = 0;
int suma = v[0];
// suma = (Σα∈[0,hasta]. v[α])
while (???) {   // Inv: suma = (Σα∈[0,hasta]. v[α])
   ???
}
???
// sumar(v,n) = (Σα∈[0,n-1]. v[α])
```

Pasos 3 y 4. Deducimos ahora la condición del bucle y el código posterior al bucle:

```
// n > 0  AND  n <= #v
int hasta = 0;
int suma = v[0];
// suma = (Σα∈[0,hasta]. v[α])
while (hasta != n-1) {  // suma = (Σα∈[0,hasta]. v[α])
   ???
}
// suma = (Σα∈[0,hasta]. v[α]) ∧ hasta = n - 1
return suma;
// sumar(v,n) = (Σα∈[0,n-1]. v[α])
```

Pasos 5 y 6. Deducimos el código interior al bucle e identificamos una función de cota que asegure su terminación:

```
// n > 0  AND  n <= #v
int hasta = 0;
int suma = v[0];
// suma = (Σα∈[0,hasta]. v[α])
while (hasta != n - 1) {  // f_cota = n – hasta
   // suma = (Σα∈[0,hasta]. v[α]) ∧ hasta ≠ n - 1
   hasta = hasta + 1;
   suma = suma + v[hasta];
   // suma = (Σα∈[0,hasta]. v[α])
}
// suma = (Σα∈[0,hasta]. v[α]) ∧ hasta = n - 1
return suma;
// sumar(v,n) = (Σα∈[0,n-1]. v[α])
```

El algoritmo iterativo diseñado es el siguiente:

```
// n > 0
int hasta = 0;
int suma = v[0];
while (hasta != n - 1) {
   // suma = (Σα∈[0,hasta]. v[α])
   hasta = hasta + 1;
   suma = suma + v[hasta];
}
return suma;
// sumar(v,n) = (Σα∈[0,n-1]. v[α])
```

Conviene documentar los bucles con el predicado invariante que nos ha abierto las puertas de su diseño ya que constituye un excelente elemento de documentación del código.

4. Diseño de algoritmos iterativos más complejos

Diseñar el siguiente algoritmo sin utilizar la operación de potenciación.

```
// n ≥ 0

???

// elevar(x,n) = x^n
```

¿ Cómo calcularías: 176.45^{103} utilizando únicamente lápiz y papel ?

Diseñar el siguiente algoritmo sin utilizar la operación de potenciación.

```
// n ≥ 0
???
// elevar(x,n) = x^n
```

¿ Cómo calcularías 176.45^{103} utilizando únicamente lápiz y papel ?

¿ Así lo harías: $176.45^{103} = 176.45 \times 176.45 \times 176.45 \times ... \times 176.45$?

¡¡ Has tenido que hacer 102 multiplicaciones !!

¿ Te animas a calcular a mano el resultado ? ¿ Es el método más eficiente ?

¿ Cómo calcularías: 176.45^{103} utilizando únicamente lápiz y papel ?

¿ Así: $176.45^{103} = 176.45 \times 176.45 \times 176.45 \times \ldots \times 176.45$?

¡¡¡ Tienes que hacer 102 multiplicaciones !!!

¿ Te animas a calcular a mano el resultado ? ¿ Es el método más eficiente ?

Puedes agrupar operaciones (¡y te ahorras 50 multiplicaciones!):

$$176.45^{103} = [\underbrace{176.45\times\ldots\times176.45}_{\text{50 multiplicaciones}}]\times176.45\times[\underbrace{176.45\times\ldots\times176.45}_{\text{50 multiplicaciones}}]$$

$$= 176.45^{51} \times 176.45 \times 176.45^{51}$$

Puedes agrupar más operaciones (¡y te ahorras otras 25 multiplicaciones!):

$$176.45^{51} = [\underbrace{176.45\times\ldots\times176.45}_{\text{24 multiplicaciones}}]\times176.45\times[\underbrace{176.45\times\ldots\times176.45}_{\text{24 multiplicaciones}}]$$

$$= 176.45^{25} \times 176.45 \times 176.45^{25}$$

Puedes seguir agrupando operaciones (¡y seguirás ahorrándote muchas multiplicaciones!): ...

Propiedades que definen un método de cálculo de x^n que permiten diseñar un algoritmo de cálculo de coste logarítmico en [n]:

- $n = 0 \rightarrow x^n = 1.0$
- $n > 0 \wedge n \,\%\, 2 = 0 \rightarrow x^n = [x^{n/2}]^2$
- $n > 0 \wedge n \,\%\, 2 \neq 0 \rightarrow x^n = x.[x^{n/2}]^2$

$176.45^{103} = 176.45 \times [176.45^{51}]^2$

$176.45^{51} = 176.45 \times [176.45^{25}]^2$

$176.45^{25} = 176.45 \times [176.45^{12}]^2$

$176.45^{12} = [176.45^{6}]^2$

$176.45^{6} = [176.45^{3}]^2$

$176.45^{3} = 176.45 \times [176.45^{1}]^2$

$176.45^{1} = 176.45 \times [176.45^{0}]^2$

$176.45^{0} = 1.0$

¡ Hay que elevar 6 veces una cantidad al cuadrado y hay que multiplicar 4 veces un par de números, es decir, sólo hay que hacer 10 multiplicaciones !

$$176.45^{103} = 176.45 \times [176.45^{51}]^2 = x.(x.x^2.x^{16}.x^{32})^2$$
$$= x.x^2.x^4.x^{32}.x^{64} = x^{103}$$

$$176.45^{51} = 176.45 \times [176.45^{25}]^2 = x.(x.x^8.x^{16})^2 = x.x^2.x^{16}.x^{32}$$
$$= x^{51}$$

$$176.45^{25} = 176.45 \times [176.45^{12}]^2 = x.(x^4.x^8)^2 = x.x^8.x^{16} = x^{25}$$

$$176.45^{12} = [176.45^6]^2 = (x^2.x^4)^2 = x^4.x^8 = x^{12}$$

$$176.45^6 = [176.45^3]^2 = (x.x^2)^2 = x^2.x^4 = x^6$$

$$176.45^3 = 176.45 \times [176.45^1]^2 = x.x^2 = x^3$$

$$176.45^1 = 176.45 \times [176.45^0]^2 = 176.45 \times 1.0 = x = x^1$$

$$176.45^0 = 1.0 = x^0$$

Se han calculado las seis potencias del número x = 176.45, es decir, $\mathbf{x^2}$, $\mathbf{x^4}$, $\mathbf{x^8}$, $\mathbf{x^{16}}$, $\mathbf{x^{32}}$ y $\mathbf{x^{64}}$ (6 multiplicaciones) y se ha multiplicado cinco de ellas (4 multiplicaciones más): $\mathbf{x^{103} = x^1.x^2.x^4.x^{32}.x^{64}}$

Por lo tanto, se han efectuado únicamente 10 multiplicaciones en total

El esquema algorítmico a diseñar es el siguiente:

```
// n ≥ 0
???
while (???) {   //  ¿ I ?
   ???
}
???
// elevar(x,n) = x^n
```

x	invariante	nVar	resultado	potencias	e
x	$x^{27} = 1.0\ [x^{27}]^1$	**27**	1.0	$\mathbf{x^1}$	1
x	$x^{27} = x\ [x^{13}]^2$	**13**	x	$\mathbf{x^2}$	2
x	$x^{27} = x^3\ [x^6]^4$	6	x^3	x^4	4
x	$x^{27} = x^3\ [x^3]^8$	**3**	x^3	$\mathbf{x^8}$	8
x	$x^{27} = x^{11}\ [x^1]^{16}$	**1**	x^{11}	$\mathbf{x^{16}}$	16
x	$x^{27} = x^{27}\ [x^0]^{32}$	0	x^{27}	x^{32}	32

Cálculos realizados: $\mathbf{x^{27} = 1.x^1.x^2.x^8.x^{16}}$

Un invariante del bucle que permita resolver el problema no es inmediato. Para construirlo introducimos las variables nVar (valor variable del exponente), `resultado` (almacena el resultado parcial y final), e (exponente potencia de 2) y `potencias` (potencias de 2):

// Invariante: $x^n = resultado.[x^{nVar}]^e \wedge potencias = x^e$

Paso 1. Esquema iterativo del algoritmo a derivar. El invariante del bucle se deduce a partir del método de cálculo a aplicar:

```
// n ≥ 0
???
while (???) {
   // Inv: x^n = resultado.[x^nVar]^e ∧ potencias = x^e
   ???
}
???
// elevar(x,n) = x^n
```

Paso 2. Diseñamos el código previo al bucle:

```
// n ≥ 0
int nVar = n, e = 1;
double resultado = 1.0, potencias = x;
// x^n = resultado.[x^nVar]^e ∧ potencias = x^e
while (???) {
   // Inv: x^n = resultado.[x^nVar]^e ∧ potencias = x^e
   ???
}
???
// elevar(x,n) = x^n
```

Pasos 3 y 4. Deducimos la condición del bucle y el código posterior a él:

```
// n ≥ 0
int nVar = n, e = 1;
double resultado = 1.0, potencias = x;
while (nVar != 0) {
   // Inv: x^n = resultado.[x^nVar]^e ∧ potencias = x^e
   ???
}
// x^n = resultado.[x^nVar]^e ∧ potencias = x^e ∧ nVar = 0
return resultado;
// elevar(x,n) = x^n
```

Pasos 5 y 6. Diseñamos el código interior del bucle y proponemos una función de cota que permite probar su terminación:

```
// n ≥ 0
int nVar = n, e = 1;
double resultado = 1.0, potencias = x;
while (nVar != 0) {      // f_cota = nVar
   // x^n = resultado.[x^nVar]^e ∧ potencias = x^e ∧ nVar ≠ 0
   if (nVar % 2 != 0) {
     resultado = potencias*resultado;
   }
   nVar = nVar/2; e = 2*e; potencias = potencias*potencias;
   // x^n = resultado.[x^nVar]^e ∧ potencias = x^e
 }
 return resultado;
 // elevar(x,n) = x^n
```

El resultado del diseño iterativo es el siguiente:

```
// n ≥ 0
int nVar = n, e = 1;
double resultado = 1.0, potencias = x;
while (nVar != 0) { // x^n = resultado.[x^nVar]^e ∧ potencias = x^e
   if (nVar % 2 != 0) {
     resultado = potencias*resultado;
   }
   nVar = nVar/2; e = 2*e; potencias = potencias*potencias;
 }
 return resultado;
 // elevar(x,n) = x^n
```

Se ha documentado el bucle con el predicado invariante que resume el método de cálculo aplicado.