Programación 2

Lección 11. Análisis de la corrección de un algoritmo: primera parte

1. **Precondición más débil y corrección de un algoritmo. Ejemplos**
2. **Análisis de la corrección de una secuencia de instrucciones. Ejemplos**
3. **Análisis de la corrección de una instrucción condicional. Ejemplos**
4. **Análisis de la corrección de una instrucción iterativa**
   - **Concepto de predicado invariante de un bucle**
   - **Ejemplos**

1. Precondición más débil y corrección de un algoritmo. Ejemplos

La **precondición más débil** de una acción **A** con postcondición **Q** es el predicado **pmd(A,Q)** más débil que hace correcto el diseño:

```
//  pmd(A,Q)
 A
//  Q
```

Los <u>axiomas</u> de las instrucciones de asignación, devolución de valor y nula indican <u>cómo calcular</u> dicha precondición más débil:

```
//   Dom(v = Exp) ∧ Q_v^Exp
v = Exp;
//   Q
```

$$// \quad \mathrm{Dom}(v = \mathrm{Exp}) \land \mathrm{Q}_{v}^{\mathrm{Exp}}$$

```
//   Dom(Exp) ∧ Q_nombreFunción(lista de parámetros)^Exp
return Exp;
//   Q
```

$$// \quad \mathrm{Dom}(\mathrm{Exp}) \land \mathrm{Q}_{\mathrm{nombreFunción(lista\ de\ parámetros)}}^{\mathrm{Exp}}$$

```
//   Q
;
//   Q
```

Si la acción A es <u>correcta</u>:

```
// Q
A
// R
```

Y se satisface que $P \Rightarrow Q$ entonces es también es <u>correcta</u>:

```
// P        // Precondición reforzada:  P ⇒ Q
A
// R
```

Y se satisface que $R \Rightarrow S$ entonces es también es <u>correcta</u>:

```
// Q
A
// S       // Postcondición debilitada: R ⇒ S
```

2. Análisis de la corrección de una secuencia de instrucciones

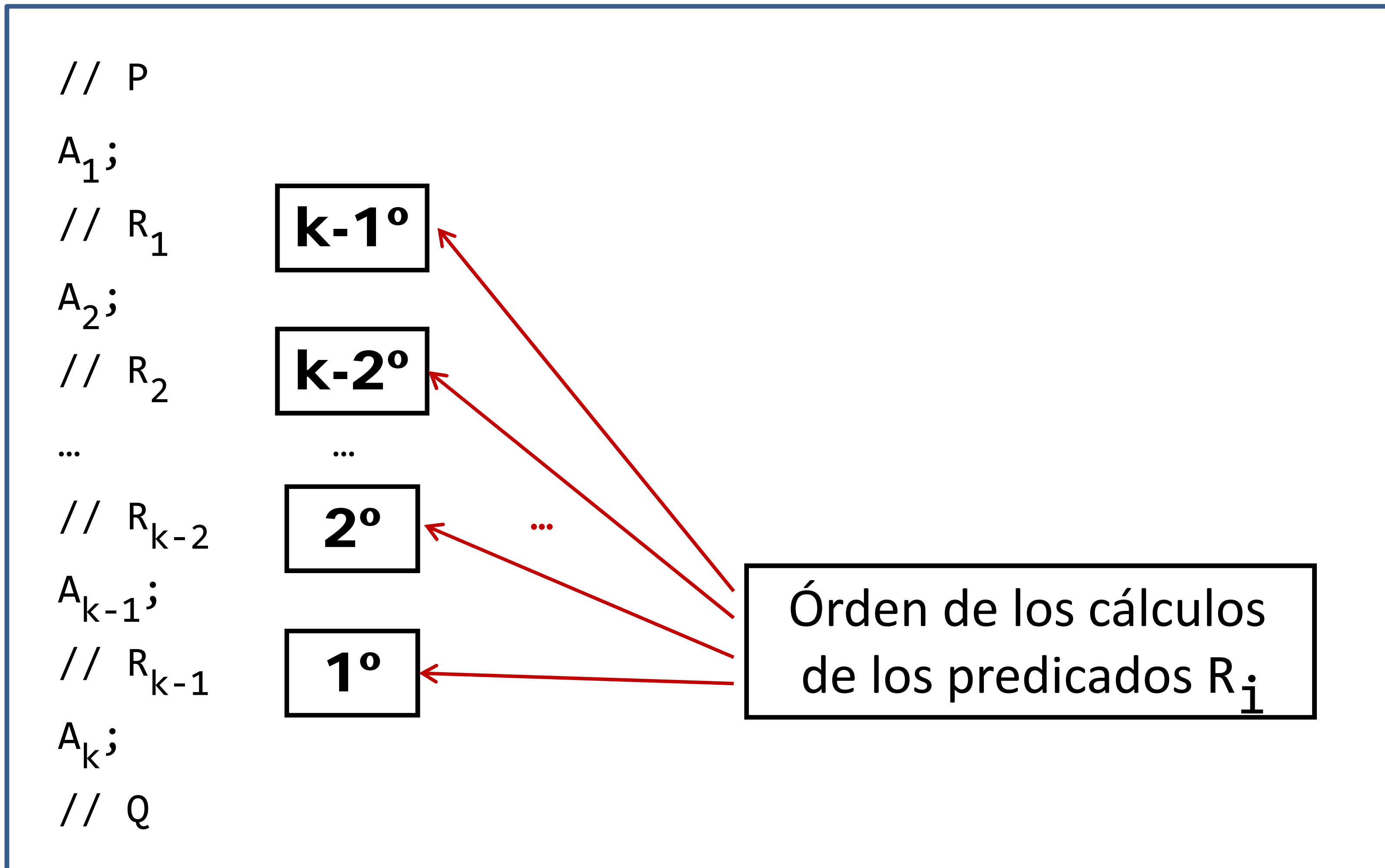
Debemos **probar** que la siguiente secuencia de instrucciones es **correcta**.

```
// x = A ∧ y = B
aux = x;
x = y;
y = aux;
// x = B ∧ y = A
```

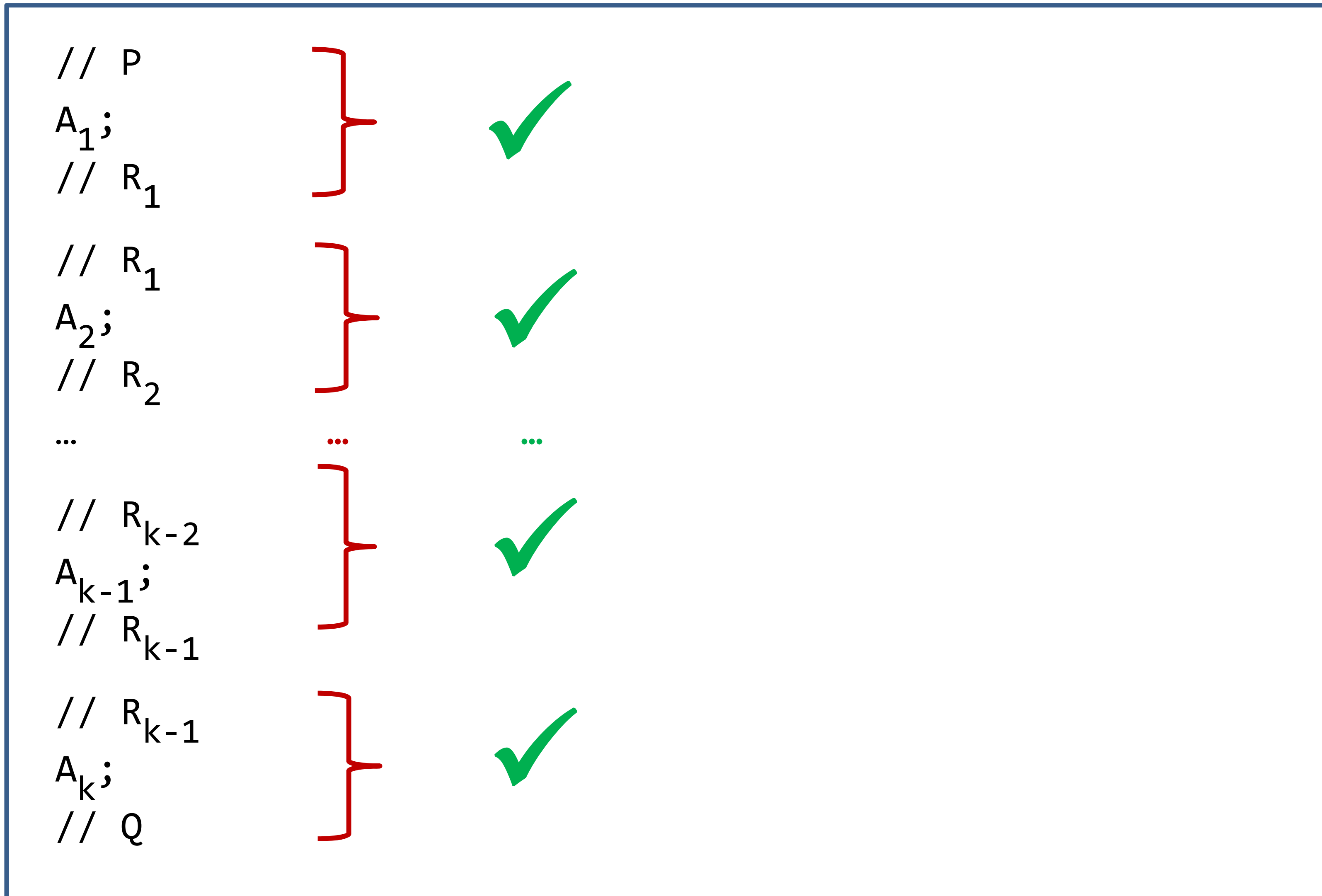
En un caso general: ¿cómo probar la corrección de una secuencia de **k** instrucciones?

```
// P
A1;
A2;
…
Ak-1;
Ak;
// Q
```

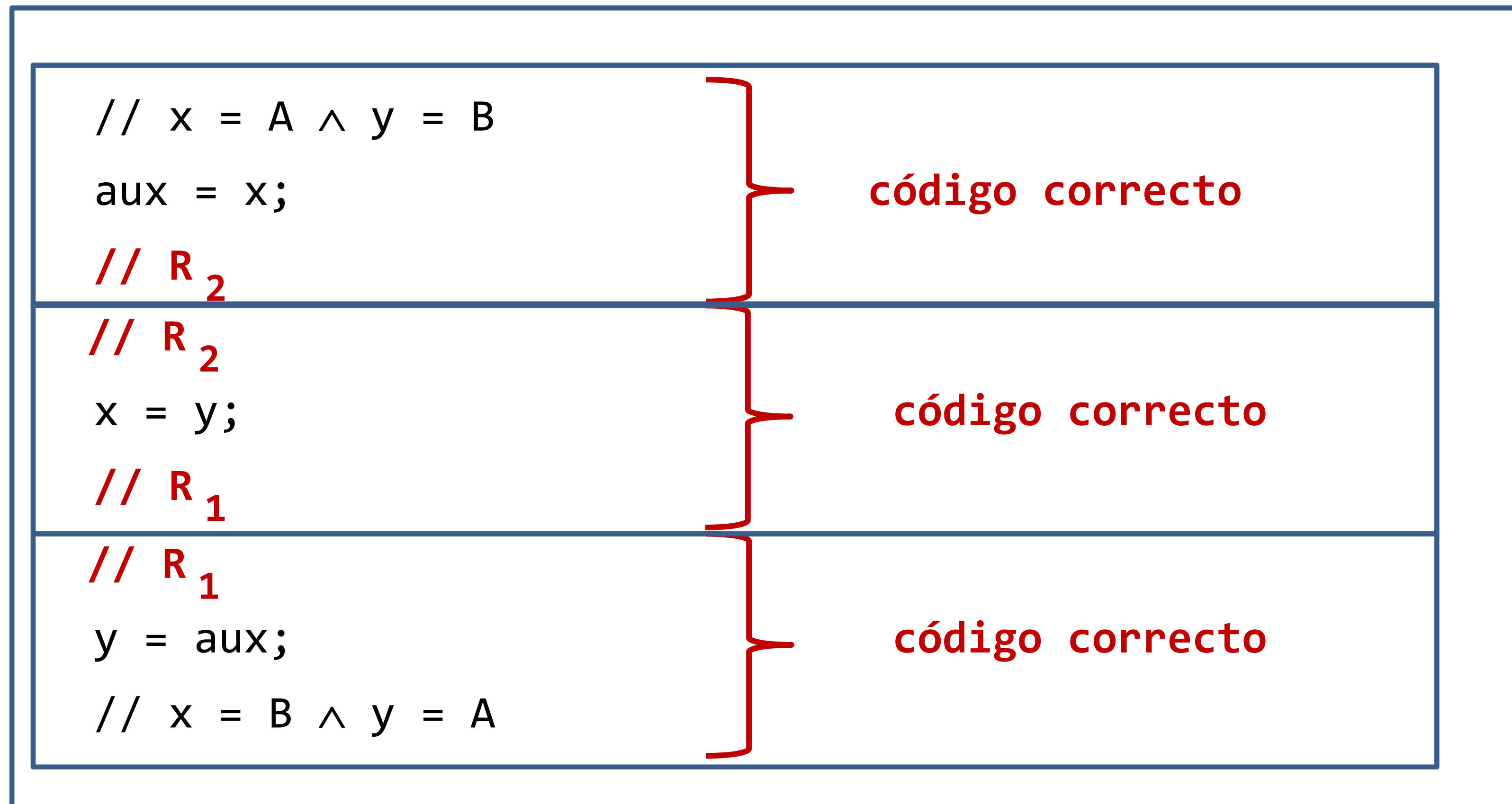
1. Plantear (calculando o, en su caso, conjeturando) predicados intermedios $\mathbf{R_i}$ que debieran satisfacerse en los puntos en los que se definen:



2. Demostrar la corrección de cada una de las **k** instrucciones **A_i** de la secuencia:



En nuestro caso particular, bastará con proponer dos predicados, R_1 y R_2, para los que sean correctas cada una de las tres instrucciones que integran la secuencia.



¿Cómo escribir o, en su caso, calcular los predicados R_1 y R_2?

En nuestro caso particular, podemos calcular los predicados intermedios, comenzando por los últimos. Para ello aplicaremos, paso a paso, los axiomas que aseguran la corrección de cada instrucción (aquí el axioma de la asignación):

```
// x = A ∧ y = B
aux = x;
x = y;
// x = B ∧ aux = A      1º  ┐
y = aux;                    ├ código correcto
// x = B ∧ y = A            ┘
```

Seguimos aplicando los axiomas que aseguran la corrección de cada instrucción:

```
// x = A ∧ y = B
aux = x;
// y = B ∧ aux = A     2º
x = y;
// x = B ∧ aux = A
y = aux;
// x = B ∧ y = A
```

código correcto

Aplicamos por tercera vez el axioma de la asignación para calcular la **precondición más débil** (**pmd**) que hace correcto el algoritmo:

```
// x = A ∧ y = B
// y = B ∧ x = A
aux = x;
// y = B ∧ aux = A
x = y;
// x = B ∧ aux = A
y = aux;
/7 x = B ∧ y = A
```

3º

código correcto

Aplicamos finalmente **la regla del reforzamiento de la precondición** a la ***pmd*** que acabamos de calcular que confirma la corrección del diseño:

```
// x = A ∧ y = B  ⇒  y = B ∧ x = A
// y = B ∧ x = A
aux = x;
// y = B ∧ aux = A
x = y;
// x = B ∧ aux = A
y = aux;
// x = B ∧ y = A
```

código correcto

Resumamos el método de cálculo de predicados intermedios, comenzando por el final, aplicado:

```
// P
A1;
A2;
…
Ak-1;
Ak;
// Q
```

1) Se han calculado, en este orden, pmd_k, pmd_{k-1}, …, pmd_2 y pmd_1

2) Se ha probado que **P** es igual o más fuerte que pmd_1

```
// P
// P  ⇒  pmd1
// pmd1          k-2º
A1;
// pmd2          k-1º
A2;
                 …
 …
// pmdk-1        2º
Ak-1;
// pmdk          1º
Ak;
// Q
```

este código es correcto

1) Se han calculado, en este orden, **pmd_k, pmd_{k-1}**, …, **pmd_2** y **pmd_1**

2) Se ha probado que P no es ni igual ni más fuerte que pmd_1

```
// P
// P  ⇏  pmd₁
// pmd₁          k-2º
A₁;
// pmd₂          k-1º
A₂;
                 …
…
// pmd(k-1)      2º
A(k-1);
// pmdk          1º
Ak;
// Q
```

este código no es correcto ✘

3. Análisis de la corrección de una instrucción condicional. Ejemplos

Debemos **probar** que la siguiente instrucción condicional es **correcta**.

```
// x = A ∧ y = B
if (x <= y) {
   mayor = y;
}
else {
   mayor = x;
}
// mayor = (Máx α∈{A,B}. α)
```

```
// x = A ∧ y = B
if (x <= y) {
  // x = A ∧ y = B ∧ x ≤ y
  mayor = y;
  // mayor = (Máx α∈{A,B}. α)
}
else {
  // x = A ∧ y = B ∧ x > y
  mayor = x;
  // mayor = (Máx α∈{A,B}. α)
}
// mayor = (Máx α∈{A,B}. α)
```

¿está libre de errores?

¿es correcto?

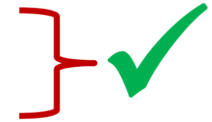
¿es correcto?

No hay riesgo de error al evaluar la condición:

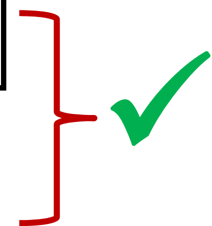
```
// x = A ∧ y = B  ⇒  Dom(x ≤ y) ≡ cierto   1º  } ✓
if (x <= y) {
  // x = A ∧ y = B ∧ x ≤ y
  mayor = y;
  // mayor = (Máx α∈{A,B}. α)
}
else {
  // x = A ∧ y = B ∧ x > y
  mayor = x;
  // mayor = (Máx α∈{A,B}. α)
}
// mayor = (Máx α∈{A,B}. α)
```

El código de la primera alternativa es correcto:

```
// x = A ∧ y = B  ⇒  Dom(x ≤ y) ≡ cierto
if (x <= y) {
  // x = A ∧ y = B ∧ x ≤ y ⇒ y = (Máx α∈{A,B}. α)
  mayor = y;
  // mayor = (Máx α∈{A,B}. α)
}
else {
  // x = A ∧ y = B ∧ x > y
  mayor = x;
  // mayor = (Máx α∈{A,B}. α)
}
// mayor = (Máx α∈{A,B}. α)
```

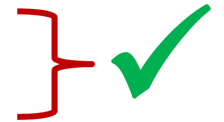


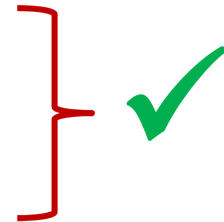
2º



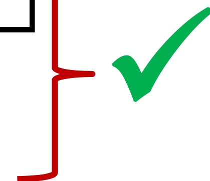
El código de la segunda alternativa también es correcto:

```
// x = A ∧ y = B  ⇒  Dom(x ≤ y) ≡ cierto
if (x <= y) {
  // x = A ∧ y = B ∧ x ≤ y ⇒ y = (Máx α∈{A,B}. α)
  mayor = y;
  // mayor = (Máx α∈{A,B}. α)
}
else {
  // x = A ∧ y = B ∧ x > y ⇒ x = (Máx α∈{A,B}. α)
  mayor = x;
  // mayor = (Máx α∈{A,B}. α)
}
// mayor = (Máx α∈{A,B}. α)
```





3º



¿Cómo probar que una instrucción condicional es **correcta**?

```
//  P
if (C) {
   A1
}
else {
   A2
}
//  Q
```

¿Cómo probar que una instrucción condicional es **correcta**?

```
//  P
//  P ⇒ Dom(C)                          1º   } ¿1?
if (C) {
   // P ∧ C   ⇒   pmd(A1,Q)             2º   ⎫
   A1                                        ⎬ ¿2?
   // Q                                      ⎭
}
else {
   // P ∧ ¬C   ⇒   pmd(A2,Q)            3º   ⎫
   A2                                        ⎬ ¿3?
   // Q                                      ⎭
}
//  Q
```

4. Análisis de la corrección de una instrucción iterativa

Queremos **probar** que la siguiente instrucción iterativa es **correcta**.

```
// n ≥ 0 ∧ n ≤ #v ∧ i = 0 ∧ suma = 0.0
while (i != n) {
    suma = suma + v[i];
    i = i + 1;
}
// suma = (Σα∈[0,n-1]. v[α])
```

La clave de la demostración está en identificar un predicado **INV** que se satisfaga antes de evaluar la condición del bucle en cada una de sus iteraciones:

```
// n ≥ 0 ∧ n ≤ #v ∧ i = 0 ∧ suma = 0.0
// INV
 while (i != n) {
    suma = suma + v[i];
    i = i + 1;
    // INV
 }
 // suma = (Σα∈[0,n-1]. v[α])
```

El predicado **INV** se satisfará antes de ejecutar el código a iterar e inmediatamente después de salir del bucle:

```
// n ≥ 0 ∧ n ≤ #v ∧ i = 0 ∧ suma = 0.0
// INV
 while (i != n) {
   // INV ∧ i ≠ n
   suma = suma + v[i];
   i = i + 1;
   // INV
 }
 // INV ∧ i = n
 // suma = (Σα∈[0,n-1]. v[α])
```

Estos predicados **INV** se conocen como predicados **invariantes** del bucle.

Siempre es posible plantear como predicado invariante de un bucle el predicado **cierto**. El problema que plantea este predicado tan débil es su inutilidad, ya que ni ayuda a comprender el diseño ni facilita el análisis de su corrección.

```
// n ≥ 0 ∧ i = 0 ∧ suma = 0.0
// cierto
while (i != n) {
  // cierto ∧ i ≠ n
  suma = suma + v[i];
  i = i + 1;
  // cierto
}
// cierto ∧ i = n
// suma = (Σα∈[0,n-1]. v[α])
```

Un predicado invariante ligeramente más fuerte que el anterior, por ejemplo el predicado **i ≥ 0 ∧ i ≤ n ∧ n ≤ #v**, tampoco ayuda demasiado a comprender el diseño ni basta para razonar sobre su corrección. No es suficientemente fuerte.

```
// n ≥ 0 ∧ n ≤ #v ∧ i = 0 ∧ suma = 0.0
// i ≥ 0 ∧ i ≤ n ∧ n ≤ #v
 while (i != n) {
   // i ≥ 0 ∧ i ≤ n ∧ n ≤ #v ∧ i ≠ n
   suma = suma + v[i];
   i = i + 1;
   // i ≥ 0 ∧ i ≤ n ∧ n ≤ #v
 }
 // i ≥ 0 ∧ i ≤ n ∧ n ≤ #v ∧ i = n
 // suma = (Σα∈[0,n-1]. v[α])
```

Los posibles predicados invariantes realmente útiles han de ser un reflejo del método empleado para resolver el problema. Consideremos el predicado:

i ≥ 0 ∧ i ≤ n ∧ n ≤ #v ∧ suma = (Σα∈[0,i-1]. v[α])

```
// n ≥ 0 ∧ n ≤ #v ∧ i = 0 ∧ suma = 0.0
// i ≥ 0 ∧ i ≤ n ∧ n ≤ #v ∧ suma = (Σα∈[0,i-1]. v[α])
 while (i != n) {
   // i ≥ 0 ∧ i < n ∧ n ≤ #v ∧ suma = (Σα∈[0,i-1]. v[α])
   suma = suma + v[i];
   i = i + 1;
   // i ≥ 0 ∧ i ≤ n ∧ n ≤ #v ∧ suma = (Σα∈[0,i-1]. v[α])
 }
 // i ≥ 0 ∧ i = n ∧ n ≤ #v ∧ suma = (Σα∈[0,i-1]. v[α])
 // suma = (Σα∈[0,n-1]. v[α])
```

Permite probar formalmente que es correcto el código (inexistente) que precede al bucle:

```
// n ≥ 0 ∧ n ≤ #v ∧ i = 0 ∧ suma = 0.0
// ⇒                                                  // p_1ª
// i ≥ 0 ∧ i ≤ n ∧ n ≤ #v ∧ suma = (Σα∈[0,i-1]. v[α])

// i ≥ 0 ∧ i ≤ n ∧ n ≤ #v ∧ suma = (Σα∈[0,i-1]. v[α])
while (i != n) {
   // i ≥ 0 ∧ i < n ∧ n ≤ #v ∧ suma = (Σα∈[0,i-1]. v[α])
   suma = suma + v[i];
   i = i + 1;
   // i ≥ 0 ∧ i ≤ n ∧ n ≤ #v ∧ suma = (Σα∈[0,i-1]. v[α])
}
// i ≥ 0 ∧ i = n ∧ n ≤ #v ∧ suma = (Σα∈[0,i-1]. v[α])

// suma = (Σα∈[0,n-1]. v[α])
```

También que es correcto el código (inexistente) que sigue al bucle:

```
// n ≥ 0 ∧ n ≤ #v ∧ i = 0 ∧ suma = 0.0
// ⇒                                                  // p_1ª
// i ≥ 0 ∧ i ≤ n ∧ n ≤ #v ∧ suma = (Σα∈[0,i-1]. v[α])
// i ≥ 0 ∧ i ≤ n ∧ n ≤ #v ∧ suma = (Σα∈[0,i-1]. v[α])
while (i != n) {
  // i ≥ 0 ∧ i < n ∧ n ≤ #v ∧ suma = (Σα∈[0,i-1]. v[α])
  suma = suma + v[i];
  i = i + 1;
  // i ≥ 0 ∧ i ≤ n ∧ n ≤ #v ∧ suma = (Σα∈[0,i-1]. v[α])
}
// i ≥ 0 ∧ i = n ∧ n ≤ #v ∧ suma = (Σα∈[0,i-1]. v[α])
// ⇒                                                  // p_2ª
// suma = (Σα∈[0,n-1]. v[α])

// suma = (Σα∈[0,n-1]. v[α])
```

Que la condición del bucle se evalúa sin errores:

```
// n ≥ 0 ∧ n ≤ #v ∧ i = 0 ∧ suma = 0.0
// ⇒                                                    // p_1ª
// i ≥ 0 ∧ i ≤ n ∧ n ≤ #v ∧ suma = (Σα∈[0,i-1]. v[α])
// i ≥ 0 ∧ i ≤ n ∧ n ≤ #v ∧ suma = (Σα∈[0,i-1]. v[α])
// ⇒
// Dom(i ≠ n) ≡ cierto                                  // p_3ª
while (i != n) {
   // i ≥ 0 ∧ i < n ∧ n ≤ #v ∧ suma = (Σα∈[0,i-1]. v[α])
   suma = suma + v[i];
   i = i + 1;
   // i ≥ 0 ∧ i ≤ n ∧ n ≤ #v ∧ suma = (Σα∈[0,i-1]. v[α])
}
// i ≥ 0 ∧ i = n ∧ n ≤ #v ∧ suma = (Σα∈[0,i-1]. v[α])
// ⇒                                                    // p_2ª
// suma = (Σα∈[0,n-1]. v[α])

// suma = (Σα∈[0,n-1]. v[α])
```

También podemos probar la corrección del bloque a iterar:

```
// n ≥ 0 ∧ n ≤ #v ∧ i = 0 ∧ suma = 0.0
// ⇒                                                          // p_1ª
// i ≥ 0 ∧ i ≤ n ∧ n ≤ #v ∧ suma = (Σα∈[0,i-1]. v[α])
// i ≥ 0 ∧ i ≤ n ∧ n ≤ #v ∧ suma = (Σα∈[0,i-1]. v[α])
// ⇒  Dom(i≠n) ≡ cierto                                      // p_3ª
while (i != n) {
   // i ≥ 0 ∧ i < n ∧ n ≤ #v ∧ suma = (Σα∈[0,i-1]. v[α])
   suma = suma + v[i];
   i = i + 1;
   // i ≥ 0 ∧ i ≤ n ∧ n ≤ #v ∧ suma = (Σα∈[0,i-1]. v[α])
}
// i ≥ 0 ∧ i = n ∧ n ≤ #v ∧ suma = (Σα∈[0,i-1]. v[α])
// ⇒                                                          // p_2ª
// suma = (Σα∈[0,n-1]. v[α])
// suma=(Σα∈[0,n-1]. v[α])
```

Las cuatro pruebas completadas. Falta probar que el bucle termina:

```
// n ≥ 0 ∧ n ≤ #v ∧ i = 0 ∧ suma = 0.0
// i ≥ 0 ∧ i ≤ n ∧ n ≤ #v ∧ suma = (Σα∈[0,i-1]. v[α])
while (i != n) {  //  ¿ La iteración termina ?
  // i ≥ 0 ∧ i < n ∧ suma = (Σα∈[0,i-1]. v[α])
  // ⇒                                                    // p_4ª
  // i+1 ≥ 0 ∧ i < n ∧ n ≤ #v ∧ suma + v[i] = (Σα∈[0,i]. v[α])
  // i+1 ≥ 0 ∧ i+1 ≤ n ∧ suma + v[i] = (Σα∈[0,i]. v[α])      2
  suma = suma + v[i];
  // i+1 ≥ 0 ∧ i+1 ≤ n ∧ n ≤ #v ∧ suma = (Σα∈[0,i+1-1]. v[α])
  i = i + 1;                                               1
  // i ≥ 0 ∧ i ≤ n ∧ n ≤ #v ∧ suma = (Σα∈[0,i-1]. v[α])
}
// i ≥ 0 ∧ i = n ∧ n ≤ #v ∧ suma = (Σα∈[0,i-1].v[α])
// ⇒
// suma = (Σα∈[0,n-1].v[α])                                // p_2ª
// suma = (Σα∈[0,n-1].v[α])
```

Las cuatro pruebas completadas. Falta probar que el bucle termina:

```
// n ≥ 0 ∧ n ≤ #v ∧ i = 0 ∧ suma = 0.0
// i ≥ 0 ∧ i ≤ n ∧ n ≤ #v ∧ suma = (Σα∈[0,i-1]. v[α])
while (i != n) {
  //  ¿ La iteración termina ?
  // i ≥ 0 ∧ i < n ∧ n ≤ #v ∧ suma = (Σα∈[0,i-1]. v[α])
  suma = suma + v[i];
  i = i + 1;
  // i ≥ 0 ∧ i ≤ n ∧ n ≤ #v ∧ suma = (Σα∈[0,i-1]. v[α])
}
// i ≥ 0 ∧ i = n ∧ n ≤ #v ∧ suma = (Σα∈[0,i-1]. v[α])
// suma = (Σα∈[0,n-1]. v[α])
```

¿Cómo probar que el bucle termina? Mediante una función de cota que define una **sucesión** **finita** de **valores decrecientes**. En este caso es apropiada para ello: $f_{cota}(i) = n - i$

```
// n ≥ 0 ∧ n ≤ #v ∧ i = 0 ∧ suma = 0.0
while (i != n) {    // f_cota (i) = n - i
  // i ≥ 0 ∧ i ≤ n ∧ n ≤ #v ∧ suma = (Σα∈[0,i-1]. v[α])
  // i = A ∧ f_cota(inicio_iteración) = n - A          1
  suma = suma + v[i];   i = i + 1;
  // i = A + 1 ∧ f_cota(fin_iteración) = n – A - 1     2
  // 1) n - A > n – A – 1
  // 2) i ≥ 0 ∧ i ≤ n  →  n – i ≥ 0
}
// suma=(Σα∈[0,n-1]. v[α])
```

cálculos para probar terminación (1, 2)

Prueba de que la iteración **termina** ya que el valor de la función de cota:

1. Decrece en cada iteración: $n - A > n - A - 1$ // p_5ª (a)
2. La sucesión de valores que toma en Z es finita ya que está acotada inferiormente: $i \geq 0 \wedge i \leq n \rightarrow n - i \geq 0$ // p_5ª (b)

¿Cómo probar que el diseño de una composición iterativa de instrucciones es correcto?

```
// P  ⇒  I                                  ¿1?
// I  ⇒  Dom(C)                             ¿2?
while (C) {   //  Terminación: f_cota       ¿3?
  // C ∧ I  ⇒  pmd(A,I)
  A                                         ¿4?
  // I
}
// ¬C ∧ I  ⇒  Q                             ¿5?
// Q
```

Terminación (**¿3?**):

1. La función de cota f_{cota} **decrece** en cada iteración:

   $$f_{cotaAntes_A} > f_{cotaDespués_A}$$

2. La sucesión de valores decrecientes que toma la función de cota es **finita** (por ejemplo, probando que la sucesión está acotada inferiormente en $\mathbb{Z}$).