Programación 2

Prueba formal de la corrección de algoritmos iterativos

Problemas 14

Problema 1. Identificar predicados invariantes de cada uno de los bucles de estas cinco funciones cuyo código se presenta más adelante:

- **mayor**(n)
- **permutar**(v,p,n)
- **cambiarSigno**(v,n)
- **autoPermutar**(v,n)
- **autoAcumular**(v,n)

Cada uno de ellos ha de ser suficientemente fuerte como para sustentar la demostración formal de la corrección del código a iterar en cada caso, así como del código que precede y que sigue a cada bucle.

```
/*
 * Pre: n ≥ 0 ∧ n = A
 * Post: mayor(n) = (Máx α∈[1,∞]. digito(A,α))
 *
 * Definición: digito(n,i) = (n / 10 i-1) % 10
 */
int mayor (int n) {
   int max = n % 10;
   n = n / 10;
   while (n != 0) {
      // ¿Invariante del bucle?
      if (n % 10 > max) {
         max = n % 10;
      }
      n = n / 10;
   }
   return max;
}
```

```
/*
 * Pre: n > 0 ∧ n <= #v ∧ n <= #p
 * Post: (∀α∈[0,n-1]. p[α] = v[n-1-α])
 */
void permutar (const double v[], double p[], const int n) {
   int i = 0;
   while (i != n) {
      // ¿Invariante del bucle?
      p[i] = v[n-1-i];
      i = i + 1;
   }
}
```

```
/*
 * Pre: n > 0 ∧ n <= #v ∧ v = Vo
 * Post: (∀α∈[0,n-1]. v[α] = -Vo[α])
 */
void cambiarSigno (int v[], const int n) {
    int i = 0;
    while (i != n) {
        // ¿Invariante del bucle?
        v[i] = -v[i];
        i = i + 1;
    }
}
```

```
/*
 * Pre: n > 0 ∧ n <= #v ∧ v = Vo
 * Post: (∀α∈[0,n-1]. v[α] = Vo[n-1-α])
 */
void autoPermutar (double v[], const int n) {
   int i = 0, medio = (n - 1) / 2;
   while (i != medio + 1) {
       // ¿Invariante del bucle?
       double aux = v[i];
       v[i] = v[n-1-i];
       v[n-1-i] = aux;
       i = i + 1;
   }
}
```

```
 /*
  * Pre: n > 0 ∧ n <= #v ∧ v = Vo
  * Post: (∀α∈[0,n-1]. v[α] = (∑β∈[0,α]. Vo[β]))
  */
void autoAcumular (double v[], const int n) {
    int i = 0;
    while (i != n-1) {
       // ¿Invariante del bucle?
       i = i + 1;
       v[i] = v[i-1] + v[i];
    }
 }
```

Una solución. Se ha identificado un predicado invariante del bucle lo suficientemente fuerte como para sustentar el conjunto de pruebas de la corrección del código de la función `mayor(n)`.

```
/*
 * Pre: n ≥ 0 ∧ n = A
 * Post: mayor(n) = (Máx α∈[1,∞]. digito(A,α))
 * Definición: digito(n,i) = (n / 10^(i-1)) % 10
 */
int mayor (int n) {
   int max = n % 10;        //  int i = 1;
   n = n / 10;
   while (n != 0) {
       // Inv: A ≥ 0 ∧ i ≥ 1 ∧ n = A / 10^i ∧
       //      max = (Máx α∈[1,i]. digito(A,α))
      if (n % 10 > max) { max = n % 10; }
      n = n / 10;            //  i = i + 1;
   }
   return max;
}
```

La variable **i** del diseño anterior se puede sustituir por la variable inicial **I** cuyo valor, en el predicado invariante del bucle, representa el número de veces que el valor inicial de **n** ha sido dividido por 10,

```
/*
 * Pre: n ≥ 0 ∧ n = A
 * Post: mayor(n) = (Máx α∈[1,∞]. digito(A,α))
 * Definición: digito(n,i) = (n / 10^(i-1)) % 10
 */
int mayor (int n) {
   int max = n % 10;
   n = n / 10;
   while (n != 0) {
       // Inv: A ≥ 0 ∧ I ≥ 1 ∧ n = A / 10^I ∧
       //      max = (Máx α∈[1,I]. digito(A,α))
       if (n % 10 > max) { max = n % 10; }
       n = n / 10;
   }
   return max;
}
```

Se ha identificado un predicado invariante del bucle lo suficientemente fuerte como para sustentar el conjunto de pruebas de la corrección del código de la función `permutar(v,p,n)`.

```
/*
 * Pre: n > 0 ∧ n <= #v ∧ n <= #p
 * Post: (∀α∈[0,n-1]. p[α] = v[n-1-α])
 */
void permutar (const double v[], double p[], const int n) {
   int i = 0;
   while (i != n) {
      // Inv: n > 0 ∧ n <= #v ∧ n <= #p ∧ i ≥ 0 ∧ i ≤ n ∧
      //      (∀α∈[0,i-1]. p[α] = v[n-1-α])
      p[i] = v[n-1-i];
      i = i + 1;
   }
}
```

Se ha identificado un predicado invariante del bucle lo suficientemente fuerte como para sustentar el conjunto de pruebas de la corrección del código de la función `cambiarSigno(v,n)`.

```
/*
 * Pre: n > 0 ∧ n <= #v ∧ v = Vo
 * Post: (∀α∈[0,n-1]. v[α] = -Vo[α])
 */
void cambiarSigno (int v[], const int n) {
    int i = 0;
    while (i != n) {
       // Inv: n > 0 ∧ n <= #v ∧ i ≥ 0 ∧ i ≤ n ∧
       //      (∀α∈[0,i-1]. v[α] = -Vo[α]) ∧
       //      (∀α∈[i,n-1]. v[α] = Vo[α])
       v[i] = -v[i];
       i = i + 1;
    }
}
```

Se ha identificado un predicado invariante del bucle lo suficientemente fuerte como para sustentar el conjunto de pruebas de la corrección del código de la función `autoPermutar(v,n)`

```
/*
 * Pre: n > 0 ∧ n <= #v ∧ v = Vo
 * Post: (∀α∈[0,n-1]. v[α] = Vo[n-1-α])
 */
void autoPermutar (double v[], const int n) {
   int i = 0, medio = (n - 1) / 2;
   while (i != medio + 1) {
      // Inv: n > 0 ∧ n <= #v ∧ i ≥ 0 ∧ i ≤ medio + 1 ∧
      //      medio = (n - 1) / 2 ∧
      //      (∀α∈[0,i-1]. v[α] = Vo[n-1-α]) ∧
      //      (∀α∈[i,n-1-i]. v[α] = Vo[α]) ∧
      //      (∀α∈[n-i,n-1]. v[α] = Vo[n-1-α])
      double aux = v[i];
      v[i] = v[n-1-i];  v[n-1-i] = aux;  i = i + 1;
   }
}
```

Se ha identificado un predicado invariante del bucle lo suficientemente fuerte como para sustentar el conjunto de pruebas de la corrección del código de la función `autoAcumular(v,n)`.

```
/*
 * Pre: n > 0 ∧ n <= #v ∧ v = Vo
 * Post: (∀α∈[0,n-1]. v[α] = (∑β∈[0,α]. Vo[β]))
 */
void autoAcumular (double v[], const int n) {
   int i = 0;
   while (i != n-1) {
      // Inv: n > 0 ∧ n <= #v ∧ i ≥ 0 ∧ i ≤ n - 1 ∧
      //      (∀α∈[0,i]. v[α] = (∑β∈[0,α]. Vo[β])) ∧
      //      (∀α∈[i+1,n-1]. v[α] = Vo[α])
      i = i + 1;
      v[i] = v[i-1] + v[i];
   }
}
```

Problema 2. Probar formalmente la corrección de la función `cambiarSigno(v,n)`.

```
/*
 * Pre: n > 0 ∧ n <= #v ∧ v = Vo
 * Post: (∀α∈[0,n-1]. v[α] = -Vo[α])
 */
void cambiarSigno (int v[], const int n) {
    int i = 0;
    while (i != n) {
       v[i] = -v[i];  i = i + 1;
    }
}
```

Sugerencia: desarrollar la colección de pruebas formales alrededor del predicado invariante del bucle identificado en el problema anterior

El predicado invariante identificado en el problema anterior va a ser pieza clave de la demostración formal de la corrección de esta función.

```
/*
 * Pre: n > 0 ∧ n <= #v ∧ v = Vo
 * Post: (∀α∈[0,n-1]. v[α] = -Vo[α])
 */
void cambiarSigno (int v[], const int n) {
    int i = 0;
    // Invariante del bucle
    while (i != n) {
       v[i] = -v[i];  i = i + 1;
       // Invariante del bucle
    }
}
```

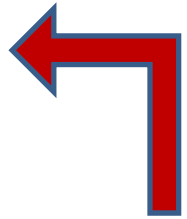
```
/*
 * Pre: n > 0 ∧ v = Vo
 * Post: (∀α∈[0,n-1]. v[α] = -Vo[α])
 */
void cambiarSigno (int v[], const int n) {
    int i = 0;
    // Inv: n > 0 ∧ n <= #v ∧ i ≥ 0 ∧ i ≤ n ∧
    //      (∀α∈[0,i-1]. v[α] = -Vo[α]) ∧
    //      (∀α∈[i,n-1]. v[α] = Vo[α])
    while (i!=n) {
        v[i] = -v[i];  i = i + 1;
        // Inv: n > 0 ∧ n <= #v ∧ i ≥ 0 ∧ i ≤ n ∧
        //      (∀α∈[0,i-1]. v[α] = -Vo[α]) ∧
        //      (∀α∈[i,n-1]. v[α] = Vo[α])
    }
}
```

```
// Pre: n > 0 ∧ n <= #v ∧ v = Vo
// Post: (∀α∈[0,n-1].v[α]=-Vo[α])
void cambiarSigno (int v[], const int n) {
    // n > 0 ∧ v = Vo
    // ¿ ⇒ ?
    // n > 0 ∧ 0 ≥ 0 ∧ 0 ≤ n ∧
    // (∀α∈[0,0-1]. v[α] = -Vo[α]) ∧
    // (∀α∈[0,n-1]. v[α] = Vo[α])
    int i = 0;
    // Inv: n > 0 ∧ n <= #v ∧ i ≥ 0 ∧ i ≤ n ∧
    //      (∀α∈[0,i-1]. v[α] = -Vo[α]) ∧
    //      (∀α∈[i,n-1]. v[α] = Vo[α])
    while (i != n) {
        v[i] = -v[i];  i = i + 1;
        // Inv: n > 0 ∧ n <= #v ∧ i ≥ 0 ∧ i ≤ n ∧
        //      (∀α∈[0,i-1]. v[α] = -Vo[α]) ∧
        //      (∀α∈[i,n-1]. v[α] = Vo[α])
    }
}
```

1º

```
// Pre: n > 0 ∧ n <= #v ∧ v = Vo
// Post: (∀α∈[0,n-1].v[α]=-Vo[α])
void cambiarSigno (int v[], const int n) {
    // n > 0 ∧ n <= #v ∧ v = Vo
    // ⇒                    // prueba que es correcto el código que precede al bucle
    // n > 0 ∧ n <= #v ∧ 0 ≥ 0 ∧ 0 ≤ n ∧
    // (∀α∈[0,0-1]. v[α] = -Vo[α]) ∧
    // (∀α∈[0,n-1]. v[α] = Vo[α])
    int i = 0;
    // n > 0 ∧ n <= #v ∧ i ≥ 0 ∧ i ≤ n ∧
    // (∀α∈[0,i-1]. v[α] = -Vo[α]) ∧ (∀α∈[i,n-1]. v[α] = Vo[α])
    while (i != n) {
        v[i] = -v[i];  i = i + 1;
        // n > 0 ∧ n <= #v ∧ i ≥ 0 ∧ i ≤ n ∧
        // (∀α∈[0,i-1]. v[α] = -Vo[α]) ∧
        // (∀α∈[i,n-1]. v[α] = Vo[α])
    }
}
```

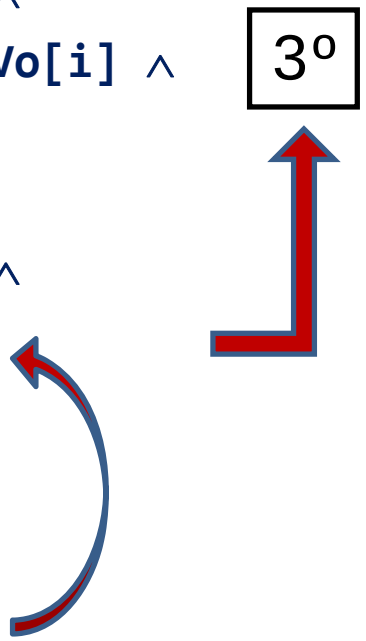
1º



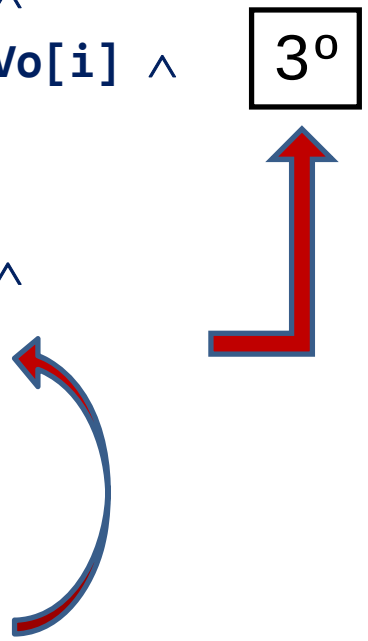
```
/*
 * Pre: n > 0 ∧ n <= #v ∧ v = Vo
 * Post: (∀α∈[0,n-1]. v[α] = -Vo[α])
 */
void cambiarSigno (int v[], const int n) {
    int i = 0;
    // n > 0 ∧ n <= #v ∧ i ≥ 0 ∧ i ≤ n ∧
    // (∀α∈[0,i-1]. v[α] = -Vo[α]) ∧ (∀α∈[i,n-1]. v[α] = Vo[α])
    // ⇒                // prueba que la condición del bucle se ejecuta sin errores
    // Dom(i != n) ≡ cierto
    while (i != n) {
       v[i] = -v[i];  i = i + 1;
    }
}
```

```
/*
 * Pre: n > 0 ∧ n <= #v ∧ v = Vo
 * Post: (∀α∈[0,n-1]. v[α] = -Vo[α])
 */
void cambiarSigno (int v[], const nt n) {
    int i = 0;
    while (i != n) {
        v[i] = -v[i];  i = i + 1;
        // n > 0 ∧ n <= #v ∧ i ≥ 0 ∧ i ≤ n ∧
        // (∀α∈[0,i-1]. v[α] = -Vo[α]) ∧(∀α∈[i,n-1]. v[α] = Vo[α])
    }
    // n > 0 ∧ n <= #v ∧ i ≥ 0 ∧ i ≤ n ∧
    // (∀α∈[0,i-1]. v[α] = -Vo[α]) ∧
    // (∀α∈[i,n-1]. v[α] = Vo[α]) ∧ i = n
    //  ⇒      // prueba que es correcto el código (ninguno) que sigue al bucle
    // (∀α∈[0,n-1]. v[α] = -Vo[α])
}
```

```
while (i != n) {
    // n > 0 ∧ n <= #v ∧ i ≥ 0 ∧ i ≤ n ∧ i ≠ n ∧
    // (∀α∈[0,i-1]. v[α] = -Vo[α]) ∧
    // (∀α∈[i,n-1]. v[α] = Vo[α])
    // ¿ ⇒ ?
    // n > 0 ∧ n <= #v ∧ i + 1 ≥ 0 ∧ i + 1 ≤ n ∧
    // (∀α∈[0,i-1]. v[α] = -Vo[α]) ∧ -v[i] = -Vo[i] ∧
    // (∀α∈[i+1,n-1]. v[α] = Vo[α])
    v[i] = -v[i];
    // n > 0 ∧ n <= #v ∧ i + 1 ≥ 0 ∧ i + 1 ≤ n ∧
    // (∀α∈[0,i+1-1]. v[α] = -Vo[α]) ∧
    // (∀α∈[i+1,n-1]. v[α] = Vo[α])
    i = i + 1;
    // n > 0 ∧ n <= #v ∧ i ≥ 0 ∧ i ≤ n
    // ∧ (∀α∈[0,i-1]. v[α] = -Vo[α])
    // ∧ (∀α∈[i,n-1]. v[α] = Vo[α])
}
```



```
while (i != n) {
    // n > 0 ∧ n <= #v ∧ i ≥ 0 ∧ i ≤ n ∧ i ≠ n ∧
    // (∀α∈[0,i-1]. v[α] = -Vo[α]) ∧
    // (∀α∈[i,n-1]. v[α] = Vo[α])
    // ⇒                    // prueba que es correcto el código a iterar en el bucle
    // n > 0 ∧ n <= #v ∧ i + 1 ≥ 0 ∧ i + 1 ≤ n ∧
    // (∀α∈[0,i-1]. v[α] = -Vo[α]) ∧ -v[i] = -Vo[i] ∧
    // (∀α∈[i+1,n-1]. v[α] = Vo[α])
    v[i] = -v[i];
    // n > 0 ∧ n <= #v ∧ i + 1 ≥ 0 ∧ i + 1 ≤ n ∧
    // (∀α∈[0,i+1-1]. v[α] = -Vo[α]) ∧
    // (∀α∈[i+1,n-1]. v[α] = Vo[α])
    i = i + 1;
    // n > 0 ∧ n <= #v ∧ i ≥ 0 ∧ i ≤ n
    // ∧ (∀α∈[0,i-1]. v[α] = -Vo[α])
    // ∧ (∀α∈[i,n-1]. v[α] = Vo[α])
}
```



```
while (i != n) {     // f_cota = n – i
   // Inv: n > 0 ∧ n <= #v ∧ i ≥ 0 ∧ i ≤ n ∧
   //      (∀α∈[0,i-1]. v[α] = -Vo[α]) ∧
   //      (∀α∈[i,n-1]. v[α] = Vo[α])

   // i = X ∧ f_cota(antes) = n – X                    4º
   v[i] = -v[i];
   i = i + 1;
   // i = X + 1 ∧ f_cota(después) = n – X – 1          5º

   // Prueba de la terminación del bucle:
   //    1. El valor de la f_cota decrece en cada iteración:
   //          f_cota(antes) > f_cota(después)
   //          ya que n – X > n – X - 1
   //    2. El valor de f_cota está acotado inferiormente en Z:
   //          Inv  →  i ≤ n  →  f_cota = n - i ≥ 0
}
```

Problema 3. Probar formalmente la corrección de la función permutar(v,p,n).

```
/*
 * Pre: n > 0 ∧ n <= #v ∧ n <= #p
 * Post: (∀α∈[0,n-1]. p[α] = v[n-1-α])
 */
void permutar (const double v[], double p[], const int n) {
   int i = 0;
   while (i != n) {
      p[i] = v[n-1+i];
      i = i + 1;
   }
}
```

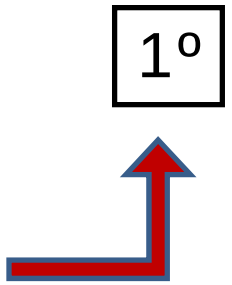
Sugerencia: desarrollar la colección de pruebas formales alrededor del predicado invariante del bucle identificado en el primer problema

El predicado invariante identificado en el primer problema va a ser pieza clave para la demostración formal de la corrección de esta función.

```
/*
 * Pre: n > 0 ∧ n <= #v ∧ n <= #p
 * Post: (∀α∈[0,n-1]. p[α] = v[n-1-α])
 */
void permutar (const double v[], double p[], const int n) {
   int i = 0;
   // Invariante del bucle
   while (i != n) {
      p[i] = v[n-1+i];
      i = i + 1;
      // Invariante del bucle
   }
}
```

```
/*
 * Pre: n > 0 ∧ n <= #v ∧ n <= #p
 * Post: (∀α∈[0,n-1]. p[α] = v[n-1-α])
 */
void permutar (const double v[], double p[], const int n) {
   int i = 0;
   // Inv: n > 0 ∧ n <= #v ∧ n <= #p ∧ i ≥ 0 ∧ i ≤ n ∧
   //      (∀α∈[0,i-1]. p[α] = v[n-1-α])
   while (i != n) {
      p[i] = v[n-1-i];
      i = i + 1;
      // Inv: n > 0 ∧ n <= #v ∧ n <= #p ∧ i ≥ 0 ∧ i ≤ n ∧
      //      (∀α∈[0,i-1]. p[α] = v[n-1-α])
   }
}
```

```
/*
 * Pre: n > 0 ∧ n <= #v ∧ n <= #p
 * Post: (∀α∈[0,n-1]. p[α] = v[n-1-α])
 */
void permutar (const double v[], double p[], const int n) {
   // n > 0 ∧ n <= #v ∧ n <= #p
   // ¿ ⇒ ?
   // n > 0 ∧ n <= #v ∧ n <= #p ∧
   // 0 ≥ 0 ∧ 0 ≤ n ∧ (∀α∈[0,0-1]. p[α] = v[n-1-α])
   int i = 0;
   // Inv: n > 0 ∧ n <= #v ∧ n <= #p ∧ i ≥ 0 ∧ i ≤ n ∧
   //      (∀α∈[0,i-1]. p[α] = v[n-1-α])
   while (i != n) {
      p[i] = v[n-1-i];
      i = i + 1;
      // Inv: n > 0 ∧ n <= #v ∧ n <= #p ∧ i ≥ 0 ∧ i ≤ n
      //      (∀α∈[0,i-1]. p[α] = v[n-1-α])
   }
}
```



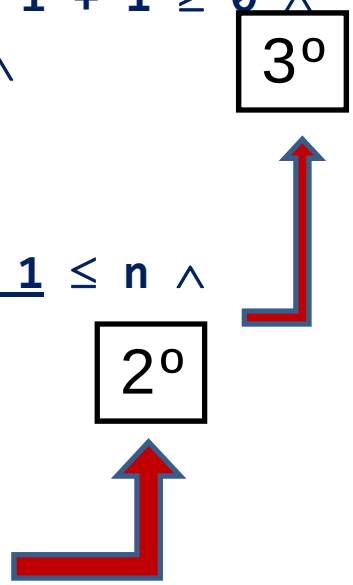
```
/*
 * Pre: n > 0 ∧ n <= #v ∧ n <= #p
 * Post: (∀α∈[0,n-1]. p[α] = v[n-1-α])
 */
void permutar (const double v[], double p[], const int n) {
   // n > 0 ∧ n <= #v ∧ n <= #p
   // ⇒                    // prueba que es correcto el código que precede al bucle
   // n > 0 ∧ n <= #v ∧ n <= #p ∧
   // 0 ≥ 0 ∧ 0 ≤ n ∧ (∀α∈[0,0-1]. p[α] = v[n-1-α])
   int i = 0;
   // Inv: n > 0 ∧ n <= #v ∧ n <= #p ∧ i ≥ 0 ∧ i ≤ n ∧
   //      (∀α∈[0,i-1]. p[α] = v[n-1-α])
   while (i != n) {
      p[i] = v[n-1-i];
      i = i + 1;
      // Inv: n > 0 ∧ n <= #v ∧ n <= #p ∧ i ≥ 0 ∧ i ≤ n
      //      (∀α∈[0,i-1]. p[α] = v[n-1-α])
   }
}
```

1º

```
/*
 * Pre: n > 0 ∧ n <= #v ∧ n <= #p
 * Post: (∀α∈[0,n-1]. p[α] = v[n-1-α])
 */
void permutar (const double v[], double p[], const int n) {
   int i = 0;
   // n > 0 ∧ n <= #v ∧ n <= #p ∧
   // i ≥ 0 ∧ i ≤ n ∧ (∀α∈[0,i-1]. p[α] = v[n-1-α])
   // ⇒    // prueba que la condición del bucle se ejecuta sin errores
   // Dom(i!=n) ≡  cierto
   while (i != n) {
       p[i] = v[n-1-i];
       i = i + 1;
   }
}
```

```
/*
 * Pre: n > 0 ∧ n <= #v ∧ n <= #p
 * Post: (∀α∈[0,n-1]. p[α] = v[n-1-α])
 */
void permutar (const double v[], double p[], const int n) {
   int i = 0;
   // n > 0 ∧ n <= #v ∧ n <= #p ∧
   // i ≥ 0 ∧ i ≤ n ∧ (∀α∈[0,i-1]. p[α] = v[n-1-α])
   while (i != n) {
      p[i] = v[n-1-i];
      i = i + 1;
      // n > 0 ∧ n <= #v ∧ n <= #p ∧ i ≥ 0 ∧ i ≤ n ∧
      // (∀α∈[0,i-1]. p[α] = v[n-1-α])
   }
   // n > 0 ∧ n <= #v ∧ n <= #p ∧ i ≥ 0 ∧ i ≤ n ∧
   // (∀α∈[0,i-1]. p[α] = v[n-1-α])
   // i = n
   // ⇒          // prueba que es correcto el código (ninguno)  que sigue al bucle
   // (∀α∈[0,n-1].p[α]=v[n-1-α])
}
```

```
while (i != n) {
    // n > 0 ∧ n <= #v ∧ n <= #p ∧ i ≥ 0 ∧ i ≤ n ∧
    // (∀α∈[0,i-1] .p[α] = v[n-1-α]) ∧ i ≠ n
    // ¿ ⇒ ?
    // i ≥ 0 ∧ i < n ∧ n > 0 ∧ n <= #v ∧ n <= #p ∧ i + 1 ≥ 0 ∧
    // i + 1 ≤ n ∧ (∀α∈[0,i-1]. p[α] = v[n-1-α]) ∧
    // v[n-1-i] = v[n-1-i]
    p[i] = v[n-1-i];
    // n > 0 ∧ n <= #v ∧ n <= #p ∧ i + 1 ≥ 0 ∧ i + 1 ≤ n ∧
    // (∀α∈[0,i+1-1]. p[α] = v[n-1-α])
    i = i + 1;
    // n > 0 ∧ n <= #v ∧ n <= #p ∧ i ≥ 0 ∧ i ≤ n ∧
    // (∀α∈[0,i-1]. p[α] = v[n-1-α])
}
```



```
while (i != n) {
    // n > 0 ∧ n <= #v ∧ n <= #p ∧ i ≥ 0 ∧ i ≤ n
    // ∧ (∀α∈[0,i-1]. p[α] = v[n-1-α]) ∧ i ≠ n
    // ⇒                    // prueba que es correcto el código a iterar en el bucle
    // i ≥ 0 ∧ i < n ∧ n > 0 ∧ n <= #v ∧ n <= #p ∧ i + 1 ≥ 0 ∧
    // i + 1 ≤ n ∧ (∀α∈[0,i-1]. p[α] = v[n-1-α]) ∧
    // v[n-1-i] = v[n-1-i]
    p[i] = v[n-1-i];
    // n > 0 ∧ n <= #v ∧ n <= #p ∧ i + 1 ≥ 0 ∧ i + 1 ≤ n ∧
    // (∀α∈[0,i+1-1]. p[α] = v[n-1-α])
    i = i + 1;
    // n > 0 ∧ n <= #v ∧ n <= #p ∧ i ≥ 0 ∧ i ≤ n ∧
    // (∀α∈[0,i-1]. p[α] = v[n-1-α])
 }
```

3º

2º

```
/*
 * Pre: n > 0 ∧ n <= #v ∧ n <= #p
 * Post: (∀α∈[0,n-1]. p[α] = v[n-1-α])
 */
void permutar (const double v[], double p[], const int n) {
   int i = 0;
   while (i != n) {   // f_cota = n - i
      // Inv: n > 0 ∧ n <= #v ∧ n <= #p ∧
      //      i ≥ 0 ∧ i ≤ n ∧ (∀α∈[0,i-1]. p[α] = v[n-1-α])
      // i = X ∧ f_cota(antes) = n - X
      p[i] = v[n-1-i];  i = i + 1;
      // i = X + 1 ∧ f_cota(después) = n - X - 1
      // Prueba de la terminación del bucle:
      //    1. El valor de la f_cota decrece en cada iteración:
      //          f_cota(antes) > f_cota(después)
      //          ya que n - X > n - X - 1
      //    2. El valor de f_cota está acotado inferiormente en Z:
      //          Inv  →  i ≤ n  →  f_cota = n - i ≥ 0
   }
}
```

4º

5º