Programación 2

Lección 4. Metodología de diseño de algoritmos recursivos

1. Diseño iterativo *vs* diseño recursivo
2. Cómo diseñar algoritmos recursivos
3. Diseño recursivo de funciones:
   a) Función que calcula el máximo común divisor de dos naturales
   b) Una función de recorrido de un vector
   c) Una función de búsqueda en un vector
   d) Una función de comparación de dos vectores
   e) Una función que modifica los elementos de un vector
   f) Una función interactiva

1. Diseño iterativo *vs* diseño recursivo

Diseño iterativo de un algoritmo para salir del aula:

```
while (no estoy en la puerta) {
   "Doy un paso hacia la puerta"      // acción a iterar
}
"Abro la puerta" y "Salgo de clase"
```

Diseño recursivo del mismo algoritmo para salir del aula:

```
if (estoy en la puerta) {
   "Abro la puerta" y "Salgo de clase"
}
else {
   "Doy un paso hacia la puerta" y
   Reaplico este mismo algoritmo  // planteamiento recursivo
}
```

Diseño iterativo de un algoritmo para salir del aula:

```
while (no estoy en la puerta) {
   "Doy un paso hacia la puerta"
}
"Abro la puerta" y "Salgo de clase"
```

Diseño recursivo* del mismo algoritmo para salir del aula:

```
if (no estoy en la puerta) {
   "Doy un paso hacia la puerta" y
   Reaplico este mismo algoritmo
}
else {
   "Abro la puerta" y "Salgo de clase"
}
```

(*) Este diseño recursivo es equivalente al anterior

2. Cómo diseñar algoritmos recursivos

```
// Pre: precondición
// Post: postcondición
tipoDato nombreFunción (lista de parámetros) {
   if (CB1) {   B1;  }          // solución base 1
   else if (CB2) {  B2;  }      // solución base 2
   else if (…) {  …  }          // solución base …
   else if (CBp) {  Bp;  }      // solución base p
   else if (CR1) {  R1;  }      // solución recurrente 1
   else if (CR2) {  R2;  }      // solución recurrente 2
   else if (…) {  …  }          // solución recurrente …
   else if (CRq) {  Rq;  }      // solución recurrente q
   else {  F;  }                // solución base o recurrente
}
```

3. Diseño recursivo de funciones

3.a) Función que calcula el máximo común divisor de dos naturales

Un primer método de cálculo del m.c.d. de dos naturales:

- $a = 0 \wedge b > 0 \rightarrow mcd(a,b) = b$ // solución base
- $b = 0 \wedge a > 0 \rightarrow mcd(a,b) = a$ // solución base
- $a>0 \wedge b>0 \wedge a \geq b \rightarrow mcd(a,b) = mcd(b,a-b)$ // sol. recurrente
- $a>0 \wedge b>0 \wedge b \geq a \rightarrow mcd(a,b) = mcd(a,b-a)$ // sol. recurrente

Un segundo método de cálculo del m.c.d. (algoritmo de Euclides):

- $b = 0 \wedge a > 0 \rightarrow mcd(a,b) = a$ // solución base
- $b > 0 \rightarrow mcd(a,b) = mcd(b,a\%b)$ // sol. recurrente

Primer diseño de la función ***mcd(a,b)***:

- $a = 0 \wedge b>0 \rightarrow mcd(a,b) = b$ // solución base
- $b = 0 \wedge a>0 \rightarrow mcd(a,b) = a$ // solución base
- $a>0 \wedge b>0 \wedge a \geq b \rightarrow mcd(a,b) = mcd(b,a-b)$ // sol. recurrente
- $a>0 \wedge b>0 \wedge b \geq a \rightarrow mcd(a,b) = mcd(a,b-a)$ // sol. recurrente

```
// Pre: a >= 0 AND b >= 0 AND (a != 0 OR b != 0)
// Post: mcd(a,b) = M AND (a % M = 0) AND (b % M = 0) AND M >= 1
//       AND (PT alfa EN Nat.
//              a % alfa != 0 OR b % alfa ! =0 OR M >= alfa)
int mcd (const int a, const int b) {
   if (a==0) { /* a=0 AND b>0 */  return b; }   // solución base
   else if (b==0) { /* b=0 AND a>0 */  return a; }   // sol. base
   else if (a>=b) { /* b>=0 AND a-b>=0 AND ((b!=0) OR (a-b!=0) */
       return mcd(b, a-b);  }              // sol. recurrente
   else { /* a>=0 AND b-a>=0 AND ((a!=0) OR (b-a!=0) */
       return mcd(a,b-a);  }             // sol. recurrente
}
```

```
// Pre: a >= 0 AND b >= 0 AND (a != 0 OR b != 0)
// Post: mcd(a,b) = M AND (a % M = 0) AND (b % M = 0) AND M >= 1
//       AND (PT alfa EN Nat.
//              a % alfa != 0 OR b % alfa ! =0 OR M >= alfa)
int mcd (const int a, const int b) {
   if (a == 0) { // a = 0 AND b > 0
       return b;                        // solución base
   }
   else if (b == 0) { // b = 0 AND a > 0
       return a;                        // solución base
   }
   else if (a >= b) { // b>=0 AND a-b>=0 AND ((b!=0) OR (a-b!=0)
       return mcd(b, a - b);            // solución recurrente
   }
   else { // a>=0 AND b-a>=0 AND ((a!=0) OR (b-a!=0)
       return mcd(a, b - a);            // solución recurrente
   }
}
```

Segundo diseño de la función ***mcd(a,b)***:

- $b = 0 \wedge a > 0 \rightarrow mcd(a,b) = a$ // solución base
- $b > 0 \rightarrow mcd(a,b) = mcd(b,a \% b)$ // solución recurrente

```
// Pre: a >= 0 AND b >= 0 AND (a != 0 OR b != 0)
// Post: mcd(a,b) = M AND (a % M = 0) AND (b % M = 0) AND M >= 1
//       AND (PT alfa EN Nat.
//                a % alfa != 0 OR b % alfa != 0 OR M >= alfa)
int mcd (const int a, const int b) {
    if (b == 0) {
        // a > 0 AND b = 0
        return a;                          //solución base
    }
    else {
        // b >= 0 AND a % b >= 0 AND (b != 0 OR a %b != 0)
        return mcd(b, a % b);              // solución recurrente
    }
}
```

3.b) Una función de recorrido de un vector

```
// Pre: n >= 0  AND  n ≤ #v
// Post: cuenta(v,n,minimo,maximo) =
//            (NUM alfa EN [0,n-1].
//                  v[alfa] >= minimo AND v[alfa] <= maximo)
int cuenta (const double v[], const int n,
            const double minimo, const double maximo) {
   if (…?…)  {
       …?…                      // unas soluciones
   }
   else {
       …?…                      // otras soluciones
   }
}
```

```
// Pre: n >= 0  AND  n ≤ #v
// Post: cuenta(v,n,minimo,maximo) =
//            (NUM alfa EN [0,n-1].
//                  v[alfa] >= minimo AND v[alfa] <= maximo)
int cuenta (const double v[], const int n,
            const double minimo, const double maximo) {
   if (n == 0)  {

       …?…                          // unas soluciones

   }
   else {

       …?…                          // otras soluciones

   }
}
```

```
// Pre: n >= 0  AND  n ≤ #v
// Post: cuenta(v,n,minimo,maximo) =
//            (NUM alfa EN [0,n-1].
//                  v[alfa] >= minimo AND v[alfa] <= maximo)
int cuenta (const double v[], const int n,
            const double minimo, const double maximo) {
   if (n == 0) {
       //  n = 0
       return 0;                  // solución base
   }
   else {
       …?…                        // otras soluciones
   }
}
```

```
// Pre: n >= 0  AND  n ≤ #v
// Post: cuenta(v,n,minimo,maximo) =
//            (NUM alfa EN [0,n-1].
//                  v[alfa] >= minimo AND v[alfa] <= maximo)
int cuenta (const double v[], const int n,
            const double minimo, const double maximo) {
   if (n == 0) {  //  n = 0
       return 0;                  // solución base
   }
   else {                         // dos soluciones recurrentes
       if (v[n-1] >= minimo && v[n-1] <= maximo) {
          //  n - 1 >= 0  AND  n - 1 ≤ #v
          return 1 + cuenta(v, n-1, minimo, maximo);
       }
       else {
          //  n - 1 >= 0  AND  n - 1 ≤ #v
          return cuenta(v, n-1, minimo, maximo);
       }
   }
}
```

3.c) Una función de búsqueda en un vector

```
// Pre: n >= 0  AND  n ≤ #v
// Post: esta(v,n,dato) = (EX alfa EN [0,n-1].v[alfa] = dato)
bool esta (const int v[], const int n, const int dato) {
   if (…?…)  {
       …?…                          // unas soluciones
   }
   else {
       …?…                          // otras soluciones
   }
}
```

```
// Pre: n >= 0  AND  n ≤ #v
// Post: esta(v,n,dato) = (EX alfa EN [0,n-1].v[alfa] = dato)
bool esta (const int v[], const int n, const int dato) {
   if (n == 0) {
       …?…                          // unas soluciones
   }
   else {
       …?…                          // otras soluciones
   }
}
```

```
// Pre: n >= 0  AND  n ≤ #v
// Post: esta(v,n,dato) = (EX alfa EN [0,n-1].v[alfa] = dato)
bool esta (const int v[], const int n, const int dato) {
   if (n == 0) {
       // n = 0
       return false;                    // solución base
    }
    else {
       // n > 0
       …?…                             // otras soluciones
    }
}
```

```
// Pre: n >= 0  AND  n ≤ #v
// Post: esta(v,n,dato) = (EX alfa EN [0,n-1].v[alfa] = dato)
bool esta (const int v[], const int n, const int dato) {
   if (n == 0) {
       // n = 0
       return false;                        // solución base
    }
    else {
       // n > 0
       if (v[n-1] == dato) {
           return true;                     // solución base
       }
       else {
           // n - 1 >= 0  AND  n - 1 ≤ #v
           return esta(v, n-1, dato);       // solución recursiva
       }
   }
}
```

3.d) Una función de comparación de dos vectores

```
// Pre: n >= 0  AND  n ≤ #v1  AND  n ≤ #v2
// Post: iguales(v1,v2,n) =
//                  (PT alfa EN [0,n-1].v1[alfa] = v2[alfa])
bool iguales (const int v1[], const int v2[], const int n) {
   if (…?…) {
      …?…                                  // unas soluciones
   }
   else {
      …?…                                  // otras soluciones
   }
}
```

```
// Pre: n >= 0  AND  n ≤ #v1  AND  n ≤ #v2
// Post: iguales(v1,v2,n) =
//                  (PT alfa EN [0,n-1].v1[alfa] = v2[alfa])
bool iguales (const int v1[], const int v2[], const int n) {
   if (n == 0) {
      // n = 0
      return true;                          // solución base
   }
    else {
       // n > 0
        …?…                                 // otras soluciones
   }
}
```

```
// Pre: n >= 0  AND  n ≤ #v1  AND  n ≤ #v2
// Post: iguales(v1,v2,n) =
//                  (PT alfa EN [0,n-1].v1[alfa] = v2[alfa])
bool iguales (const int v1[], const int v2[], const int n) {
   if (n == 0) {
      // n = 0
      return true;                              // solución base
   }
   else {
      // n > 0
      if (v1[n-1] != v2[n-1]) {
         return false;                          // solución base
      }
      else {
         // n - 1 >= 0  AND  n - 1 ≤ #v1  AND  n - 1 ≤ #v2
         return iguales(v1, v2, n-1);           // solución recursiva
      }
   }
}
```

3.e) Una función que modifica los elementos de un vector

```
// Pre: n >= 0   AND  n ≤ #v  AND  v = Vo
// Post: (PT alfa EN [0,n-1].(Vo[alfa] = x -> v[alfa] = y) AND
//                          (Vo[alfa] != x -> v[alfa] = Vo[alfa]) )
void sustituir (int v[], const int n, const int x, const int y) {
   if (…?…) {
      …?…                                  // unas soluciones
   }
   else {
      …?…                                  // otras soluciones
   }
}
```

```
// Pre: n >= 0   AND  n ≤ #v  AND  v = Vo
// Post: (PT alfa EN [0,n-1].(Vo[alfa] = x -> v[alfa] = y) AND
//                          (Vo[alfa] != x -> v[alfa] = Vo[alfa]) )
void sustituir (int v[], const int n, const int x, const int y) {
   if (n > 0) {
     // n > 0
      …?…                                  // otras soluciones
   }
   else {
      /* ninguna acción */                  // solución base
   }
}
```

```
// Pre: n >= 0  AND  n ≤ #v  AND  v = Vo
// Post: (PT alfa EN [0,n-1].(Vo[alfa] = x -> v[alfa] = y) AND
//                          (Vo[alfa] != x -> v[alfa] = Vo[alfa]) )
void sustituir (int v[], const int n, const int x, const int y) {
   if (n > 0) {
      // n > 0
      if (v[n-1] == x) {
         v[n-1] = y;
      }
      // n - 1 >= 0  AND  n - 1 ≤ #v  AND
      // (Vo[n-1] = x -> v[n-1] = y)  AND
      // (Vo[n-1] != x -> v[n-1] = Vo[n-1])  AND
      // (PT alfa EN [0,n-2].v[alfa] = Vo[alfa])
      sustituir(v, n-1, x, y);      // solución recursiva
   }
}
```

3.f) Una función interactiva

La función ***leerRespuesta****(pregunta, minimo, maximo)* devuelve la primera respuesta válida proporcionada por el operador:

```
Escriba un número del intervalo [1,49]: 14
```

La función devuelve el valor 14.

Y en este nuevo caso:

```
Escriba un número del intervalo [1,49]: 50
Escriba un número del intervalo [1,49]: 0
Escriba un número del intervalo [1,49]: -13
Escriba un número del intervalo [1,49]: 52202
Escriba un número del intervalo [1,49]: 37
```

La función devuelve ahora el valor 37.

```
// Pre: cierto
// Post: Presenta [pregunta] por pantalla, lee la respuesta
//       del operador y, si la respuesta es un entero comprendido
//       en [minimo,maximo], entonces devuelve ese valor; en caso
//       contrario reitera los pasos anteriores.
int leerRespuesta (const char pregunta[], const int minimo,
                   const int maximo) {
   // Plantea la pregunta
   cout << pregunta << flush;
   int  respuesta;
   // Lee la respuesta
   cin >> respuesta;
   if (…?…) {
      …?…                                  // unas soluciones
   }
   else {
      …?…                                  // otras soluciones
   }
}
```

```
// Pre: cierto
// Post: Presenta [pregunta] por pantalla, lee la respuesta
//       del operador y, si la respuesta es un entero comprendido
//       en [minimo,maximo], entonces devuelve ese valor; en caso
//       contrario reitera los pasos anteriores.
int leerRespuesta (const char pregunta[], const int minimo,
                   const int maximo) {
   // Plantea la pregunta
   cout << pregunta << flush;
   int  respuesta;
   // Lee la respuesta
   cin >> respuesta;
   if (respuesta >= minimo && respuesta <= maximo) {
      return respuesta;                    // solución base
   }
   else {
      …?…                                  // otras soluciones
   }
}
```

```
// Pre: cierto
// Post: Presenta [pregunta] por pantalla, lee la respuesta
//       del operador y, ..., reitera los pasos anteriores.
int leerRespuesta (const char pregunta[], const int minimo,
                   const int maximo) {
   // Plantea la pregunta y lee la respuesta
   cout << pregunta << flush;
   int  respuesta;
   cin >> respuesta;
   if (respuesta >= minimo && respuesta <= maximo) {
      return respuesta;                    // solución base
   }
   else {                                  // solución recursiva
      const int MAX_LONG = 132;
      char linea[MAX_LONG];
      cin.getline(linea, MAX_LONG);     // desprecia resto línea
      return leerRespuesta(pregunta, minimo, maximo);
   }
}
```