Programación 2

Tema 2.
Diseño recursivo de algoritmos

Departamento de
Informática e
Ingeniería de Sistemas
Universidad Zaragoza

Tema 2: Diseño recursivo

- Lección3: **Introducción a la recursividad**
- Lección 4: **Metodología de diseño de algoritmos recursivos**
- Lección 5: **Diseño de algoritmos recursivos por inmersión**
- Lección 6: **Diseño de algoritmos recursivos que trabajan con ficheros**

Programación 2

Lección 3. Introducción a la recursividad

Departamento de
Informática e
Ingeniería de Sistemas
Universidad Zaragoza

Definiciones recursivas

- Una definición es recursiva si en ella se hace referencia a lo definido.
- Una definición recursiva consta de una base que describe una parte de la realidad a definir y de una recurrencia que permite completar la definición a partir de lo previamente definido.
- Ejemplos...

Definiciones recursivas

- El conjunto de los números naturales:

Base de la definición: El **0** es un número natural

Recurrencia: Si **n** es un número natural
entonces **n+1** es también un número natural

De la definición anterior se deduce que el conjunto de los números naturales es { 0, 1, 2, 3, 4, ... }

Definiciones recursivas

- El conjunto de los números pares (enteros):

Base de la definición: El **0** es un número par
Recurrencia: Si **n** es un número par
entonces **n+2** es también un número par
Recurrencia: Si **n** es un número par
entonces **n-2** es también un número par

Definiciones recursivas

- El conjunto de los múltiplos de 7:

Base: El **...** es un número múltiplo de 7

Recurrencia: Si ... es un múltiplo de 7 entonces

... es también múltiplo de 7

De la definición anterior debe deducirse que el conjunto de los números múltiplos de 7 es { ..., -21, -14, -7, 0, 7, 14, 21, ... }

- ¿El conjunto de los números capicúas?

Definiciones recursivas

- Estamos definiendo conjuntos, pero también se pueden definir **cálculos.**
- Ejemplo: **cálculo del factorial:**

Base de la definición: El factorial de 0 es igual a 1, es decir, $0!=1$

Recurrencia: Para $n>0$ se define el factorial de n del modo siguiente $n! = n \times (n-1)!$

De la definición anterior se deduce que:

$0! = 1$ $1! = 1$ $2! = 2$ $3! = 6$ $4! = 24$ $5! = 120$ Etc., etc.

Algoritmos recursivos

- Basándonos en la definición podemos diseñar un método que calcule el factorial (sin bucles!):

```
/* Pre: n ≥ 0 ∧ n = X */
/* Post: factorial(n) = (∏α∈[1,X].α)   */
public int factorial (int n) {
   if (n==0)
      return 1;
   else
      return n*factorial(n-1);
}
```

Departamento de
Informática e
Ingeniería de Sistemas
Universidad Zaragoza

Algoritmos recursivos

- Se dice que un algoritmo es recursivo si su ejecución provoca, de forma directa o indirecta, la invocación una o más veces del propio algoritmo.
- ¿Qué ocurre durante la ejecución del algoritmo recursivo?

Algoritmos recursivos

- Se observa que:
  - Cada invocación de un algoritmo a sí mismo equivale a la generación de un ***instancia de ejecución*** diferente de ese mismo algoritmo, que a todos los efectos equivale a la invocación de un algoritmo distinto
  - Para cada instancia se crea un ***registro de activación***, que se va almacenando en una estructura de datos pila, en la que se guarda información sobre el punto de retorno del flujo de control de la ejecución tras la terminación de la llamada, los valores de los parámetros, de las variables locales y el resultado devuelto
  - Cada instancia maneja su propio ***léxico local*** con independencia del resto de las llamadas, de modo que los identificadores tanto de la lista de parámetros como de las variables locales y el propio nombre del algoritmo, hacen referencia en cada instancia a objetos diferentes.

Algoritmos recursivos

- Definición recursiva de la sucesión de Fibonacci:

```
Base de la definición: El 0 es primer elemento de la
   sucesión de Fibonacci
Base de la definición: El 1 es segundo elemento de la
   sucesión de Fibonacci
Recurrencia: Para i>2 el i-ésimo elemento de la
   sucesión de Fibonacci es igual a la suma de los
   dos elementos que le preceden en dicha sucesión
```

De la definición anterior se deduce que los números que integran la sucesión de números de Fibonacci son los siguientes:

```
{ 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ... }
```

Algoritmos recursivos

- Ejemplo: método que calcula la sucesión de Fibonacci:

```
/* Pre: n > 0 */
/* Post: (n = 1 ∨ n = 2 → fibonacci(n) = n - 1)
         ∧ (n > 2 → fibonacci(n) =
                        fibonacci(n-2) + fibonacci(n-1))
*/
public int fibonacci (int n) {
   if (n <= 2)
        return n-1;
   else
        return fibonacci(n-2) + fibonacci(n-1);
}
```

Algoritmos recursivos

- En este caso la invocación de la función genera dos invocaciones recursivas.
- ¿Qué ocurre en la ejecución del algoritmo?
- ¿Es eficiente?

Algoritmos recursivos

- Tipos de recursividad:
  - Múltiple: se invoca a sí mismo más de una vez dentro de una misma instancia (ej. Fibonacci)
  - Lineal: se invoca a sí mismo como mucho una vez dentro de una misma instancia (ej. Factorial)
  - Lineal final: se invoca a sí mismo como mucho una vez dentro de una misma instancia y esta llamada recursiva es la última acción que efectúa como parte del cuerpo de su definición (el algoritmo concluye tras la llamada recursiva)
  - Indirecta: el algoritmo A invoca a otro que incluye una invocación a A

Programación 2

Lección 4. Metodología de diseño de algoritmos recursivos

Departamento de
Informática e
Ingeniería de Sistemas
Universidad Zaragoza

Diseño recursivo

- La clave:
  - Debemos identificar uno o varios casos que constituyan la base del problema
  - Planteamiento recursivo: obtener la solución a partir de la solución para datos más simples.

Diseño recursivo

- Cálculo del máximo común divisor de dos enteros:

```
b = 0   → mcd(a,b) = a                         // solución base
a≥0 ∧ b>0 → mcd(a,b) = mcd(b, a%b)             // sol. recurrente
```

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

Departamento de
Informática e
Ingeniería de Sistemas
Universidad Zaragoza

Diseño recursivo

- Esquema de un método recursivo:

```
/* Pre: precondicion */
/* Post: postcondicion */
public tipoDato nombreMétodo (lista de parámetros) {
  if (CB1) {  B1; }        // solución base
  else if (CB2) {  B2; }   // solución base
  else if (…) {  …  }      // solución base
  else if (CBp) {  Bp; }   // solución base

  else if (CR1) {  R1; }   // solución recurrente
  else if (CR2) {  R2; }   // solución recurrente
  else if (…) {  …  }      // solución recurrente
  else if (CRq) {  Rq; }   // solución recurrente
}
```

Diseño recursivo

- Cálculo del máximo común divisor de dos enteros:

```
a = 0   → mcd(a,b) = b                        // solución base
b = 0   → mcd(a,b) = a                        // solución base
a>0 ∧ b>0 ∧ a≥b → mcd(a,b) = mcd(b,a-b)  // sol. recurrente
a>0 ∧ b>0 ∧ b≥a → mcd(a,b) = mcd(a,b-a)  // sol. Recurrente
```

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

Diseño recursivo

- ¿En qué difieren el planteamiento recursivo y el iterativo?

$$
\begin{tikzcd}
P \arrow[r, "\text{sobre}"] \arrow[d] & D \arrow[d] \\
(\,p' \arrow[r, "\text{sobre}"] & d'\,)\ {*}n
\end{tikzcd}
$$

Planteamiento iterativo

$$
\begin{tikzcd}
P \arrow[r, "\text{sobre}"] \arrow[d] & D \arrow[d] \\
P \arrow[r, "\text{sobre}"] & d
\end{tikzcd}
$$

Planteamiento recursivo

Departamento de
Informática e
Ingeniería de Sistemas
Universidad Zaragoza
1542

Diseño recursivo

- La recursión es muchas veces la forma más natural de describir funciones y tipos de datos.
- Su uso genera programas más compactos que las correspondientes versiones iterativas.
- Familia de lenguajes funcionales.
- Sin embargo su uso no siempre es recomendable...

Diseño recursivo

- Ejemplo: método que calcula la sucesión de Fibonacci:

```
/* Pre: n > 0 */
/* Post: (n = 1 ∨ n = 2 → fibonacci(n) = n - 1)
          ∧ (n > 2 → fibonacci(n) =
                         fibonacci(n-2) + fibonacci(n-1))
*/
public int fibonacci (int n) {
   if (n <= 2)
       return n-1;
   else
       return fibonacci(n-2) + fibonacci(n-1);
}
```

Diseño recursivo + tablas

- Consulta de información almacenada en una tabla:

```
/* Pre: desde>=0 AND desde<=hasta AND hasta<=T.length-1 */
/* Post: cuenta(T,desde,hasta,minimo,maximo) =
          (NUM alfa EN [desde,hasta].
                          T[alfa]>=minimo AND T[alfa]<=maximo) */
public int cuenta (double[] T, int desde, int hasta,
                   double minimo, double maximo) {…}
```

- Análisis de casos:

–Si se satisface que *desde* = *hasta* basta comprobar si el elemento *T[desde]* ha de ser contabilizado (base)

–Si se satisface que *desde* < *hasta* hay comprobar si el elemento *T[desde]* ha de ser contabilizado y hay que reaplicar el método *cuenta* sobre la subtabla *T[desde+1..hasta]* *(recurrencia)*

Diseño recursivo + tablas

```
/* Pre: desde>=0 AND desde<=hasta AND hasta<=T.length-1 */
/* Post: cuenta(T,desde,hasta,minimo,maximo) =
          (NUM alfa EN [desde,hasta].
                          T[alfa]>=minimo AND T[alfa]<=maximo) */
public int cuenta (double[] T, int desde, int hasta,
                   double minimo, double maximo) {
   if (desde==hasta)                 // soluciones base
       if (T[desde]>=minimo && T[desde]<=maximo)  return 1;
       else return 0;
   else                              // dos soluciones recurrentes
       if (T[desde]>=minimo && T[desde]<=maximo)
          return 1 + cuenta(T, desde+1, hasta, minimo, maximo);
       else
          return cuenta(T, desde+1, hasta, minimo, maximo);
}
```

Diseño recursivo + tablas

- Búsqueda en una tabla:

```
/* Pre: n<=T.length */
/* Post: está(T,n,dato) = (EX alfa EN [0,n-1].T[alfa]=dato)  */
public int está (Dato[] T, int n, Dato dato) {…}
```

- Análisis de casos:
  - Si se satisface que $n \leq 0$ se concluye que el valor *dato* no puede estar almacenado en la subtabla *T[0..n-1]* (base)
  - Si se satisface que $n > 0$ y que $T[n - 1] = dato$ se concluye que el valor *dato* está almacenado en la subtabla *T[0..n-1]* (base)
  - Si se satisface que $n > 0$ y que $T[n-1] \neq dato$ se concluye que el valor *dato* estará almacenado en la subtabla *T[0..n-1]* si y sólo si lo está en la subtabla *T[0..n-2]* (recurrencia)

Departamento de Informática e Ingeniería de Sistemas
Universidad Zaragoza

Diseño recursivo + tablas

```
/* Pre: n<=T.length */
/* Post: está(T,n,dato) = (EX alfa EN [0,n-1].T[alfa].equals(dato))  */
public int está (Dato[] T, int n, Dato dato) {
   if (n<=0)
        return false;                              // solución base
   else (n>0)
        if (T[n-1].equals(dato))
           return true;                            // solución base
        else
           return está(T, n-1, dato);              // solución recurrente
}
```

Diseño recursivo + tablas

- Comparación de tablas:

```
/* Pre: desde>=0 AND hasta<=T1.length-1 AND hasta<=T2.length-1 */
/* Post: iguales(T1,T2,desde,hasta) =
                 (PT alfa EN [desde,hasta].T1[alfa]=T2[alfa])  */
public boolean iguales (int[] T1, int[] T2,
                        int desde, int hasta) {…}
```

- Análisis de casos:
  - Si *desde* > *hasta* entonces el intervalo a considerar define un conjunto vacío de índices y, por lo tanto, ambas tablas almacenan datos idénticos (base)
  - Si *desde* ≠ *hasta* y *T1*[*desde*] ≠ *T2*[*desde*] se concluye que, para el intervalo *[desde,hasta]*, almacenan datos que no todos son idénticos (base)
  - Si *desde* ≠ *hasta* y *T1*[*desde*] = *T2*[*desde*] se concluye que, para el intervalo *[desde,hasta]*, ambas tablas almacenan datos idénticos si y sólo si para el intervalo *[desde+1,hasta]* ambas tablas almacenan datos idénticos (*re*currencia)

Diseño recursivo + tablas

```
/* Pre: desde>=0 AND hasta<=T1.length-1 AND hasta<=T2.length-1 */
/* Post: iguales(T1,T2,desde,hasta) =
                 (PT alfa EN [desde,hasta].T1[alfa]=T2[alfa])  */
public boolean iguales (int[] T1, int[] T2,
                         int desde, int hasta) {
   if (desde>hasta)
        return true;                              // solución base
   else (desde<=hasta)
      if (T[desde]!=T2[desde])
          return false;                         // solución base
      else
         return iguales(T1, T2, desde+1, hasta);  // sol. recur.

}
```

Departamento de
Informática e
Ingeniería de Sistemas
Universidad Zaragoza

Diseño recursivo + tablas

- Modificación de la información de una tabla:

```
/* Pre: desde>=0 AND hasta<=T.length-1 AND T=To*/
/* Post: (PT alfa EN [desde,hasta].
            (To[alfa]=x -> T[alfa]=y)
            AND (To[alfa]!=x -> T[alfa]=To[alfa]) )  */
public void sustituir (int[] T1, int desde, int hasta
                           int x, int y) {…}
```

Diseño recursivo + tablas

- Análisis de casos:
  - –Si *desde* > *hasta* entonces no es necesario programar ninguna acción ya que ningún elemento de la subtabla *T[desde,hasta]* ha de ser modificado (base)
  - –Si *desde* <= *hasta* y *T*[*desde*] = *x* debe asignarse al elemento *T[desde]* el valor *y* y deben sustituirse en la subtabla *T[desde+1,hasta]* todos aquellos elementos cuyo valor sea igual al de parámetro *x* por el valor del parámetro *y* (base)
  - –Si *desde* <= *hasta* y *T*[*desde*] ≠ *x* basta con sustituir en la subtabla *T[desde+1,hasta]* todos aquellos elementos cuyo valor sea igual al del parámetro *x* por el valor del parámetro *y*.

Diseño recursivo + tablas

```
/* Pre: desde>=0 AND hasta<=T.length-1 AND T=To*/
/* Post: (PT alfa EN [desde,hasta].
            (To[alfa]=x -> T[alfa]=y)
            AND (To[alfa]!=x -> T[alfa]=To[alfa]) )  */
public void sustituir (int[] T1, int desde, int hasta
                           int x, int y) {
   if (desde>hasta)                        // solución base
      {}
   else /*desde<=hasta*/ {
      if (T[desde]==x) T[desde] = y;       // sol. recurrente
      sustituir(T1, desde+1, hasta, x, y)  // sol. recurrente
   }
}
```