Universidad de **Valladolid**
Departamento de informática
Campus de Segovia

Estructura de datos

Tema 1: Recursividad

Prof. Montserrat Serrano Montero

ÍNDICE

•Conceptos básicos
•Ejemplos recursivos
•Recursividad indirecta
•Métodos para problemas recursivos
•Recursividad e iteración

CONCEPTOS BÁSICOS

- Un **objeto** es **recursivo** cuando se define en función de sí mismo, es decir, interviene en su propia definición.

- La recursividad es la propiedad mediante la cual un subprograma o rutina puede llamarse a sí mismo.

- Utilizando la recursividad, la resolución de un problema se reduce a uno esencialmente igual pero algo menos complejo.

- Características que deben cumplir los problemas recursivos:
  - La recursividad debe terminar alguna vez: **caso base**
  - Cada **nueva formulación** estamos **más cerca del caso final (o base)**.

EJEMPLO DE PROBLEMA RECURSIVO

- FACTORIAL DE UN NÚMERO ENTERO POSITIVO

Sin recursividad:

$$n! = n\ (n-1)\ (n-2)........1$$

$$(n-1)! = (n-1)\ (n-2).......1$$

Ley de recurrencia:

$$n! = n\ (n-1)! \quad \text{si } n>0$$

$$0! = 1 \qquad \text{si } n=0$$

IMPLEMENTACIÓN EN PASCAL MEDIANTE FUNCIONES

Función no recursiva o iterativa

```
function factorial (n: word): real;
    var i: word; aux: real;
    begin
    aux := 1;
    for i := 1 to n do aux := i * aux;
    factorial := aux
    end;
```

Función recursiva

```
function factorial (n: word): real;
    begin
    if n > 0
        then factorial := n * factorial (n-1)
        else factorial := 1
    end;
```

PROCESO DE EJECUCIÓN. PILA RECURSIVA

- **Ejecución del programa**:

  – Bajo una llamada recursiva el sistema reserva espacio (**tabla de activación**) donde almacenar una copia de los objetos locales y parámetros del subprograma en ese momento.

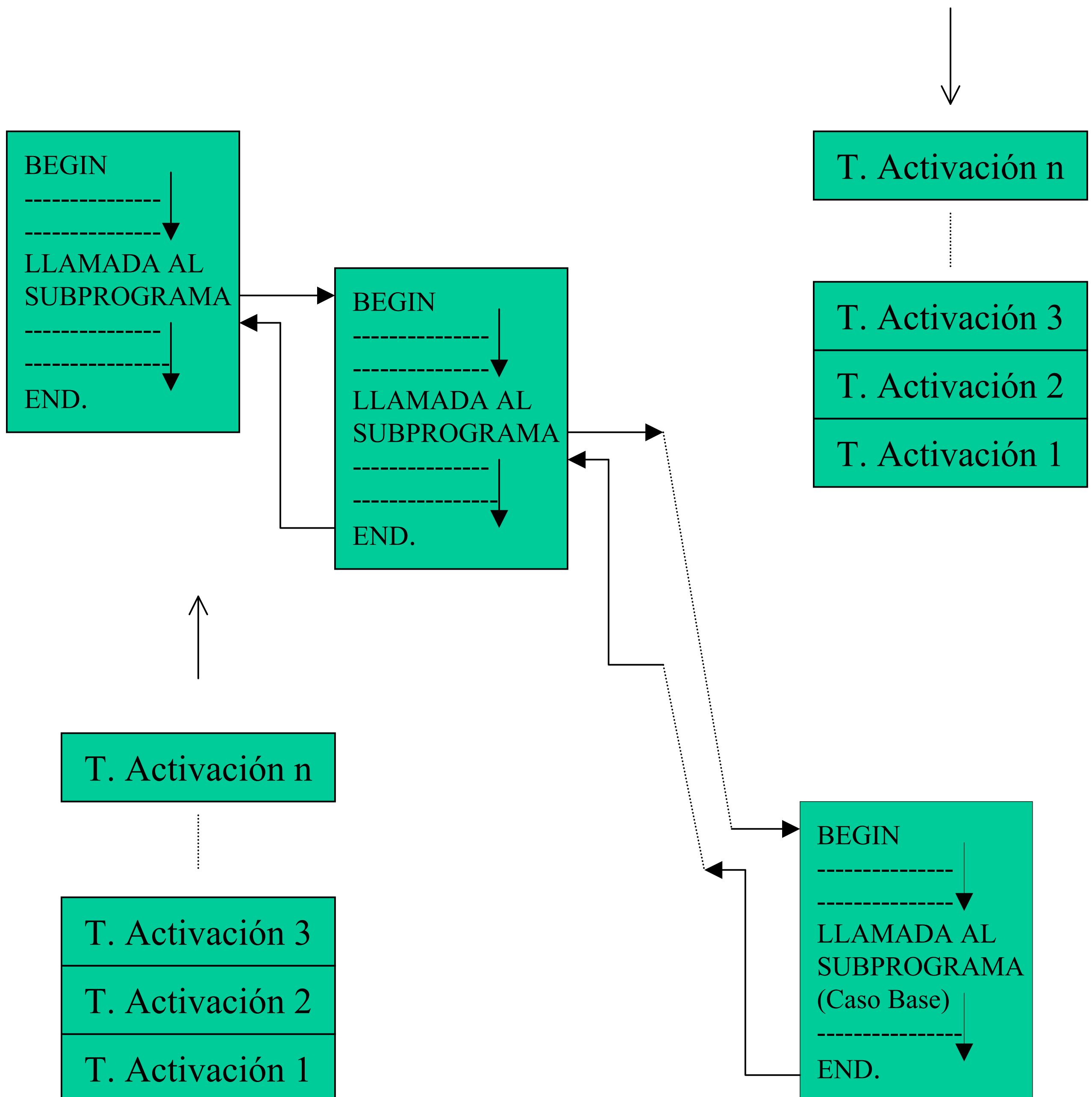
  – La tabla de activación se amontona sobre las llamadas recursivas anteriores formando lo que se conoce como **pila recursiva**.

  – Este proceso termina cuando un nuevo valor del parámetro no produce una nueva llamada recursiva (**se alcanza el caso base**).

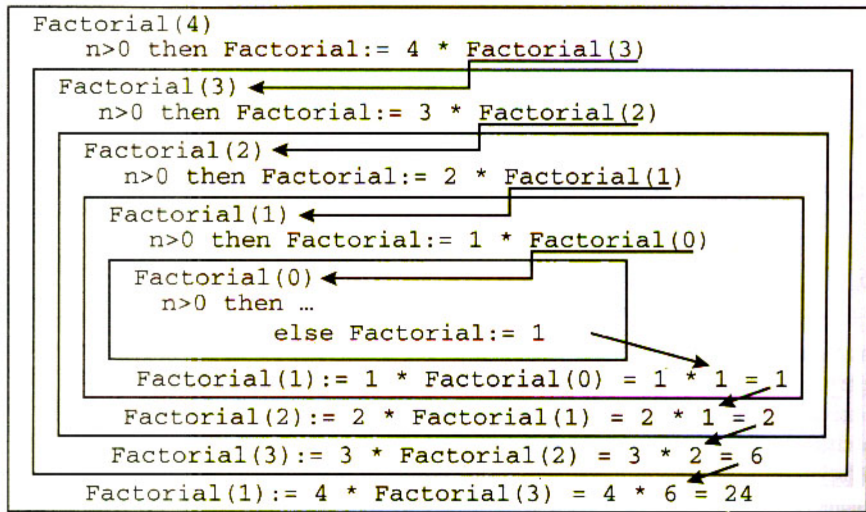
  – Una vez alcanzada esta situación el sistema va liberando el espacio reservado conforme los subprogramas se van ejecutando sobre su tabla de activación.

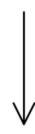
  – Este proceso finaliza con la llamada inicial.

REPRESENTACIÓN GRÁFICA DEL PROCESO DE EJECUCIÓN DE UN PROGRAMA RECURSIVO



PROCESO DE EJECUCIÓN PARA n = 4





factorial (4) = 4* factorial (3)

factorial (4) = 4*(3* factorial (2))

factorial (4) = 4*(3*(2* factorial (1)))

factorial (4) = 4*(3*(2*(1* factorial (0))))

Fac(1)=4*Fac(0)
Fac(2)=4*Fac(1)
Fac(3)=4*Fac(2)
Fac(4)=4*Fac(3)

factorial = Fac

- Para obtener la solución final se deshacen las llamadas anteriores siguiendo el orden inverso (pila recursiva).

factorial (4) = 4*(3*(2*(1*1)))

factorial (4) = 4*(3*(2*1))

factorial (4) = 4*(3*2)

factorial (4) = 4*6 = 24

Fac(1)=4*Fac(0)
Fac(2)=4*Fac(1)
Fac(3)=4*Fac(2)
Fac(4)=4*Fac(3)

EJEMPLO RECURSIVO CON PROCEDIMIENTO

- **Escribir una cadena de caracteres al revés:**

  Ley de recurrencia:

  EscribirInverso(Cadena, Longitud):

  1º.- EscribirUltimo(Cadena, Longitud)

  2º.- EscribirInverso(Cadena, Longitud-1)

  Implementación en Pascal:

```
procedure EscribirInverso (cad:string;lg:integer);
begin
    if lg > 0 then
          begin
              write (cad[lg]);
              EscribirInverso (cad, lg-1);
          end
end;
```

DESARROLLO DE UNA RUTINA RECURSIVA

1. **Análisis del problema:** Obtener la ley de recurrencia

a) Identificar el/los casos base.

b) Identificar el/los casos recursivos que deben tender al caso base.

Ej: Calclar a^n

1	si	$n = 0$
$a * a^{(n-1)}$	si	$n > 0$

2. **Implementación:** Traslado a una sentencia condicional

if <condicion del caso base> **then**
final := <caso base>
else
final :=
final (<valor próximo caso base>)
	if n = 0 **then**
	final := 1
	else
	final := $a * a^{(n-1)}$

DESARROLLO DE UNA RUTINA RECURSIVA

3. Verificación:

- Comprobar que se definen todos los casos base.
- Comprobar que se define todo el dominio.
- La rutina debe invocarse a sí misma con elementos del dominio.
- La llamada recursiva se hace siempre con elementos estrictamente menores, más próximos al caso base.
- Asegurarse que existe una llamada o punto en el que finalizan las llamadas recursivas.
- La recursividad no sólo debe ser finita, sino también pequeña para evitar problemas de saturación de memoria.

RECURSIVIDAD INDIRECTA O MUTUA

- Existen dos tipos o formas de recursividad en TurboPascal:

  1. Directa o simple: Un subprograma se llama a sí mismo directamente. Es el tipo de recursividad empleado en los ejemplos vistos hasta ahora.

  2. Indirecta, mutua o cruzada: Un subprograma A llama a otro subprograma B y éste a su vez llama al subprograma A

- Problema: Una subrutina no puede ser utilizada antes de ser declarada. Este problema, en Pascal, se resuelve mediante la palabra reservada "**forward**". Este identificador indica al compilador que la rutina a la que hace referencia se declarará posteriormente.

RECURSIVIDAD INDIRECTA: DECLARACIÓN

```
PROCEDURE Segundo (parámetros); forward;
PROCEDURE Primero (parámetros);
........
Begin {Primero}
.........
Segundo(parámetros)
.........
End; {Primero}

PROCEDURE Segundo (parámetros);
........
Begin {Segundo}
.........
Primero(parámetros)
.........
End; {Segundo}
```

RECURSIÓN INDIRECTA: EJEMPLO

- Programa que determina la paridad de un entero positivo empleando para ello dos funciones mutuamante recurrentes:

```
function par (n:integer) : boolean;
function impar (n:integer) : boolean;
```

```
program Paridad;
var
   n:integer;
function par(n:integer): boolean; forward;
function impar(n:integer): boolean;
   begin
        if n = 0 then
                impar := false
        else
                impar := par(n-1)
   end;{Impar}
function par (n:integer): boolean;
    begin
        if n=0 then
                par := true
        else
                par := impar(n-1)
   end;{Par}
```

```
begin {Paridad}
    writeln('Introduce un número (negativo para
    terminar) ');
    readln(n);
    while n > 0 do
        begin
          if par (n) then
            writeln(n, '  es par ')
          else
            writeln(n, '  es impar ');
          writeln('Introduce un número (negativo para
    terminar) ');
          readln (n)
        end
end. {Paridad}
```

MÉTODOS PARA LA RESOLUCIÓN DE PROBLEMAS QUE USAN RECURSIVIDAD

Divide y vencerás

- Consiste en dividir el problema en dos o más subproblemas, cada uno de los cuales es similar al original pero más pequeño en tamaño.

- Con las soluciones a los subproblemas se debe poder construir fácilmente una solución del problema general.

- Cada problema se resuelve mediante 2 opciones:

  1. Problema pequeño de inmediata solución.

  2. Aplicar de forma recursiva la técnica *divide y vencerás*.

DIVIDE Y VENCERÁS: EJEMPLO

- Implementación recursiva del procedimiento de búsqueda binaria de un elemento en un vector ordenado:

```
function Bbinaria (A: Vector; elem: integer): integer;
function Busqueda (A: Vector; elem: integer; i,j: integer):
    integer;
var m: integer;
begin {Busqueda}
    if i>j then Busqueda := 0
    else
      begin
        m := (i+j) div 2;
        if A[m] = elem then
          Busqueda := m
        else
          if A[m] > elem then
            Busqueda := Busqueda (A, elem, i , m-1)
          else
            Busqueda := Busqueda (A, elem, m+1,j )
      end;
end; {Busqueda}
begin {Bbinaria}
Bbinaria := Busqueda (A, elem, 1, n)
end; {Bbinaria}
```

MÉTODOS PARA LA RESOLUCIÓN DE PROBLEMAS QUE USAN RECURSIVIDAD

Retroceso (backtracking)

- Basado en el tanteo sistemático, en el cual la solución del problema se divide en pasos y la solución del paso i-ésimo se define en función de la solución del paso i-ésimo+1.

- Las técnicas de retroceso se pueden emplear con la finalidad de encontrar una, todas o la solución óptima al problema planteado.

RECURSIÓN E ITERACIÓN

- Un programa que se invoca a sí mismo se repite un cierto número de veces.
- Por esta razón cualquier cálculo recursivo puede ser expresado como una iteración y viceversa.
- La elección entre uno y otro método vendrá dado por motivos de eficiencia y legibilidad.
  - Un problema recursivo posee una solución más legible si se resuelve mediante un algoritmo recursivo.
  - Una solución recursiva ocupa más memoria que una solución iterativa.

CASOS EN LOS QUE DEBE EVITARSE EL USO DE LA RECURSIVIDAD

- Los algoritmos recursivos en los que distintas llamadas recursivas producen los mismos cálculos es conveniente convertirlos en iterativos.

- La sucesión de Fibonacci: 0, 1, 1, 2, 3, 5, 8, ...
  (Ley de recurrencia)
  - f (0) = 0 si n = 0 y f (1)= 1 si n = 1
  - f (n) = f (n-1) + f (n-2) si n > 1

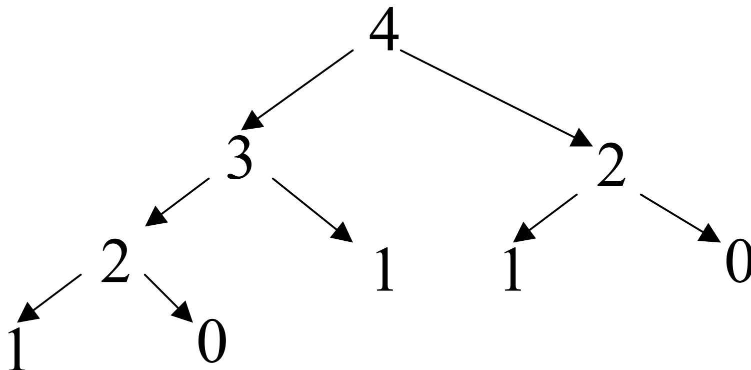
- Implementación en Pascal:

```
function fibonacci (n:integer): integer;
 begin
   if n = 0 then
     fibonacci := 0
   else
     if  n = 1 then fibonacci:= 1
     else fibonacci := fibonacci (n-1)+fibonacci (n-2)
 end; {fibonacci}
```

CASOS EN LOS QUE DEBE EVITARSE EL USO DE LA RECURSIVIDAD

- En ocasiones una llamada recursiva genera por diferentes vías llamadas repetidas.

- Función de Fibonacci: fibonacci (4)



- Esta solución es mala, pues hay que repetir el cálculo de los valores inferiores para distintas ramas del resultado. Al diseñarlo de forma repetitiva el tiempo de ejecución disminuirá.

IMPLEMENTACIÓN ITERATIVA PARA FIBONACCI

```
function fibonacci (n: integer): integer;
  var
    i, j, k: integer;
  begin
     fibonacci := 0; i := 0; j := 1;
     for k := 2 to n do
       begin
         j := j + i; {se queda con el último valor}
         i := j - i; { retiene el penúltimo valor de j}
       end;
    if n > 0 then fibonacci := j
 end;
```

El diseño recursivo tiene complejidad exponencial.

El diseño iterativo tiene complejidad lineal.

- En general, la recursividad debe ser evitada cuando el problema se pueda resolver mediante una estructura iterativa con un tiempo de ejecución similar o más bajo. La recursividad debiera evitarse siempre en subprogramas con recursión en extremo final que calculen valores definidos en forma de relaciones recurrentes simples.

TRANSFORMACIÓN RECURSIVO-ITERATIVA

- Todo algoritmo recursivo puede ser transformado en otro de tipo iterativo, pero para ello a veces se necesita utilizar pilas donde almacenar los cálculos parciales y el estado actual del subprograma recursivo.

- Es posible estandarizar los pasos necesarios para convertir un algoritmo recursivo en iterativo, aunque el algoritmo así obtenido requerirá una posterior labor de optimización.

- Vemos dos tipos de transformaciones:

  a) Recursivas finales

  b) Recursivas no finales

RECURSIVAS FINALES

Esquema recursivo

```
function f (x: t1): t2;
  begin
    if <Condicion (x)> then
      f := <CasoBase>
    else
      f := f (Pred (x))
  end;
```

Esquema iterativo

```
function f (x: t1): t2;
  begin
   while not <Condicion (x)> do
      x := Pred (x);
    f := <CasoBase>
  end;
```

RECURSIVAS FINALES: EJEMPLO

- **Transformar el algoritmo recursivo, que calcula el resto de la división entera de dos enteros, a su correspondiente algoritmo iterativo:**

  Equivalencias:

  x = (a, b)

  t1= integer

  t2 = integer

  <Condicion (a, b)> = a < b

  Pred (a, b) = (a-b, b)

  <CasoBase> = a, si a <b

- **Nota:** Para probar los algoritmos, sustituir mod por otro identificador.

RECURSIVAS FINALES: EJEMPLO

Esquema recursivo

function mod (a,b:integer): integer;
  begin
    if a < b **then**
      mod := a
    else
      mod := mod (a-b,b)
  end;

Esquema iterativo

function mod (a,b:integer): integer;
  begin
    while not (a < b) **do**
      a := a - b;
    mod := a
  end;

RECURSIVAS NO FINALES

Esquema recursivo

```
function f (x: t1): t2;
  begin
    if <Condicion (x)> then
      f := <CasoBase>
    else
      f := C(x, f (Pred (x)))
  end;
```

Esquema iterativo

```
function f (x: t1): t2;
  var result: t2;
  begin
    result := <CasoBase>;
    while not <Condicion (x)> do
      begin
      result := C (result, x);
      x := Pred (x);
      end;
    f := result;
end;
```

RECURSIVAS NO FINALES: EJEMPLO

- **Transformar el algoritmo recursivo, que calcula la suma de los elementos de un vector, a su correspondiente algoritmo iterativo:**

  Equivalencias:

  x = (A, i)

  t1= (vector, integer)

  t2 = integer

  <Condicion (A, i)> = i < 1

  <CasoBase> para (A, 0) = 0

  C (x, f (Pred(x))) = A[i] +Suma (A, i-1)

  C (result, x) = result + A[i]

  Pred (x) = i -1

- **result es la pila donde se almacenan las llamadas.**

RECURSIVAS NO FINALES: EJEMPLO

Esquema recursivo

```
function suma (A: Vector; i: integer): integer;
  begin
    if  i < 1 then
      suma := 0
    else
      suma := A[i] + suma (A, i-1)
  end;
```

Esquema iterativo

```
function suma (A: Vector; i: integer): integer;
  var result: integer;
  begin
   result := 0;
    while not (i < 1) do
      begin
      result := result + A[i];
       i := i-1;
      end;
    suma := result;
 end;
```