Departamento de Lenguajes y Ciencias de la Computación

UNIVERSIDAD DE MÁLAGA

Apuntes para la asignatura

Informática

Facultad de Ciencias (Matemáticas)

http://www.lcc.uma.es/personal/pepeg/mates

Tema 11. Diseño de algoritmos recursivos

Bibliografía

- *Programming with TopSpeed Modula-2*. Barry Cornelius. Addison-Wesley.
- *Pascal y estructuras de datos*. Nell Dale y Susan C. Lilly. McGraw-Hill.
- *Fundamentos de programación*. L. Joyanes. McGraw-Hill.

11.1. Recursión

Una definición es *recursiva* cuando el concepto que se introduce está definido en base a sí mismo. Un gran número de problemas se plantean de forma recursiva. Esto significa que su solución se apoya en la solución del mismo problema pero para un caso más fácil.

En matemáticas es frecuente definir conceptos en términos de sí mismos. Por ejemplo la siguiente definición de factorial de un número *n* positivo es recursiva:

$$n! = \begin{cases} 1, & si \;\; n = 0 \\ n \times (n-1)!, & si \;\; n \neq 0 \end{cases}$$

La definición es recursiva porque estamos definiendo el factorial de *n* como el producto de *n* por el factorial de *n-1*, si *n* es distinto de cero. El concepto que definimos (factorial) aparece en la propia definición. Si queremos calcular, por ejemplo, el factorial de 4 con la definición anterior procederíamos del siguiente modo:

$$4! = 4 \times 3! = 4 \times 3 \times 2! = 4 \times 3 \times 2 \times 1! = 4 \times 3 \times 2 \times 1 \times 0! = 4 \times 3 \times 2 \times 1 \times 1 = 24$$

El siguiente ejemplo es una definición recursiva del sumatorio de los primeros *n* números naturales:

$$\sum_{i=0}^{n} i = \begin{cases} 0, & si \;\; n = 0 \\ n + \sum_{i=0}^{n-1} i, & si \;\; n \neq 0 \end{cases}$$

Un último ejemplo es la definición de una potencia con exponente entero positivo:

$$b^e = \begin{cases} 1, & si \; e = 0 \\ b \times b^{(e-1)}, & si \; e \neq 0 \end{cases}$$

Vemos que la recursividad es una técnica que permite dar definiciones simples y elegantes. En *Modula-2* es posible definir subprogramas recursivos, es decir, subprogramas en cuyo cuerpo aparezcan llamadas a sí mismos.

La definición anterior de la función factorial nos lleva a la siguiente función *Modula-2*:

```
PROCEDURE Factorial (n : CARDINAL) : CARDINAL;
BEGIN
  IF n = 0 THEN
    RETURN 1
  ELSE
    RETURN n * Factorial(n-1)
  END
END Factorial;
```

Obsérvese que hay una llamada a la propia función que se está definiendo (`Factorial`) en la parte ELSE. Esto no presenta ningún problema ya que las reglas de ámbito de *Modula-2* lo permiten.

Por otro lado, la función presenta más de una instrucción RETURN. Esta práctica es común a la hora de escribir funciones recursivas. En el caso de funciones no recursivas, es mejor estilo de programación utilizar una única sentencia RETURN al final del subprograma.

Veamos ahora qué es lo que ocurre si un programa ejecuta la sentencia `IO.WrCard(Factorial(3), 0)` que llama a la función recursiva anterior:

1) El subprograma `WrCard` necesita conocer el valor de su argumento `Factorial(3)` para poder imprimirlo por lo cual se entra en la función `Factorial`. Como el argumento está declarado por valor se crea una nueva variable `n` y se asigna a ésta el valor 3. Llamemos a esta variable $n^{(1}$.

2) Como $n^{(1}$ es distinto de cero el valor que devuelve la llamada a `Factorial(3)` es la correspondiente al ELSE, o sea, $n^{(1}$ `* Factorial(`$n^{(1}$`-1)`. Para poder calcular este producto es necesario calcular `Factorial(`$n^{(1}$`-1)`que provoca una nueva llamada a la función `Factorial`, en este caso con argumento 2, ya que $n^{(1}$ vale 3.

3) Se vuelve a entrar en la función `Factorial`, por lo que se crea una nueva variable que llamaremos $n^{(2}$ y cuyo valor es 2. Como el valor de $n^{(2}$ es distinto de cero esta llamada devuelve $n^{(2}$ `* Factorial(`$n^{(2}$`-1)`. De nuevo para calcular este producto se debe conocer el valor de la segunda parte lo que provoca una tercera llamada a Factorial con argumento 1, ya que $n^{(2}$ vale 2.

4) Se crea $n^{(3}$ con valor 1, y se devuelve $n^{(3}$ `* Factorial(`$n^{(3}$`-1)`. Se produce una ultima llamada a factorial con argumento 0.

5) Al entrar en la función `Factorial` se crea una nueva variable $n^{(4}$ con valor 0. Como $n^{(4}$ vale 0 la función `Factorial` devuelve el valor indicado en la parte IF, o sea, 1.

6) El producto del punto 4) puede ser calculado, por lo que la tercera llamada a factorial devuelve el valor $n^{(3}$ `* Factorial(`$n^{(3}$`-1)`= 1 * 1 = 1.

7) El producto del punto 3) puede ser calculado, por lo que la segunda llamada a factorial devuelve el valor $n^{(2}$ `* Factorial(`$n^{(2}$`-1)`= 2 * 1 = 2.

8) El producto del punto 2) puede ser calculado, por lo que la primera llamada a factorial devuelve el valor $n^{(1}$ `* Factorial(`$n^{(1}$`-1)`= 3 * 2 = 6.

9) El valor de Factorial(3) ha sido calculado y puede ser impreso por pantalla.

11.1.1 Verificación de los subprogramas recursivos

Hemos visto que la ejecución de un subprograma recursivo hace, en general, que éste sea llamado varias veces. Para asegurar que este proceso termina es necesario que el subprograma recursivo definido cumpla las siguientes condiciones:

1) Hay al menos una posible llamada al subprograma que produce un resultado sin provocar una nueva llamada recursiva. A esto se le llama *caso base* de la recursión. En el caso de la función `Factorial` el caso base es n=0.

   La siguiente definición de la función `Factorial` no es correcta ya que no posee caso base:

```
PROCEDURE Factorial (n : CARDINAL) : CARDINAL;
BEGIN
   RETURN n * Factorial(n-1)
END Factorial;
```

2) Todas las llamadas recursivas se refieren a un caso que es más cercano al caso base. En el caso de la función `Factorial`, cada llamada disminuye en uno el valor del argumento, por lo que éste se va acercando a cero, que es el caso base.

   A partir de la expresión $n! = (n-1)! \times n$, y la sustitución $n = m+1$ se obtiene que $(m+1)! = m! \times (m+1)$, de donde despejando se deduce $m! = (m+1)! \,/\, (m+1)$. Con esta expresión podemos construir el siguiente subprograma recursivo:

```
PROCEDURE Factorial (m : CARDINAL) : CARDINAL;
```

```
BEGIN
   IF m = 0 THEN
      RETURN 1
   ELSE
      RETURN Factorial(m+1) DIV (m+1)
   END
END Factorial;
```

Este subprograma presenta un caso base, sin embargo no es correcto ya que no cumple la condición 2). Cada llamada recursiva incrementa en uno el valor del argumento por lo que éste se va alejando del caso base y la recursión nunca termina.

3) El caso base debe acabar alcanzándose.

   El siguiente subprograma cumple las condiciones 1) y 2), pero no es correcto. Si lo llamamos con un argumento cuyo valor sea impar, el caso base nunca se alcanza.

```
PROCEDURE Factorial (n : CARDINAL) : CARDINAL;
BEGIN
   IF n = 0 THEN
      RETURN 1
   ELSE
      RETURN n * (n-1) * Factorial(n-2)
   END
END Factorial;
```

Para algunos algoritmos recursivos el caso base puede ser múltiple. Un ejemplo es la sucesión de *Fibonacci*, que se define del siguiente modo: el primer número de *Fibonacci* es 0, el segundo es 1 y cualquier otro número de *Fibonacci* se obtiene sumando los dos números de *Fibonacci* que le preceden:

$$Fibonacci(n) = \begin{cases} 0, & si\ \ n = 1 \\ 1, & si\ \ n = 2 \\ Fibonacci(n-1) + Fibonacci(n-2), & en\ otro\ caso \end{cases}$$

Los casos bases en este ejemplo son $n = 1$ y $n = 2$. Una función en *Modula-2* que calcule el *n*-ésimo número de *Fibonacci* según esta definición es:

```
PROCEDURE Fibonacci (n : CARDINAL) : CARDINAL;
BEGIN
   IF n = 0 THEN
      WrStr ("Error: Fibonacci(0) no está definido");
      HALT
   ELSIF n = 1 THEN
      RETURN 0
   ELSIF n = 2 THEN
      RETURN 1
   ELSE
      RETURN Fibonacci(n-1) + Fibonacci(n-2)
   END
END Factorial;
```

11.2 Recursividad frente a iteración

Cualquiera de los problemas resueltos de modo recursivo puede ser resuelto de modo iterativo. En algunos casos, como en el de la función `Factorial`, la solución iterativa es bastante simple:

```
PROCEDURE Factorial (n : CARDINAL) : CARDINAL;
VAR i, Resultado : CARDINAL;
```

```
BEGIN
  Resultado := 1;
  FOR i := 1 TO n DO
    Resultado := Resultado * i
  END;
  RETURN Resultado
END Factorial;
```

Podemos observar que, en la versión iterativa, la recursión ha sido sustituida por una iteración (un bucle FOR, en este caso). También ha sido necesario introducir, en la versión iterativa, dos variables locales (`i` y `Resultado`). En general, en los algoritmos recursivos se suele utilizar una sentencia de selección (IF), mientras que en los iterativos equivalentes se usa una sentencia de iteración (FOR, WHILE, REPEAT o LOOP) y es necesario introducir variables locales que almacenen los resultados intermedios del cómputo.

En otros casos la solución iterativa al problema es más complicada. En un apartado posterior veremos una solución recursiva al problema de las *Torres de Hanoi*. No es nada fácil encontrar una solución iterativa a este problema.

Un poco más complicada, aunque también fácil, es la solución iterativa para la recurrencia de *Fibonacci*. En este caso, necesitamos almacenar en dos variables (`FibNMenos1` y `FibNMenos2`) los valores de las dos iteraciones previas:

```
PROCEDURE Fibonacci (n : CARDINAL) : CARDINAL;
VAR FibN, FibNMenos1, FibNMenos2, i : CARDINAL;
BEGIN
  IF n = 0 THEN
    WrStr ("Error: Fibonacci(0) no está definido");
    HALT
  ELSIF n = 1 THEN
    FibN := 0
  ELSE
    FibNMenos1 := 0;
    FibN       := 1;
    FOR i := 2 TO n-1 DO
      FibNMenos2 := FibNMenos1;
      FibNMenos1 := FibN;
      FibN := FibNMenos2 + FibNMenos1;
    END
  END;
  RETURN FibN
END Fibonacci;
```

11.3 Eficiencia de los algoritmos recursivos

Los algoritmos recursivos suelen ser menos eficientes (en tiempo) que los iterativos correspondientes. Esto es así por dos motivos principalmente:

- La sobrecarga debida al funcionamiento interno de la recursividad.
- La posible sobrecarga debida a la solución recursiva.

11.3.1 Sobrecarga debida al funcionamiento interno

En la primera sección de este tema hicimos un seguimiento en detalle de la evaluación de una llamada a la función recursiva `Factorial`. Podemos observar que, en general, una llamada a una función recursiva produce varias llamadas más a la propia función. Cada una de estas llamadas requiere cierto tiempo para:

- Hacer la llamada.

- Crear las variables locales y copiar los parámetros por valor.
- Ejecutar las instrucciones en parte ejecutiva de la función.
- Destruir las variables locales y parámetros por valor.
- Salir de la función.

Si se realizan unas cuantas llamadas recursivas a una función, puede ocurrir que el tiempo necesario para hacer las llamadas y crear y destruir variables prevalezca sobre el tiempo necesario para realizar el cómputo propiamente dicho.

Este tipo de sobrecarga *aparece en todos* los algoritmos recursivos. Si utilizamos una función iterativa equivalente esta sobrecarga no aparece, ya que la función iterativa no da lugar a varias llamadas a sí misma.

11.3.2 Sobrecarga debida a la solución recursiva

Este tipo de sobrecarga no se debe al modo en que funciona la recursividad en sí misma, sino a que la solución recursiva al problema es mala. Esta sobrecarga aparece en la función recursiva que calcula los números de *Fibonacci*. Si analizamos el algoritmo podemos observar que los mismos valores son calculados varias veces. En la siguiente figura podemos observar las llamadas que se producen al intentar evaluar la expresión `Fibonacci(5)`:

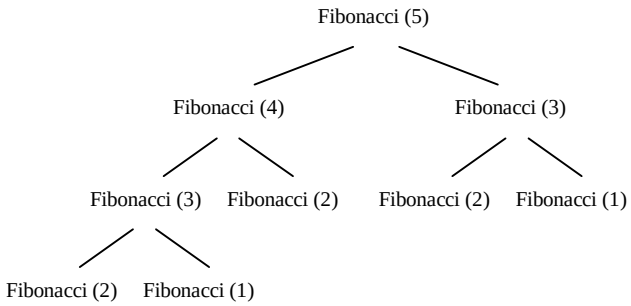


Figura 1. Llamadas realizadas al evaluar `Fibonacci(5)`.

Podemos observar que han sido necesarias un total de 9 llamadas a la función para calcular este valor y que el cálculo de `Fibonacci(2)`, por ejemplo, ha sido *repetido* 3 veces. Se ha realizado un montón de trabajo repetido.

Este tipo de sobrecarga no aparece en todos los algoritmos recursivos (por ejemplo, no aparece en el caso de la función `Factorial` recursiva). Al evaluar `Factorial(3)`, no se repite ningún cómputo:

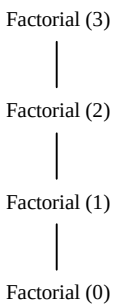


Figura 2. Llamadas realizadas al evaluar `Factorial(3)`.

Es posible mejorar el algoritmo recursivo para el calculo de los números de *Fibonacci* de modo que se evite este tipo de sobrecarga. En concreto, necesitamos añadir un par de parámetros extras en la función que almacenen el valor de los dos números de *Fibonnaci* anteriores. En la primera llamada estos argumentos tomarán valores 0 y 1 (los dos primeros números de la recurrencia):

```
PROCEDURE Fibonacci (n : CARDINAL) : CARDINAL;
  PROCEDURE FibAux (n, FibN, FibNMas1 : CARDINAL) : CARDINAL;
  BEGIN
    IF n = 1 THEN
      RETURN FibN
    ELSE
      RETURN FibAux(n-1, FibNMas1, FibN+FibNMas1)
    END
  END FibAux;
BEGIN
  IF n = 0 THEN
    WrStr("Error: Fibonacci(0) no está definido");
    HALT
  ELSE
    RETURN FibAux(n, 0, 1)
  END
END Fibonacci;
```

Si calculamos `Fibonacci(5)` con este algoritmo no repetimos cálculo:

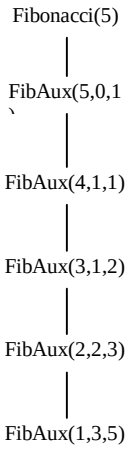


Figura 3. Llamadas realizadas al evaluar `Fibonacci(5)`.

Sin embargo, la nueva solución no es tan directa como la original, lo cual era la ventaja fundamental de la solución recursiva.

11.3.3 Iteración o recursión

El hecho de que la recursividad presente cierta sobrecarga (en tiempo de ejecución del algoritmo) frente a la iteración no significa que no se deba utilizar, sino que hay que utilizarla cuando realmente sea apropiada.

Cuando el problema a resolver es relativamente simple, suele ser igualmente fácil escribir la versión iterativa y la recursiva a éste. En este caso es preferible utilizar la solución iterativa ya que es más eficiente. Sin embargo, las soluciones a ciertos problemas más complicados pueden ser mucho más fáciles si utilizamos para ello la recursividad (ver el ejemplo de las *Torres de Hanoi*). En este caso, la claridad y simplicidad del algoritmo recursivo debe prevalecer frente

al tiempo extra que consume la recursión. Este enfrentamiento entre simplicidad y eficiencia aparece a menudo en la programación de ordenadores.

Otro uso común de la recursividad es como herramienta para desarrollar *prototipos* de programas. Podemos comenzar con una solución recursiva simple, que desarrollemos rápidamente, y posteriormente, si necesitamos un programa más eficiente, mejorarlo utilizando una solución iterativa.

El verdadero valor de la recursividad es como herramienta para resolver problemas para los que no hay soluciones iterativas simples.

11.4 Ejemplos clásicos de recursividad

11.4.1 Las torres de Hanoi

El primero de los ejemplos que estudiaremos es fácilmente resuelto mediante un algoritmo recursivo. Sin embargo, una solución iterativa al problema es mucho más complicada. El enunciado del problema es el siguiente:

> *Sean tres estacas verticales y n discos de distintos radios que pueden insertarse en las estacas formando torres. Inicialmente los n discos están situados todos en la primera estaca por orden decreciente de radios (ver figura 3). Se trata de pasar los n discos de la primera estaca a la última siguiendo las siguientes reglas:*
>
> 1) *En cada paso se mueve un único disco.*
> 2) *En ningún paso se puede colocar un disco sobre otro disco de radio menor.*
> 3) *Puede usarse la estaca central como estaca auxiliar.*

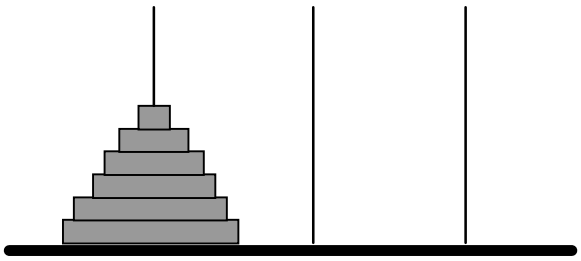
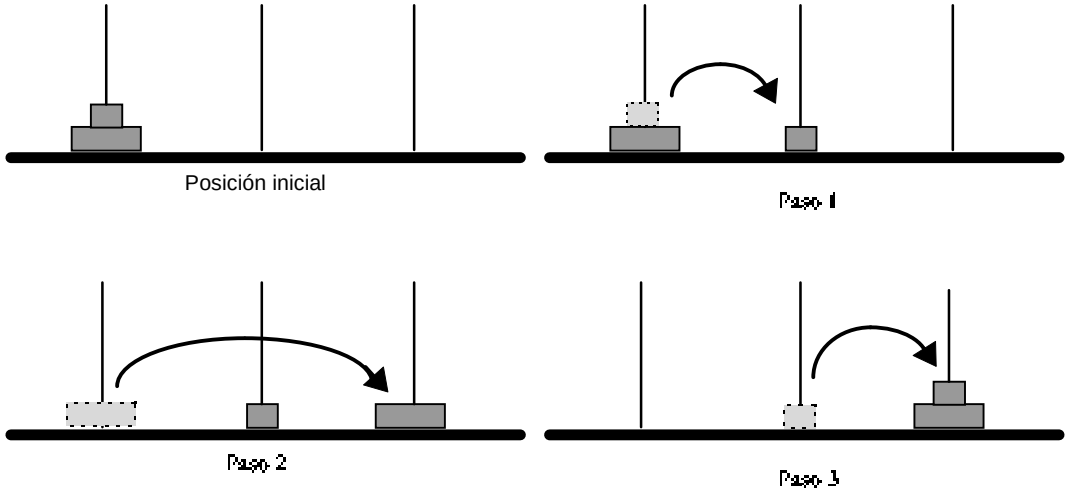


Figura 4. Situación inicial del juego.

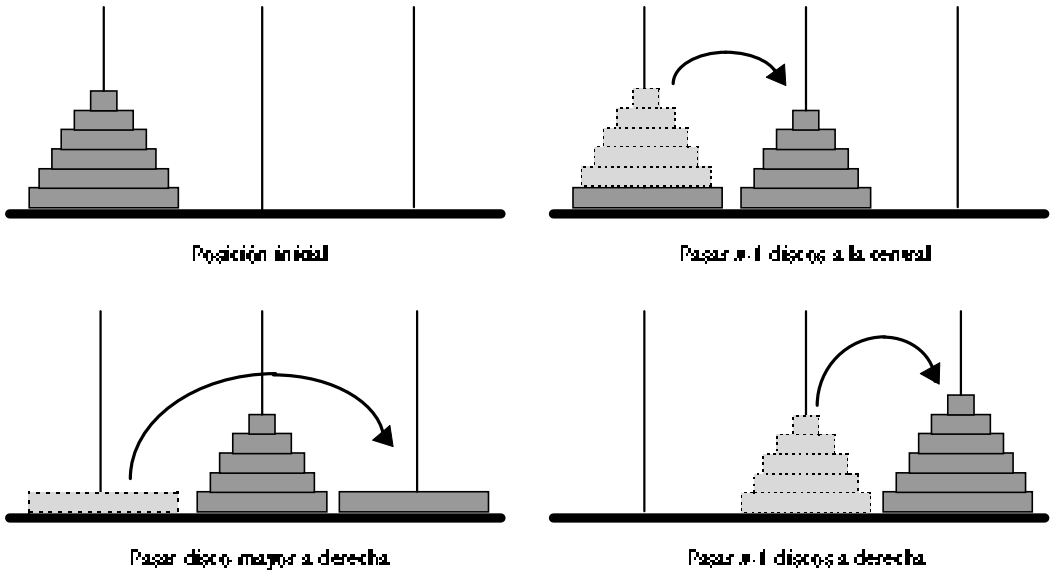
En el caso de una torre de altura uno, la solución es trivial:

La solución en el caso de una torre de dos discos consta de solo tres pasos:



Si suponemos resuelto el problema para una torre de altura *n*-1 discos (es decir sabemos mover *n*-1 discos de una estaca a otra usando la que queda como auxiliar), el problema de pasar *n* discos desde la estaca izquierda a la derecha queda resuelto del siguiente modo:

1) Pasar *n*-1 discos de la estaca izquierda a la estaca central, utilizando la derecha como auxiliar.
2) Mover el disco mayor de la estaca izquierda a la estaca derecha.
3) Mover *n*-1 discos de la estaca central a la derecha usando la izquierda como auxiliar.



Obsérvese que para solucionar el problema para una torre de altura *n* hemos supuesto resuelto el problema para una torre de altura *n*-1; la solución es recursiva.

El programa en *Modula-2* que usa la estrategia anterior para mostrar por pantalla los pasos necesarios para solucionar una torre cuya altura se ha leído desde el teclado es:

```
MODULE Hanoi;
FROM IO IMPORT WrStr, WrLn, RdCard, RdLn;
CONST
  IZQ = "Izquierda ";
  DER = "Derecha ";
  CEN = "Central ";
VAR
```

```
  Altura : CARDINAL;

  PROCEDURE WrMovimientos (N : CARDINAL;
                           Izq, Der, Cen : ARRAY OF CHAR);
  BEGIN
     IF N = 1 THEN
        WrStr ("Mover de "); WrStr(Izq); WrStr("a "); WrStr(Der);
        WrLn
     ELSE
        WrMovimientos(N-1, Izq, Cen, Der);
        WrStr ("Mover de "); WrStr(Izq); WrStr("a "); WrStr(Der);
        WrLn;
        WrMovimientos(N-1, Cen, Der, Izq)
     END
  END WrMovimientos;

BEGIN
  WrStr ("Dame la altura: ");
  Altura := RdCard(); RdLn;

  WrMovimientos(Altura, IZQ, DER, CEN)
END Hanoi.
```

La clave del programa anterior es el subprograma recursivo

```
PROCEDURE WrMovimientos (N : CARDINAL;
                         Izq, Der, Cen : ARRAY OF CHAR);
```

que muestra por pantalla los movimientos necesarios para mover `N` discos desde la estaca cuyo nombre es `Izq` a la estaca cuyo nombre es `Der`, usando para ello la estaca `Cen` como estaca auxiliar.

Este subprograma tiene como caso base aquél en el cual la altura de la torre a resolver es 1. Para resolver este caso (mover una torre de altura 1 que esta situada en la estaca izquierda a la estaca derecha) basta con realizar un único movimiento: mover un disco desde la estaca izquierda a la derecha.

En otro caso, si la altura de la torre es mayor a 1:

1) Escribimos primero los pasos necesarios para mover `N`-1 discos desde la estaca izquierda a la central usando la derecha como auxiliar. Esto se consigue con la llamada recursiva

   ```
   WrMovimientos(N-1, Izq, Cen, Der);
   ```

   Obsérvese que `Izq` aparece como primer parámetro en esta llamada, por lo cual la estaca desde la que se mueven los `N`-1 discos es desde la izquierda. `Cen` es el segundo argumento, por lo que la estaca hacia la que se mueven los discos es la central. Por último `Der` es el tercer argumento e indica que se use la estaca derecha como auxiliar.

2) Movemos el disco que queda en la estaca izquierda a la derecha

   ```
   WrStr ("Mover de "); WrStr(Izq); WrStr("a "); WrStr(Der);
   ```

3) Escribimos los pasos necesarios para mover `N`-1 discos desde la torre central a la derecha usando la izquierda como auxiliar.

   ```
   WrMovimientos(N-1, Cen, Der, Izq);
   ```

Podemos comprobar que la solución cumple las tres condiciones de corrección:

1) Existe un caso base (la torre de altura 1).

2) Cada llamada recursiva se aproxima más al caso base ya que se decrementa en 1 el valor de `N`.

3) El caso base será alcanzado. Dado que en cada llamada recursiva decrementamos el valor de `N` exactamente uno, en algún momento `N` será 1.

La llamada que activa el subprograma desde el programa principal hace que se muestren los pasos necesarios para mover tantos discos como indique la variable `Altura` desde la estaca izquierda a la derecha, utilizando la central como auxiliar. Si ejecutamos el algoritmo para una altura inicial de 3 discos obtenemos la siguiente salida por pantalla:

```
Dame la altura: 3
Mover de Izquierda a Derecha
Mover de Izquierda a Central
Mover de Derecha a Central
Mover de Izquierda a Derecha
Mover de Central a Izquierda
Mover de Central a Derecha
Mover de Izquierda a Derecha
```

El lector puede comprobar que la solución obtenida es correcta.

11.4.2 Algoritmos de búsqueda recursivos

En el tema 7 estudiamos varios algoritmos para buscar un elemento en un vector. Veremos ahora como podemos expresar estos algoritmos de forma recursiva.

11.4.2.1 Búsqueda lineal recursiva

Recordemos que la búsqueda lineal consiste en ir inspeccionando cada elemento del vector de izquierda a derecha hasta que, o bien se encuentra el elemento buscado, o bien se ha recorrido todo el vector. En este último caso el elemento buscado no está en el vector.

Realizaremos la recursión sobre la longitud del vector. Así, podemos expresar esta idea como:

1) Si buscamos un elemento en un vector de tamaño cero podemos concluir que el elemento no está en éste

2) Si el elemento buscado coincide con el primer elemento del vector entonces lo hemos encontrado.

3) En otro caso deberemos buscar (llamada recursiva) a partir de la siguiente posición del vector.

El siguiente subprograma implementa esta búsqueda recursiva:

```
PROCEDURE BuscarLineal ( A : ARRAY OF ELEMENTO;
                         Elemento : ELEMENTO ) : INTEGER;
  PROCEDURE BuscarDesde( A : ARRAY OF ELEMENTO;
                         Elemento : ELEMENTO;
                         Primero, Maximo :CARDINAL ) : INTEGER;
  CONST
    NOENCONTRADO = -1;
  BEGIN
    IF Primero > Maximo THEN
      RETURN NOENCONTRADO
    ELSIF A[Primero] = Elemento THEN
      RETURN Primero
    ELSE
      RETURN BuscarDesde(A, Elemento, Primero+1, Maximo)
    END
```

```
  END BuscarDesde;
BEGIN
  RETURN BuscarDesde(A, Elemento, 0, HIGH(A))
END BuscarLineal;
```

La función devuelve la posición en la que se encontró el elemento o el valor -1 si no se encontró. El algoritmo recursivo se implementa en `BuscarDesde`. Esta función busca recursivamente el elemento `Elemento` en el vector `A` . En todo momento, el segmento del vector que queda por inspeccionar es el determinado por las variables `Primero` y `Maximo`. Con cada llamada recursiva el tamaño del segmento a inspeccionar se reduce en una unidad. El caso 1) se produce cuando (`Primero` > `Maximo`). El caso 2) es el correspondiente a la parte ELSIF mientras que el 3) aparece tras la cláusula ELSE. La función `BuscarLineal` hace que comience la búsqueda, estableciendo como segmento inicial la totalidad del vector.

Obsérvese que al ser `BuscarDesde` una función recursiva, será probablemente llamada varias veces. Al comienzo de cada llamada se reservará espacio para cada uno de los parámetros por valor de esta función. Si el tamaño de la variable `A` es relativamente grande, reservar espacio para ella en cada llamada puede ser costoso (en tiempo y consumo de memoria). Para solucionar este problema podemos, aunque su valor no sea modificado, declarar este parámetro por referencia (**VAR** `A:`**ARRAY OF** `ELEMENTO`), ya que no se crean nuevas copias con cada llamada para este tipo de parámetros.

11.4.2.2 Búsqueda binaria recursiva

La búsqueda binaria puede ser también fácilmente implementada de modo recursivo. Si el vector está ordenado en orden ascendente, el algoritmo es el siguiente:

1) Si buscamos un elemento en un vector de tamaño cero podemos concluir que el elemento no está en éste
2) Si el elemento que ocupa la posición central del vector coincide con el buscado, lo hemos encontrado en la posición central.
3) Si el elemento buscado es menor que el que ocupa la posición central del vector, buscar en la mitad izquierda del segmento.
4) En otro caso, buscar en la mitad derecha del segmento.

El siguiente subprograma implementa esta búsqueda recursiva:

```
PROCEDURE BuscarBinaria  (  A : ARRAY OF ELEMENTO;
                            Elemento : ELEMENTO ) : INTEGER;
  PROCEDURE BuscarEn (A : ARRAY OF ELEMENTO;
                      Elemento : ELEMENTO;
                      Primero, Ultimo :CARDINAL ) : INTEGER;
  CONST
    NOENCONTRADO = -1;
  VAR
    Central : CARDINAL;
  BEGIN
    IF Primero > Ultimo THEN
      RETURN NOENCONTRADO
    ELSE
      Central := (Primero + Ultimo) DIV 2;
      IF A[Central] = Elemento THEN
        RETURN Central
      ELSIF A[Central] > Elemento THEN
        RETURN BuscarEn(A, Elemento, Primero, Central-1)
      ELSE
        RETURN BuscarEn(A, Elemento, Central+1, Ultimo)
      END
    END
```

```
  END BuscarEn;
BEGIN
  RETURN BuscarEn(A, Elemento, 0, HIGH(A))
END BuscarLineal;
```

En este caso, el tamaño del segmento donde realizar la búsqueda queda reducido a la mitad en cada llamada recursiva. Las mismas consideraciones de eficiencia estudiadas para la búsqueda lineal recursiva son aplicables a este algoritmo.

Relación de problemas (Tema 11)

1. El *máximo común divisor* de dos números enteros se puede definir con ayuda de las siguientes propiedades elementales:

$$mcd(a,b) = mcd'(|a|,|b|),$$
$$mcd'(a,a) = a,$$
$$mcd'(a,0) = a,$$
$$mcd'(a,b) = mcd'(b,a),$$
$$mcd'(a,b) = mcd'(a-b,b), \quad \text{si } a>b$$

Escribe un programa recursivo que calcule el *máximo común divisor* de dos números enteros usando las propiedades citadas.

Sigue las llamadas realizadas por el algoritmo que diseñes para calcular el máximo común divisor de 6 y 32.

2. Observe que la última propiedad del ejercicio anterior permite reducir el cálculo del máximo común divisor de dos números al mismo problema pero con dos números más pequeños, utilizando para ello la operación diferencia. Tal operación puede sustituirse por el resto de la división entera, ya que si *k* es un divisor de *a* y *b* (siendo *a*>*b*), entonces *k* también es un divisor del resto de la división de *a* entre *b*:

$$mcd'(a,b) = mcd'(a \,\mathbf{MOD}\, b, b), \quad \text{si } a>b$$

Dicho resto puede ser cero, y en ese caso el máximo común divisor es el valor de *b* (puesto que *a* sería un múltiplo de *b*). Esta propiedad permite acelerar el método del ejercicio anterior.

Escribe un programa recursivo que calcule el *máximo común divisor* de dos números enteros usando las propiedades citadas, junto con las del ejercicio anterior.

Sigue las llamadas realizadas por el algoritmo que diseñes para calcular el máximo común divisor de 6 y 32.

3. Escribe un programa recursivo que calcule números combinatorios teniendo en cuenta las siguientes propiedades:

$$\binom{m}{n} = 1, \quad \text{si } n = 0 \text{ o si } m=n$$
$$\binom{m}{n} = m, \quad \text{si } n = 1 \text{ o si } n=m-1$$
$$\binom{m}{n} = \binom{m-1}{n-1} + \binom{m-1}{n}$$

Sigue las llamadas que realiza el algoritmo para calcular 5 sobre 2.

4. Escribe un programa recursivo que calcule la suma de los primeros `N` números pares naturales.

5. La función de *Ackermann* se describe del siguiente modo:

$$A(m,n) = n+1, \quad \text{si } m = 0$$
$$A(m,n) = A(m-1,1), \quad \text{si } n = 0$$
$$A(m,n) = A(m-1, A(m,n-1)), \quad \text{si } m > 0 \text{ y } n>0$$

Esta función recursiva es muy conocida porque aun pareciendo muy simple da lugar a cálculos muy complicados. Por ejemplo la llamada *A*(2, 2) produce 27 llamadas recursivas para devolver el valor 7.

Escribe un subprograma recursivo que calcule la función de *Ackermann* para valores de *m* y *n* dados. Utiliza éste para escribir un programa que produzca la siguiente tabla:

m \ *n*	0	1	2	3	4	5	6	7
0	1	2	3	4	5	6	7	8
1	2	3	4	5	6	7	8	9
2	3	5	7	9	11	13	15	17
...	...	...	...	...	...	...	...	...

Puede que notes que el ordenador tarda un poco en calcular los últimos valores de la tabla. Esto no es extraño ya que el cálculo de *A*(2, 6), por ejemplo, da lugar a 172.233 llamadas recursivas.

NOTA: Para la versión 1 de *TopSpeed* se produce un error al intentar evaluar *A*(3, *n*) si *n* > 7, *A*(4, *n*) si *n* > 0 o *A*(*m*,*n*) si *m* > 4.

6. Escribe un programa recursivo que lea un número entero positivo desde teclado y lo escriba por pantalla en orden inverso (si se lee 124 se escribirá 421).

7. Escribe un programa recursivo que calcule la suma de las cifras de un número entero positivo leído de teclado.

8. Escribir un programa recursivo que lea un número entero positivo expresado en base 10 desde teclado y lo pase a otra base entre 2 y 10 también leída de teclado. No se conoce a priori el número de cifras del número en la base destino, por lo cual **no** se podrá utilizar un array para ir almacenando las cifras calculadas.

9. Supongamos que tenemos un ordenador que sólo trabaja con números naturales y para el cual sólo existen dos funciones aritméticas:

```
PROCEDURE Suc(n : CARDINAL) : CARDINAL;
BEGIN
  RETURN n+1
END Suc;

PROCEDURE Pred(n : CARDINAL) : CARDINAL;
BEGIN
  IF n = 0 THEN
    WrStr("ERROR, número negativo");
    HALT
  ELSE
    RETURN n-1
  END
END Pred;
```

Escribe los siguientes subprogramas de un modo recursivo:

```
PROCEDURE Suma      (a, b : CARDINAL) : CARDINAL;
PROCEDURE Resta       (a, b : CARDINAL) : CARDINAL;
PROCEDURE Producto    (a, b : CARDINAL) : CARDINAL;
```

```
PROCEDURE Cociente    (a, b : CARDINAL) : CARDINAL;
```

Teniendo en cuenta las siguientes propiedades:

$Suma(a,0) = a$

$Suma(a,b) = Suc(Suma(a, Pred(b)), \quad \text{si } b \neq 0$

$Resta(a,0) = a$

$Resta(0,a) = \text{ERROR, numero negativo}$

$Resta(a,b) = 0, \quad \text{si } a = b$

$Resta(a,b) = Resta(Pred(a), Pred(b)), \quad \text{si } a \neq b$

$Producto(a,0) = 0$

$Producto(a,b) = Suma(Producto(a, Pred(b)), a)$

$Cociente(a,0) = \text{ERROR, division por cero}$

$Cociente(a,b) = Suc(Cociente(Resta(a,b),b)), \quad \text{si } a \geq b$

$Cociente(a,b) = 0, \quad \text{si } a < b$

Recuerda que no puedes usar +, -, *, **DIV** o **MOD** en estos cuatro subprogramas.

Intenta deducir tú las propiedades de *Resto*(*a*,*b*) (valor del resto obtenido al dividir *a* por *b*) y escribe el correspondiente subprograma recursivo.

10. Escribir un programa recursivo que lea una cadena de caracteres acabada en punto desde el teclado y la imprima al revés por pantalla. No se conoce el tamaño máximo de la cadena a priori, por lo cual el programa **no** podrá almacenar la cadena en un array tal como se va leyendo. Utiliza lectura múltiple de datos (sin RdLn).

Nota: Hablamos de programa recursivo cuando utiliza subprogramas recursivos.

Relación complementaria (Tema 11)

11. Escribe un programa recursivo que calcule potencias enteras de números reales, usando para ello las siguientes propiedades:

$$\begin{aligned} a^0 &= 1 \\ a^n &= a \cdot a^{n-1} \quad si \quad n > 0 \\ a^n &= \left(\frac{1}{a}\right)^{-n} \quad si \quad n < 0 \end{aligned}$$

12. Escribe un procedimiento recursivo que escriba `N` saltos de línea por pantalla, siendo `N` un argumento de este subprograma.

13. Escribe un programa recursivo que calcule el capital C_n obtenido, situando el capital C_0 a interés compuesto durante n años al interés anual r (expresado en porcentaje). La recurrencia que permite calcular C_n es:

$$C_1 = C_0 \cdot (1 + r/100)$$
$$C_2 = C_1 \cdot (1 + r/100)$$
$$\ldots$$
$$C_n = C_{n-1} \cdot (1 + r/100)$$

14. Obtén una fórmula no recursiva para C_n .Escribe un programa que calcule C_n a partir de C_0 y r utilizando la fórmula obtenida.

15. Escribir un programa recursivo que lea un número entero positivo expresado en base b desde teclado (con $b \leq 9$) y lo pase a base 10. (Utiliza el tipo LONGCARD)

16. Escribe una función recursiva que devuelva el elemento de mayor valor en un array de enteros.

17. Escribe una función recursiva que devuelva la suma de los valores almacenados en un array de enteros.

18. Escribe una función recursiva que devuelva el n-ésimo elemento de menor valor en un array de enteros. Por ejemplo, si el array contiene los valores 12 7 3 9 1 5 21 y n =1 devuelve 1, si n=2 devuelve 3, si n=5 devuelve 9 y si n=28 se produce un error.