1. *a*)
```
procedure FechaAInt (sfecha: string; var dia, mes, anyo: integer);
(* Entrada : fecha en formato ddmmaaaa
   Salida  : fecha en forma de tres enteros
   Pre     : entrada correcta (8 caracteres dígitos, representando una fecha)
*)
```

b)
```
function edad (fnac, factual: string): integer;
(* Entrada       : dos fechas en formato ddmmaaaa
   Valor devuelto: número de años completos transcurridos
                          entre la primera y la segunda
   Pre           : las entradas representan realmente fechas y
                         la primera es posterior o igual a la segunda,
                         en el orden de fechas
*)
```

c)
```
function ValeMovAlfil (pini, pfin: TPosicion;
                       var tablero: TTablero): boolean;
(* Entrada       : dos posiciones en un tablero de ajedrez
                    la configuración completa de las piezas  en el tablero
   Valor devuelto: cierto si un alfil puede ir en un movimiento
                          de la primera posición a la segunda
                    falso si no
   Pre           : En la posición 'pini' de 'tablero' hay un alfil
   Observaciones :
          Se tiene en cuenta:
            - que la posición de llegada sea posible desde la de partida
                 por un alfil
            - que la posición de llegada esté libre u
                 ocupada por una pieza de distinto color
            - que no haya piezas en la trayectoria
          El color del alfil puede obtenerse del contenido de tablero
```

Ha de suponerse que están definidos tipos adecuados:

```
type TPosicion = record
                  fila : 'A'..'H';
                  columna : 1 .. 8
                 end;
     end;
   TColor    = (blanco, negro);
   TPieza    = record
                 figura : (peon, torre, caballo, alfil, reina, rey);
                 color : TColor;
               end;
   TTablero  = array ['A'..'H', 1..8] of TPieza;
```

Otra solución, más simple y menos potente, con el mismo tipo `TPosicion`:

```
function ValeMovAlfil (pini, pfin: TPosicion): boolean;
(* Entrada        : dos posiciones en el tablero de ajedrez
   Valor devuelto : cierto si un alfil puede ir en un movimiento
                          de la primera posición a la segunda
   Observaciones  : No se tiene en cuenta el resto de las piezas
*)
```

d)
```
procedure MuestraModifica (var r : Tregistro; var f : Tfichero);
(* Entrada-Salida : un fichero de registros
```

```
                    un registro de dicho fichero
   Otros          : entrada de usuario
                    salida por pantalla
   Objetivo       : mostrar al usuario los valores del registro
                      y permitir que el usuario los modifique
                      actualizando el fichero si es el caso
   Pre            : el registro está en el fichero
   Post           : el registro y el fichero quedan actualizados si el usuario
                       ha realizado alguna modificación
*)
```

Otra solución:

```
procedure MuestraModifica (var r : Tregistro; var modificado : boolean);
(* Entrada-Salida : un registro
   Salida         : 'modificado' indica si el usuario ha modificado o no
                       el registro
   Otros          : entrada de usuario
                    salida por pantalla
   Objetivo       : mostrar al usuario los valores del registro
                    y permitir que el usuario los modifique
   Observaciones  : se actualizan los valores del registro.
                    Si 'modificado' indica que se han modificado,
                       el fichero no se actualiza (deberá realizarse
                       la actualización en el fichero posteriormente)
*)
```

2. *a*) Un algoritmo que, tomando como entrada una serie de datos devuelve los mismos datos ordenados por algún criterio.

   La serie puede estar en un vector, fichero u otro tipo estructurado. Los datos deben ser de un tipo ordenado (es decir, en el que haya definido un orden, por ejemplo mediante un operador relacional $\leq$: números enteros o reales, enumerados, caracteres, cadenas de caracteres ...) o bien contener alguna componente de un tipo ordenado (por ejemplo, registro, uno de cuyos campos es de tipo ordenado). El algoritmo devolverá, en el mismo tipo de estructura, los datos en orden (creciente o decreciente) según la componente elegida.

   b) La idea fundamental es la de insertar un elemento en una serie que ya está ordenada, desplazando los elementos para hacer lugar al nuevo si es necesario.

   Inicialmente, se parte del primer elemento de la serie de partida, que obviamente constituye una serie ordenada.

   El segundo elemento de la serie de partida se inserta en la posición que le corresponda de la parte ya ordenada (delante o detrás; si es delante, deberá desplazarse para dejar sitio; si no, se pone a continuación).

   En general, cuando se considera el n-ésimo elemento de la serie de partida, los n-1 anteriores formarán una subserie ordenada, y se buscará la posición del nuevo, desplazando los posteriores de la subserie, si los hay, para hacer hueco.

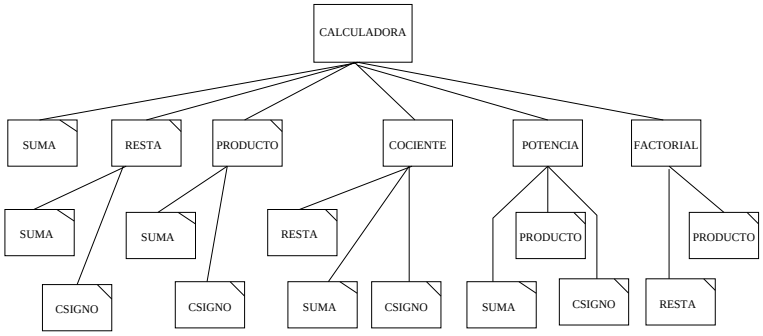

Figura 1: Un diseño modular

3. Diseño modular:Figura 1

   Interfaces y precondiciones:

Módulo	Entradas	Salidas	Precondiciones
Suma	a, b	a+b	-
Csigno	a	-a	-
Producto	a, b	a*b	-
Cociente	a, b	a div b	$a \neq 0$
Potencia	a, b	a^b	$b \geq 0$ y $(a,b) \neq (0,0)$
Factorial	a	a!	$a \geq 0$

   Todas las entradas y salidas son elementos del tipo adecuado (entero "largo").

   Explicación de los módulos (no se pedía):

   Suma se implanta mediante el módulo dado

   Resta Se define $a - b$ como $a + (-b)$, es decir $suma(a, csigno(b))$

   Producto Si $b = 0$ el producto es 0.

   Si $b > 0$, el producto es una iteración de b sumas. Para controlar la iteración se requiere un contador al que vaya sumándosele 1.

   Si $b < 0$, $a * b = -(a * (-b))$ siendo $-b$ positivo, de modo que se requiere un cambio de signo.

   Cociente Ha de suponerse el cociente entero. Si $b = 0$ no puede calcularse a/b.

   Si $0 < a < b$, el cociente a/b es 0.

   Si $0 < b \leq a$, el cociente a/b puede calcularse como una iteración de restas. Por lo tanto se necesita la resta y la suma para calcular el número de iteraciones.

   Si alguno de los números a dividir es negativo, se necesitarán cambios de signo.

   Potencia Entre enteros no tiene mucho sentido plantearse exponentes negativos.

   Si $b > 0$ la potencia a^b se calcula mediante una iteración de productos (nuevamente para llevar la cuenta de la iteración se necesita la suma).

   Si $b = 0$ y a no es 0, la potencia es 1.

   El valor 0^0 no está definido.

   Factorial Se calcula mediante una serie de productos. Se necesita la resta y sólo se realiza para enteros positivos.

   Ha de suponerse en cualquier caso que se dispone de un código de comparación entre números.

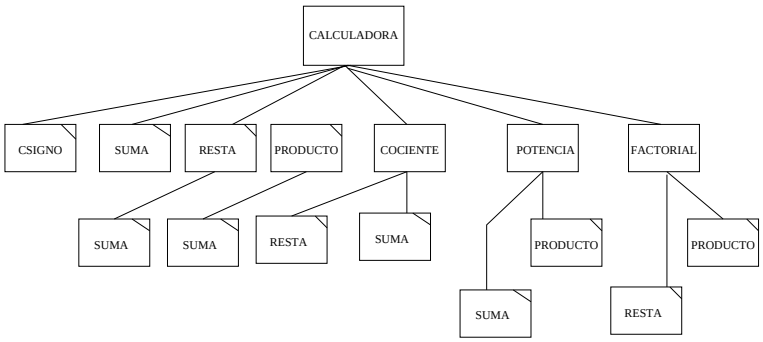

Figura 2: Otro diseño modular

Otra posibilidad es dejar al procedimiento principal que se encargue de los signos y trabajar con un módulo 'producto' preparado sólo para multiplicador positivo y un módulo cociente preparado sólo para positivos:

Diseño modular:Figura 2

Interfaces y precondiciones

Módulo	Entradas	Salidas	Precondiciones
Suma	a, b	a+b	-
Csigno	a	-a	-
Producto	a, b	a*b	$b > 0$
Cociente	a, b	a div b	$a > 0$ $b \geq 0$
Potencia	a, b	a^b	$b \geq 0$ y $(a, b) \neq (0, 0)$
Factorial	a	a!	$a \geq 0$

4. *a*) Versión iterativa: hay muchas. Se proponen dos usando una cadena local:

```
function inversa (c: string): string;
 var i : integer;
     inversaProv : string;
begin
        inversaProv := '';
        for i := length (c) downto 1 do
                inversaProv := inversaProv + c[i];
        inversa := inversaProv
end;

function inversa (c: string): string;
 var i : integer;
     inversaProv : string;
begin
        inversaProv := '';
        for i := 1 to  length (c) do
                inversaProv := c[i] + inversaProv;
        inversa := inversaProv
end;
```

Solución calculando las posiciones reflejadas:

```
function inversa (c: string): string;
var     i : integer;
        provisional : string;
        l : integer;
begin
        provisional := ''; l := length(c);
        for i := 1 to l do
                provisional := provisional + copy (c, l+1-i, 1);
                (* la posición reflejada de la i
                   respecto al centro es l+1-i *)
        inversa := provisional
end;
```

Solución trabajando sobre la cadena parámetro, basada en intercambio:

```
function inversa (c: string): string;
var     m, i: integer;
        a: char;
begin
        m := length (c) div 2;
        for i := 1 to m do
                begin
                        a := c[i];
                        c[i] := c[length(c)+1-i];
                        c[length(c)+1-i]:= a
                end;
        inversa := c
end;
```

b) Versión recursiva:

```
function inversaRecursiva (c: string): string;
begin
        if length(c) in [0,1] then inversaRecursiva := c
        else
                inversaRecursiva :=
                        inversaRecursiva(copy (c, 2, length(c)-1)) +
                        copy(c, 1, 1)
end;
```

5. Desplazamiento a la izquierda de los elementos de una matriz. También hay varias soluciones.

   Solución trabajando sobre la matriz parámetro, en la que

   - se retiene el primer elemento ([1,1])
   - se tratan las primeras filas, desplazando los primeros elementos a la izquierda, y separando el caso del último de cada fila, que debe tomar el valor del primer elemento de la siguiente
   - se trata la última fila, desplazando los primeros elementos a la izquierda, y separando el caso del último, que debe tomar el valor retenido al principio

En esta solución se ha optado por pasar como parámetros de entrada las dimensiones de la matriz.

```
procedure shiftL1 (var A: tmatriz; m, n: integer);
var     i, j: integer;
        x : integer;
begin
        x := a[1, 1];
        (* m-1 primeras filas: *)
        for i := 1 to m-1 do
                begin   (* i < m *)
                        (* n-1 primeros elementos: *)
                        for j := 1 to n-1 do
                            a [i, j] := a[i, j+1];
                        (* último de la fila: *)
                        a[i, n] := a[i+1, 1]
                end;
        (* última fila: *)
        (* primeros: *)
        for j := 1 to n-1 do
                a [m, j] := a[m, j+1];
        (* último: *)
        a[m,n] := x
end;
```

Una solución usando un vector de m elementos, para almacenar el primer elemento de cada fila antes de hacer los desplazamientos de los restantes. Después se colocan los guardados en sus posiciones finales de la última columna.

En esta solución no se ha pasado más parámetro que la matriz a modificar:

```
procedure shiftL3 (var A: tmatriz);
var i, j    : integer;
    guardas : array [1..M] of integer;
begin
        for i := 1 to M do
                guardas [i] := a[i, 1];
        for i := 1 to M do
                for j := 1 to N-1 do
                        a [i, j] := a[i, j+1];
        for i := 1 to M-1 do
                a[i, N] := guardas[i+1];
        a[M, N] := guardas[1]
end;
```

Una solución que en primer lugar translada los elementos de la matriz a un vector, para realizar en él los desplazamientos más fácilmente, y repone después los valores en la matriz. Utiliza más memoria local:

```
procedure shiftL7 (var a: tmatriz);
var vector : array [1.. m*n] of integer;
    i, f, c : integer;
    aux : integer;
begin   (* traslado a vector *)
        i := 1;
        for f := 1 to m do
            for c := 1 to n do
                begin
                    vector [i] := a[f,c];
                    i := i+1
                end;
(* desplazamiento *)
        aux := vector [1];
        for i := 1 to m*n-1 do
            vector [i] := vector[i+1];
        vector[m*n] := aux;
(* traslado a matriz *)
        i := 1;
        for f := 1 to m do
            for c := 1 to n do
                begin
                    a[f,c] := vector [i] ;
                    i := i+1
                end;
end;
```

Una curiosa solución basada en intercambio: si se intercambian los valores de cada posición con el de la siguiente, al final todos quedan desplazados a la izquierda, y el primero pasa a la última posición. Hay que hacer un intercambio menos que elementos. Por ejemplo: figura 3

```
procedure shiftL18 (var a: tmatriz; m, n: integer);
var i, j : integer;
    x : integer;
begin
    for i := 1 to m-1 do
        begin
           for j := 1 to n-1 do
               begin
                  x := a[i,j]; a[i,j] := a[i, j+1]; a[i, j+1] := x
               end;
            x := a[i, n]; a[i, n] := a[i+1, 1]; a[i+1, 1] := x
        end;
    for j := 1 to n-1 do
        begin
           x := a[m,j]; a[m,j] := a[m, j+1]; a[m, j+1] := x
        end;
end;
```

Otra solución en la que la guía es el elemento que hay que escribir, recorriendo la matriz en orden inverso. Primero habrá que escribir el elemento $[1, 1]$ en la última posición, pero salvaguardando el que había en ella. A partir de ese momento, se escribe el que se había guardado, previa reserva del que había en la posición que se escribe:

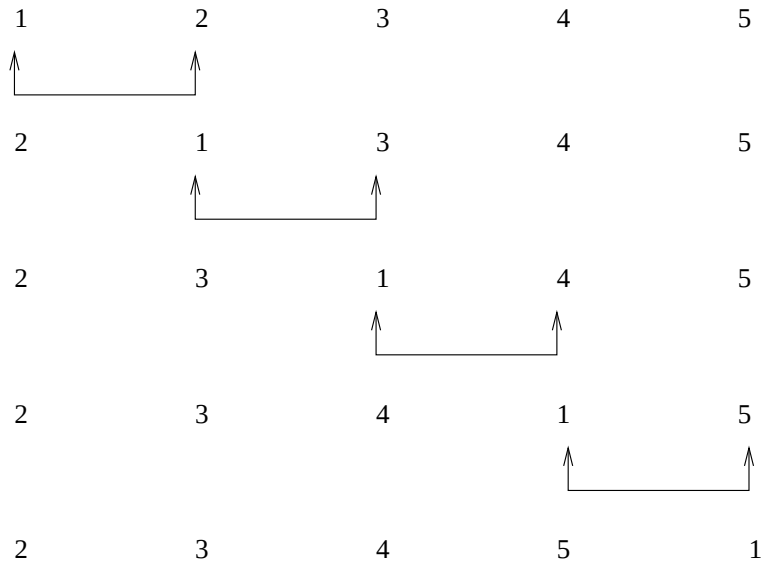

Figura 3: Efecto de intercambios sucesivos

```
procedure shiftL13 (var a: tmatriz; m, n: integer);
var i, j : integer;
    escribir, guardar : integer;
begin
        escribir := a[1,1];
        for i := m downto 1 do
            for j := n downto 1 do
                begin
                    guardar := a[i,j];
                    a[i, j] := escribir;
                    escribir := guardar
                end;
end;
```

Finalmente, dos soluciones parecidas, en las que se recorren los elementos de la matriz separando cada caso. Es compacta pero requiere muchas evaluaciones sobre los índices de los bucles.

```
procedure shiftL16 (var A: tmatriz; m, n: integer);
var     i, j: integer;
        aux: integer;
begin
     aux := A[1,1];
     for i := 1 to m do
         for j := 1 to n do
             begin
                 if (i=m) and (j=n) then A[i,j]:= aux      (* último elemento *)
                 else if (j=n)      then A[i,j] := A[i+1,1] (* última columna *)
                 else                A[i,j] := A[i, j+1];
             end;
end;
```

En la que sigue, el método es el mismo, algo más oculto:

```
procedure shiftL15 (var A: tmatriz; m, n: integer);
var     i, j: integer;
        aux: integer;
begin
        for i := 1 to m do
                for j := 1 to n do
                        begin
                                if (i=1) and (j=1) then aux := A[i,j];
                                if (i=m) and (j=n) then A[i,j]:= aux
                                else if (j=n) then A[i,j] := A[i+1,1]
                                else A[i,j] := A[i, j+1];
                        end;
end;
```

En esta última solución, es muy importante codificar en el interior del bucle dos alternativas consecutivas

```
    if (i=1) and (j=1) then aux := A[i,j];
    if (i=m) and (j=n) then  ...
```

en lugar de

```
    if (i=1) and (j=1) then aux := A[i,j]
    else if (i=m) and (j=n)  ...
```

y en ese orden.

6.
```
program FundeFich (f1, f2, fs);
(* Pre: f1 y f2 tienen el mismo número de elementos *)

const L1 = 10; L2 = 6;

type  tv1 = array [1..L1] of integer;
      tv2 = array [1..L2] of integer;
      tv3 = array [1..L1+L2] of integer;
      tf1 = file of tv1;
      tf2 = file of tv2;
      tf3 = file of tv3;

var   v1: tv1;
      v2: tv2;
      v3: tv3;
      f1 : tf1;
      f2 : tf2;
      fs : tf3;

procedure fundir (var v1: tv1; var v2: tv2; var v3: tv3);
(* E: v1, v2 S: v3 *)
var i : integer;
begin
    for i := 1 to L1 do v3[i] := v1[i];
    for i := 1 to L2 do v3[i+L1] := v2[i]
end;

Begin
    assign (f1, 'vect10.dat');
    assign (f2, 'vect6.dat');
    assign (fs, 'juntos.dat');
    reset (f1);
    reset (f2);
    rewrite (fs);
    while not eof (f1) do
          begin
              read (f1, v1); read (f2, v2);
              fundir (v1,v2,v3);
              write (fs, v3)
          end;
    close (f1);
    close (f2);
    close (fs);
End.
```