

Programación I – 4 de Febrero de 2002

1. Eliminar repetidos (3 puntos)

```
type Vector_de_Enteros is array (1..1000) of Integer;
```

Especificar y escribir un subprograma que, dado un vector del tipo anterior, obtenga uno equivalente que no contenga elementos repetidos.

2. Vocales fantasma (3 puntos)

```
type Vector_de_Caracteres is array (Integer range <>) of Character;
```

Especificar y escribir un subprograma que, dado un vector de caracteres que contiene únicamente letras y espacios, escriba en la pantalla las palabras pertenecientes al vector, una en cada línea, eliminando sus vocales.

Por ejemplo, dado el siguiente vector:

	H	o	y		m	e		v	o	y			a		l	a		p	l	a	y	a
--	---	---	---	--	---	---	--	---	---	---	--	--	---	--	---	---	--	---	---	---	---	---

El resultado será:

Hy
M
Vy
L
ply

3. Puzzle (4 puntos)

Las siguientes declaraciones representan un puzzle formado por piezas cuadradas

```
M : constant Integer:= 10;
N : constant Integer:= 15;
R : constant Integer:= 50;

subtype Lado is Integer range 0 .. 9999;

type Pieza is array (1 .. 4) of Lado;

type Puzzle is array (1 .. M, 1 .. N) of Pieza;

type Tabla_Piezas is array (1 .. R) of Pieza;

type Lista_Piezas is record
  Elementos : Tabla_Piezas;
  Cuantos : Integer;
end record;
```

En el puzzle puede haber huecos sin llenar, siempre de una sola pieza. En el puzzle se representa un hueco con una pieza cuyo valor es 0 en sus 4 lados. Escribe el subprograma de nombre Completar_Huecos_Internos que se especifica de esta manera:

```
procedure Completar_Huecos_Internos
  (P: in out Puzzle;
  L: in Lista_Piezas;
  Resto: out Lista_Piezas);
-- Pre: el puzzle P está sin completar y tiene varios huecos en su parte
--       interior.
--       Cada hueco está rodeado de 4 piezas y en cada uno cabe una sola
--       pieza.
--       Todas las piezas necesarias para completar el puzzle están en L.
--       En cada hueco cabe una sola pieza de L.
-- Post: se han llenado todos los huecos del puzzle con las piezas de L.
--       Las piezas sobrantes se encuentran en la lista Resto.
```

Nota: una pieza cabe en un hueco si los cuatro valores que tiene en sus cuatro lados coinciden con los valores de las piezas de alrededor (las piezas no se pueden girar). Por ejemplo, la pieza descrita a continuación cabe en el hueco del centro del puzzle de la figura, porque sus valores y los de las piezas de alrededor (T3, P4, Q1, R2) coinciden.

Puzzle

		P2	
		P1 P3	
		P4	
T2		0	
T1 T3		0 0	Q2
T4		0	Q1 Q3
			Q4
		R2	
		R1 R3	
		R4	

Pieza a insertar

P4
T3 Q1
R2

1. Baloncesto (1,5 puntos)

Se ha definido una estructura para guardar información correspondiente a un partido de baloncesto. En esa estructura se guardará el nombre de los equipos participantes, los puntos conseguidos por cada equipo y la lista de jugadores que intervinieron, guardándose por cada jugador su nombre y los puntos conseguidos en el partido (el reglamento admite que en un equipo jueguen 10 jugadores como máximo).

```
type T_Jugador is record
  Nombre : String(1..20);
  Puntos: Natural;
end record;

Max_Jugadores: constant Integer := 10;

subtype T_Num_Jugadores is Integer range 0 .. Max_Jugadores;

type T_Tabla_Jugadores is array(1..Max_Jugadores) of T_Jugador;

type Tipo_Equipo is record
  Nombre_Equipo: String(1..20);
  Puntos: Natural;
  Numero_de_Jugadores: T_Num_Jugadores;
  --indica el nº de jugadores de este equipo
  Jugadores: T_Tabla_Jugadores;
end record;

type Tipo_Partido is array (1 .. 2) of Tipo_Equipo;
```

Dadas las definiciones anteriores, escribir un subprograma que resuelva el siguiente problema:

- Datos:** estructura de tipo Tipo_Partido
un entero para seleccionar uno de los dos equipos (1 ó 2)
- Postcondición:** el resultado es un valor booleano:
- *true* si los puntos de ese equipo son igual al resultado de calcular
la suma de los puntos conseguidos por los jugadores de ese equipo
- *false* en caso contrario

2. Lotería primitiva (4 puntos)

Se tiene una estructura de datos con información sobre boletos de una versión simplificada de la lotería primitiva, con la siguiente información por boleto:

- Número de boleto (un entero positivo).
- Número de apuestas (valor entre 1 a 6).
- Por cada apuesta 6 números en orden ascendente (valores entre 1 y 49).

Nº boleto: 19934023	Nº. apuestas: 2
1 8 15 22 29 36 43 2 9 16 23 30 37 44 3 10 17 24 31 38 45 4 11 18 25 32 39 46 5 12 19 26 33 40 47 6 13 20 27 34 41 48 7 14 21 28 35 42 49	1 8 15 22 29 36 43 2 9 16 23 30 37 44 3 10 17 24 31 38 45 4 11 18 25 32 39 46 5 12 19 26 33 40 47 6 13 20 27 34 41 48 7 14 21 28 35 42 49
1 8 15 22 29 36 43 2 9 16 23 30 37 44 3 10 17 24 31 38 45 4 11 18 25 32 39 46 5 12 19 26 33 40 47 6 13 20 27 34 41 48 7 14 21 28 35 42 49	1 8 15 22 29 36 43 2 9 16 23 30 37 44 3 10 17 24 31 38 45 4 11 18 25 32 39 46 5 12 19 26 33 40 47 6 13 20 27 34 41 48 7 14 21 28 35 42 49
1 8 15 22 29 36 43 2 9 16 23 30 37 44 3 10 17 24 31 38 45 4 11 18 25 32 39 46 5 12 19 26 33 40 47 6 13 20 27 34 41 48 7 14 21 28 35 42 49	1 8 15 22 29 36 43 2 9 16 23 30 37 44 3 10 17 24 31 38 45 4 11 18 25 32 39 46 5 12 19 26 33 40 47 6 13 20 27 34 41 48 7 14 21 28 35 42 49

Estas son las definiciones de tipos de datos correspondientes:

```

type Apuesta is array(1..6) of Integer;
type Apuestas_de_Boleto is array(1..6) of Apuesta;
type Boleto is record
    Num_Boleto: Integer;
    Num_Apuestas: Integer; -- valor entre 1 y 6
    Apuestas: Apuestas_de_Boleto;
end record;

type Tabla_de_Boletos is array(1..10000) of Boleto;
type Lista_de_Boletos is record
    Num_Boletos: Integer; -- valor entre 0 y 10000
    Boletos: Tabla_de_Boletos;
end record;

```

A. (1 Punto) Las categorías de premios son cuatro: 6 aciertos, 5 aciertos, 4 aciertos y 3 aciertos. Un boleto puede tener varias apuestas acertadas en cada categoría. Como máximo hay 100 boletos premiados en cada categoría. Declarar una estructura de datos (el nombre del tipo será T_Datos_Premios) para almacenar los números de boleto premiados en cada categoría de aciertos.

B. (3 Puntos) En el fichero llamado "RESULTADOS.DAT" se tienen los números agraciados en el último sorteo: seis números ordenados de menor a mayor (todos entre 1 y 49).

Especificar, diseñar y codificar un subprograma llamado Calcular_Premios que obtenga una estructura del tipo definido en el ejercicio A a partir de una lista de boletos del tipo mencionado en el preámbulo (Lista_de_Boletos) y devuelva además el dinero recaudado en el sorteo. El precio de una apuesta es 3 euros.

3. Matriz concéntrica (2,5 puntos)

```

N: constant Integer := 7;
type matriz_caracteres is array (1..N, 1..N) of Character;
type Matriz_Cuadrada is record
  rango: Integer;
  matriz: matriz_caracteres;
end record;

```

Diremos que una matriz cuadrada es concéntrica cuando está formada por una serie de anillos concéntricos, donde cada uno de los anillos tiene todos sus caracteres iguales. Por ejemplo, las matrices (1) y (2) son concéntricas, mientras que la (3) no lo es, porque en el tercer anillo más externo hay un carácter (%) diferente de los demás.

(1)

@	@	@	@	@	@	@
@	*	*	*	*	*	@
@	*	X	X	X	*	@
@	*	X	O	X	*	@
@	*	X	X	X	*	@
@	*	*	*	*	*	@
@	@	@	@	@	@	@

(3)

@	@	@	@	@	@	@	@
@	*	*	*	*	*	*	@
@	*	X	%	X	*	*	@
@	*	X	O	X	*	*	@
@	*	X	X	X	*	*	@
@	*	*	*	*	*	*	@
@	@	@	@	@	@	@	@

(2)

*	*	*	*
*	X	X	*
*	X	X	*
*	*	*	*

Se pide diseñar un subprograma recursivo que, dada una matriz cuadrada, diga si es concéntrica o no. (1,5 puntos).

Para resolver este problema puede ser de ayuda la implementación del siguiente subprograma no recursivo (1 punto):

```

function Anillo_Correcto( M:      in Matriz_Cuadrada;
                           Anillo: in Integer)           return Boolean
-- Precondición: Anillo está entre 1 y (nº filas matriz + 1)/2
--   Este valor indica el anillo correspondiente a la fila dada.
--   Por ejemplo, en la matriz 1), los siguientes valores de
--   Anillo indican lo siguiente:
--     Anillo = 1: el anillo más externo (formado por caracteres @)
--     Anillo = 2: el anillo formado por caracteres *
--     Anillo = 3: el anillo formado por caracteres X
--     Anillo = 4: el anillo formado por caracteres O
-- Postcondición: el resultado es True si el Anillo tiene todos los
--   caracteres iguales, y False si no

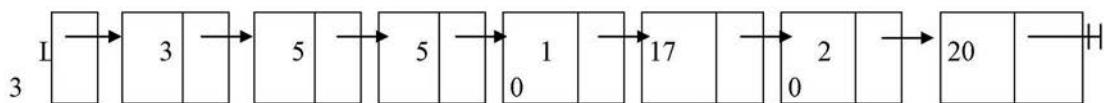
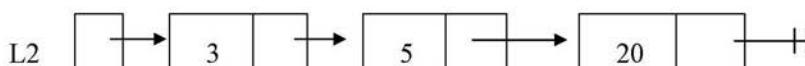
```

```
-- Por ejemplo, en la matriz 3)
-- Anillo = 1: el resultado es True
-- Anillo = 2: el resultado es True
-- Anillo = 3: el resultado es False (hay un carácter % diferente)
-- Anillo = 4: el resultado es True
```

4. Fusionar listas (2 puntos)

Escribir un subprograma que, dadas dos listas ligadas de enteros ordenadas ascendenteamente cree una nueva lista resultado de fusionarlas. Es decir, la lista resultado está ordenada y contiene los elementos de las listas originales, incluyendo repeticiones.

Por ejemplo, el resultado de fusionar las listas L1 y L2 es la lista L3.



En la solución se valorará la eficiencia del algoritmo.

1. Lotería (2 Puntos)

```
Max_Premios_por_Categoría: constant Integer := 100;
type Tabla_Boletos is array(1.. Max_Premios_por_Categoría) of Integer;
subtype Valor_entre_0_y_Tope is Integer range 0 .. Max_Premios_por_Categoría;
type Lista_Boletos_Premiados is record
    Cuantos: Valor_entre_0_y_Tope;
    Boletos: Tabla_Boletos;
end record;

type T_Datos_Premios is array(3 .. 6) of Lista_Boletos_Premiados;
```

La estructura de datos anterior contiene los números de boleto premiados en una edición de la lotería primitiva. Se premia a los boletos de 3, 4, 5 y 6 aciertos, y por cada una de las 4 categorías de aciertos se tiene una lista de los boletos con premio en esa categoría. Un boleto puede aparecer más de una vez en una lista de premios.

Diseñar y codificar el subprograma Total_Dinero cuya cabecera se indica a continuación que, a partir de un número de boleto, el dinero recaudado y una estructura de tipo T_Datos_Premios, calcule cuánto dinero le corresponde a ese boleto en premios. La asignación de dinero se calcula de la siguiente manera: la mitad del dinero recaudado se destina a premios. Esta cantidad se distribuye a partes iguales entre cada una de las 4 categorías. Dentro de cada categoría, se distribuye de manera proporcional entre sus acertantes.

```
function Total_Dinero (Recaudacion: in Natural;
                        Num_Boleto: in Integer;
                        Datos_Premios: in T_Datos_Premios) return Natural;
```

2. Agenda (4,5 Puntos)

```

type Hora is record
  Ocupada: Boolean; -- si vale True esta hora está ocupada
  Empresa: String(1..20); -- solo contiene un valor cuando Ocupada vale True
end record;

type Dia is array(8 .. 21) of Hora; -- desde las 8:00 hasta las 21:00 horas

type Agenda is array(1 .. 31) of Dia;

```

La estructura de datos anterior guarda los datos de una agenda. En ella se apuntan, para cada día del mes, los trabajos que realiza un trabajador. Por cada día y cada hora se apunta si el trabajador tiene un trabajo asignado y, en caso afirmativo, en qué empresa lo hace.

- A. (3 Puntos)** Al final del mes, un trabajador quiere saber cuál es la empresa a la que más horas ha dedicado, junto con el número de horas que ha invertido en ella. Se pide escribir un subprograma que lo calcule. Como máximo, un trabajador puede trabajar para 20 empresas diferentes. En caso de que hubiera más de una empresa con el mismo número de horas, se devolverá una de ellas.

```

procedure Calcular_Empresa_con_mas_Horas_Trabajadas (
  Ag           : in Agenda;
  Nombre_Empresa : out String;
  Horas        : out Integer) is
-- Precondición:
-- Postcondición: el resultado es el nombre de la empresa que más
--                 horas ocupa en la agenda, junto con las horas invertidas en ella.
--                 En caso de que hubiera varias empresas con el mismo
--                 número de horas, se devolverá cualquiera de ellas.

```

Se tiene la siguiente estructura de datos:

```

Numero_de_Trabajadores: constant Integer := 100;

type Empresa is array(1 .. Numero_de_Trabajadores) of Agenda;

```

- B. (1,5 Puntos)** Una empresa tiene 100 trabajadores, y se guarda una agenda con los horarios de cada uno de ellos. De cara a hacer reuniones, se quiere obtener una estructura de datos donde se indiquen las horas en las que todos los trabajadores están libres (es decir, tienen esa hora marcada en su agenda como libre).

```

procedure Calcular_Horas_Libres ( E           : in Empresa;
                                 Horas_Libres : out Agenda);
-- Precondicion:
-- Postcondicion: El resultado sera una agenda donde estaran marcadas como
--                 libres las horas que todos los trabajadores tengan libre, y las
--                 demas estaran como ocupadas (es decir, si algun trabajador
--                 tiene esa hora ocupada)

```

3. Palíndromo (1,5 puntos)

```
N: constant Integer := 100;  
type Frase is array(1 .. N) of Character;
```

Diremos que una frase es un palíndromo si se lee igual de izquierda a derecha y de derecha a izquierda, exceptuando espacios. Por ejemplo, las frases (1), (2) y (3) son palíndromos, mientras que las frases (4), (5) y (6) no lo son.

(1)

h	a	y		s	a	l	a	s			y				a	h	
---	---	---	--	---	---	---	---	---	--	--	---	--	--	--	---	---	--

(2)

		a	b	c				c						b			a
--	--	---	---	---	--	--	--	---	--	--	--	--	--	---	--	--	---

(3)

a	b	c	d	c	b	a											
---	---	---	---	---	---	---	--	--	--	--	--	--	--	--	--	--	--

(4)

a	b															a	b
---	---	--	--	--	--	--	--	--	--	--	--	--	--	--	--	---	---

(5)

				a			b								c	
--	--	--	--	---	--	--	---	--	--	--	--	--	--	--	---	--

(6)

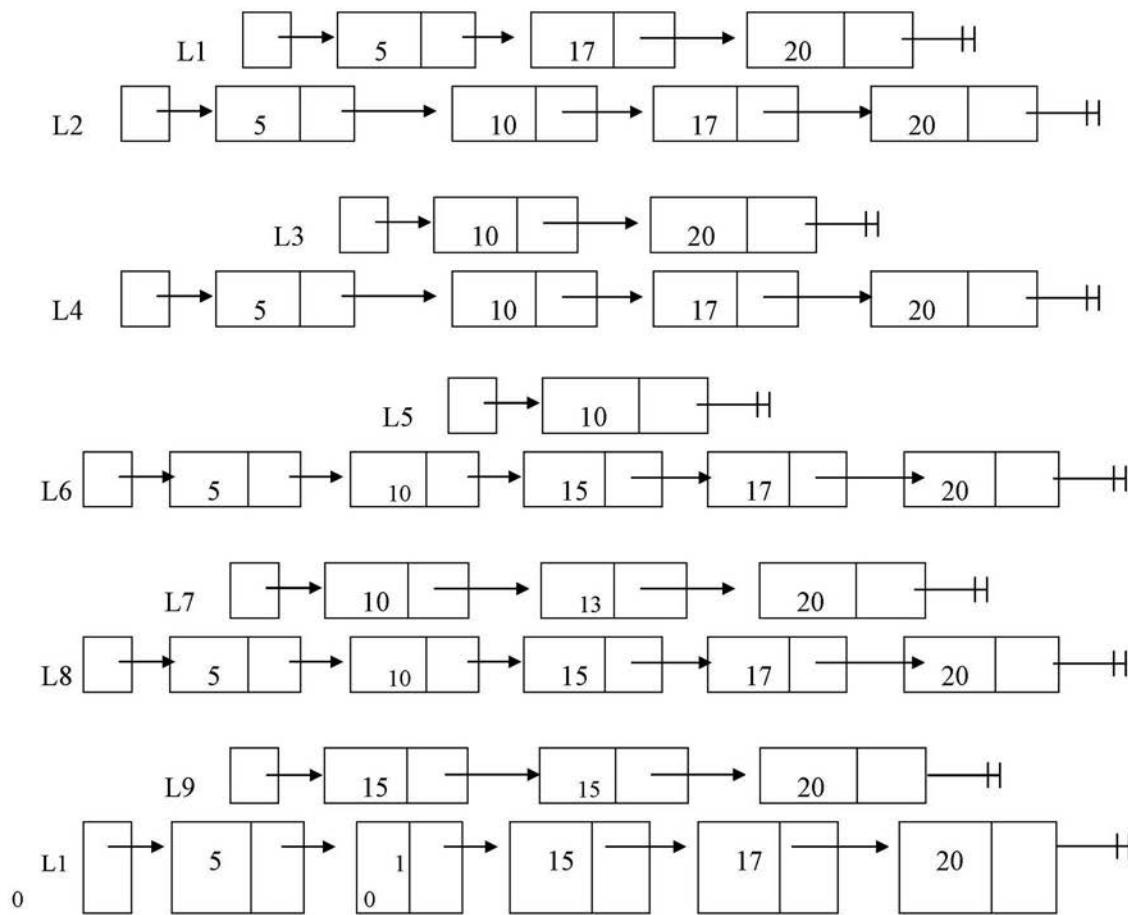
				a	b	c										
--	--	--	--	---	---	---	--	--	--	--	--	--	--	--	--	--

Se pide diseñar un subprograma recursivo que, dada una frase, diga si es un palíndromo o no. (1,5 puntos).

4. Sublista intermitente (2 puntos)

Dadas dos listas ligadas L1 y L2 ordenadas ascendentemente, escribir un subprograma que diga si todos los elementos de L1 se encuentran en L2 en el mismo orden.

Por ejemplo, L1 es sublista intermitente de L2, L3 lo es de L4, y L5 también de L6, mientras que L7 no es sublista intermitente de L8, ni L9 lo es de L10.



En la solución se valorará la eficiencia del algoritmo (sólo se puede recorrer cada lista una sola vez).

3. Urbanización (4 puntos)

Se tienen las siguientes definiciones de tipos de datos para representar los apartamentos de una urbanización. Por cada apartamento se quieren guardar diferentes informaciones: propietario actual y anterior (nombre, apellido y DNI por cada uno) y precio. La urbanización tiene 20 bloques de edificios. Cada edificio tiene 10 plantas, y en cada planta se tienen varios apartamentos, pudiendo variar su número entre 1 y 8.

```
type Identificacion is record
    Nom,
    Apel : String (1 .. 20);
    Dni : Integer;
end record;

type Info_Apartamento is record
    Precio : Integer;
    Propietario_Actual : Identificacion;
    Propietario_Anterior : Identificacion;
end record;

Maximo_De_Apartamentos_Por_Planta: constant Integer := 8;

type Tabla_Apartamentos is array
    (1 .. Maximo_De_Apartamentos_Por_Planta) of Info_Apartamento;

type Planta is record
    Apartamentos: Tabla_Apartamentos;
    Num_Apartamentos: Integer; -- valor entre 1 y 8
end record;

Max_Plantas : constant Integer := 10;

type Edificio is array (1 .. Max_Plantas) of Planta;
Num_Bloques : constant Integer := 20;

type Urbanizacion is array (1 .. Num_Bloques) of Edificio;
```

Escribir el subprograma correspondiente a las siguiente especificación:

```
function El_de_Mayor_Patrimonio(U: in Urbanización) return Integer
-- precondición:
-- postcondición: el resultado es el DNI de la
-- persona de la urbanización que tiene un mayor patrimonio
-- (el patrimonio es la suma de los valores de los pisos que posee)
```

Programación I – 3 de septiembre de 2003

1. Facultades

Una determinada Universidad tiene 5 Facultades, a saber, Informática, Medicina, Filosofía, Derecho y Pedagogía. La Secretaría de esta Universidad recibe preinscripciones a las diferentes Facultades de un número de alumnos.

De cara a facilitar la labor de la Dirección para determinar qué alumnos admite en cada Facultad, la Secretaría dispone de la siguiente información relativa a las cualidades recomendables (imaginación, capacidad de abstracción, minuciosidad, memoria, ... así hasta N cualidades diferentes, donde N es un valor constante) que un alumno debe poseer para una determinada carrera:

```
Num_Cualidades: constant Integer := N;
-- Hay N cualidades, numeradas del 1 al N

type Vector_de_Cualidades is array(1..Num_Cualidades) of Boolean;
-- true si el alumno posee esa cualidad, y false si no

subtype Facultad is Integer range 1..5;
-- 1: Informatica, 2:Medicina, ..., 5: Pedagogia

type Vector_Facultades is array(Facultad) of Vector_de_Cualidades;
```

A. **(1,5 puntos)** Implementar el siguiente subprograma, donde VF son las cualidades exigidas por una Facultad, y VA son las cualidades que posee un alumno:

```
function Cumple_Cualidades(VF:      in Vector_de_Cualidades;
                            VA:      in Vector_de_Cualidades)
            return Boolean
-- pre: VF contiene datos de una Facultad y VA datos
--       de un alumno
-- post: el resultado será cierto si el alumno (VA) cumple
--       todas las cualidades exigidas en la Facultad (VF), y
--       falso si no
```

Nota: se valorará especialmente la eficiencia de la solución.

Al comienzo del curso todos los alumnos rellenan unos cuestionarios que determinan qué cualidades posee cada uno, que se encuentran guardados en la siguiente estructura de datos:

```
subtype Cadena is String(1..20);

type Info_Alumno is record
    Nombre      : Cadena;
    Instituto   : Cadena;
    Cualidades  : Vector_de_Cualidades;
end record;

Max_Alumnos: constant Integer := 20000;

type Vector_de_Alumnos is array(1..Max_Alumnos) of Info_Alumno;

type Lista_de_Alumnos is record
    Tabla       : Vector_de_Alumnos;
    Num_Alumnos : Integer;
end record;
```

B. **(2,5 puntos)** Implementar un procedimiento que, dada una lista de alumnos (de tipo `Lista_de_Alumnos`) y los datos de las Facultades (de tipo `Vector_Facultades`), obtenga una estructura de datos con 5 listas, una por Facultad, de los alumnos que cumplen todas las cualidades necesarias para inscribirse en ella.

Si un mismo alumno cumple los requisitos de varias Facultades, aparecerá en la lista de cada una de esas Facultades.

Nota: para resolver este ejercicio se puede utilizar la función `Cumple_Cualidades` del apartado A.

C. **(2,5 puntos)** Implementar un procedimiento que, dada la estructura con 5 listas de alumnos del ejercicio anterior como datos, obtenga una lista que contenga únicamente los alumnos que han sido admitidos en **todas** las carreras.

Para ello puede ser útil implementar el siguiente subprograma:

```
function Intersección(L1, L2: in Lista_de_Alumnos)
    return Lista_de_Alumnos
-- pre:
-- post: el resultado es la lista de alumnos que pertenecen
--        a las dos listas L1 y L2
```

2. Matriz transpuesta (1,5 puntos)

```
N: constant Integer := 1000;  
  
type Matriz is array (1..N, 1..N) of Integer;
```

Especificar y escribir un subprograma recursivo que, dada una matriz cuadrada, obtenga su transpuesta:

```
procedure Transponer( ... )  
-- Postcondición: el resultado es la matriz transpuesta a la original
```

Ejemplo, dada la matriz M1, su transpuesta será la matriz M2.

1	2	4	8
3	7	9	11
6	12	24	32
5	10	20	30

(M1)

1	3	6	5
2	7	12	10
4	9	24	20
8	11	32	30

(M2)

Un método para transponer una matriz es el siguiente: intercambiar los valores de la primera fila con los de la primera columna, y repetir el proceso con el resto de la matriz.

3. Lista de palabras (2 puntos)

Dadas las siguientes definiciones de tipos de datos:

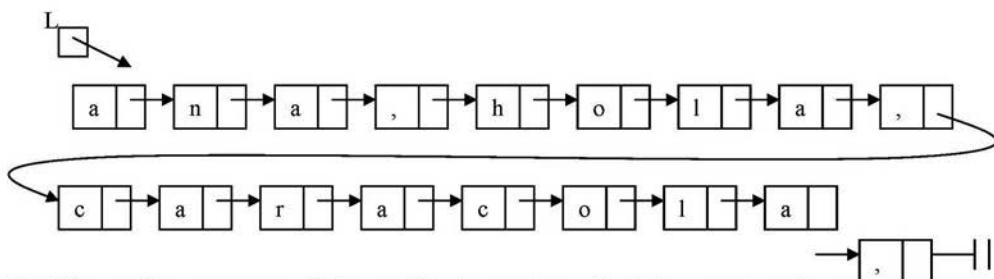
```
Max_Letras : constant Integer := 20;

type Palabra is record
    Letras: String (1 .. Max_Letras);
    Cont : Integer;
end record;

type Tabla_Palabras is array(1..100) of Palabra;

type Todos_los_Vocablos is record
    Vocablos: Tabla_Palabras;
    Cuantos_Vocablos: Integer;
end record;

type Nodo;
type Lista is access Nodo;
type Nodo is record
    Info: Character;
    Sig : Lista;
end record;
```



Escribir un subprograma que, dada una lista de caracteres, donde los caracteres forman palabras y estas palabras acaban siempre con una coma, generará una estructura de datos de tipo **Todos_los_Vocablos**, con la lista de palabras de **L**.

Por ejemplo, dada la lista **L** de la figura, el resultado contendrá las 3 palabras (ana, hola, caracola)

```
procedure Obtener_Vocablos(L : in Lista;
                           V : out Todos_los_Vocablos)
-- pre: L contiene una secuencia de palabras
--       todas las palabras de L acaban con una coma
-- post: V contiene los vocablos de L
```

Programación I – 29 de Enero de 2004

1. Enteros grandes (3 puntos)

```
Maximo_de_Digitos: constant Integer:= 100;  
  
type Entero_Grande is array (1 .. Maximo_de_Digitos) of Integer;
```

Suponer un vector del tipo anterior utilizado para representar valores enteros muy grandes. Por ejemplo, el valor 1223334445678556111 se representaría de la siguiente manera:

1	2	3	...	89	90	91	92	93	94	95	96	97	98	99	100
0	0	0	...	4	4	5	6	7	8	5	5	6	1	1	1

Escribir un subprograma que obtenga la resta de dos enteros utilizando esta clase de números.

```
function Resta(E1, E2 : in Entero_Grande) return Entero_Grande is  
    -- Pre: E1 y E2 tienen ceros en las cifras no ocupadas más significativas  
    -- (más a la izquierda)  
    -- E1 y E2 son positivos  
  
    -- Post: si E2 > E1  
    -- entonces el resultado es cero  
    -- si no el resultado es la resta de los dos valores de entrada, E1 - E2
```

2. Lista de enteros (4 puntos)

```
type Tabla is array (1..1000) of Integer;  
  
type Lista_Numeros is record  
  T: Tabla;  
  Cont: Integer;  
end record;
```

- a) Especificar y escribir un subprograma que, dada una lista de números (de tipo `Lista_Numeros`), genere otra lista que contenga los números que mantienen el orden ascendente, tomando como referencia el primer elemento de la lista.

Por ejemplo, dada la lista de valores `v1`, el resultado sería la lista `v2`:

<code>v1</code>	1	2	3	4	5	6	7	8	9	...
	12	14	3	11	4	17	15	28	24	

<code>v2</code>	1	2	3	4	5	6	7	8	9	...
	12	14	17	28						

```
function Subsecuencia_Ascendente(L: in Lista_Numeros) return Lista_Numeros is
  -- Precondición:
  -- Postcondición: el resultado es una lista que contiene los números de L que
  --                 mantienen el orden ascendente, tomando como referencia el
  --                 primer elemento de la lista
```

- b) Especificar y escribir un subprograma que, dada una lista de números (de tipo `Lista_Numeros`) ordenada ascendentemente y un valor entero, inserte ese elemento en la lista manteniendo el orden ascendente.

Ejemplo:

<code>v2</code>	1	2	3	4	5	6	7	8	9	...
	12	14	17	28						

después de insertar el valor 15, la lista quedaría de esta manera:

<code>v2</code>	1	2	3	4	5	6	7	8	9	...
	12	14	15	17	28					

```
procedure Insertar_Ord(L: in out Lista_Numeros; V: in Integer) is
  -- Precondición: L está ordenada ascendentemente
  -- Postcondición: se ha insertado V en L manteniendo el orden ascendente
```

3. Agenda (3 puntos)

Se quiere utilizar una agenda para organizar las actividades de una persona. La información será la que aparece en el siguiente cuadro, a saber: por cada día de la semana aparecerán una serie de tareas ordenadas según la hora. Cada actividad tiene una hora de comienzo y otra de final, sin sobrepasar los límites de un día (8:30 y 20:00).

2 actividades el lunes
930-1400 Visita abogado
1800-1900 Inglés
4 actividades el martes
900-1000 Reunión con Jon
1000-1130 Reunión con Ane
1130-1300 Preparar informe
1730-1830 Polideportivo
3 actividades el miércoles
830-1100 Reunión con Mikel
1330-1600 Comida
1800-1930 Inglés
2 actividades el jueves
900-1100 Preparar informe
1730-1830 Polideportivo
1 actividad el viernes
1800-2000 Teatro
0 actividades el sábado
0 actividades el domingo

Para utilizar ese tipo de agendas, hemos definido los siguientes tipos de datos:

```
type Dia_de_la_Semana is (Lunes, Martes, Miércoles, Jueves, Viernes, Sábado, Domingo);

Primera_Hora: constant Integer := 830;
Última_Hora: constant Integer := 2000;
subtype Hora is Integer range Primera_Hora .. Última_Hora;

type Intervalo_Horario is record
    Comienzo, Fin : Hora;
end record;

Max_Caracteres_Actividad : constant Integer := 25;
subtype Nombre_Actividad is String (1 .. Max_Caracteres_Actividad);

type Actividad is record
    Intervalo : Intervalo_Horario;
    Nombre : Nombre_Actividad ;
end record;

Max : constant Integer := 50;
type Tabla_de_Actividades is array (1 .. Max) of Actividad;

type Lista_de_Actividades is record
    Info : Tabla_de_Actividades;
    Cuantas : Integer range 0 .. Max;
end record;

type Agenda_Semanal is array (Dia_de_la_Semana) of Lista_de_Actividades;
```

Dada una agenda, obtener el día de la semana con más horas ocupadas:

```
function Dia_Mas_Ocupado (A : in Agenda_Semanal) return Dia_de_la_Semana is
-- Pre: las actividades comienzan y acaban en intervalos de media hora,
--       desde las 830 hasta las 2000
-- Post: el resultado es el día de la semana con más horas ocupadas
--       en caso de que haya varios días con el número máximo de horas ocupadas,
--       devolverá el primero
```

En el ejemplo, el día más ocupado es el miércoles, con 6.5 horas ocupadas.

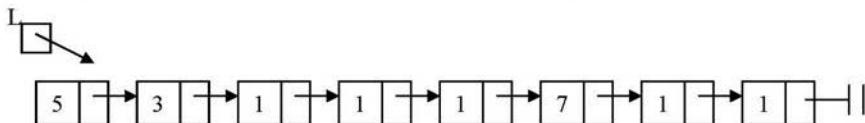
Programación I – 10 de junio de 2004

1. Eliminar subsecuencias (2 puntos)

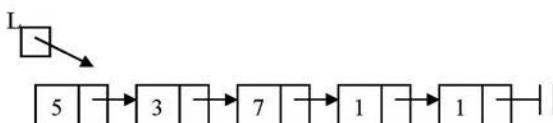
Se pide implementar el siguiente subprograma:

```
procedure Eliminar_Subsecuencias_De_Unos (L : in out Lista; N: in Integer ) is
  -- Pre:
  -- Post: se han eliminado de L todas las subsecuencias de N unos consecutivos
  --       en caso de que haya subsecuencias con más de N unos, se eliminarán
  --       de N en N
```

Por ejemplo, después de la llamada `Eliminar_Subsecuencias_De_Unos(L, 3)`



la lista quedaría de la siguiente manera:



2. Escribir múltiplos (2 puntos)

```
type Tabla is array (Integer range 1 .. N) of Integer;
```

Especificar y escribir un subprograma recursivo que, dada una tabla y un número X, escriba en pantalla los números de la tabla hasta el número X que sean múltiplos de la posición en la que se encuentra X.

```
procedure Escribir_Multiplos (T : in Tabla;
                               X : in Integer;
                               i : in integer;
                               Pos : out Integer ) is
  -- Pre: X se encuentra en T
  --       i indica la parte de la tabla que no se ha examinado todavía
  -- Post: Pos contiene la posición donde se encuentra X
```

1	2	3	4	5	6	7	8	9	10	...	N
8	36	7	24	10	55	56	18	12	...		

Por ejemplo, la llamada `Escribir_Multiplos(T, 55, 1, P)` escribiría en pantalla 24 y 36.

P tendría valor 6, porque el 55 se encuentra en la posición 6 de la tabla, y 24 y 36 son múltiplos de 6.

3. Venta de CDs (6 puntos)

Se tienen las siguientes definiciones de tipos de datos para representar los CDs que han sacado una serie de cantantes:

```
Max_Cantantes: constant Integer := 10000;  
  
type Lista_Cantantes_con_CDs is record  
    Num_Cantantes: Integer; -- 0 .. Max_Cantantes  
    Cantantes: Tabla_Cantantes_CDs;  
end record;  
  
type Tabla_Cantantes_CDs is array (1 .. Max_Cantantes) of Cantante;  
  
type Cantante is record  
    Nom: String (1 .. 20);  
    Num_CDs : Integer;  
    CDs: Tabla_CDs;  
end record;  
  
type Tabla_CDs is array (1 .. 40) of CD;  
  
type CD is record  
    Titulo : String (1 .. 20);  
    PVP : Integer;  
    Num_Canciones: Integer;  
end record;
```

Por cada cantante, se tiene una lista con los CDs que ha sacado, indicando por cada uno el título, el precio de venta y el número de canciones que contiene. Los CDs de un cantante se hallan ordenados por orden ascendente del precio (PVP).

Por ejemplo, la lista podría ser de la siguiente manera:

Madonna	ACDC	U2	Oasis	Sabina
3	4	2	1	3
CD1, 15				
CD2, 20	CD2, 25	CD2, 20		CD2, 20
CD3, 40	CD3, 35			CD3, 40
	CD4, 60			

Por otro lado, cuando una persona quiere hacer un pedido de CDs de unos determinados cantantes, sus datos quedan guardados en una estructura de datos de la manera siguiente:

```
type Carro_de_la_Compra is record  
    Num_Cantantes: Integer; -- 0 .. N  
    TC: Tabla_Cantantes;  
end record;  
type Tabla_Cantantes is array (1 .. N) of Info_Compra; -- N es una cte.  
  
type Info_Compra is record  
    Nom : String (1 .. 20); -- nombre del cantante  
    Indice_Cantante : Integer;  
    Indice_CD : Integer;  
end record; -- inicialmente solo contiene el nombre del cantante
```

a) (2 puntos) Escribir el subprograma correspondiente a la siguiente especificación:

```
procedure Buscar_Cantantes(LCD : in      Lista_Cantantes_con_CDs;
                           CC  : in out Carro_de_la_Compra)
-- precondition: CC contiene nombres de cantantes en el campo Nom
--               Los campos Indice_Cantante e Indice_CD de CC tienen valor indefinido.
--               Todos los cantantes de CC existen en LCD
-- postcondición: por cada cantante de CC, se ha añadido en el campo
--                 Indice_Cantante la posición que ocupa en LCD, es decir,
--                 el lugar donde se encuentran los datos de ese cantante.
--                 El campo Indice_CD contiene el índice del CD más caro de ese cantante
```

b) (4 puntos) A la hora de hacer un pedido, un cliente especifica en el carro de la compra una lista de los cantantes que quiere, junto con la cantidad de dinero disponible. Dados estos datos, queremos desarrollar un subprograma que obtenga una lista de los CDs a comprar, teniendo en cuenta las siguientes restricciones:

1. Los cantantes del carro de la compra se hallan ordenados de mayor a menor prioridad. Esta prioridad se tendrá en cuenta en el caso de que el dinero no sea suficiente para comprar los CDs.
2. Se intentará comprar los últimos CDs (los más caros) de cada cantante, si llega.
3. En caso de que el dinero no sea suficiente para comprar los CDs se probará con el siguiente CD más barato del cantante de menor prioridad (al comienzo es el último cantante de CC). En caso de que tampoco sea suficiente con el CD más barato del cantante de menor prioridad, se cogerá el segundo cantante de menor prioridad, y así sucesivamente, hasta encontrar una selección de CDs cuyo precio sea menor o igual a la cantidad especificada.
4. Siempre se comprará un CD por cada uno de los cantantes pedidos.
5. En caso de que ni siquiera cogiendo el CD más barato de cada cantante fuera suficiente, se devolverá como resultado el carro con los CDs más baratos.

El subprograma tendrá la siguiente especificación:

```
procedure Hacer_la_Compra(LCD      : in      Lista_Cantantes_con_CDs;
                           CC       : in out Carro_de_la_Compra;
                           Precio_Max : in      Integer)
-- precondition: CC contiene nombres de cantantes en el campo Nom
--               Los campos Indice_Cantante y Indice_CD de CC tienen valor indefinido
--               Todos los cantantes de CC existen en LCD
-- postcondición: por cada cantante de CC, se ha añadido en el campo
--                 Indice_Cantante la posición que ocupa en LCD, es decir,
--                 el lugar donde se encuentran los datos de ese cantante
--                 El campo Indice_CD contiene el índice de los CDs que se han comprado
```

Por ejemplo, dada la lista del ejemplo anterior, el resultado de la llamada:

```
Hacer_la_Compra(LCD,
                  [(ACDC, ?, ?) (Oasis, ?, ?), (Sabina, ?, ?)]
                  56)
```

devolvería [(ACDC, 2, 2) (Oasis, 4, 1), (Sabina, 5, 1)]

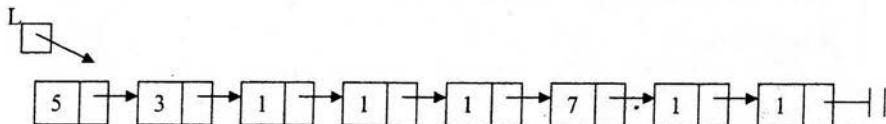
Programación I – 6 de septiembre de 2004

1. Números iguales (3 puntos)

Se pide implementar el siguiente subprograma **usando recursividad**:

```
function Iguales (L           : in Lista;
                  Num        : in Integer;
                  N_Cifras : in Integer )
    return boolean

-- Pre: Num >= 0
--     N_Cifras contiene el número de cifras de Num
--     L contiene un número entero positivo, con un dígito por cada nodo,
--     ...
-- Post: el resultado es true si los dígitos de L, tomados de izquierda a
--       derecha, forman el número Num
```



Por ejemplo, dada la lista anterior:

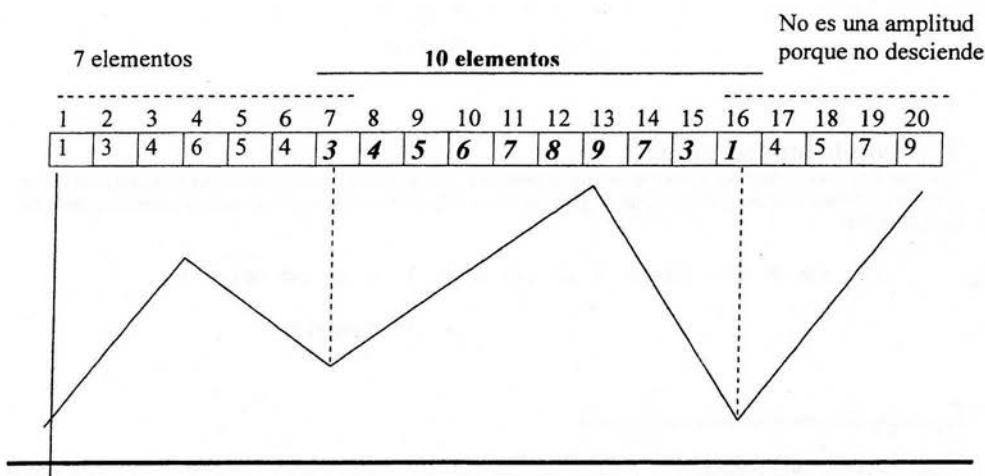
- la llamada `Iguales(L, 53111711, 8)` devolverá `true`
- la llamada `Iguales(L, 431117119, 9)` devolverá `false`
- la llamada `Iguales(L, 53211711, 8)` devolverá `false`
- la llamada `Iguales(L, 531, 3)` devolverá `false`
- la llamada `Iguales(L, 531117119, 9)` devolverá `false`

2. Amplitud (2,5 puntos)

```
type Vector_de_Enteros is array (Integer range <>) of Integer;
```

Queremos escribir un subprograma que tome como entrada los datos de una función matemática representada por un vector de enteros. Este vector estará formado por subsecuencias ascendentes y descendentes que representan amplitudes de segmentos, donde un segmento está formado por una serie ascendente de valores seguida de una serie descendente. El subprograma deberá devolver la longitud del segmento de mayor amplitud.

Ejemplo:



Dado el vector anterior, la función devolverá 10, que es el tamaño del mayor segmento que contiene.

```
function Calcular_Amplitud_Max (VA: in Vector_de_Enteros) return Integer
-- Pre: El vector VA tendrá al menos dos elementos
-- -- Comenzará con una subsecuencia ascendente
-- Post: El resultado es la longitud del segmento de mayor amplitud
-- -- En caso de que no haya ningún segmento el resultado será cero
```

3. Cine (4,5 puntos)

Se tienen las siguientes definiciones de tipos de datos para representar una sala de cine:

```
NF: constant Integer := 100; -- número de filas
NBF: constant Integer := 200; -- número de butacas por fila

Num_Billete_Max: constant Integer := 1000000;

subtype NBillete is Integer range 1 .. Num_Billete_Max;

type Butaca is record
    Ocupada: Boolean; -- true si ocupada y false si libre
    Billete: NBillete; -- solo si la butaca está ocupada
end record;

type Sala_de_Cine is array (1 .. NF, 1 .. NBF) of Butaca;
```

Por ejemplo, la siguiente figura representa un ejemplo de una sala, donde las posiciones en blanco representan butacas libres (el campo Ocupada vale False):

F1		T 222				T 232				
F2			T 469	T 422	T 411	T 241	T 111	T 321	T 543	
F3	T 333	T 354	T 534							T 567
F4	T 987		T 814		T 666					T 589
F5		T 123	T 288	T 999	T 989	T 787	T 872	T 765		T 259
F6	T 555			T 898			T 873			T 247

3.a) Diseñar e implementar el siguiente subprograma (2 puntos):

```
procedure Obtener_Hueco_Mas_Grande (Sala : in Sala_de_Cine;
                                      Fil, Col : out Integer;
                                      NButacas : out Integer)

-- Pre:
-- Post: En caso de que no hubiera ninguna butaca libre, NButacas valdrá
--        cero y Fil, Col tendrán un valor indeterminado
--        si no, se devolverá la posición de comienzo (fila y columna),
--        dada por Fil y Col, del mayor "hueco" del cine, donde un hueco es una
--        secuencia de butacas libres consecutivas en una misma fila
--        Si hubiera más de un hueco con la longitud máxima, se devolverá la
--        posición del primero
--        NButacas contendrá la longitud de ese hueco
```

En el ejemplo anterior, el resultado sería la fila 3, columna 4, con 6 butacas libres.

3.b) Especificar, diseñar e implementar un subprograma que tome como entrada una sala de cine parcialmente llena y una lista de personas que quieren asistir a esa sesión. Por cada persona se tiene su DNI y el número de billete que ha comprado. El subprograma deberá asignar esas personas a butacas de manera que el mayor número de ellas estén en butacas consecutivas, del modo siguiente:

- Se buscarán sucesivamente los huecos más grandes, a los que se irán asignando las personas de la lista.
- El proceso acabará cuando no haya elementos en la lista o la sala esté completamente llena.

```
procedure Procesar_Espectadores (Sala : in out Sala_de_Cine;
                                 LP   : in out Lista_Personas)
-- Pre: LP contiene una lista de personas que quiere entrar en la sala
-- Post: se han introducido los elementos de LP en la sala, empezando por los
--        huecos de mayor tamaño
--        En caso de que la sala se llene, LP contendrá las personas que se
--        han quedado sin butaca
```

Se puede resolver este problema de dos maneras alternativas, de las que **deberás elegir una, según la implementación del tipo de datos Lista_Personas:**

3.b.1) Implementación estática (arrays y/o registros) de la lista de personas **(1,5 puntos)**

Suponer que como mucho hay 1000 personas en la lista.

3.b.2) Implementación dinámica (apuntadores) de la lista de personas **(2,5 puntos)**

En los dos casos se pide:

- Definiciones de tipos de datos necesarios (Lista_Personas y relacionados)

Diseño e implementación de la solución al problema propuesto.

Programación I – 1 de febrero de 2005

1) Cuadrado mágico (2,5 puntos)

Escribir un subprograma que dado un valor entero n (un valor impar), escriba un cuadrado mágico de tamaño n . Por ejemplo, el cuadrado mágico de tamaño 7 es el siguiente:

```
0 * * * * * 0
* 0 * * * 0 *
* * 0 * 0 * *
* * * 0 * * *
* * 0 * 0 * *
* 0 * * * 0 *
0 * * * * * 0
```

2) Compilador (3,5 puntos)

Un error de compilación muy común en los programas es el que proviene de abrir paréntesis y dejarlos sin cerrar o cerrar paréntesis que no se han abierto. Se pide hacer un programa en Ada que sea capaz de detectar errores de parentización.

```
if (a > b) then { b := a+n ) ; z := x+y }
```

↑
Error!!!

Se pide implementar el siguiente subprograma:

```
type Cadena is record
  Long: Integer;
  Caracteres: String(1 .. 100);
end record;

function Balanceado( C: in Cadena ) return Boolean
-- pre: C contiene un línea de un programa, que puede contener
--       paréntesis y llaves
--       Ejemplo:      if (a > b) then { b := a+n ) ; z := x+y }
--       post: el resultado es true si C está balanceada y false si no
```

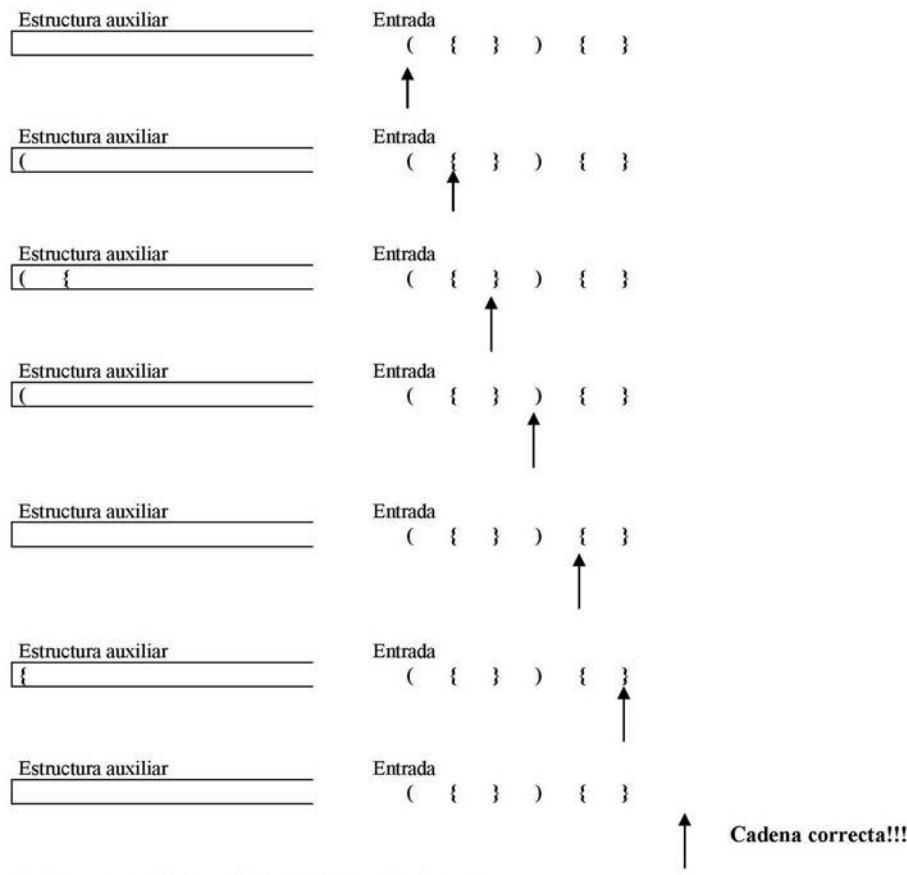
El proceso a seguir es usar un vector auxiliar y recorrer la cadena de entrada carácter a carácter:

- si se encuentra un paréntesis o una llave de apertura, entonces se mete en la estructura auxiliar
- si se encuentra un paréntesis o una llave de cierre, entonces debe coincidir con el último elemento introducido en la estructura auxiliar, si lo hubiera. En este caso se debe eliminar el paréntesis o llave de apertura de la estructura auxiliar, ya que ha sido emparejado correctamente.
- Los caracteres que no son paréntesis o llaves se ignorarán

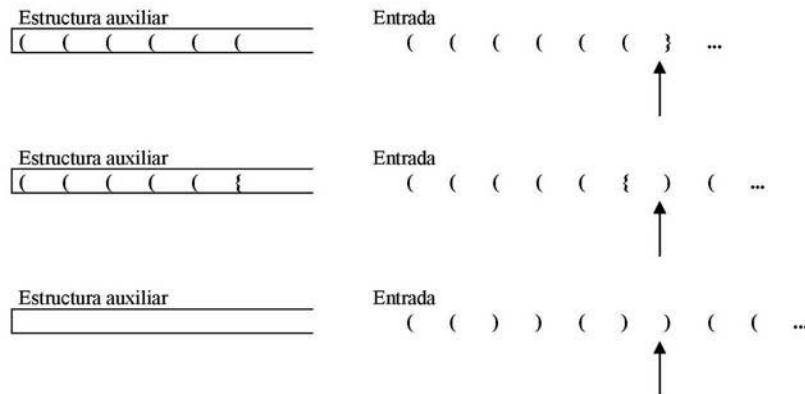
El proceso acaba cuando se encuentra uno de los siguientes casos:

- a) Se acaba el recorrido de la cadena de entrada.

Ejemplo:



- b) Se ha encontrado una de las siguientes configuraciones:



3. Agenda (4 puntos)

Se quiere utilizar una agenda para organizar las actividades de una persona. La información será la que aparece en el siguiente cuadro, a saber: por cada día de la semana aparecerán una serie de tareas ordenadas según la hora. Cada actividad tiene una hora de comienzo y otra de final, sin sobrepasar los límites de un día (8:30 y 20:00).

2 actividades el lunes
930-1400 Visita abogado
1800-1900 Inglés
4 actividades el martes
900-1000 Reunión con Jon
1000-1130 Reunión con Ane
1130-1300 Preparar informe
1730-1830 Polideportivo
3 actividades el miércoles
830-1100 Reunión con Mikel
1330-1600 Comida
1800-1930 Inglés
2 actividades el jueves
900-1100 Preparar informe
1730-1830 Polideportivo
1 actividad el viernes
1100-1200 Inglés
1800-2000 Teatro
0 actividades el sábado
0 actividades el domingo

Para utilizar ese tipo de agendas, hemos definido los siguientes tipos de datos:

```
type Dia_de_la_Semana is (Lunes, Martes, Miércoles, Jueves, Viernes, Sábado, Domingo);

Primera_Hora: constant Integer := 830;
Última_Hora: constant Integer := 2000;
subtype Hora is Integer range Primera_Hora .. Última_Hora;

type Intervalo_Horario is record
    Comienzo, Fin : Hora;
end record;

Max_Caracteres_Actividad : constant Integer := 25;
subtype Nombre_Actividad is String (1 .. Max_Caracteres_Actividad);

type Actividad is record
    Intervalo : Intervalo_Horario;
    Nombre : Nombre_Actividad;
end record;

Max : constant Integer := 50;
type Tabla_de_Actividades is array (1 .. Max) of Actividad;

type Lista_de_Actividades is record
    Info : Tabla_de_Actividades;
    Cuantas : Integer range 0 .. Max;
end record;

type Agenda_Semanal is array (Dia_de_la_Semana) of Lista_de_Actividades;
```

Dada una agenda, obtener el nombre de la actividad que se repite más veces:

```
function Actividad_Mas_Frecuente (A : in Agenda_Semanal) return Nombre_Actividad is
-- Post: el resultado es el el nombre de la actividad que se repite más veces
-- Si no hay ninguna actividad, el resultado será una cadena formada por blancos
-- Si varias actividades de igual frecuencia, devolver la primera
```

En el ejemplo, la actividad más frecuente es Inglés (3 veces).

3. Copia de exámenes (6 puntos)

De cara a evitar las copias de exámenes, una profesora ha diseñado un método que se basa en la frecuencia de aparición de las palabras del examen. El método se basa en codificar un examen como la lista de las palabras que aparecen en él, junto con el número de apariciones de cada palabra. Para ello, se han hecho las siguientes definiciones de tipos de datos:

```
Max_Palabras: constant Integer := 1000;

type Datos_Palabra is record
  Palabra: String(1..20);  -- hay espacios en las posiciones no llenas
  N_Apariciones: Integer;
end record;

type Tabla_Palabras is array (1 .. Max_Palabras) of Datos_Palabra;

type Examen is record
  Num_Palabras_Diferentes: Integer;
  Palabras: Tabla_Palabras;
end record;
```

La idea básica para detectar copias es que los programas copiados tengan el mismo número de apariciones de variables (los nombres pueden no coincidir, ya que es un método habitual de copia el cambiar únicamente los nombres de las variables). Por ejemplo, los procedimientos a y b son sospechosos de copia, pero no si los comparamos con el c:

<pre>procedure a is x1, x2: integer; begin if x1 > x2 then v4 := 0; end if; end a;</pre>	<pre>procedure b is y1, y2: integer; begin if y1 > y2 then indice := 0; end if; end b;</pre>	<pre>procedure c is z1, z2, z3: integer; begin if z1 > z2 + z3 then z3 := z3 + 1; end if; end c;</pre>
---	---	---

Para codificar un examen, la profesora ha cogido las palabras que aparecen en él y las ha metido en una estructura de datos, contando por cada palabra cuántas veces aparece. En este proceso las palabras se guardan ordenadas alfabéticamente y además no se cuentan las palabras reservadas (if, while, for, procedure, ...). Por ejemplo, esta podría ser la representación de 4 exámenes diferentes:

examen 1	examen 2	examen 3	examen 4
5	5	5	6
i 12	elem 7	elem 7	cont 12
indice 15	indice2 15	indice2 15	cont2 3
var3 7	j 9	j 12	j 6
var5 9	x 12	var3 9	w 24
var7 12	z 12	var19 9	z1 17
			z5 13

Por ejemplo, los exámenes 1 y 2 son sospechosos de copia, al contener el mismo número de variables y además, las frecuencias de aparición coinciden (15, 12, 12, 9, 7). Por contra, el examen 1 y el 3 no son sospechosos de copia, porque las frecuencias son diferentes ((15, 12, 12, 9, 7) y (15, 12, 9, 9, 7)). El examen 4 no es copia de los demás porque su número de palabras es diferente de los otros 3.

a) (2,5 puntos) Escribir el subprograma correspondiente a la siguiente especificación:

```
function Se_Han_Copiado(Ex1, Ex2 : in Examen) return Boolean
-- precondición: Ex1 y Ex2 están ordenadas alfabéticamente
-- postcondición: El resultado es true si el número de palabras de Ex1 y Ex2
--                 es el mismo y además todas las palabras coinciden en su número
--                 de apariciones
```

b) (3,5 puntos) Se tiene una matriz con los datos del examen de programación. Cada elemento de la matriz representa dónde se sentó cada alumno. Por cada alumno se tiene su número de identificación junto con la codificación del examen que ha realizado. Se quiere sacar una lista de las parejas de alumnos que son sospechosos de copia. Sabemos que un alumno puede copiar a los alumnos que se encuentran delante, detrás, a su derecha o a su izquierda.

					4444														
				8888	2222	3333													
							5555												

```
Max_Filas: constant Integer := 20;
Max_Columnas: constant Integer := 20;

type Datos_Mesa is record
    Ocupada: Boolean;    -- true si hay algún alumno y false si no
    Ident: Integer;
    Ex: Examen;
end record;

type Aula is array (1 .. Max_Filas, 1 .. Max_Columnas) of Datos_Mesa;
```

Se pide diseñar e implementar el siguiente subprograma:

```
procedure Escribir_Sospechosos(Aula1: in Aula)
-- precondición: Aula1 contiene los datos de los alumnos que han
--                 realizado un examen
-- postcondición: se han escrito en la pantalla las parejas de alumnos
--                 sospechosos de haber copiado. En la lista no deberá haber
--                 repeticiones de parejas, es decir, si aparece la pareja (X, Y), no
--                 se deberá escribir la pareja (Y, X)
```

Programación I – 31 de enero de 2006

1) Codificación (2,5 puntos)

Escribir un subprograma que dado un vector de enteros obtenga otro vector con los valores originales codificados.

Para codificar un entero, el método consistirá en desplazar la cifra de menor peso a la izquierda. Por ejemplo, el número 1234567 se codificará como 7123456.

Ejemplo: dado el vector T1:

T1	1	2	3	4	5	...	100
1234567	898989	111333					

el resultado será:

T2	1	2	3	4	5	...	100
7123456	989898	311133					

```
procedure Codificar (T1: in Vector_de_Enteros;
                     T2: out Vector_de_Enteros)
-- Pre:
-- Post: T2 contiene los valores del vector T1 codificados.
--        Cada número se codifica desplazando la última cifra a la
--        izquierda
```

2) Baloncesto (3,5 puntos)

Se ha definido una estructura para guardar información correspondiente a un partido de baloncesto. En esa estructura se guardará el nombre de un equipo, los puntos conseguidos por ese equipo y la lista de jugadores que intervinieron, guardándose por cada jugador su nombre y los puntos conseguidos en el partido (el reglamento admite que en un equipo jueguen 10 jugadores como máximo).

```
type T_Jugador is record
  Nombre : String(1..20);
  Puntos: Natural;
  FPersonales: Natural;
end record;

Max_Jugadores: constant Integer := 10;
subtype T_Num_Jugadores is Integer range 0 .. Max_Jugadores;

type T_Tabla_Jugadores is array(1..Max_Jugadores) of T_Jugador;

type Tipo_Equipo is record
  Nombre_Equipo: String(1..20);
  Puntos_equipo: Natural;
  Numero_de_Jugadores: T_Num_Jugadores;
  --indica el nº de jugadores de este equipo
  Jugadores: T_Tabla_Jugadores;
end record;

type Tipo_Partido is array (1 .. 2) of Tipo_Equipo;
```

Dadas las definiciones anteriores, escribir un subprograma que escriba por pantalla la lista de los jugadores que han cometido 1, 2, 3, 4 y 5 faltas personales (5 es el número máximo de faltas personales), ordenados de menor a mayor número de faltas. Por ejemplo:

```
1 falta(s) personal(es):  
    José Pérez  
    Juan Gómez  
2 falta(s) personal(es):  
    Alberto Scola  
    Jon Garrido  
3 falta(s) personal(es):  
    no hay ningún jugador  
4 falta(s) personal(es):  
    Javier Clemente  
    Joseba Alvez  
    Unai Agirre  
5 falta(s) personal(es):  
    Jon Dávalillo  
    Richard Scott
```

```
procedure Escribir_Faltas (Par: in tipo_partido)  
-- Pre:  
-- Post: se ha escrito por pantalla la lista de los jugadores que han  
--        cometido 1, 2, 3, 4 y 5 faltas personales (5 es el número máximo  
--        de faltas personales), ordenados de menor a mayor número de faltas  
--        En caso de que para un determinado número de faltas no haya ningún  
--        jugador se escribiría el mensaje "no hay ningún jugador"
```

Notas:

- ❖ Para resolver el problema solo se podrá recorrer la estructura "Par" una sola vez.
- ❖ En caso de ser necesario, se pueden declarar estructuras de datos adicionales.
- ❖ Se recomienda el uso de subprogramas

3. Elecciones (4 puntos)

Se tienen las siguientes definiciones de tipos de datos para representar los resultados de las elecciones en la comunidad autónoma:

- Constantes establecidas

```
Max_Num_Partidos: constant integer := 10;
Num_Escasos: constant integer := 25;
```

- Rangos

```
subtype t_rango_partidos is integer range 0..Max_Num_Partidos;
subtype T_Nombre is String(1..15);
```

- Lista de partidos con sus votos (lista_partidos_votos)

```
type t_info_Partido_Votos is record
  Nombre : T_Nombre;
  Votos : natural;
end record;
type T_Tabla_Partido is array (1 .. Max_Num_Partidos)
  of t_info_Partido_Votos;
type t_lista_Partidos_Votos is record
  Num_Partidos: T_Rango_Partidos;
  Tabla_Partidos: T_Tabla_Partido;
end record;
```

- Lista de partidos con sus escaños (t_lista_escasos)

```
type T_Info_Partido_Escasos is record
  Nombre : T_Nombre;
  Escasos: Natural;
end record;
type T_Tabla_Escasos is array (1 .. Max_Num_Partidos)
  of T_Info_Partido_Escasos;
type T_Lista_Escasos is record
  Num_Partidos : T_Rango_Partidos;
  Tabla_Escasos : T_Tabla_Escasos;
end record;
```

Se quiere implementar el siguiente programa:

```
procedure Calcular_Escasos (
  LPV_Bizkaia, LPV_Araba, LPV_Gipuzkoa : in t_lista_Partidos_Votos;
  Resultado: out T_Lista_Escasos)
-- Pre: las 3 variables de entrada contienen los resultados de las elecciones en
--      cada una de las 3 provincias
-- Post: el resultado es el número de escaños de cada partido. Este resultado se
--       calcula asignando los 25 escaños a repartir en cada provincia,
--       y acumulándolos finalmente por cada partido
```

Por ejemplo, dados estos resultados electorales:

Bizkaia	Araba	Gipuzkoa			
PNV/EA	264774	PNV/EA	51601	PNV/EA	147498
PSE-PSOE	151347	pp	43765	EHAK	78088
pp	113867	PSE-PSOE	43765	pp	70577
EHAK	65431	EHAK	14180	PSE-PSOE	51163
EB-IU	36258	EB-IU	8395	EB-IU	20278
ARALAR	10187	ARALAR	2541	ARALAR	15273

Después de aplicar la asignación de escaños por el método D'Hont, el número de escaños por provincia será:

Bizkaia	Araba	Gipuzkoa			
PNV/EA	11	PNV/EA	8	PNV/EA	10
PSE-PSOE	6	pp	7	EHAK	5
pp	5	PSE-PSOE	7	pp	5
EHAK	2	EHAK	2	PSE-PSOE	3
EB-IU	1	EB-IU	1	EB-IU	1
				ARALAR	1

El resultado final sería:

Total escaños	
PNV/EA	29
PSE-PSOE	16
pp	17
EHAK	9
EB-IU	3
ARALAR	1

Notas:

- ❖ El resultado no tiene por qué estar ordenado (ver resultado final)
- ❖ Para resolver este ejercicio, se puede usar el siguiente subprograma (no hay que implementarlo):

```
procedure Repartir_Escaños (
    I_P : in T_Lista_Partidos_Votos;
    Escanos : out T_Lista_Escaños ) is
-- post: el resultado es el número de escaños de cada partido,
--        aplicando la ley D'Hont
--        Se han repartido 25 escaños de acuerdo a los votos conseguidos
```

Programación I – 16 de junio de 2006

1. Lista de jugadores (2 puntos)

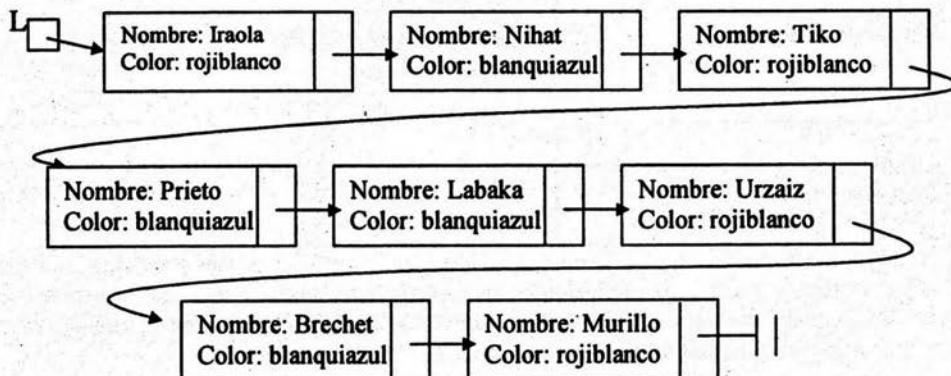
Con la siguiente definición de datos:

```
type Nodo;
type Lista is access Nodo;
type Nodo is record
    Nombre: String(1..20);
    Color: String(1..30);
    Sig : Lista;
end record;
```

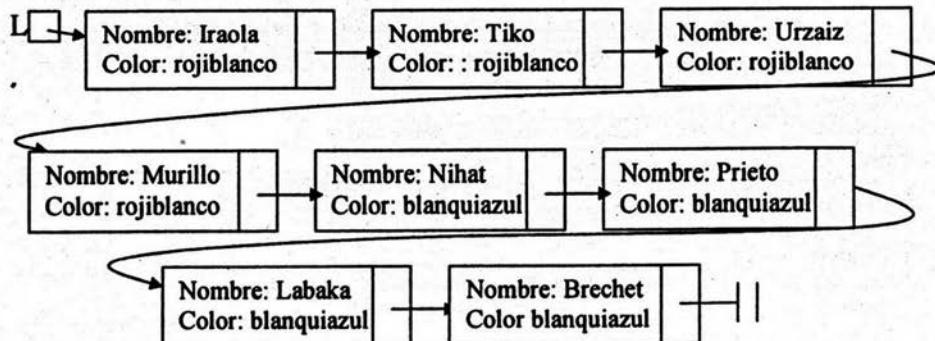
Se pide implementar el siguiente subprograma de **agrupar_en_equipo**:

```
procedure Agrupar_en_equipo (L: in out lista_jugadores ) is
-- Pre: Aparecen en L jugadores blanquiazules y rojiblancos intercalados sin ningún
orden determinado
-- Post: Aparecerán los jugadores agrupados por colores de forma que aparecerán
todos los rojiblancos primero y luego todos los blanquiazules
```

Ejemplo



L debería quedar por ejemplo así:



2. Lista en orden ascendente (2 puntos)

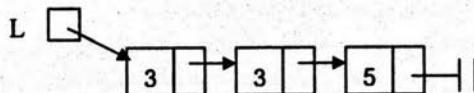
Con la siguiente definición de datos:

```
type Nodo;
type Lista is access Nodo;
type Nodo is record
    Info : Integer;
    Sig : Lista;
end record;
```

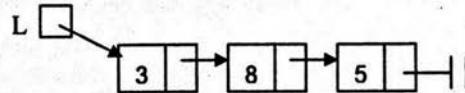
- 1) Implementar el siguiente subprograma `esta_ordenado_en_orden_ascendente` **RECUSIVAMENTE**:
- 2) Escribir la primera llamada (la llamada desde el programa principal)

```
function esta_ordenada_en_orden_ascendente (?????????) return boolean is
-- Pre: L está formada por números positivos
-- Post: el resultado es true si los elementos de L están ordenados
ascendente, y false en cualquier otro caso
```

Ejemplos:



El resultado será *true*



El resultado será *false*

3. Proyectos de fin de carrera (6 puntos)

Se quiere realizar un aplicación para asignar los proyectos de fin de carrera de la Ingeniería Informática. Se tiene una lista de NP¹ proyectos, donde por cada proyecto a considerar se indican: identificador del proyecto, título y director del proyecto. Por otra parte, se tiene una lista de NA¹ alumnos matriculados para el proyecto de fin de carrera. Por cada alumno tenemos:

- Código del alumno, nombre y apellidos, y nota media durante la carrera
- Lista de uno o más proyectos en los que el alumno está interesado, ordenados de mayor a menor prioridad de elección; es decir, el primer proyecto de la lista es el más interesante para el alumno, mientras el último de la lista es el que menos le interesa.

A) (3,5 puntos) Dadas las estructuras de datos de la figura 1, se quiere realizar la asignación de proyectos a alumnos. El método de asignación de proyectos es el siguiente:

- Se busca al alumno con mejor nota media
- A este alumno se le asigna el primer proyecto de su lista que no haya sido ya asignado

Se repite el proceso hasta que no se pueden asignar más proyectos. En el ejemplo mostrado se tiene que hay más proyectos que alumnos. La asignación siguiendo el método descrito será:

- La alumna de código 333, Naiara Abasolo, se lleva el proyecto 666
- El alumno de código 555, Joseba Anguiano, se lleva el proyecto 222
- La alumna de código 222, María Abasolo, se queda sin proyecto
- La alumna de código 444, Ainara Gómez, se lleva el proyecto 111

Realizar la especificación, diseño del algoritmo y la implementación de la solución al problema planteado. En caso de ser necesario el uso de alguna estructura de datos adicional o modificar las estructuras existentes, se deberá incluir su definición. El programa deberá obtener en pantalla las asignaciones de proyectos a alumnos.

Las definiciones ADA de las estructuras de datos aparecen en la Figura 1:

```
NP: constant Integer := ...; -- un valor constante X
NA: constant Integer := ...; -- un valor constante Y

type Datos_Proyecto is record
  Codigo: Integer;
  Titulo, Director: String(1 .. 50);
  Asignado: Boolean; -- inicialmente vale false
  A_Quien: Integer; -- código del alumno
end record;
type Tabla_Proyectos is array (1 .. NP) of Datos_Proyecto;

type Tabla_Proyectos_Elegidos is array(1 .. NP) of Integer; -- cod de proyecto

type Lista_Proyectos_Elegidos is record
  Cuantos: Integer;
  T: Tabla_Proyectos_Elegidos;
end record;

type Datos_Alumno is record
  Codigo: Integer;
  Nombre_y_Apellidos: String(1..50);
  Nota_Media: Float;
  LPE: Lista_Proyectos_Elegidos;
  Examinado: Boolean; -- true si este alumno ha sido tratado y false si no
end record;
type Tabla_Alumnos is array (1 .. NA) of Datos_Alumno;

function Libre (Cod: in Integer; Tp: in Tabla_Proyectos) return Integer is
  -- pre: Cod es un código de proyecto que pertenece a TP
  -- post: devuelve la posición de Cod en TP si Cod no ha sido asignado y 0 si no
```

¹ NP y NA son nombres de constantes. No tienen por qué ser iguales, es decir, puede suceder que haya tantos proyectos como alumnos, o bien más proyectos que alumnos o más alumnos que proyectos.

Lista de proyectos	
Código: 111	
Título: <i>Diseño de un sistema poco inteligente</i>	
Director: José Pérez	
Código: 222	
Título: <i>Diseño de un sistema muy inteligente</i>	
Director: Amaia Pérez	
Código: 666	
Título: <i>Diseño de un sistema demasiado inteligente</i>	
Director: Amaia Pérez	
Código: 444	
Título: <i>Diseño de un sistema poco eficiente</i>	
Director: Pedro Pérez	
Código: 555	
Título: <i>Diseño de un sistema muy eficiente</i>	
Director: Idoia Pérez	

Lista de alumnos	
Código: 444	
Nombre_y_Apellidos: Ainara Gomez	
Nota_Media: 5	
Lista_Proyectos_Elegidos	
Cuantos: 3	
T:	666 222 111
Código: 222	
Nombre_y_Apellidos: Maria Abasolo	
Nota_Media: 7	
Lista_Proyectos_Elegidos	
Cuantos: 2	
T:	222 666
Código: 333	
Nombre_y_Apellidos: Naiara Abasolo	
Nota_Media: 9	
Lista_Proyectos_Elegidos	
Cuantos: 4	
T:	666 111 222 444
Código: 555	
Nombre_y_Apellidos: Joseba Anguiano	
Nota_Media: 8	
Lista_Proyectos_Elegidos	
Cuantos: 2	
T:	666 222

B) (2,5 puntos) Se quiere saber cuáles han sido los proyectos más solicitados por los alumnos. Para ello se tendrá que implementar un subprograma que pase de la estructura de datos inicial, donde cada alumno tiene una lista de proyectos escogidos, a otra en la que por cada proyecto aparezca una lista de los alumnos que lo han escogido. Se pide:

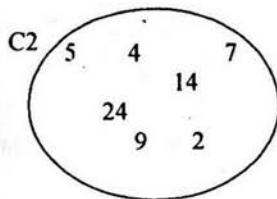
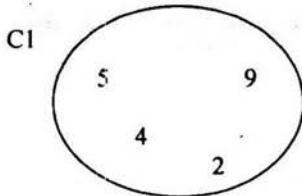
- Definir las estructuras de datos en ADA necesarias para resolver este problema (1 punto)
- Especificación, diseño e implementación del subprograma resultante (1,5 puntos)

Por ejemplo, dada la estructura anterior, el resultado a obtener será:

Lista de proyectos con alumnos	
Proyecto111. <i>Diseño de un sistema poco inteligente.</i>	José Pérez
Alumno333	Alumno444
Proyecto222. <i>Diseño de un sistema muy inteligente.</i>	Amaia Pérez
Alumno333	Alumno555 Alumno222 Alumno444
Proyecto666. <i>Diseño de un sistema demasiado inteligente.</i>	Amaia Pérez
Alumno333	Alumno555 Alumno222 Alumno444
Proyecto444. <i>Diseño de un sistema poco eficiente.</i>	Pedro Pérez
Alumno666	
Proyecto555. <i>Diseño de un sistema muy eficiente.</i>	Idoia Pérez
Sin alumnos	

Nota: en la puntuación se valorará el uso de subprogramas

2. Subconjunto (2 puntos)



Dada la siguiente definición de tipos de datos que representa un conjunto de enteros:

```
N: constant Integer := 1000;  
  
type Conjunto is record  
    Cuantos_Elementos: Integer;  
    T: Vector_de_Enteros(1 .. N);  
end record;
```

Se pide implementar el siguiente subprograma de forma recursiva:

```
function Subconjunto (C1: in Conjunto; C2: in Conjunto) return boolean  
-- Pre:  
-- Post: la función devolverá true si todos los elementos de C1  
-- pertenecen a C2 y false si no
```

En el ejemplo anterior, C1 sí es subconjunto de C2, ya que todos sus elementos (5, 9, 4 y 2) se encuentran contenidos en C2.

Nota: si se considera útil, se pueden usar los siguientes subprogramas auxiliares:

```
function Es_Vacio (C: in Conjunto) return boolean  
-- Pre:  
-- Post: la función devolverá true si C es el conjunto vacío  
-- y false si no  
  
function Pertenece (N: in Integer; C: in Conjunto) return boolean  
-- Pre:  
-- Post: la función devolverá true si N pertenece a C y  
-- false si no  
  
function Primero (C: in Conjunto) return Integer  
-- Pre: C no está vacío  
-- Post: la función devolverá el valor del primer elemento de C  
  
function Resto (N: in Integer; C: in Conjunto) return Conjunto  
-- Pre: N pertenece a C  
-- Post: la función devolverá el conjunto resultante de  
-- quitar N de C
```

2) Mejor trasbordo (7,5 puntos)

Se quiere encontrar la mejor conexión para llegar a un determinado destino en autobús, y se sabe qué autobús se quiere tomar inicialmente (será un dato de entrada junto con la calle en la que se toma dicho autobús). Se sabe que sólo hay que hacer un trasbordo pero hay que encontrar cuál es el mejor, es decir, cuál supone el trayecto con menos paradas.

Como precondiciones sabemos que:

- 1) partimos **siempre** de la primera parada del autobús inicial,
- 2) que **hay siempre al menos un autobús** que nos permitirá hacer trasbordo hasta el destino,
- 3) que solo es necesario enlazar dos autobuses y que los autobuses como **máximo coinciden en una única parada**. Es decir, no habrá que intentar buscar enlaces tal que haga falta enlazar más de dos autobuses, *porque solo se quiere coger dos autobuses*.

Por ejemplo: Se quiere ir de la calle A (en la primera parada del 1ºautobús) , tomando como primer autobús el 13, a la calle K

Autobús 13	Autobús 38	Autobús 48	Autobús 72	Autobús 58	Autobús 85	Autobús 27
Calle A	Calle E	Calle M	Calle K	Calle X	Calle Z	Calle N
Calle B	Calle N	Calle N	Calle E	Calle Y	Calle J	Calle V
Calle X	Calle V	Calle N	Calle V	Calle M	Calle L	Calle N
Calle Z	Calle W	Calle O	Calle Z	Calle J	Calle P	
Calle W	Calle Z	Calle P	Calle J	Calle L	Calle Q	
Calle D		Calle K	Calle L	Calle P		
			Calle S	Calle Q		
			Calle O	Calle K		

Como se puede ver solo los autobuses 38 (parada Calle W), 72 (parada Calle Z), 58 (parada Calle X), 85 (parada Calle Z) coinciden con el 13, de estos solo el 72 y el 58 tienen la parada K entre sus paradas.

El mejor trasbordo correspondería al autobús 72, ya que nos supone un recorrido de 9 paradas (A,B,X,Z,J,L,S,O,K) versus 10 paradas que nos supone el autobús 58 (A,S,X,Y,M,J,L,P,Q,K).

Para ello se dispone de los siguientes tipos de datos:

```
type Parada is string (1..7);

type Tabla_paradas is array (1..10) of Parada;

type Autobus is record
    Numero_id: integer;
    Cuantas_paradas: integer;
    Paradas: Tabla_paradas;
end record;

type Tabla_autobuses is array (1..30) of Autobus;

type L_Autobuses is record
    Cuantos: integer;
    Autobuses: Tabla_autobuses;
end record;
```

Se propone utilizar al menos los siguientes subprogramas

1) (Puntos 3)

Subprograma Buscar_trasbordos_posibles:

Dados los datos **Autos: L_autobuses,**
 Primer_autobús: Autobus;
 Parada_destino: parada

Obtendrá **Trasbordos_posibles: Información_trasbordo**, es decir los trasbordos posibles hasta llegar al destino.

Para ello puede ser interesante codificar y utilizar los siguientes subprogramas

Function autobuses_con_el_destino (Autobuses: in L_autobuses; Parada_destino: in parada)
 returns L_autobuses

Pre: Entrará al menos un autobús que tiene la parada destino entre sus paradas .

Post: Se devolverán los autobuses que tienen entre sus paradas la de destino

Function autobuses_trasbordables (Autobuses: in L_autobuses; Primer_autobus: in Autobus)
 returns L_autobuses

Pre: Entrará al menos un autobús que tenga alguna parada coincidente con el primer autobus

Post: Se devolverán los autobuses que tienen entre sus paradas alguna que enlace con el primer autobus

2) (Puntos 3)

Subprograma Buscar_mejor_trasbordo:

Dado el dato que deberíais haber calculado ya

Trasbordos_posibles: Información_trasbordo, calculará cual de todos supone el mínimo número de paradas.(quizás os hagan falta más datos de entrada, todo dependerá de qué información guardéis en la estructura que tenéis que crear)

Obtendrá el número del autobús que suponga el mejor trasbordo

(Puntos 1.5) Se tendrá que definir el tipo de dato **Información_trasbordo** tal que permita guardar suficiente información de cada trasbordo posible para poder calcular cuál supone menos paradas. Es decir, que contendrá información sobre varios trasbordos, donde por cada trasbordo se guardarán las información que os haga falta para el cálculo posterior.

Información sobre el enlace en la Calle X en el autobús inicial 13	Información sobre el enlace en la Calle X en el autobús de trasbordo 58
Información sobre el enlace en la Calle Z en el autobús inicial 13	Información sobre el enlace en la Calle Z en el autobús de trasbordo 58
.....

Por supuesto vosotros tendréis que saber que información y como la queréis guardar.

Programación I – 18 de junio de 2007

1. Restaurante (1 punto)

El tipo de datos T_Comedor consta de la lista de personas invitadas a un restaurante los siete días de una semana.

```
Type Persona is record
    Nombre_persona: string(1..25);
    ✓Vegetariano: boolean;
End persona;
type Ttabla_Personas is array (integer range<=>) of Persona;

type Tlista_Personas is record
    Personas: Ttabla_Personas(1..50);
    NPersonas: Natural;
end record;
type Tmenu is record
    InfoMenu: string(1..500);
    Menu_vegetariano: boolean;
End Tmenu;
type Torganizacion_Diaria is record
    Invitados: Tlista_Personas;
    El_Menu: Tmenu;
end record;

type TComedor is array(1..7) of Torganizacion_Diaria;
```

Implementa un subprograma **informe_restaurante**, cuya cabecera se facilita a continuación, que partiendo de un valor de tipo T_Comedor escriba en la salida estándar por cada día si existe algún invitado que muestre incompatibilidad con respecto al menú, y si lo(s) existiera escribir el nombre de dichos invitados. (Mostrará incompatibilidad aquel invitado que siendo vegetariano esté invitado a un menú NO vegetariano).

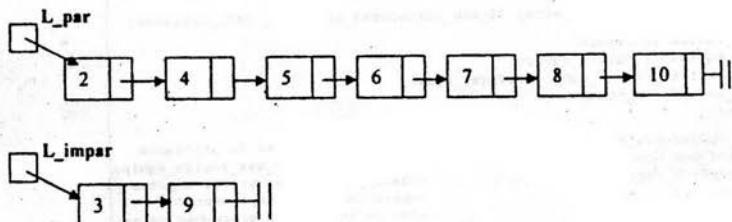
```
Procedure informe_restaurante (El_Comedor: in TComedor);
```

2. Listas pares e impares (1.75 puntos)

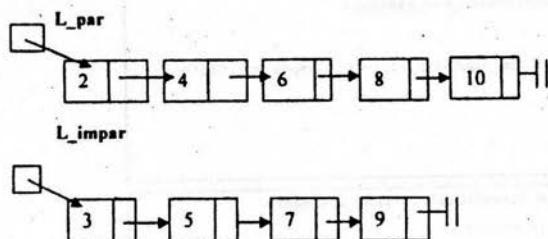
Tenemos dos listas ligadas. La primera está formada por números pares seguidos ordenados en orden ascendente. La segunda es una lista formada por los números impares complementarios de los primeros, es decir, números impares, seguidos y ordenados en orden ascendente. Pero se han colado algunos errores de forma que en la lista de números pares hay algún impar.

Diseñar un subprograma que elimine los impares de la lista de los pares, insertándolos ordenadamente en la lista de los impares. No hay que hacer ninguna comprobación. El que falta en L_impar está seguro traspapelado en L_par.

Antes:



Después:



3. Listas pares e impares _ recursividad (2.25 puntos)

Se busca hacer lo mismo que en el ejercicio anterior pero de forma recursiva.

NOTA: Si se utiliza algún subprograma adicional, éste también debe de estar implementado de forma recursiva. No se utilizarán más listas que las que se proporcionan.

Programación I – 10 de septiembre de 2007

1. Número mayor (2 puntos)

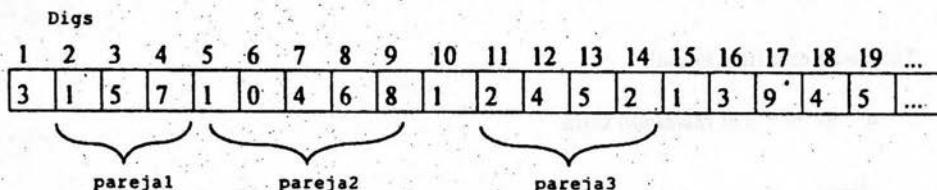
```
subtype Dígito is Integer range 0..9;
type Vector_de_Dígitos is array (1..10000) of Dígito;
type Vector_de_Indices is array (1..100) of Integer;
```

Los elementos de tipo Vector_de_Dígitos contienen, en cada posición, un dígito decimal (valor entre 0 y 9).

Los elementos de tipo Vector_de_Indices contienen, en cada posición, un valor entre 1 y 10000. Cada pareja de valores (a, b), donde $a \leq b$, hace referencia a 2 posiciones de un vector de tipo Vector_de_Dígitos.

Por ejemplo, dados:

```
Digs: Vector_de_Dígitos;
Inds: Vector_de_Indices;
```



Inds

1	2	3	4	5	6	7	8	9	10	11	12	13	14	...
2	4	5	9	11	14	123	132

pareja1 pareja2 pareja3 pareja4

La primera pareja de Inds hace referencia a las posiciones 2 a 4 del vector Digs, es decir, el entero 157. La segunda pareja representa las posiciones entre 5 y 9, es decir, el entero 10468. La tercera pareja representa las posiciones entre 11 y 14, es decir, el entero 2452, ...

Especificar y escribir el siguiente subprograma:

```
function Mayor_Valor(Digs: in Vector_de_Dígitos;
                      Inds: in Vector_de_Indices) return Integer
-- pre:
-- post: el resultado es el valor del número más grande representado en
-- las parejas del vector Inds
```

En el ejemplo, el entero más grande de la parte visible de los vectores es el 10468.

2. Agencia de alquiler (5 puntos)

Una agencia de alquiler de pisos dispone de una lista de pisos sin alquilar y de otra de pisos ocupados (esta última en orden alfabético de los nombres de los inquilinos), cada una con su información correspondiente. De cada piso se conoce, según sea pertinente, la siguiente información: código del piso, nombre del propietario, coste del alquiler, ciudad, calle, número, planta, nombre del inquilino y fechas inicial y final del contrato de alquiler.

```
type Fecha is record
    Dia, Mes, Año : Integer;
end record;

subtype Cadena is String(1..20);

type Info_Piso is record
    Codigo          : Integer;
    Propietario     : Cadena;
    Coste_Alquiler : Integer;
    Ciudad, Calle  : Cadena;
    Numero, Planta : Integer;
    <Inquilino      : Cadena;
    <=Fecha_Inicio_Alq,
    -Fecha_Fin_Alq : Fecha;
end record;

Max_Pisos: constant Integer := 2000;

type Tabla_Pisos is array(1..Max_Pisos) of Info_Piso;

type Lista_de_Pisos is record
    Tabla    : Tabla_Pisos;
    Num_Pisos: Integer;
end record;
```

Se pide:

- (2,5 puntos) Un procedimiento que, dados los datos de un piso recién alquilado (de tipo Info_Piso), lo inserte ordenadamente en la lista de pisos alquilados.
- (2,5 puntos) Un procedimiento que actualice ambas listas en el momento mismo en que se firme un contrato de una vivienda desocupada (dados los datos de código del piso a alquilar, nombre del inquilino y fechas de inicio y fin del contrato).

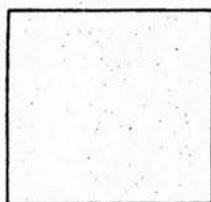
3. Dibujar cuadrados (1,5 puntos)

Sabemos que la ejecución del procedimiento auxiliar:

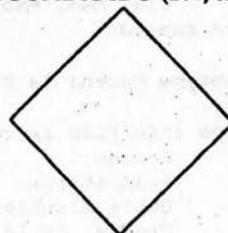
```
procedure DIBCUADRADO(lado: in Float; orientación: in Boolean);
```

produce el dibujo de un cuadrado, centrado en un punto predeterminado, cuya longitud de lado viene representada por su primer parámetro, y orientación true indica que se debe dibujar sobre su base y orientación false sobre su vértice. Véanse los siguientes ejemplos:

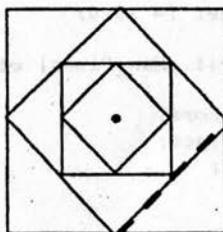
DIBCUADRADO (5.0, true)



DIBCUADRADO (2.4, false)



Escribir un subprograma recursivo de nombre Dibujar que dibuje el mayor número posible de cuadrados concéntricos según el formato:



tomando entre los datos de entrada dos números reales A, B, donde A representa la longitud del lado del mayor cuadrado que eventualmente se dibujará, y B la longitud mínima que el lado podrá tomar.

NOTAS:

- Tener en cuenta que $\text{hipotenusa}^2 = \text{cateto1}^2 + \text{cateto2}^2$
- Para calcular la raíz cuadrada de un valor real, utilizar la siguiente función:

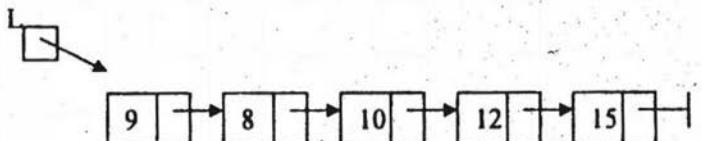
```
function sqrt(X: in Float) return Float
-- pre: X >= 0
-- post: el resultado es el valor de la raíz cuadrada de X
```

4. Mover a partir de posición N (1,5 puntos)

Escribir un subprograma que, dada una lista ligada de enteros L, y un valor entero positivo N (donde $N \leq$ longitud de L), mueva los valores de la lista a partir de la posición N al comienzo de la lista.

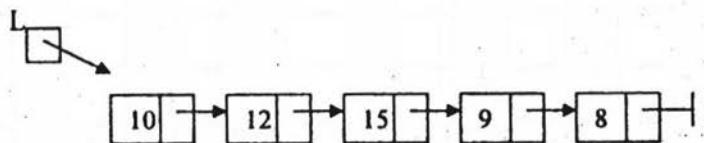
No se podrá crear ningún nuevo nodo, ni tampoco crear otra lista.

Ejemplo:

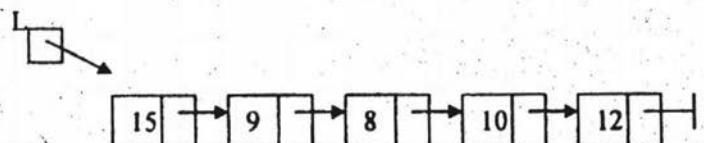


Tomando esta lista inicial:

- Si $N = 3$ el resultado sería



- Si $N = 5$ el resultado sería



Programación I 5 Febrero de 2008

1. Elementos iguales (2 puntos)

Dado un vector formado por 1000 números reales, codificar el subprograma **apariciones** tal que nos devuelva una estructura que guarde cada número real cuantas veces ha aparecido en el vector inicial. (Se valorará la claridad). Para ello definir un nuevo tipo de dato **Lista_reales**. Será de gran ayuda utilizar algún subprograma auxiliar

```
N: constant integer :=1000;
Type vector_de_reales is array (1..N) of float;
Type Lista_reales is ....
```

Ejemplo

1	2	3	4	5	6	7	995	996	997	998	999	1000
2,2	4,0	2,1	0,5	6,1	10,0	222,0	56,4	23,2	6,3	2,2	66,3	7,5

Y debería de devolver como resultado

El 0,5 ha aparecido 20 veces

El 2,1 ha aparecido 56 veces

El 2,2 ha aparecido 80 veces etc

```
Function apariciones(reales: in vector_de_reales) returns Lista_reales is
```

–Pre: reales contiene N números reales que podrán repetirse

–Pos: se devolverá una estructura con información acerca de cuantas veces ha aparecido cada número real.

2. Comparar estructuras (3 puntos)

Comparar si dos estructuras contienen exactamente los mismos elementos y en la misma cantidad aunque estén en distinto orden

```
Type T_numeros is array(1.. N) of integer;
Type Lista_números is record
  Cuantos: integer;
  Numeros: T_numeros;
end record;
```

Ejemplos:

1	2	3	4	5
12	5	3	3	15

1	2	3	4	5
12	3	3	5	15

Se consideraran iguales

1	2	3	4	5
12	5	3	3	15

1	2	3	4	5
12	5	3	5	15

No son iguales

1	2	3	4	5
12	5	3	3	15

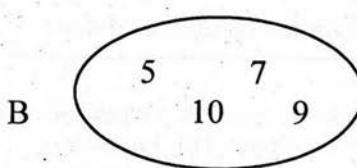
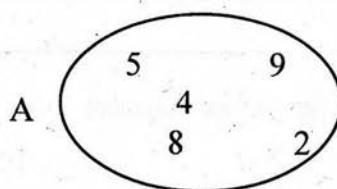
1	2	3	4	5	6
12	5	3	3	15	3

No son iguales

```
Function son_iguales(L1, L2: in Lista_de_numeros) returns boolean is
–Pre: L_numeros contiene varios números enteros que podrán repetirse y que no estarán ordenados
–Pos: se devolverá true si L1 y L2 tienen los mismos elementos y en la misma cantidad.
```

Programación I 03 de Junio de 2008

1. Diferencia (2,5 puntos)



Dada la siguiente definición de tipos que representa un conjunto:

```
type Conjunto is record
    Cuantos_Elementos: integer;
    T: Vector_de_Enteros(1 .. 100);
end record;
```

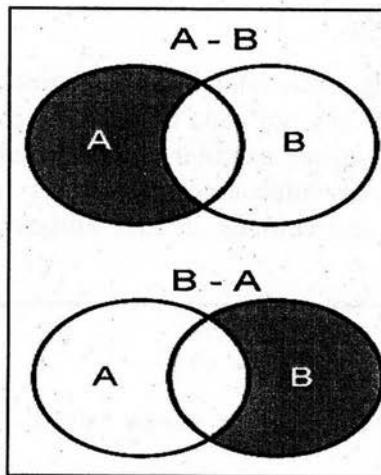
Se pide implementar el siguiente subprograma de **forma recursiva**:

```
subprograma Diferencia (A: ¿? Conjunto;
                        B: ¿? Conjunto) ¿?) ¿?
    -- Post: en A quedarán los elementos únicos a A, es decir, aquellos que
    -- no pertenecen a B
```

En el ejemplo anterior la diferencia sería (4, 2, 8) dado que inicialmente tenía A (4, 2, 8, 5, 9) y de ellos (5, 9) están en B.

NOTA: si se considera útil, se pueden usar los siguientes subprogramas:

```
function Es_Vacio (C: in Conjunto) return boolean
    -- Post: Devuelve true si C es el conjunto vacío y
    -- false si no
function Pertenece (N: in Integer; C: in Conjunto)
    return boolean
    -- Post: Devuelve true si N pertenece a C y false si no
function Primero (C: in Conjunto) return Integer
    -- Pre: C no está vacío
    -- Post: Devuelve el valor de un elemento de C
function Resto (N: in integer; C: in Conjunto)
    return Conjunto
    -- Pre: N pertenece a C
    -- Post: El resultado es el conjunto tras quitar N de C.
```

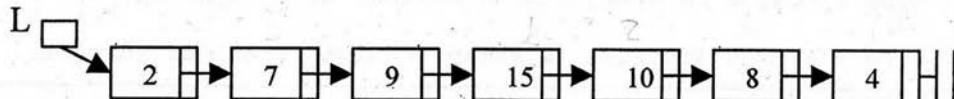


NOTA (de Wikipedia): Sean A y B dos conjuntos cualesquiera. En teoría de conjuntos, se denomina **conjunto diferencia** de A y B , representado por $A - B$, al **conjunto formado por todos los elementos que están en A , pero no están en B** .

2. Orden piramidal (2,5 puntos)

$S = \langle s_1, s_2, \dots, s_n \rangle$ es una secuencia ordenada de mayor a menor de N (N será impar) números, partiendo de esta secuencia almacenada en un array, se pide crear una lista $P = \langle p_1, p_2, \dots, p_n \rangle$ tal que los números en P aparezcan en orden piramidal. El orden piramidal supone que el mayor de los números estará en el centro, y los siguientes que sean justo menores a él aparecerán a sus lados (el mayor a la derecha) y así sucesivamente, hasta agotar S .

Ejemplo, la secuencia $S = \langle 15, 10, 9, 8, 7, 4, 2 \rangle$ en orden piramidal:



```
type secuencia is record
    N: Integer;
    T: Vector_de_Enteros(1 .. 100);
end record;
```

```
procedure orden_piramidal (S: in secuencia; L: in out Lista) is
    -- Pre: Los elementos de  $S$  están ordenados de mayor a menor,  $N \geq 1$ 
    -- Post: Se creará una lista con los elementos de  $S$  en
    -- orden piramidal
```

Programación I

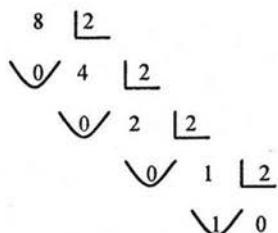
8 de Septiembre del 2008

1. Suma de primos (2 puntos)

Se pide implementar el siguiente subprograma de forma recursiva:

```
subprograma pasar_de_decimal_a_binario(numero: in integer; ¿??????) ¿??  
--pre:  
--post:
```

Ejemplo:

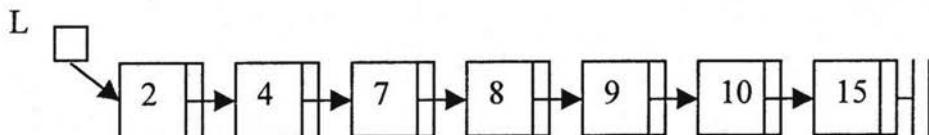
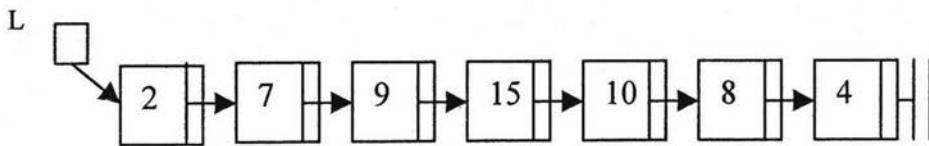


El resultado debería de ser 1000

2. Ordenar ascendente (2 punto)

Codificar un subprograma tal que dada una lista ordene ésta de forma ascendente

```
procedure ordenar_lista (L: in out lista) is  
--pre: L podría contener elementos que no estén ordenados de forma  
ascendente  
--post: Los elementos de L estarán ordenados ascendente
```



Las estructuras de datos a utilizar serán las siguientes:

```
subtype Clave_receptor is Integer range 1 ..3000;
subtype Clave_donante is Integer range 0..10000;
type Sangre is
  (cero_neg,cero_pos,A_neg,A_pos,B_neg,B_pos,AB_neg,AB_pos);
  ---equivale a (0-,0+,A-,A+,B-,B+,AB-,AB+);

type Receptor is record
  Codigo_receptor: Clave_donante;
  Tipo_receptor: Sangre;
  Litros: Integer;
end record;

type T_receptores_sin_asignar is array (1..3000) of Receptor;

type Lista_receptores_sin_asignar is record
  Cuantos: Integer;
  Receptores_sin_asignar: T_receptores_sin_asignar;
end record;

type Donante is record
  Codigo_donante: Clave_donante;
  Tipo_donante: Sangre;
  Litros: Integer;
end record;

type T_donantes_sin_asignar is array (1..10000) of Donante;

type Lista_donantes_sin_asignar is record
  Cuantos: Integer;
  Donantes_sin_asignar: T_donantes_sin_asignar;
end record;

type T_tipos is array (Sangre) of boolean;
type Matriz_compatibilidades is array(Sangre)
of T_tipos;

type Asignacion is record
  Codigo_receptor: Clave_receptor;
  Codigo_donante: Clave_donante;
  Litros: Integer;
end record;

type T_asig_realizada is array (1..3000) of Asignacion;

type Lista_asignaciones_realizadas is record
  Cuantas: Integer;
  Asignaciones_realizadas: T_asig_realizada;
end record;
```

3. Donantes de sangre (6 puntos)

Nos piden hacer un programa que gestione la asignación de donantes de sangre a receptores que necesitan una donación. Como es sabido no todos los tipos de sangre son compatibles entre sí. A continuación se muestra una la tabla de compatibilidad. DATO A RECORDAR: En esta tabla los tipos de sangre de los donantes están ordenados con respecto a la universalidad (el 0- es el tipo más universal y el AB+ el menos universal)

		Donante							
		O-	O+	B-	A-	B+	A+	AB-	AB+
Receptor	AB+	X	X	X	X	X	X	X	X
	AB-	X		X	X			X	
	A+	X	X		X		X		
	A-	X			X				
	B+	X	X	X		X			
	B-	X		X					
	O+	X	X						
	O-	X							

[Tabla obtenida de Wikipedia]

Por un lado tendremos esta matriz, y por otro lado tendremos otras 3 estructuras más, una con información sobre posibles donantes, otra sobre los receptores, y por último una estructura que contendrá la información de las asignaciones que nuestro programa realice.

Por cada **donante** tendremos: código de donante, el grupo (A, B, AB, 0) y tipo (+, -) de sangre, y cuántos litros ha donado.

Por cada **receptor** se tendrá: código, el grupo y tipo de sangre, y cuántos litros necesita.

Y por cada **asignación realizada** se guardará código del receptor, su grupo y tipo de sangre, el código del donante, su grupo y tipo de sangre y cantidad donada al receptor, el código del donante será 0 porque o bien no se ha encontrado donante compatible, o sí se ha encontrado alguno pero ninguna donación es suficiente como para cubrir la necesidad del receptor. SOLO puede haber un donante (con código > 0 o 0 para indicar que no se ha encontrado donante compatible) por cada receptor (UN DONANTE, UN RECEPTOR). Así se pide,

1) (1 pto) Dados los datos de un receptor, dos donantes y la tabla de compatibilidades, codificar un subprograma que diga cuál de los donantes es el menos universal.

```
function donante_preferido(recep1: in Receptor; dona1,dona2: in Donante;
                           compatibilidades: in Matriz_compatibilidades)
    return integer is
--pre: Los donantes no tienen por que ser compatibles con el receptor
--post: Devolverá 0,1,0 2. Si ambos donantes son compatibles devolver 1 si
el primero es el menos universal (si son iguales cualquiera vale), sino
devolver 2. Si ninguno fuese compatible devolver 0.
```

NOTAS: Recordad que la tabla de compatibilidades está ordenada por universalidad (de más a menos universal). Así un receptor del grupo AB+ puede recibir sangre de cualquier grupo y tipo, pero el último en la tabla (AB+) será el que prefiramos como donante porque es el menos universal.

NO HAY QUE CODIFICAR SOLO UTILIZAR

```
procedure buscar_posicion(recep_actual: in Receptor; dona_actual: in Donante;
                           compatibilidades: in Matriz_compatibilidades;
                           x,y: out integer) is
  --pre:
  --post: x será la posición del tipo de sangre del receptor (de 1 a 8) en la
          matriz de compatibilidades e y la posición del donante en la misma matriz.
          x e y valdrán 0 si el donante no es compatible.
```

- 2) (2 ptos) Dadas la variables de tipo *Lista_donantes_sin_asignar*, *receptor* y *Matriz de compatibilidades* y utilizando los subprogramas codificados previamente, codificar el subprograma *mejor_donante*, que nos devuelva la posición en la variable de tipo *Lista_donantes_sin_asignar* del mejor donante, es decir aquel que es compatible, además de compatible es el menos universal y que tenga suficiente sangre para cubrir la necesidad del receptor. En caso contrario devolverá 0.

```
function el_mejor_donante (recep_actual: in Receptor;
                           donantes: in Lista_donantes_sin_asignar;
                           compatibilidades: in Matriz_compatibilidades)
                           return integer is
  --pre:
  --post: Devolverá la posición del mejor donante, es decir uno que sea
          compatible, con sangre suficiente y que sea el menos universal (Si hubiese
          varios iguales, cualquiera). Si no lo encontrásemos entonces devolver 0.
```

- 3) (3 puntos) Dados todos los subprogramas anteriores, codificar el programa *asignar_donantes_a_receptores* tal que dadas las variables de tipo *Lista_receptores_sin_asignar*, *Lista_donantes_sin_asignar* y por último *Matriz_compatibilidades* rellene una variable de tipo *Asignaciones_realizadas*, de forma que por cada receptor se guarde información a cerca del donante que el subprograma *el_mejor_donante* haya encontrado, junto con los litros que necesitaba el receptor, actualizando las variables de tipo *Lista_receptores_sin_asignar* e *Lista_donantes_sin_asignar*. Si no fuese posible encontrar ningún donante, entonces el código de donante en este caso será el 0.

IMPORTANTE: Por cada asignación las estructuras de donantes y la de receptores deberán actualizarse (restando la cantidad donada, pudiendo resultar que haya que eliminar al donante si no le queda sangre. Al receptor habrá que eliminarlo en cualquier caso).

```
procedure asignar_donantes_a_receptores (
  compatibilidades: in Matriz_compatibilidades;
  receptores: in out Lista_receptores_sin_asignar;
  donantes: in out Lista_donantes_sin_asignar;
  asignaciones: out Lista_asignaciones_realizadas)
  --pre:receptores tendrá al menos 1 receptor, y donantes al menos 2 donantes
  --post: Devolverá las asignaciones correspondientes, de forma que cada
          receptor tenga como máximo un donante. Podría suceder que no encontrásemos un
          donante compatible, y con suficiente sangre entonces se le asignará al
          receptor el código del donante 0.
```

1) El arca de Noé (6 puntos)

El viejo Noé está muy preocupado porque se aproximan fuertes tormentas, y quiere subir a los animales de su granja a su arca para zarpar antes de que esto ocurra. El ejercicio consistirá en construir un programa que ayude a Noé a organizar a los animales y las plantas dentro de su arca.

Existen en su granja 5 lugares de almacenaje llenos de animales y plantas, más 2 vacíos para meter las parejas que se formen:

- 1 lugar para las plantas, (plantas de tipo `info_especies`)
- 4 rediles con animales (machos y hembras): 2 para los animales carnívoros (1 redil para machos y otro para hembras) 2 para los herbívoros (1 para machos y otro para hembras). Podrá haber varios machos y varias hembras de la misma especie. Esta información la almacenaremos en las variables `machos_carnívoros`, `hembras_carnívoras`, `machos_herbívoros`, `hembras_herbívoras` de tipo `info_especies`.

Así que el primer trabajo consistirá meter en los rediles vacíos por cada especie **UNA Y SOLO UNA PAREJA** formada por un macho y una hembra de dicha especie. Así cuando insertemos una pareja en el redil de las parejas, deberemos eliminar los animales que forman esa pareja del redil de los animales. Habrá un redil de parejas para los carnívoros y otro para los herbívoros, con objeto de que no se coman entre ellos. Las plantas carecen de distinción por sexo, así es que no habrá tareas de organización ha hacer con ellas. A la hora de generar las parejas se cogerá el primer macho de una especie y la primera hembra de la misma especie (aunque hubiese más).

2 ptos) Se podrá implementar con listas o con arrays (si se implementa con listas daros cuenta de que el siguiente apartado valdrá 1,5 más).

0,5 puntos

```
procedimiento formar_todas_las_parejas (machos_carni, hembras_carni, machos_herbí,
                                         hembras_herbí: in out Info_especies;
                                         redil_parejas_carni, redil_parejas_herbí: out Info_especies) is
  --pre: los machos y hembras no están ordenados con respecto a ningún criterio,
         habrá por lo menos una pareja de herbívoros y otra de carnívoros.
  --post: En redil_parejas_carni, redil_parejas_herbí habrá UNA Y SOLO UNA pareja por
         cada especie. Las repetidas (si las hubiera), se quedarán en los rediles
         iniciales.
```

1,5 ptos)

```
procedimiento formar_parejas (machos, hembras: in out Info_especies;
                               parejas: out Info_especies) is
  --pre: los machos y hembras no están ordenados con respecto a nada. Habrá por lo menos
         una pareja.
  --post: parejas contendrá UNA Y SOLO UNA pareja por cada especie. Los excedentes de cada
         especie (si los hubiera). Se quedarán en las estructuras iniciales
```

Para ello se podrán utilizar el siguiente subprograma (NO ES NECESARIO CODIFICARLO)

```
procedimiento eliminar_del_redil (animales: in out Info_especies,
                                   animal_1: in Info_especie) is
  --pre: -
  --post: Se habrá eliminado el primer animal_1 del redil, es decir se habrá actualizado
         redil.
```

Una vez que el trabajo en la granja ha finalizado nuestro objetivo será llenar el arca pero cuidado, el peso debe estar bien repartido:

1) **Para que el peso esté bien repartido. Se seguirán los siguientes pasos:**

- 1.- Calcular el centro del arca (porque será a partir del centro donde empezemos a colocar las parejas)
- 2.- Colocar la especie más pesada en el centro (cualquiera que sea, carnívoro, herbívoro o planta)
- 3.- Colocar a drcha e izda de la especie central las siguientes especies más pesadas. (a su drcha la más pesada, izda la siguiente más pesada).
- 4.- Siguiendo hacia los extremos a drcha y a izda, la siguientes dos menos pesadas, luego las dos más pesadas y así sucesivamente, hasta que ya no se pueda seguir.

Se pide implementar el procedimiento `llenar_el_arca`. Para ello se podrán utilizar los siguiente subprogramas (NO ES NECESARIO CODIFICARLOS, SÓLO UTILIZARLOS)

```
procedure ordenar_especies (especies: in out Info_especies) is
  --pre: las especies no están ordenadas con respecto a ningún criterio
  --post: las especies están ordenadas en modo decreciente con respecto al peso.
```

2,5 o 4 ptos) Se podrá implementar con listas o con arrays (si se implementa con listas el ejercicio valdrá 4)

```
procedure llenar_el_arca (redil_de_parejas_carni, redil_de_parejas_herb: in out
                           Info_especies; plantas: in out Info_especies;
                           arca: out Info_arca) is
  --pre: Habrá al menos una especie de carnívoros, otra de herbívoros y una de plantas
  El número de huecos del arca es impar, es decir que existe un hueco central. Hay espacio
  para todas las especies.
  --post: Los animales estarán organizados en parejas.
```

León-120kl
Tigre-90kl
León-90kl
Perro-25...
Redil_machos_carnivor.

León-80kl
León-100kl
Perro-15kl
Tigre-60kl ...
Redil_hembras_carnivor.

Vaca-100kl
Canario-0.3kl
Escarabajo-0.01kl
Vaca-150kl.
Redil_machos_herbivor.

Cabra-15kl
Canario-0.2kl
Cabra-10kl
Vaca-150kl...
Redil_hembras_herbivor.

Secuoya
2000kl
Peral 50kl
Ficus 0,5kl
Plantas

Vaca	Cabra	Conejo	Ardilla	Canario	Escarabajo	...
300 kl	40 kl	8 kl	1 kl	0,5 kl	0,02 kl	

León	Tigre	Perro	Gato	...
200	150 kl	41 kl	8 kl	
kl				

Redil_de_parejas_herb

Redil_de_parejas_carni

Secuoya	Peral	Ficus
2000 kl	50 kl	0,5 kl

Plantas

	Ardilla	Peral	Canario	León	Secuoya	Vaca	Escarab	Tigre	Ficus	
	1 kl	50 kl	0,5 kl	200 kl	2000kl	300 kl	0,02 kl	150 kl	0,5 kl	

Arca

Centro

-- opción con arrays

Max_especies: constant Integer := 101;---Cambiar esto para ajustarlo, tiene que ser impar

subtype Rango_max_especies is Integer range 0 .. Max_especies;

TYPE Info_Especie IS RECORD

 familia: String(1..20);

 Especie : String(1..20);

 Peso: float;

END RECORD;

TYPE T_Especies IS ARRAY (Rango_Max_Especies) OF Info_Especie;

TYPE Info_Especies IS RECORD

 Cuantas: Integer;

 Tabla_especies:T_especies;

END RECORD;

type Info_Arca is record

 Ind_comienzo:integer;

 Ind_final:integer;

 Tabla_especies:T_especies;

end record;

Type Info_especie is record

 Familia: string(1..20);

 Especie: string(1..20);

 Peso: float;

end record;

Type nodo_especie;

Type Lista is access nodo_especie;

Type nodo_especie is record

 Info: Info_especie;

 Sig: lista;

end record;

2.- Divisible por 11 (3 puntos)

Dado un número formado por 1000 cifras se pide implementar un subprograma recursivo que permita decidir si dicho número es o no divisible entre 11. Para saber si un número es divisible entre 11, el método que se debe seguir es el de sumar las cifras que se encuentran en las posiciones pares, y las cifras que se encuentran en las posiciones impares, para luego restar ambas sumas. Si el resultado de la resta fuese múltiplo de 11, se puede concluir que el número inicial también lo es. Tomemos como ejemplo el siguiente número 000....0000198230, la suma de las cifras de las posiciones pares es $9+2+0=11$; y la de las posiciones impares es $1+8+3=11$. Beraz, $11-11=0$, y como 0 es divisible entre 11 diremos que el número inicial también lo es. Otro ejemplo 000...0000279818, $12-23=-11$. Este número también sería divisible entre 11.

Dado el gran tamaño del número inicial, éste se almacenará en un array del siguiente tipo:

```
Type T_Num is array (1..1000) of Integer;
```

1. Ganador y los 2 equipos que descienden

Basándonos en los tipos de datos mostrados, dispondremos de variables para almacenar información sobre diferentes una temporada de fútbol (temporada=año).

Solo se almacena información sobre los partidos de primera división. Por cada partido, se almacenará quién es el equipo local, quien el visitante y por cada uno su nombre y cuantos goles a marcado al rival. Al equipo que resulte ganador, se le sumarán 3 puntos. El perdedor no obtendrá ningún punto. Si el partido hubiese resultado en empate, entonces se le asignarán 1 punto a cada uno de los equipos.

Se nos pide encontrar el ganador de la temporada junto con los 2 equipos que bajarán a segunda división (es decir, los que menos puntos tienen, en caso de empate nos da igual quien baja). Se nos pide implementar la **solución más eficiente**.

1. **(1.5 puntos)** Para ello, seguramente deberéis crear una nueva estructura de datos en el paquete datos, que actuará de "cuaderno". También deberéis modificar Info_Temporada, para poder albergar información sobre el ganador y los perdedores.

2. (3 puntos)

with datos; use datos;

```
procedure Asignar_los_puntos_del_partido (.....) is
---entrada: Información sobre un partido y vuestra nueva estructura
---pre: --
---salida: vuestra nueva estructura
---post: La nueva estructura quedará actualizada con la información sobre el partido, asignándole
      3 puntos al ganador o 1 punto a cada equipo en caso de empate
```

3. (5.5 puntos)

with datos; use datos;

```
procedure Encontrar_Ganador_y_los_2_que_descienden (Temporada: in out Info_temporada) is
---entrada: Información sobre la temporada
---pre: --
---salida: Información sobre la temporada
---post: La información sobre la temporada ha quedado actualizada, de forma que cada temporada
      guarda su ganador y los 2 equipos que descienden
```

Programación básica (Versión B)
ULTIMA PRUEBA INDIVIDUAL - 19 Diciembre de 2012

```
package datos is

  MaxTemp: constant integer:=150;---Las temporadas empezaron en 1929;
  MaxJornada: constant integer:=42;--esto es, 2*(MaxEquipos - 1)
  MaxEquipos: constant integer:=22;---temporadas 1995/96 1996/97, participaron 22 en vez de 20
  subtype S-nombre is string(1..50);
  subtype S-partido is integer range 5.. MaxEquipos/2;
  subtype S-jorn is integer range 18.. MaxJornada;
  type Info_equipo is record
    nombre: S-nombre;
    goles: integer;
  end record;
  type Info_Un_partido is record
    local: Info_equipo;
    visitante: Info_equipo;
  end record;
  type T_partidos is array (1..MaxEquipos/2) of Info_Un_partido;
  type Info_partidos is record
    CuantosPart: S-partido;---Por cada jornada se puede calcular el número de partidos
    --- con la siguiente fórmula CuantasJorn/2+1,
    --- podíamos no haber duplicado la información, no añadiendo el campo
    --- CuantosPar dado que la inf. Que almacenará se puede calcular como
    --- hemos comentado, pero por simplificar, lo hemos puesto explícitamente.
    partidos: T_partidos;
  end record;
  type T_jornadas is array (1..MaxJorn) of Info_partidos;
  type Info_temporada is record
    CuantasJorn: S-jorn;
    jornadas: T_jornadas;
  end record;
end datos;
```

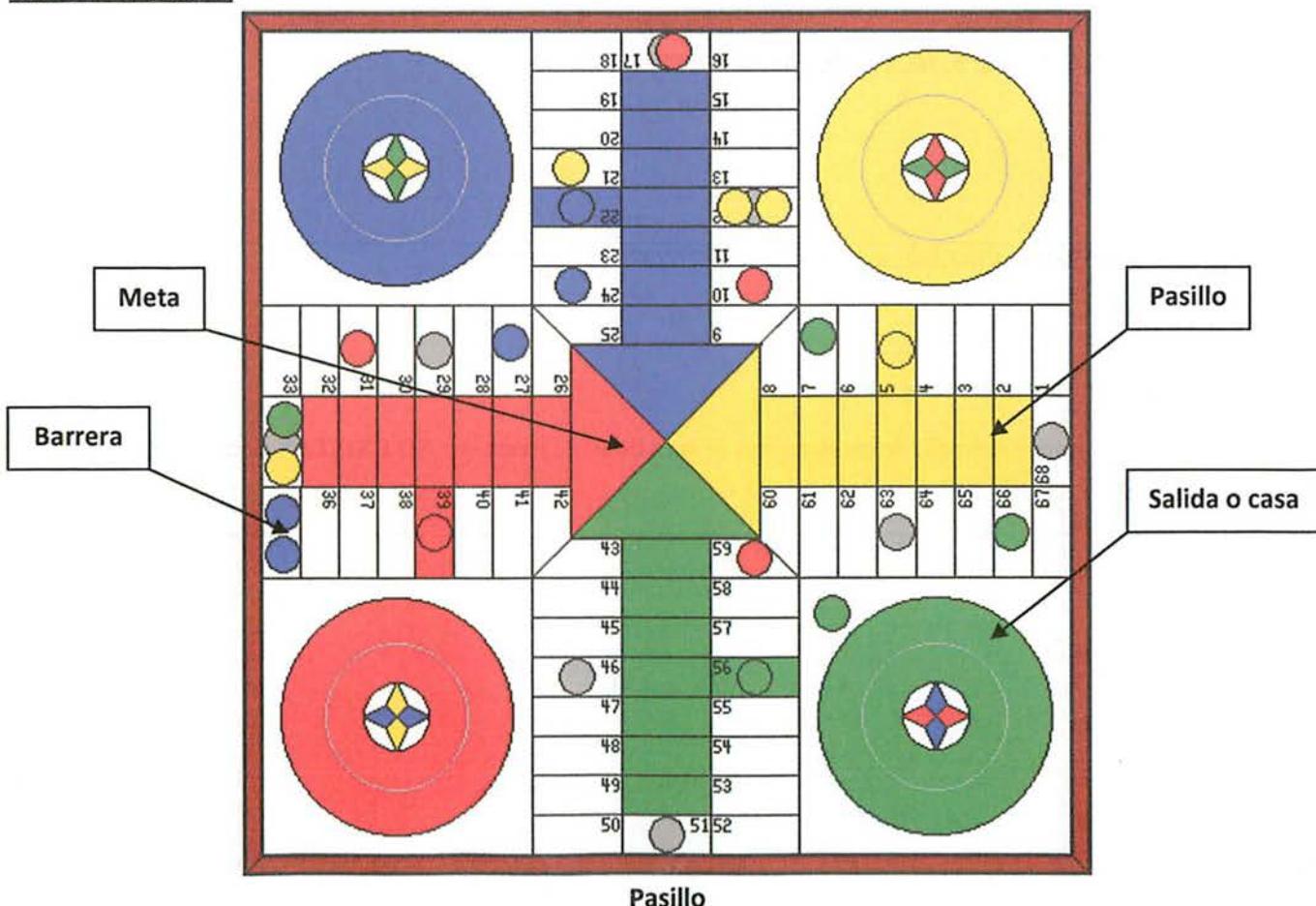
Nombre:

Parchís (7.5 puntos)

Este ejercicio se basa en una versión simplificada del conocido juego del parchís. Además, en este ejercicio solo se pide un subprograma de los que serían necesarios para poder jugar una partida; concretamente, el que permite determinar qué posibilidades de movimiento existen para las fichas de un determinado color (*t_color*), dada la configuración actual del tablero (*t_tablero*, en el que puede haber entre 1 y 16 fichas) y la tirada de un dado (*t_tiradaDado*). Así, se distinguen tres situaciones para las fichas que pueden moverse: (1) las fichas que al moverse comen a otra ficha; (2) las que al moverse aseguran la ficha; y (3) las que pueden moverse pero no cumplen ninguna de las dos situaciones anteriores. Puede ocurrir que, debido a la existencia de barreras, ciertas fichas no puedan moverse.

Definiciones de datos para el ejercicio

```
package parchis is
    subtype t_tiradaDado is integer range 1..6;
    subtype t_rangoCasillas is integer range 1..68;
    type t_color is (Amarillo, Azul, Rojo, Verde);
    type t_fichas is array(1..2) of t_color;
    type t_casilla is record
        cuantos: integer; -- su valor será 0, 1 ó 2 en función del número de fichas que contenga
        fichas: t_fichas;
        segura: boolean; -- en las casillas seguras su valor será true (por ejemplo, 5, 12, 17...)
    end record;
    type t_tablero is array(t_rangoCasillas) of t_casilla;
end parchis;
```

Tablero del parchís

Se considera el tablero como un array de 68 casillas. El tablero es circular, lo que significa que la siguiente casilla a la 68 es la número 1 (aunque no se cumpla para las fichas amarillas, como se verá enseguida). Cada casilla se identifica por la posición que ocupa en el tablero (valor entre 1 y 68), y tiene capacidad para dos fichas. De esta forma, una casilla puede ofrecer diferentes configuraciones a lo largo de una partida:

- Estar libre: no contiene ninguna ficha (*cuantos* vale 0).
- Tener una única ficha (*cuantos* vale 1 y la ficha se encuentra en la posición 1 de la casilla).
- Tener dos fichas iguales formando barrera (*cuantos* vale 2): puede ocurrir en cualquier casilla del tablero.
- Tener dos fichas diferentes que NO forman barrera (*cuantos* vale 2): esta situación solamente tiene sentido en las casillas en las que el booleano *segura* vale true (casillas 5, 12, 17...).

Hay cuatro fichas de cada color, pero según se desarrolla una partida no todas ellas están siempre dentro del tablero: pueden estar en casa, en la meta o en los pasillos. Si una ficha se encuentra en alguna de estas casillas especiales, no se analizarán sus posibles movimientos (para empezar, porque ni siquiera estarán en la estructura de tipo *t_tablero*).

Cuando se tira el dado, se intenta mover una ficha del color correspondiente, lo que significa avanzar con ella tantas casillas como haya indicado la tirada del dado, siempre que sea posible. De este modo, se distinguen cuatro situaciones:

1. La ficha no puede moverse. Esto ocurre cuando (1) en el camino entre el origen y el destino existe alguna barrera, aunque sea del mismo color que la ficha actual, o (2) en el destino no hay sitio para la ella (esto es, hay ya dos fichas en esa casilla).
2. La ficha come a otra. Esto ocurre cuando (1) no existen barreras en el camino; (2) la casilla de destino no es segura; y (3) en el destino hay una única ficha contrincante (es decir, de otro color).
3. La ficha se asegura. Esto ocurre cuando (1) no existen barreras en el camino; (2) se alcanza un destino seguro; y (3) en ese destino seguro hay sitio para una ficha más.
 - o NOTA: Se dice que se alcanza un destino seguro cuando (a) el booleano "segura" de la casilla de destino vale true; (b) se alcanza el pasillo o (c) en la casilla de destino la ficha forma barrera con otra de su mismo color.
 - o NOTA 2: Una ficha alcanza el pasillo cuando sobrepasa...
 - la casilla 17 si es azul.
 - la casilla 34 si es roja.
 - la casilla 51 si es verde.
 - la casilla 68 si es amarilla.
 - o NOTA 3: Las barreras formadas dentro del pasillo no se tendrán en cuenta para este ejercicio.
4. La ficha hace un movimiento normal. Esto ocurre cuando no se da ninguna de las situaciones anteriores.

Se pide:

- 1) Definir la estructura de datos *Info_posibleMovimientos* de manera que sea la más adecuada para almacenar los resultados que se piden. Es posible que haya que definir alguna otra estructura adicional. (1.5 puntos)
- 2) Implementar los siguientes dos subprogramas (0.5 puntos + 5.5 puntos):

IMPORTANTE: Se recomienda utilizar más subprogramas, aunque no se pidan explícitamente, si con ello se consigue mejorar la claridad y la modularidad. Asimismo, se recuerda que las soluciones implementadas deben ser eficientes (por ejemplo, téngase en cuenta que como mucho habrá 4 fichas de ^{Página 62} color en el tablero, aunque no tengan por qué estar las 4).

2a) function entraEnPasillo (color: t_color; posEnTablero: t_rangoCasillas) return boolean

-- Pre:

-- Pos: devuelve *true* cuando en la siguiente casilla a la casilla *posEnTablero*, se entra en el pasillo del *color*.

-- Ejemplo: para el azul cuando *posEnTablero* valga 17.

2b) procedure estudiarMovimientos (tablero: in t_tablero; colorFicha: in t_color; tirada: in t_tiradaDado; salida {in out} Info_posiblesMovimientos)

-- Pre: En *tablero* hay al menos una ficha de color *colorFicha*

-- Post: se ha almacenado la información referente a los posibles movimientos que se pueden realizar con las fichas de color *colorFicha* a partir de la situación de *tablero* y de *tirada*, distinguiendo entre las fichas que comen, se aseguran o realizan un movimiento normal. La posición de las fichas en el tablero es la misma que en la entrada, es decir, este subprograma no realiza ningún movimiento de fichas (únicamente analiza las posibilidades).

-- Ejemplo 1: según la distribución reflejada en la figura anterior, siendo el turno del color **Rojo** y el valor de la tirada de dado **4**, el resultado esperado sería:

- la 17 come.
- la 31 asegura
- la 59 asegura.

(Obsérvese que la ficha de la casilla 10 no se puede mover porque tiene una barrera en el camino)

-- Ejemplo 2: según la distribución reflejada en la misma figura, siendo el turno del color **Verde** y el valor de la tirada de dado **2**, el resultado esperado sería:

- la 7 mueve, pero no come ni asegura.
- la 66 asegura.

(Obsérvese que la ficha de la casilla 34 no se puede mover porque tiene una barrera en el camino, y que la ficha que está en casa no se está teniendo en cuenta en este ejercicio)

-- Ejemplo 3: según la distribución reflejada en la misma figura, siendo el turno del color **Azul** y el valor de la tirada de dado **1**, el resultado esperado sería:

- la 24 mueve, pero no come ni asegura.
- la 27 mueve, pero no come ni asegura.
- la 35 mueve, pero no come ni asegura.

(Obsérvese que en la casilla 35 hay dos fichas azules, pero solamente se contabiliza un movimiento)

Unión de Listas (2.5 puntos)

Se dispone de 2 listas (de tipo ListaEnteros) ordenadas de menor a mayor de números enteros. Se busca crear una tercera lista formada por la *unión ordenada* de los elementos de las dos listas iniciales, sin elementos duplicados. Como la eficiencia es importante, se valorará que cada lista no se recorra más que una única vez.

Se pide:

- 1) definir el tipo de datos ListaEnteros. Como máximo se podrán almacenar 100 enteros.
- 2) implementar el subprograma *uniónDeListas*, que dadas dos listas de enteros ordenadas, obtenga la unión ordenada de ellas SIN DUPLICADOS.

Ejemplo:

Lista1: 2, 5, 7, 8, 10, 17, 28

Lista2: 3, 4, 5, 6, 7, 8, 9, 20, 24, 25, 26, 27, 29, 30

ListaResultado: 2, 3, 4, 5, 6, 7, 8, 9, 10, 17, 20, 24, 25, 26, 27, 28, 29, 30

```
procedure/function uniónDeListas(lis1,lis2: in listaEnteros ??????????????????????????)
```

--pre: las listas estarán ordenadas de menor a mayor

--post: el resultado será la unión ordenada de los elementos de las 2 listas, de forma que **NO EXISTAN** elementos duplicados.