Departamento de Lenguajes y Ciencias de la Computación

UNIVERSIDAD DE MÁLAGA

Apuntes para la asignatura

Informática

Facultad de Ciencias (Matemáticas)

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

Tema 8. Tipos de datos estructurados

Bibliografía

- *Programación 1*. José A. Cerrada y Manuel Collado. Universidad Nacional de Educación a Distancia.
- *Programming with TopSpeed Modula-2*. Barry Cornelius. Addison-Wesley.

Introducción

¿???

8.1 Estructuras de datos

En el tema anterior se estudiaron los *tipos de datos simples*. La principal característica de estos tipos es que una variable de un tipo simple almacena un único valor en un momento concreto. Así, una variable de tipo INTEGER puede valer 5 o 300, pero no puede contener dos valores enteros a la vez.

Ahora vamos a estudiar otros tipos de datos, los *tipos estructurados*. Una variable de tipo estructurado puede almacenar varios datos a la vez.

Una *estructura de datos* es una colección de datos que puede ser caracterizada por su organización y las operaciones que se definen en ella.

Los tipos de datos estructurados más frecuentes en programación son:

- *Tipos estáticos:*
    - ARRAY (vectores y matrices)
    - STRING (cadenas de caracteres)
    - RECORD (registros)
    - SET (conjuntos)
    - FILE (ficheros)
- *Tipos dinámicos:*
    - Listas
    - Pilas
    - Colas
    - Árboles
    - Grafos

Las *estructuras de datos estáticas* son aquellas cuyo tamaño máximo está definido antes de que el programa se ejecute y no puede modificarse durante la ejecución del programa.

Mediante el uso de un tipo de datos específico llamado puntero (POINTER) es posible construir *estructuras de datos dinámicas*. El tamaño de estas estructuras puede variar (crecer o decrecer) durante la ejecución del programa. El tamaño máximo de estas estructuras sólo está acotado por la memoria del ordenador donde se está ejecutando el programa. Estas estructuras ofrecen soluciones eficaces y efectivas en la solución de problemas complejos.

8.2 El tipo ARRAY

Un *array* es un tipo de datos estructurado formado por un número *finito fijo* de elementos del *mismo* tipo. Un array se caracteriza porque se puede *acceder*, de manera *directa*, a cada uno sus elementos utilizando para ello el nombre de la variable seguido de un *índice* (que indica la posición del elemento en el array).

A los arrays *unidimensionales* se les conoce como *vectores*. En Modula-2 un tipo vector se declara de la siguiente forma:

```
TYPE
  TIPOVECTOR =ARRAY TIPOINDICE OF TIPOELEMENTO;
```

donde `TIPOVECTOR` es el nombre del nuevo tipo que se declara, `TIPOINDICE` indica las dimensiones del array y `TIPOELEMENTO` es el tipo de cada elemento dentro del array.

El `TIPOINDICE` suele ser un tipo subrango. El siguiente ejemplo declara el tipo de un vector de cinco valores reales, con posiciones desde la 1 a la 5:

```
CONST
  MAXINDICE   = 5;
TYPE
  RANGO       = [1..MAXINDICE];
  VECTOR5=  ARRAY RANGO OF REAL;
VAR
  Vector : VECTOR5;
```

Podemos representar gráficamente la variable `Vector` como:

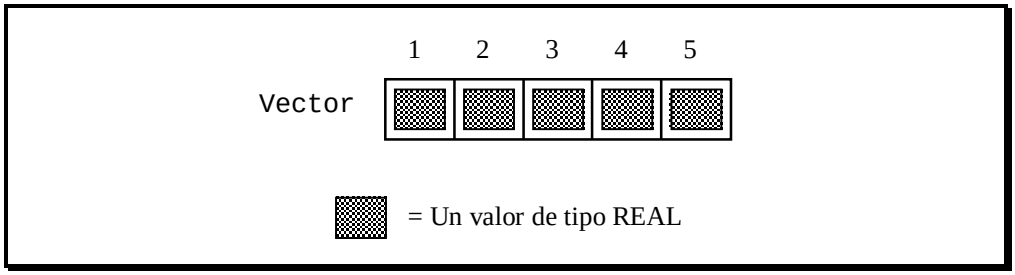
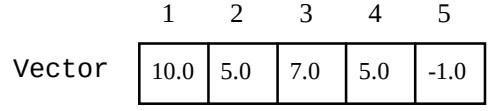


Figura 1. Un ARRAY de cinco reales

`Vector` es una variable donde se pueden almacenar cinco valores reales (en Modula-2 el tamaño de un vector debe ser conocido en tiempo de compilación, por lo que el rango del subíndice debe venir dado por constantes, no por variables). Para modificar esta variable se indica la posición que se quiere modificar escribiendo, tras el nombre de la variable, la posición deseada entre corchetes. Después de ejecutar el programa

```
...
BEGIN
  Vector[1] := 10.0;
  Vector[2] :=  5.0;
  Vector[3] :=  7.0;
  Vector[4] :=  5.0;
  Vector[5] := -1.0;
END ...
```

el contenido de la variable `Vector` es:

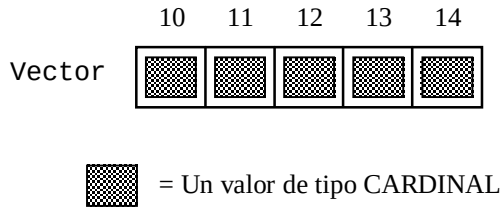


Por supuesto, el índice del array no tiene que comenzar por uno. Las siguientes declaraciones son también válidas:

```
CONST
  MINIMO = 10;
  MAXIMO = 14;
```

```
TYPE
  RANGO  = [MINIMO..MAXIMO];
  VECTOR = ARRAY RANGO OF CARDINAL;
VAR
  Vector : VECTOR;
```

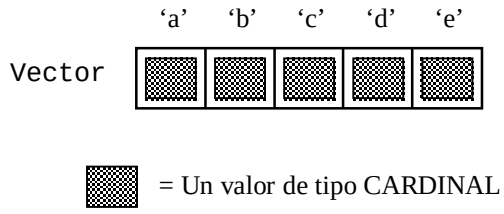
Genera el siguiente vector:



El tipo índice no tiene que ser entero:

```
CONST
  MINIMO = 'a';
  MAXIMO = 'e';
TYPE
  RANGO  = [MINIMO..MAXIMO];
  VECTOR = ARRAY RANGO OF CARDINAL;
VAR
  Vector : VECTOR;
```

dando lugar al vector:



En este último caso se accede a la primera posición del vector escribiendo `Vector['a']`.

El tipo índice puede ser cualquier tipo ordinal, es decir, INTEGER, CARDINAL, CHAR, BOOLEAN, un tipo enumerado o un tipo subrango (los dos primeros casos corresponden a un ARRAY con tipo índice [MIN(INTEGER)..MAX(INTEGER)] o [MIN(CARDINAL) .. MAX(CARDINAL)], que no suele caber en la memoria del ordenador). El tipo base del array puede ser cualquier tipo incluyendo los tipos estructurados.

Se produce un error si se intenta acceder a una posición del array no existente (p.e. `Vector[10]`). Este error es detectado por el compilador si la expresión índice es una constante o se puede producir en tiempo de ejecución si la expresión índice depende de una variable.

Se define el *rango o tamaño de un vector* como el número de sus elementos. En muchos casos el tamaño del vector es un parámetro del programa que podría tener que cambiarse para adaptarlo a distintas circunstancias. En esos casos conviene declararlo como constante antes de la declaración de tipo, quedando el programa parametrizado por dicha constante.

Las operaciones de escritura y lectura de arrays completos no están predefinidas excepto para los arrays de caracteres o STRING. Es el programador el encargado de escribir subprogramas para ello. Sí se puede, sin embargo, leer y escribir posiciones individuales del array. Esto es una característica común a todos los tipos estructurados.

Todas las operaciones disponibles sobre el tipo base pueden ser realizadas con los elementos individuales del array (un elemento del array puede aparecer donde pueda aparecer una variable del tipo base del array). Los valores índices pueden ser cualquier expresión de tipo índice.

Es posible la asignación de arrays completos, si son del mismo tipo:

```
TYPE
  VECTOR1  = ARRAY [0..9] OF REAL;
  VECTOR2= ARRAY [10..19] OF REAL;
  VECTOR3= ARRAY [0..9] OF REAL;
VAR
  V1a, V1b  : VECTOR1;
  V2        : VECTOR2;
  V3        : VECTOR3;
BEGIN
  ...
  V1a  := V1b; (* Correcto       *)
  V2   := V1b; (* Incorrecto !!! *)
  V3   := V1b; (* Incorrecto !!! *)
  ...
```

Modula-2 considera tipos distintos a dos o más tipos array definidos idénticamente, pero con distinto nombre (`VECTOR1` y `VECTOR3` son tipos distintos). En el caso de que los dos arrays tengan el mismo tipo, la asignación completa hace que se copien todos los elementos del destino en el origen:

```
...
V1a  := V1b;
...
```

es equivalente a:

```
...
V1a[0] := V1b[0];
V1a[1] := V1b[1];
V1a[2] := V1b[2];
...
V1a[9] := V1b[9];
...
```

En *TopSpeed* también se pueden comparar arrays del mismo tipo. La expresión `V1a = V1b` será `TRUE` sólo si todos los valores son iguales, posición a posición, en ambos arrays. Sin embargo esto no es válido en Modula-2 estándar, por lo que no debe usarse esta característica si queremos que nuestros programas sean portables. Es mejor solución escribir una función booleana que compare dos arrays.

8.2.1.- Arrays como parámetros

Los arrays completos también se pueden pasar como argumentos a un subprograma, siempre que los tipos del valor formal y real coincidan. El siguiente programa calcula la media de 10 valores reales leídos desde el teclado, usando para ello un array:

```
MODULE Media10;
FROM IO IMPORT RdReal, RdLn, WrStr, WrReal;
CONST
  NUMEROELEMENTOS  = 10;
  PRECISION          = 5;
TYPE
  RANGO     = [1.. NUMEROELEMENTOS];
  VECTOR =  ARRAY RANGO OF REAL;
VAR
  Valores: VECTOR;
```

```
  Media     : REAL;

  PROCEDURE LeerVector (VAR Vector : VECTOR);
  VAR
    i : RANGO;
  BEGIN
    FOR i := MIN(RANGO) TO MAX(RANGO) DO
      WrStr ("Dame un valor: ");
      Vector[i] := RdReal(); RdLn
    END
  END LeerVector;

  PROCEDURE SumaVector (Vector : VECTOR1) : REAL;
  VAR
    i    : RANGO;
    Suma : REAL;
  BEGIN
    Suma := 0.0;
    FOR i := MIN(RANGO) TO MAX(RANGO) DO
      Suma := Suma + Vector[i]
    END;
    RETURN Suma
  END SumaVector;

BEGIN
  LeerVector(Valores);
  Media := SumaVector(Valores) / FLOAT(NUMEROELEMENTOS);
  WrStr ("La media de los números leídos es: ");
  WrReal (Media, PRECISION, 0)
END Media10.
```

Recordemos que el paso por valor de un parámetro implica la creación de una copia del valor original a la hora de realizar la llamada al subprograma. En el caso de paso por referencia se accede directamente al parámetro real dentro del subprograma. Si el parámetro es un array de gran tamaño y utilizamos paso de parámetros por valor, el programa resultante puede resultar más lento o incluso agotar la memoria del ordenador con el espacio ocupado por la copia, por lo que puede ser necesario pasar el array por referencia aunque su valor no vaya a ser modificado. Sin embargo, en los programas escritos de este modo debemos inspeccionar todas las líneas de un subprograma para averiguar si el array parámetro es modificado por el subprograma o no, lo cual hace los programas menos claros, así que nosotros no usaremos esta convención.

8.2.2.- Arrays multidimensionales

El tipo base de un array puede ser cualquiera, en particular puede ser otro array, lo cual nos permite crear arrays de varias dimensiones. Por ejemplo:

```
  CONST
    MAXFILAS  = 5;
    MAXCOLS= 10;
  TYPE
    RANGOFILAS= [1..MAXFILAS];
    RANGOCOLS   = [1..MAXCOLS];
    FILA        =  ARRAY RANGOCOLS OF REAL;
    MATRIZ      = ARRAY RANGOFILAS OF FILA;
  VAR
    Matriz : MATRIZ;
```

Gráficamente, la variable `Matriz` es:

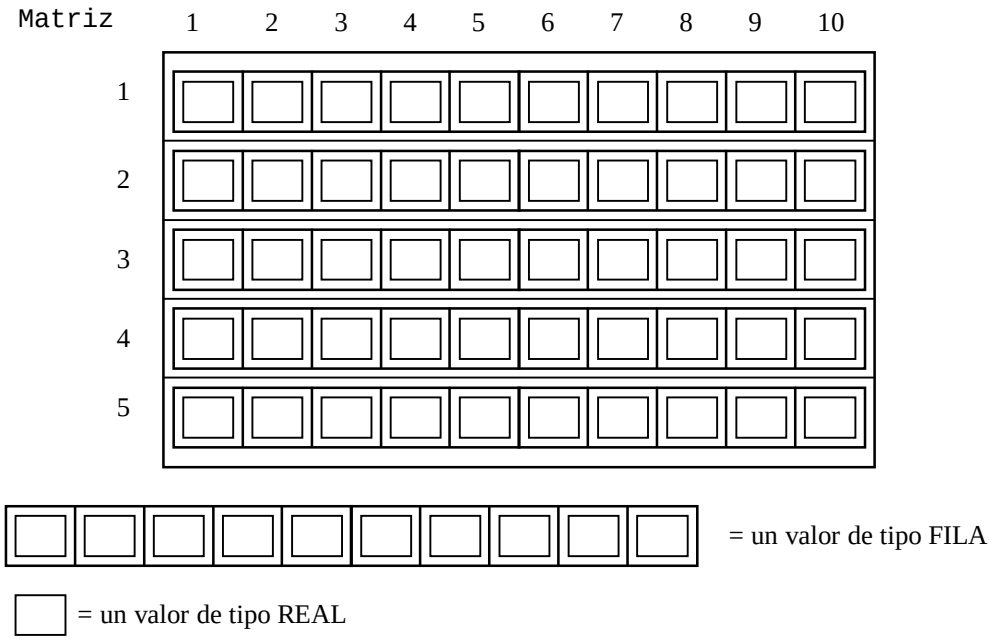


Figura 2. Una matriz 5 x 10

La declaración anterior se puede abreviar de la siguiente forma:

```
TYPE
     MATRIZ = ARRAY RANGOFILAS OF ARRAY RANGOCOLS OF REAL;
```

Otra posibilidad es:

```
TYPE
     MATRIZ = ARRAY RANGOFILAS, RANGOCOLS OF REAL;
```

El acceso a las componentes de una variable `Matriz` de tipo `MATRIZ` se puede escribir `Matriz[i][j]` (con `i` ∈ [1,5], `j` ∈ [1,10]) si la matriz se declaró del primer modo, o bien `Matriz[i,j]` si se declaró del segundo modo. En *TopSpeed* podremos usar cualquiera de los dos métodos para acceder a un componente de la matriz independientemente de como haya sido declarada (esto es una característica no estándar y por ello hace los programas no portables).

El compilador no detecta si el valor del índice de un array está comprendido dentro del rango permitido. Si se quiere que el compilador compruebe este tipo de error hay que usar la directiva de compilación (*#**check(index=>on)***) o bien activar la opción correspondiente dentro del submenú **Runtime checks** del menú **Project**.

La sintaxis BNF para la declaración de un tipo array y el acceso a componentes es:

Tipo_Array	**::=**	ARRAY Lista_Tipos_Ordinales OF Tipo
Lista_Tipos_Ordinales	**::=**	Tipo_Ordinal **{** , Tipo_Ordinal **}**
Tipo_Ordinal	**::=**	Identificador_Tipo_Ordinal **|** Tipo_Enumerado **|** Tipo_Subrango
Identificador_Tipo_Ordinal	**::=**	Identificador
Variable_array	**::=**	Identificador_de_Variable **{** [Lista_de_Expresiones] **}**

8.1.3. Arrays abiertos como parámetros

Una de las operaciones que se pueden realizar con arrays completos es pasarlos como argumento de un procedimiento o función. Sin embargo, en la declaración del subprograma se tiene que indicar un tipo concreto para el parámetro, lo cual fija el tamaño del vector que se puede pasar como argumento, por lo que sería necesario trabajar con procedimientos distintos para arrays de distintos tamaños.

En Modula-2 existe otra forma de declarar un parámetro de tipo ARRAY para un procedimiento sin especificar el rango del array en la cabecera del procedimiento, sino especificando únicamente el tipo base de los elementos del parámetro array, de la forma:

ARRAY OF T

donde T es un nombre de tipo (tipo base). A los parámetros formales declarados de esta forma se les llama *arrays abiertos como parámetros*.

Nota: un array abierto como parámetro solo puede aparecer en la lista de parámetros de la declaración de un subprograma, pero nunca en una declaración de variable. Dentro de un procedimiento que toma como parámetro un array abierto, lo único que podemos hacer con este parámetro es acceder o modificar una posición concreta de él, o pasarlo como parámetro en una llamada a otro subprograma. No podemos asignar el array completo a otro array.

En un parámetro abierto no se especifica el tipo del índice, permitiendo que el programador suministre como parámetro actual cualquier array que tenga como tipo base a T. La ventaja que se obtiene es poder tratar arrays genéricos (de diferentes tamaños y tipos índice) con un mismo procedimiento.

El tipo del índice del parámetro formal dentro del procedimiento es CARDINAL y la primera componente tiene como índice asociado el cero (0). La componente más alta del array varía dependiendo de la longitud del array que el programador suministra como parámetro actual. Si A es un array abierto (SOLO SI ESTO ES CIERTO), el índice de la componente más alta podrá obtenerse usando la función estándar **HIGH**(A). Puesto que la primera componente del array tiene el índice cero, el número total de componentes del mismo será **HIGH**(A) + 1.

Ejemplo:

```
VAR  A : ARRAY [3..5] OF CHAR;

PROCEDURE P (V : ARRAY OF CHAR) ;
   ...
BEGIN
   P (A) ;
   ...
```

Las componentes de los vectores que aparecen en este programa son:

- En el programa principal: A[3], A[4], A[5].
- En el procedimiento: V[0], V[1], V[2]. HIGH(V) = 2

Cuando dos o más parámetros son arrays abiertos los parámetros actuales no tienen que tener la misma longitud.

Veamos un ejemplo de uso de arrays abiertos como parámetros. Se trata de una función que calcula el productor escalar de dos vectores que toma como parámetros. Recordemos que para que el producto escalar esté definido los dos vectores deben tener la misma dimensión, y que el producto escalar es la suma de los productos de los elementos que ocupan la misma posición en los dos vectores:

```
MODULE ProdEsc;
FROM IO IMPORT WrReal, WrStr;
CONST
 DIGITOS   = 3;
 MAXVECTOR = 5;
TYPE
 VECTOR = ARRAY [1..MAXVECTOR] OF REAL;
VAR
 V1, V2 : VECTOR;

 PROCEDURE ProductoEscalar (A, B : ARRAY OF REAL) : REAL;
 VAR i        : CARDINAL;
     Producto : REAL;
 BEGIN
   IF HIGH(A) <> HIGH(B) THEN
     WrStr ("ERROR:el tamaño de los dos vectores debe coincidir");
     HALT
   ELSE
     Producto := 0.0;
     FOR i := 0 TO HIGH(A) DO
       Producto := Producto + A[i]*B[i]
     END
   END;
   RETURN Producto
 END ProductoEscalar;

BEGIN
 V1[1] := 10.0; V1[2] :=  5.0;
 V1[3] :=  7.0; V1[4] :=  9.0;
 V1[5] := 15.0;

 V2[1] :=  8.0; V2[2] := 12.0;
 V2[3] := 20.0; V2[4] := 17.0;
 V2[5] :=  3.0;

 WrStr("El producto escalar es: ");
 WrReal (ProductoEscalar(V1,V2), DIGITOS ,0);
END ProdEsc.
```

La función anterior puede ser usada para vectores de cualquier dimensión con tipo base REAL.

8.1.4. Cadenas de Caracteres (Strings)

Hemos visto como manipular caracteres en el lenguaje Modula-2 mediante el tipo CHAR. Sin embargo, generalmente en un programa se necesita tratar con secuencias de 1 o más caracteres. A estas secuencias se las llama *cadenas de caracteres* o *Strings*. Estas secuencias se suelen almacenar mediante arrays de tipo base CHAR, de modo que una cadena de caracteres no es más que una variable del tipo

ARRAY [0..N] **OF CHAR**

con N mayor o igual a cero. Ejemplos:

```
TYPE
  CADENAPEQUENYA =ARRAY [0..9] OF CHAR;
  CADENAGRANDE = ARRAY [0..99] OF CHAR;
VAR
  Palabra   : CADENAPEQUENYA;
  Frase     : CADENAGRANDE;
```

En el capítulo 2 vimos que Modula-2 permite escribir las cadenas de caracteres como una secuencia de caracteres entre comillas simples ‘ o dobles “, pero la inicial y final deben ser iguales. En cualquier caso el carácter que usemos para delimitar la cadena no puede aparecer dentro de la cadena. Los diferentes modos en que podemos usar una cadena de caracteres en Modula-2 son:

- Como una expresión constante en una declaración de constante:

  ```
  CONST Mensaje = “Hola cómo estas”;
  ```

- Como una expresión en una sentencia de asignación:

  ```
  Frase := “Mi casa”;
  ```

- Como un parámetro real en la llamada a un subprograma:

  ```
  WrStr (“Dame un número: “);
  ```

Si una cadena de caracteres se asigna a una variable:

- La variable debe ser de tipo ARRAY.
- Debe tener como tipo base el tipo CHAR.
- Debe tener suficientes componentes para albergar todos los caracteres en el literal.

En operaciones de este tipo hay que tener en cuenta que cuando la longitud de la constante asignada a la variable de tipo cadena no coincide con el número de componentes de dicha variable:

- Si la constante tiene menos componentes que la variable, la primera posición libre de la variable se rellena automáticamente con el carácter 0C (este es el carácter correspondiente a la posición 0 de la tabla ASCII, esto es CHR(0), y no tiene nada que ver con el dígito cero o ‘0’). Esto indica que parte de la cadena tiene datos válidos y que parte no.
- Si la constante tiene un número de componentes igual al de la variable, se asignan todos los caracteres posibles y no se introduce ningún carácter 0C como terminador.

Hay que tener en cuenta que esta operación sólo está permitida cuando el operando a asignar es un literal cadena constante, por lo que no sería válido:

```
Frase := Palabra;
```

ya que `Frase` y `Palabra` tienen tipos distintos.

Así tenemos que en Modula-2 una cadena de caracteres acaba cuando encontramos un carácter CHR(0) o cuando llegamos al final del array.

En la librería de entrada y salida (`IO`) existen procedimientos para la escritura (`WrStr`) y lectura (`RdStr`) de cadenas de caracteres. `RdStr`, a diferencia del resto de las funciones de entrada, es un procedimiento. Hay que tener en cuenta que:

- Se leen caracteres del teclado a partir del primer carácter de la siguiente línea. Esto implica que todos los caracteres que quedasen en el teclado serán ignorados (como si se

hubiese hecho `RdLn` previamente, aunque si después de `RdStr` realizamos otra operación de lectura, obtendremos estos caracteres previos).

- Se leen caracteres del teclado en la variable string hasta que se cumple una de las siguientes condiciones:
  - Se encuentra los caracteres correspondiente a la pulsación de la tecla ENTER (CHR(13) y CHR(10)). Estos caracteres son leídos, pero no se almacenan en el string. En este caso, el carácter 0C se escribe automáticamente en la primera posición libre.
  - El string se llena. En este caso, no se escribe el carácter 0C y todos los caracteres restantes, incluyendo los correspondientes a la tecla ENTER, son leídos.

Como ejemplo veamos una función que toma como parámetro una cadena de caracteres y devuelve su longitud:

```
MODULE Ejemplo;
IMPORT IO;
TYPE
  CADENA1 = ARRAY [0..30] OF CHAR;
  CADENA2 = ARRAY [1..20] OF CHAR;
VAR
  Cadena1 : CADENA1;
  Cadena2 : CADENA2;

  PROCEDURE Longitud (Cadena : ARRAY OF CHAR) : CARDINAL;
  VAR
    LaLongitud : CARDINAL;
  BEGIN
    LaLongitud := 0; (* Todos los arrays abiertos comienzan por 0 *)

    (* Mientras no se llegue al final del array (HIGH)  *)
    (* y no sea el final del String (0C) ...           *)
    WHILE (LaLongitud <= HIGH(Cadena)) AND
          (Cadena[LaLongitud] <> 0C) DO
      INC(LaLongitud)
    END;
    RETURN LaLongitud
  END Longitud;

BEGIN
  IO.WrStr("Introduzca una cadena: ");
  IO.RdStr(Cadena1);
  IO.WrStr("Su longitud es: ");
  IO.WrCard(Longitud(Cadena1),0);

  IO.WrStr("Introduzca otra cadena ");
  IO.RdStr(Cadena2);
  IO.WrStr("Su longitud es: ");
  IO.WrCard(Longitud(Cadena2),0);

  IO.WrStr("La longitud de 'Hola' es: ");
  IO.WrCard(Longitud("Hola"),0);
END Ejemplo.
```

8.3. El tipo RECORD

Un *registro* es una colección de elementos con tipos posiblemente distintos. A cada uno de los elementos que constituyen el registro se les llama *campo*. Para declarar un tipo registro se debe definir el nombre y el tipo de cada campo componente del mismo.

La descripción del tipo registro en Modula-2 se hace en la forma:

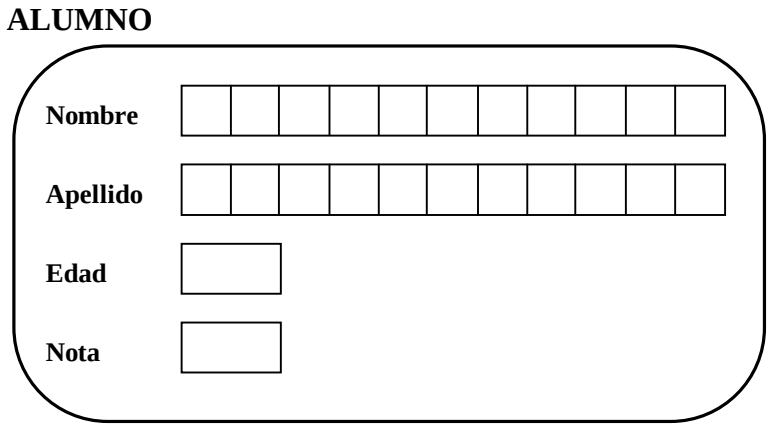
```
TYPE REGISTRO = RECORD
                   Campo1 : TIPO1;
                   Campo2 : TIPO2;
                   ...
                END;
```

donde cada pareja Campo : TIPO define un campo. Los campos de un registro pueden ser de cualquier tipo, incluyendo arrays y registros.

Veamos como usar el tipo registro para definir un tipo `ALUMNO`, de modo que una variable de dicho tipo guarde la información referente a un alumno:

```
CONST
   MAXCADENA = 10;
   MAXALUMNOS= 100;
TYPE
   NOTA = [0..10];
   EDAD   = [0..120];
   CADENA =  ARRAY [0..MAXCADENA] OF CHAR;
   ALUMNO = RECORD
      Nombre    : CADENA;
      Apellido  : CADENA;
      Edad      : EDAD;
      Nota      : NOTA
   END;
   CURSO = ARRAY [1..MAXALUMNOS] OF ALUMNO;
VAR
   ElDelegado         : ALUMNO;
   PrimeroAInformatica: CURSO;
```

Con la declaración anterior, obtenemos la siguiente estructura para la variable `ElDelegado`:



La variable `PrimeroAInformatica` es un array de cien componentes, donde cada componente almacena la información referente a un alumno.

Al igual que en un array podíamos acceder a una componente a través de un índice, podemos acceder a una componente de un registro mediante el nombre del campo. Para ello se escribe el nombre de la variable registro seguida de un punto y el nombre del campo al cual queremos acceder:

```
ElDelegado.Nombre   := “Pedro”;
ElDelegado.Apellido  := “Gómez”;
ElDelegado.Edad    := 19;
ElDelegado.Nota    := 7;

PrimeroAInformatica[10].Nombre := "José Luis" ;
```

Un campo de un registro se puede aparecer en un programa en cualquier punto donde pueda aparecer otro dato del mismo tipo:

```
IF ElDelegado.Nota >= 7 THEN ...
```

El valor de un dato tipo RECORD completo se puede asignar directamente a otra variable de su mismo tipo:

```
PrimeroAInformatica[1] := ElDelegado;
```

es equivalente a:

```
PrimeroAInformatica[1].Nombre   := ElDelgado.Nombre;
PrimeroAInformatica[1].Apellido:= ElDelgado.Apellido;
PrimeroAInformatica[1].Edad    := ElDelgado.Edad;
PrimeroAInformatica[1].Nota    := ElDelgado.Nota;
```

También es posible pasar como argumento un registro completo a una función o procedimiento.

Nota: En *TopSpeed* podemos devolver un registro en una función y comparar dos registros del mismo tipo con los operadores de igualdad y desigualdad, aunque esta característica es una extensión no estándar del lenguaje Modula-2.

Cuando en una parte de un programa se referencia más de un campo de un mismo registro, podemos usar la sentencia WITH para acceder a los campos sin tener que escribir el nombre del registro cada vez:

```
ElDelegado.Nombre   := “Pedro”;
ElDelegado.Apellido  := “Gómez”;
ElDelegado.Edad    := 19;
ElDelegado.Nota    := 7;
```

es equivalente a

```
WITH ElDelegado DO
  Nombre := "Pedro";
  Apellido  := "Gómez";
  Edad      := 19;
  Nota      := 7
END;
```

Los nombres de los campos de un registro son locales a él, por lo que no hay conflicto con otros nombres en el programa. Esto significa que dos tipos registros distintos pueden tener un campo con el mismo nombre o que una variable del programa puede tener el mismo nombre que un campo de un registro.

Otros ejemplos de tipos registros son:

```
TYPE
  DIA    = [1..31];
  MES    = [1..12];
  ANYO   = [0..2000];
  FECHA  = RECORD
             Dia : DIA;
             Mes  : MES;
             Anyo : ANYO
           END;

  PALO   = (Oros, Copas, Espadas, Bastos);
  VALOR  = (As, Dos, Tres, Cuatro, Cinco, Seis, Siete,
            Sota, Caballo, Rey);
  CARTA  = RECORD
             Palo   : PALO;
             Valor  : VALOR
           END;
```

```
  HORA      = [0..24];
  MINUTO    = [0..59];
  SEGUNDO=  MINUTO;
  TIEMPO    = RECORD
                Hora      : HORA;
                Minuto    : MINUTO;
                Segundo: SEGUNDO
              END;
VAR
  UnaBaraja        : ARRAY [1..40] OF CARTA;
  LaHora           : TIEMPO;
  FechaNacimiento: FECHA;
```

8.3.1. Registros Variantes

Hay situaciones en la que un registro puede tener unas u otras componentes dependiendo de los valores de otras componentes del registro. Este tipo de registros se denomina *registros variantes*.

Veamos un ejemplo en el que queremos almacenar información de una persona que incluya:

- el nombre,
- fecha de nacimiento
- el estado civil
  - dependiendo del estado civil:
    - si es casado:
      - el nombre del cónyuge
      - fecha de nacimiento del cónyuge
      - el número de hijos
    - si es soltero
      - el sueldo
    - si es viudo
      - la fecha en que enviudó

Lo ideal en este caso es definir un registro en el que, dependiendo del campo estado civil, se almacenen los campos correspondientes. La declaración de este registro sería:

```
CONST
  MAXCADENA   = 20;
TYPE
  ESTADOCIVIL = (Casado, Soltero, Viudo);
  CADENA      = ARRAY [0..MAXCADENA] OF CHAR;
  EDAD        = [0..120];

  DIA    = [1..31];
  MES    = [1..12];
  ANYO   = [0..2000];
  FECHA  = RECORD
             Dia  : DIA;
             Mes  : MES;
             Anyo : ANYO
           END;

  PERSONA = RECORD
```

```
    Nombre            : CADENA;
    FechaNacimiento: FECHA;
    CASE EstadoCivil : ESTADOCIVIL OF
      Casado:
        NombreConyuge          : CADENA;
        FechaNacimientoConyuge: EDAD;
        NumeroHijos            : CARDINAL  |
      Soltero:
        Sueldo              : LONGCARD  |
      Viudo:
        FechaEnviudamiento  : FECHA
    END
  END;
```

Si el campo `EstadoCivil` de una variable de tipo `PERSONA` vale `Casado`, para dicha variable es legal acceder a los campos:

- `Nombre`
- `FechaNacimiento`
- `EstadoCivil`
- `NombreConyuge`
- `FechaNacimientoConyuge`
- `NumeroHijos`

Sin embargo, si el campo `EstadoCivil` vale `Soltero` los campos válidos de la variable serán:

- `Nombre`
- `FechaNacimiento`
- `EstadoCivil`
- `Sueldo`

Esto nos permite definir variables que aún siendo del mismo tipo difieren en el número, nombre y tipo de sus componentes.

Nota: dado un registro variante es ilegal acceder a campos de la parte variante no activa. Sin embargo, la mayoría de las implementaciones del lenguaje Modula-2, incluyendo *TopSpeed*, no comprueban este tipo de situaciones, por lo que hay que tener mucha precaución a la hora de trabajar con registros variantes.

El formato de descripción de un registro con variantes es:

```
RECORD
  CamposFijos : TIPOS;
  CASE Discriminante : TIPO OF
    Valor1 : Variante1 |
    Valor2 : Variante2 |
    ....
  ELSE
    Variante
  END
END;
```

Cada variante es una colección de campos, aplicables en cada caso concreto. El campo discriminante debe ser de tipo ordinal. Hay que tener en cuenta las siguientes observaciones:

- Un nombre de campo usado dentro de una parte variante no debe repetirse dentro del mismo registro.
- En un registro puede haber más de una parte variante.

- Una parte variante puede contener a su vez partes variantes.
- Una parte variante puede estar vacía.
- Un registro puede contener únicamente partes variantes.
- Puede haber una parte ELSE, que será la última parte variante.

8.3.2 Sintaxis BNF para registros

Las reglas BNF para la definición de tipos registros son las siguientes:

```
Tipo_Registro                ::= RECORD Secuencia_de_Listas_de_Campos END
Secuencia_de_Listas_de_Campo ::=  Lista_de_Campos { ; Lista_de_Campos }
Lista_de_Campos              ::= Parte_Fija | Parte_Variante
Parte_Fija                   ::= Lista_de_Identificadores : Tipo
Parte_Variante               ::= CASE Identificador  :  Identificador_de_Tipo_Ordinal OF
                                     Variante { |  Variante }
                                     [ ELSE Secuencia_de _Listas_de_Campos ]
                                 END
Variante                     ::= Lista_de_Valores :[ Secuencia_de_Listas_de_Campos ]
Lista_de_Valores             ::=  Valores { , Valores }
Valores                      ::=  Expresión_Constante [ .. Expresión_Constante ]
Identificador_de_Tipo_Ordinal ::= Identificador
```

Obsérvese que el tipo de un campo que establece una parte variante debe ser un identificador de tipo previamente definido (no se puede introducir aquí un tipo anónimo).

En cuanto a la referencias a variables o componentes de registros:

```
Variable  ::= ( Identificador_de_Variable | Identificador_de_Campo )
              { .  Identificador_de_Campo | [ Lista_de_Expresiones ] }
```

Por último la sintaxis de la sentencia WITH es:

```
Sentencia_With  ::= WITH Variable DO Secuencia_de_Sentencias END
```

8.4. El tipo SET

En Modula-2, un *conjunto* es una colección no ordenada de valores distintos pertenecientes a un mismo tipo de datos que se llama tipo base. El tipo base debe ser simple y ordinal.

La definición de un tipo conjunto en Modula-2 tiene la siguiente forma:

```
TipoConjunto =  SET OF TipoBase
```

donde TipoBase debe ser un tipo ordinal.

Nota: Cada implementación del lenguaje suele establecer limitaciones en el tipo base. En el caso de *TopSpeed* el tipo base puede tener como máximo 65536 valores distintos. Debido a esto el tipo base no puede ser INTEGER. *TopSpeed* además no permite tipos base que sean subrango del tipo INTEGER, aunque sí subrangos del tipo CARDINAL.

Por ejemplo, podemos definir un tipo que incluya días de la semana:

```
TYPE
  DIASSEMANA     = (Lunes, Martes, Miercoles, Jueves, Viernes,
                    Sabado, Domingo);
  CONJUNTODIAS= SET OF DIASSEMANA;
VAR
  DiasLaborables, DiasNoLaborables : CONJUNTODIAS;
```

De modo que la variable `DiasLaborables` puede almacenar cero, uno o más valores del tipo `DIASSEMANA`. Para construir un valor de tipo conjunto se especifican sus elementos entre llaves precedidos del nombre del tipo:

```
BEGIN
  DiasLaborables   := CONJUNTODIAS {};
  DiasLaborables   := CONJUNTODIAS {Lunes, Martes, Miercoles,
                                    Jueves, Viernes};
  DiasNoLaborables := CONJUNTODIAS {Domingo};
  ...
```

La primera sentencia asigna a `DiasLaborables` el conjunto vacío. Podemos también asignar un rango de valores continuos del tipo base especificando dos constantes separadas por dos puntos:

```
DiasLaborables := CONJUNTODIAS {Lunes..Viernes};
DiasLaborables := CONJUNTODIAS {MIN(DIASSEMANA)..MAX(DIASSEMANA)};
```

o una combinación de ambos métodos:

```
  DiasLaborables := CONJUNTODIAS {Lunes..Miercoles, Viernes};
```

Las reglas BNF para la declaración de conjuntos y la construcción de estos son:

Tipo_Conjunto	**::=**	SET OF TipoOrdinal
Tipo_Ordinal	**::=**	Tipo
Construcción_de_Conjunto	**::=**	Identificador_de_Tipo { Lista_de_Elementos }
Lista_de_Elementos	**::=**	**[** Elementos **{** , Elementos **}** **]**
Elementos	**::=**	Expresión_Constante **[** .. Expresión_Constante **]**

8.4.1. Operadores sobre Conjuntos

Los siguientes operadores, que también se emplean en la aritmética de números, están sobrecargados y tienen los siguientes significados sobre conjuntos:

- `+` Unión de conjuntos
- `-` Diferencia de conjuntos
- `*` Intersección de conjuntos
- `/` Diferencia Simétrica de conjuntos (Unión - Intersección)

Veamos algunos ejemplos de usos de estos operadores. Dados los conjuntos:

```
Trabajo1 := CONJUNTODIAS{Lunes, Martes, Miercoles};
Trabajo2 := CONJUNTODIAS{Jueves, Lunes, Viernes};
```

Los valores de las siguientes expresiones son:

```
Trabajo1 + Trabajo2 -> {Lunes, Martes, Miercoles, Jueves, Viernes}
Trabajo1 - Trabajo2-> {Martes, Miercoles}
Trabajo1 * Trabajo2-> {Lunes}
Trabajo1 / Trabajo2-> {Martes, Miercoles, Jueves, Viernes}
```

También existen operadores relacionales definidos sobre conjuntos:

- = igualdad de conjuntos
- <> , # desigualdad de conjuntos
- <= subconjunto (A<=B: TRUE si A subconjunto de B)
- => subconjunto (A=>B: TRUE si B subconjunto de A)
- **IN** pertenencia de un elemento a un conjunto

Dados los conjuntos:

```
Trabajo1 := CONJUNTODIAS {Lunes, Martes, Miercoles};
Trabajo2 := CONJUNTODIAS {Jueves, Lunes, Viernes};
Trabajo3 := CONJUNTODIAS {Miercoles, Lunes} ;
```

Los valores de las siguientes expresiones son:

```
Trabajo1 = Trabajo2  --> FALSE
Trabajo2 # Trabajo3 --> TRUE
Trabajo3 <= Trabajo1  --> TRUE
Lunes IN Trabajo1    --> TRUE
Sabado IN trabajo3     --> FALSE
```

Modula-2 proporciona además dos procedimientos predefinidos que permiten modificar los valores de un conjunto.

- **INCL**(S,X) incluye el elemento X en el conjunto S.
- **EXCL**(S,X) excluye el elemento X en el conjunto S.

8.5 Algoritmos de búsqueda

Un problema que suele aparecer con cierta frecuencia en la programación de ordenadores es el de la *búsqueda*: dado un vector que almacena una serie de datos y un dato, determinar si este último aparece dentro del vector. El problema ha sido muy estudiado. Pasemos a ver algunas de las soluciones obtenidas.

8.5.1 Búsqueda lineal

En el caso más general, el algoritmo a aplicar consiste en ir comprobando, para cada posición del array, si el elemento que almacena coincide con el elemento buscado. Una implementación de este algoritmo, que busca de izquierda a derecha, es:

```
MODULE Lineal;
FROM IO IMPORT RdChar, RdLn, WrStr, WrCard;
CONST
  MAXVECTOR = 10;
TYPE
  INDICE    = [1..MAXVECTOR];
  ELEMENTO  = CHAR;
  VECTOR    = ARRAY INDICE OF ELEMENTO;
VAR
  UnVector : VECTOR;
  UnaLetra : ELEMENTO;
  Posicion : INDICE;

PROCEDURE BuscarLineal (Vector : VECTOR; Elemento : ELEMENTO;
                        VAR Posicion : INDICE) : BOOLEAN;
  TYPE
```

```
    INDICEBUSQUEDA = [1..MAXVECTOR+1];
  VAR
    i          : INDICEBUSQUEDA;
    Encontrado : BOOLEAN;
  BEGIN
     i := 1;
     Encontrado := FALSE;
     WHILE (NOT Encontrado) AND (i <= MAXVECTOR) DO
        IF Vector[i] = Elemento THEN
           Encontrado := TRUE
        ELSE
           INC (i)
        END
     END;
     IF Encontrado THEN
      Posicion := i
     END;
     RETURN Encontrado
  END BuscarLineal;

BEGIN
  UnVector[ 1] :=  'A';  UnVector[ 2] :=  'X';
  UnVector[ 3] :=  'D';  UnVector[ 4] :=  'E';
  UnVector[ 5] :=  'F';  UnVector[ 6] :=  'B';
  UnVector[ 7] :=  'A';  UnVector[ 8] :=  'V';
  UnVector[ 9] :=  'H';  UnVector[10] :=  'G';

  WrStr ("Dame una letra: ");
  UnaLetra := RdChar(); RdLn;

  IF BuscarLineal(UnVector, UnaLetra, Posicion) THEN
    WrStr ("El elemento fue encontrado en la posición ");
    WrCard (Posicion, 0)
  ELSE
    WrStr ("El elemento no fue encontrado")
  END
END Lineal.
```

La función `BuscarLineal` busca el elemento `Elemento` en el vector `Vector`. En el caso de que lo encuentre, devuelve `TRUE` y la variable `Posicion` contendrá la posición de la primera aparición del elemento buscado. Si el elemento no se encuentra en el vector, la función devuelve `FALSE` y el valor de `Posicion` será información no interesante. Obsérvese que la variable `i` debe ser declarada con tipo `INDICEBUSQUEDA`, ya que si el elemento buscado no se halla en el vector alcanzará el valor `MAXVECTOR+1` en la última iteración.

Se suele considerar que un algoritmo de búsqueda es tanto mejor cuanto menos operaciones de comparación realiza. En el algoritmo anterior se realizan comparaciones de datos enteros (`i<=MAXVECTOR`), de datos booleanos (`NOT Encontrado` es equivalente a `Encontrado=FALSE`) y de datos cuyo tipo coincide con el tipo base del vector (`Vector[i]=Elemento`). Las operaciones de comparación de datos booleanos suelen ser realizadas de un modo muy eficiente por los ordenadores actuales por lo que las despreciaremos y solo contaremos los otros dos tipos de comparaciones.

Vamos a contar las comparaciones de datos no booleanos que realiza el algoritmo original en el mejor y en el peor caso:

- *mejor caso*: lo mejor que nos puede ocurrir, si utilizamos el algoritmo de búsqueda lineal, es que el elemento buscado sí este en el vector, y que además esté en la posición primera. En este caso el algoritmo realiza una comparación de enteros y una de datos con tipo base en la única iteración del bucle que realiza. Recordemos que la expresión condicional del bucle **`WHILE`** se evalúa en cortocircuito por lo que en la segunda

iteración, dado que **NOT** Encontrado es falso, la comparación de enteros no llega a realizarse.

- *peor caso*: lo peor que le puede ocurrir al algoritmo anterior es que el elemento buscado no se encuentre en el vector, ya que en este caso ha de recorrer todo el vector para asegurarse de ello. En este caso el bucle **WHILE** se ejecuta MAXVECTOR veces (para valores de i: 1, 2, ..., MAXVECTOR). En cada una de estas iteraciones se realizan 2 comparaciones. Cuando i toma el valor MAXVECTOR+1 se realiza otra comparación adicional (i<=MAXVECTOR). Sumando obtenemos (2*MAXVECTOR+1) comparaciones.

8.5.2 Búsqueda lineal con centinela

Podemos mejorar sensiblemente el algoritmo anterior si utilizamos un vector con una posición más y antes de empezar a buscar un elemento lo colocamos en la última posición del vector. En este caso podemos asegurar que el elemento será encontrado en el vector (por lo menos estará en la última posición ya que lo hemos puesto allí), lo cual nos permite ahorrar una comparación de enteros (i<=MAXVECTOR) en cada iteración del algoritmo. Si después de salir del bucle el valor de i estrictamente mayor que MAXVECTOR podremos concluir que el elemento buscado no se encontraba en el vector original.

```
MODULE Centinela;
FROM IO IMPORT RdChar, RdLn, WrStr, WrCard;
CONST
  MAXVECTOR = 10;
TYPE
  INDICE    = [1..MAXVECTOR+1];
  ELEMENTO  = CHAR;
  VECTOR    = ARRAY INDICE OF ELEMENTO;
VAR
  UnVector : VECTOR;
  UnaLetra : ELEMENTO;
  Posicion : INDICE;

PROCEDURE BuscarCentinela (Vector : VECTOR; Elemento : ELEMENTO;
                           VAR Posicion : INDICE) : BOOLEAN;
  TYPE
    INDICEBUSQUEDA = [1..MAXVECTOR+1];
  VAR
    i          : INDICEBUSQUEDA;
    Encontrado : BOOLEAN;
  BEGIN
     Vector[MAXVECTOR+1] := Elemento;
     i := 1;
     Encontrado := FALSE;
     WHILE NOT Encontrado DO
        IF Vector[i] = ElementoTHEN
           Encontrado :=TRUE
        ELSE
           INC (i)
        END
     END;
     IF (i > MAXVECTOR) THEN
        Encontrado :=FALSE
     ELSE
        Posicion := i
     END;
     RETURN Encontrado
END BuscarCentinela;

BEGIN
```

```
  UnVector[ 1] :=  'A';  UnVector[ 2] :=  'X';
  UnVector[ 3] :=  'D';  UnVector[ 4] :=  'E';
  UnVector[ 5] :=  'F';  UnVector[ 6] :=  'B';
  UnVector[ 7] :=  'A';  UnVector[ 8] :=  'V';
  UnVector[ 9] :=  'H';  UnVector[10] :=  'G';

  WrStr ("Dame una letra: ");
  UnaLetra := RdChar(); RdLn;

  IF BuscarCentinela(UnVector, UnaLetra, Posicion) THEN
    WrStr ("El elemento fue encontrado en la posición ");
    WrCard (Posicion, 0)
  ELSE
    WrStr ("El elemento no fue encontrado")
  END
END Centinela.
```

El análisis para este algoritmo es:

- *mejor caso*: el elemento buscado está en la primera posición del vector, y se realiza una comparación da datos en la única iteración del bucle y una comparación de enteros después del bucle.
- *peor caso*: el elemento no se encuentra en el vector. En este caso el bucle **`WHILE`** se ejecuta `MAXVECTOR`+1 veces y en cada iteración se efectúa una comparación de datos. Después del bucle se realiza una comparación de enteros. En total se realizan `MAXVECTOR`+2 comparaciones. Hemos reducido el número de comparaciones en un orden de la mitad con respecto al algoritmo de búsqueda lineal original.

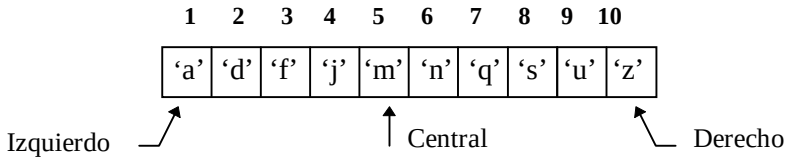
8.5.3 Búsqueda binaria

Presentamos a continuación un algoritmo de búsqueda más eficiente pero que sólo puede ser usado si sabemos que los elementos dentro del vector se encuentran ordenados. Supondremos que el orden dentro del vector es de menor a mayor (el elemento menor está en la primera posición del vector y el mayor en la última) aunque un algoritmo análogo se puede obtener para el caso inverso (de mayor a menor). La descripción del algoritmo para buscar un elemento `Elemento` en un vector `Vector` con tamaño `MAXVECTOR` es la siguiente:

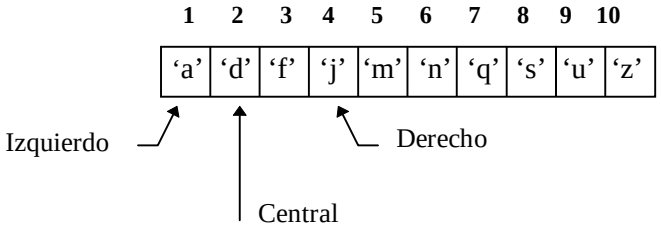
- utilizamos dos variables `Izquierdo` y `Derecho` que en cada momento indican el segmento de vector en el que estamos buscando el elemento. Como comenzamos buscando en todo el vector estas variables se inicializan a 1 y `MAXVECTOR` respectivamente.
- calculamos la posición central del vector con la siguiente fórmula: `Central := (Izquierdo+Derecho) DIV 2`
- si el elemento que ocupa la posición dada por `Central` coincide con el buscado lo hemos encontrado y acabamos.
- si el elemento buscado es menor que el que ocupa la posición central nos quedamos con la mitad izquierda del vector, esto es `Derecho := Central-1`.
- si el elemento buscado es mayor nos quedamos con la mitad derecha, o sea `Izquierdo := Central+1`
- repetir este proceso hasta que el encontremos el elemento o hasta que `Izquierdo` sea estrictamente mayor a `Derecho`. En este último caso el segmento del vector que estamos buscando es vacío lo cual indica que el elemento buscado no se encuentra en el vector.

El algoritmo es similar al que utilizamos cuando buscamos una palabra en el diccionario. Abrimos el diccionario por la mitad, y si la palabra que buscamos es menor nos quedamos con la primera mitad de páginas o si es mayor con la segunda.

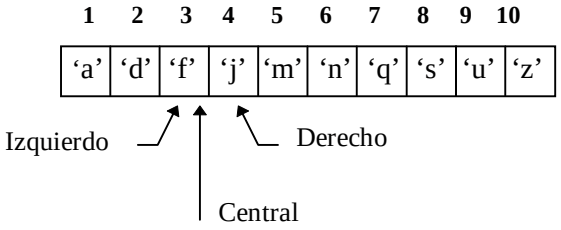
Veamos como funciona el algoritmo con un ejemplo. Se trata de buscar el elemento `'f'` en el vector que aparece en la figura:



Como el elemento buscado es menor que el que está en la posición dada por `Central` nos quedamos con la mitad izquierda del vector:

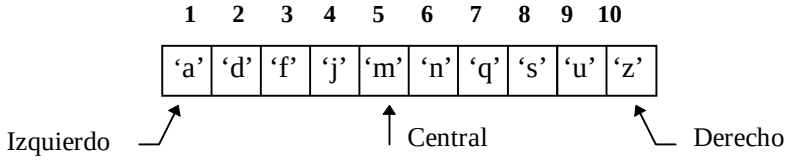


Como el elemento buscado es mayor que el que ocupa la posición dada por `Central` nos quedamos con la parte derecha del vector:

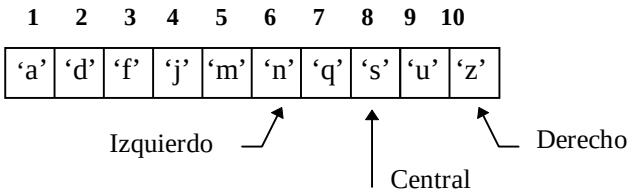


El elemento buscado coincide con el `Central`, por lo que ha sido encontrado en la posición 3.

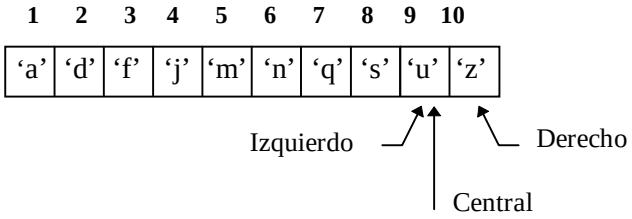
Veamos que ocurre si buscamos un elemento que no se encuentra en el vector, por ejemplo `'t'`:



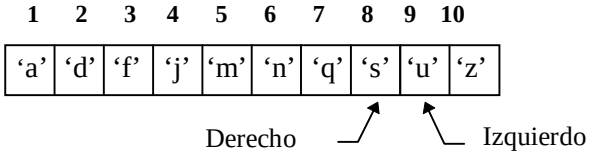
El elemento buscado es mayor que el que determina `Central`, así que tomamos la parte derecha del vector:



El elemento buscado sigue siendo mayor que el de la posición 8, por lo que volvemos a tomar la parte derecha del vector:



El elemento buscado es menor que el que está en la posición dada por `Central`, así que nos quedamos con la parte izquierda del vector:



`Izquierdo` es mayor que `Derecho`, por lo que podemos asegurar que el elemento buscado no se encuentra en el vector.

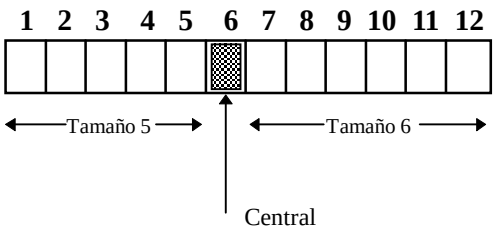
Una implementación de este algoritmo en *Modula-2* es la siguiente:

```
MODULE Binaria;
FROM IO IMPORT RdChar, RdLn, WrStr, WrCard;
CONST
  MAXVECTOR = 10;
TYPE
  INDICE    = [1..MAXVECTOR];
  ELEMENTO  = CHAR;
  VECTOR    = ARRAY INDICE OF ELEMENTO;
VAR
  UnVector : VECTOR;
  UnaLetra : ELEMENTO;
  Posicion : INDICE;

PROCEDURE BuscarBinaria (Vector : VECTOR; Elemento : ELEMENTO;
                         VAR Posicion : INDICE) : BOOLEAN;
  TYPE
    INDICEBUSQUEDA = [0..MAXVECTOR+1];
  VAR
    Izquierdo, Derecho, Central : INDICEBUSQUEDA;
    Encontrado                  : BOOLEAN;
  BEGIN
     Izquierdo := 1;
     Derecho := MAXVECTOR;
     Encontrado := FALSE;
```

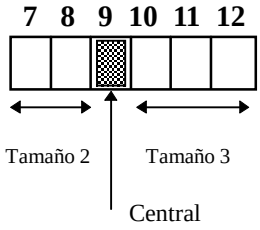
```
    WHILE (NOT Encontrado) AND (Izquierdo <= Derecho) DO
      Central := (Izquierdo+Derecho) DIV 2;
      IF Vector[Central] = Elemento THEN
        Encontrado :=TRUE
      ELSIF Vector[Central] < Elemento THEN
        Izquierdo := Central + 1
      ELSE
        Derecho := Central - 1
      END
    END;
    IF Encontrado THEN
      Posicion := Central
    END;
    RETURN Encontrado
  END BuscarBinaria;

BEGIN
  UnVector[ 1] :=  'A';  UnVector[ 2] :=  'D';
  UnVector[ 3] :=  'F';  UnVector[ 4] :=  'J';
  UnVector[ 5] :=  'M';  UnVector[ 6] :=  'N';
  UnVector[ 7] :=  'Q';  UnVector[ 8] :=  'S';
  UnVector[ 9] :=  'U';  UnVector[10] :=  'Z';

  WrStr ("Dame una letra: ");
  UnaLetra := RdChar(); RdLn;

  IF BuscarBinaria(UnVector, UnaLetra, Posicion) THEN
    WrStr ("El elemento fue encontrado en la posición ");
    WrCard (Posicion, 0)
  ELSE
    WrStr ("El elemento no fue encontrado")
  END
END Binaria.
```

Analicemos el comportamiento del algoritmo anterior en el mejor y en el peor caso:

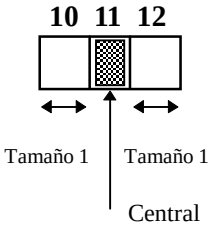
- *mejor caso*: el elemento buscado está en la posición central del vector. En este caso se realiza una comparación de enteros (`Izquierdo<=Derecho`) y otra de datos (`Vector[Central]=Elemento`).
- *peor caso*: lo peor que nos puede pasar es que el elemento buscado no se encuentre en el vector. Si el tamaño del vector (`MAXVECTOR`) es un número par, al dividir el vector en dos partes obtenemos dos vectores de tamaños `MAXVECTOR DIV 2` y (`MAXVECTOR DIV 2)-1`.En el caso de que el tamaño del vector sea impar obtenemos dos vectores de tamaño `MAXVECTOR DIV 2`.



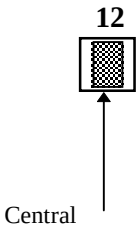
Lo peor que podría ocurrir sería que el elemento buscado estuviese en la parte de mayor tamaño



De nuevo en este caso lo peor sería tener que seguir buscando por la parte derecha:



Ahora ambos lados tienen la misma longitud, supongamos que seguimos por la derecha



En este caso el elemento o está en la casilla o no está.

Podemos observar lo siguiente:

- Para un vector de tamaño 1 el bucle se ejecutaría una sola vez.
- Para un vector de tamaño 3 el bucle se ejecutaría una vez más que para un vector de tamaño 1.
- Para un vector de tamaño 6 el bucle se ejecutaría una vez más que para un vector de tamaño 3.

En general, si llamamos $T(n)$ al número máximo de veces que se ejecuta el bucle para un vector de tamaño n, obtenemos la siguiente ecuación recurrente:

$$T(1) = 1$$
$$T(n) = 1 + T(n\ DIV\ 2),\ si\ n \neq 1$$

cuya solución es:

$T(n) = 1 + TRUNC(Log_2 n)$ (se puede demostrar por inducción).

En cada paso, lo peor que puede ocurrir es que el elemento buscado sea menor que el central actual y en este caso tenemos una comparación de enteros (`Izquierdo<=Derecho`) y dos comparaciones de elementos (`Vector[Central]=Elemento` y `Vector[Central]<Elemento`) por cada iteración. Cuando `Izquierdo` es mayor que `Derecho`, se produce una última comparación de

enteros que evita que volvamos a entrar en el bucle. Sumando obtenemos 3*(1+*TRUNC*(*Log* $_2$ MAXVECTOR))+1 comparaciones en el peor caso.

8.5.4 Consideraciones sobre los algoritmos de búsqueda

Si representamos el número de comparaciones que realizan los tres algoritmos podemos observar que la búsqueda con centinela es siempre mejor (excepto en el caso de un vector con un único elemento donde es igual) que la búsqueda lineal. Por otro lado la búsqueda binaria es mejor que la lineal con centinela a partir de un vector con 11 elementos. Como son pocas las situaciones reales en las que se usan ordenadores para buscar elementos en vectores tan pequeños, diremos que siempre que los elementos dentro del vector estén ordenados, es mejor utilizar la búsqueda binaria. El orden de reducción de comparaciones es bastante considerable, ya que el logaritmo crece mucho más despacio que la recta (p.e. para un vector de 1000 elementos la búsqueda binaria puede realizar 31 comparaciones como máximo frente a las 1002 que puede realizar la búsqueda lineal con centinela). Sólo cuando los elementos no estén ordenados dentro del vector, utilizaremos la búsqueda lineal con centinela.

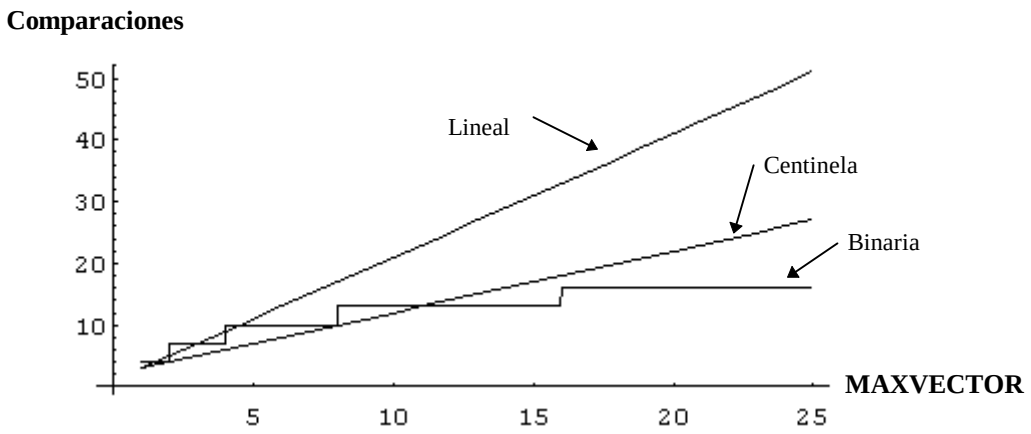


Figura 3. Comparaciones efectuadas por los distintos algoritmos de búsqueda para distintos tamaños de vector.

8.6 Algoritmos de ordenación

Éste es otro de los problemas típicos para los que se ha usado el ordenador tradicionalmente. Consiste en, dado un vector que contiene una serie de elementos desordenados, ordenar estos elementos dentro del mismo vector siguiendo para ello cierto criterio. Normalmente cada elemento del vector será un registro y uno de los campos determinará el orden del elemento. A este campo se le llama *clave*. Los algoritmos que vamos a considerar ordenarán el vector en orden ascendente (el elemento con menor clave quedará en la primera posición del vector y el de mayor clave en la última) aunque se pueden escribir algoritmos análogos para ordenar de modo descendente. Un aspecto importante de los algoritmos de ordenación es que deben ser *internos*: el algoritmo debe tomar como entrada un vector con los datos originales y producir como salida el mismo vector con los datos ordenados. En este proceso no se podrá utilizar otra estructura para realizar el proceso de ordenación, es decir el algoritmo debe ir ordenando los datos dentro de la propia estructura. Esto es debido a que los vectores que se suelen ordenar en problemas reales son de gran tamaño, por lo cual es razonable suponer que sólo podremos mantener en memoria una copia de la estructura a la vez.

Veremos a continuación algunos de los algoritmos simples clásicos para el problema de la ordenación: ordenación por intercambio, por selección y por inserción. Estudiaremos la eficiencia de cada uno de ellos y veremos que los dos últimos son razonablemente eficientes.

8.6.1 Ordenación por intercambio o método de la burbuja

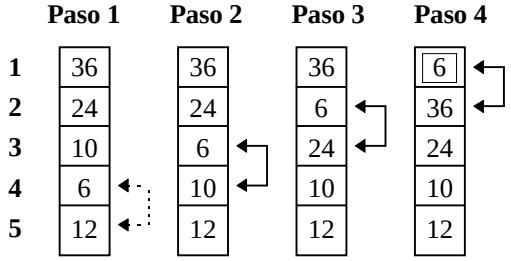
Supongamos que tenemos un vector que contiene `MAXVECTOR` elementos desordenados y queremos ordenarlos. Un algoritmo para este problema es el siguiente:

- Desde la posición `MAXVECTOR` a la posición 2, si `Vector[j-1]` es mayor que `Vector[j]`, intercambiar `V[j-1]` y `V[j]`. Después de este proceso, en `V[1]` tendremos el elemento menor del vector.
- Repetir ahora el proceso anterior desde la posición `MAXVECTOR` del vector hasta la posición 3. Ahora en `Vector[2]` tenemos el 2º elemento menor.
- ...
- Repetir el proceso desde la posición `MAXVECTOR` a `MAXVECTOR` y el vector estará ordenado.

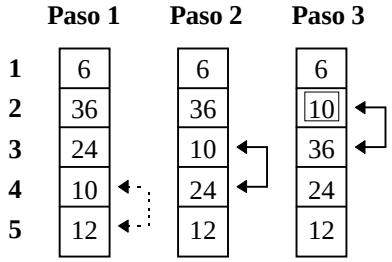
Veamos como podemos usar el algoritmo anterior para ordenar un vector con 5 enteros. El vector original es:

1	**2**	**3**	**4**	**5**
36	24	10	6	12

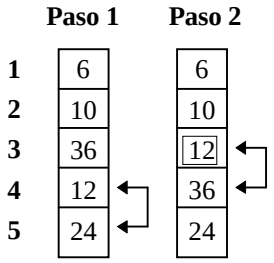
En la primera iteración del algoritmos comparamos los elementos en las posiciones 5 y 4, la 4 y la 3, la 3 y la 2 y por último la 2 y la 1. Cada flecha continua en el diagrama representa una comparación y un intercambio de elementos, mientras que una discontinua una comparación sin intercambio. Tras esta iteración el elemento menor del vector (el 6) ya se encuentra situado en la primera posición del vector:



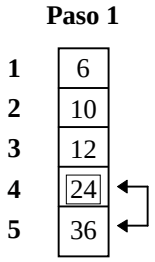
En la segunda iteración del algoritmo partimos de comparar los elementos en las posiciones 5 y 4 y acabamos comparando la 3 y la 2. El segundo elemento menor (el 10) queda colocado en la casilla 2:



En la tercera iteración del algoritmo colocamos el elemento 12:



Tras la última iteración el vector queda ordenado:



Podemos observar que en cada iteración del algoritmo un elemento del vector asciende a su posición. Es por esto que se a este algoritmo también se le llama de la burbuja.

Una implementación de este algoritmo en Modula-2 es la siguiente:

```
MODULE Burbuja;
FROM IO IMPORT WrInt, WrChar, WrLn;
CONST
  MAXVECTOR = 5;
TYPE
  INDICE   = [1..MAXVECTOR];
  ELEMENTO = INTEGER;
  VECTOR   = ARRAY INDICE OF ELEMENTO;
VAR
  UnVector : VECTOR;

  PROCEDURE OrdenarBurbuja (VAR Vector : VECTOR);
  VAR i, j : INDICE;

    PROCEDURE Intercambiar (VAR a, b : ELEMENTO);
    VAR Temp : ELEMENTO;
    BEGIN
      Temp := a;
      a    := b;
      b    := Temp
    END Intercambiar;

  BEGIN
    FOR i := 2 TO MAXVECTOR DO
      FOR j := MAXVECTOR TO i BY -1 DO
        IF Vector[j-1] > Vector[j] THEN
          Intercambiar (Vector[j-1], Vector[j])
        END
      END
    END
  END OrdenarBurbuja;

  PROCEDURE WrVector (Vector : VECTOR);
  VAR i : INDICE;
  BEGIN
    FOR i := 1 TO MAXVECTOR DO
      WrInt (Vector[i], 0);
```

```
      WrChar (' ')
    END;
    WrLn
  END WrVector;

BEGIN
  UnVector[1] := 36;
  UnVector[2] := 24;
  UnVector[3] := 10;
  UnVector[4] :=  6;
  UnVector[5] := 12;
  OrdenarBurbuja(UnVector);
  WrVector(UnVector)
END Burbuja.
```

En el caso de los algoritmos de ordenación, la eficiencia se mide contando el número de comparaciones y movimientos efectuados sobre los elementos en el vector. Estudiemos primero la comparaciones realizadas por el algoritmo de intercambio:

- La primera vez que se ejecuta el bucle más interno se realizan (MAXVECTOR-1) comparaciones (j varía desde MAXVECTOR a 2).
- La segunda vez que se ejecuta el bucle más interno se realizan (MAXVECTOR-2) comparaciones (j varía desde MAXVECTOR a 3).
- ...
- La última vez que se ejecuta el bucle más interno se realiza 1 comparación (j varía desde MAXVECTOR a MAXVECTOR).

Sumando obtenemos que el número de comparaciones es:

$$\begin{aligned} \textit{total comparaciones} &= (\mathtt{MAXVECTOR}-1)+(\mathtt{MAXVECTOR}-2)+\cdots+2+1 \\ &= \underbrace{\mathtt{MAXVECTOR}+\mathtt{MAXVECTOR}+\cdots+\mathtt{MAXVECTOR}}_{(\mathit{MAXVECTOR}-1)\ \mathit{veces}}- \\ &\quad (1+2+\cdots+(\mathtt{MAXVECTOR}-1)) \\ &= (\mathtt{MAXVECTOR}-1)\cdot\mathtt{MAXVECTOR}- \\ &\quad (\underbrace{1}_{\mathit{primero}}+\underbrace{(\mathtt{MAXVECTOR}-1)}_{\mathit{ultimo}})\cdot\underbrace{(\mathtt{MAXVECTOR}-1)}_{n^o\ \mathit{terminos}} \\ &= \frac{(2\cdot\mathtt{MAXVECTOR}-\mathtt{MAXVECTOR})\cdot(\mathtt{MAXVECTOR}-1)}{2}= \\ &= \frac{\mathtt{MAXVECTOR}\cdot(\mathtt{MAXVECTOR}-1)}{2} \end{aligned}$$

En el peor caso (el array se encuentra inicialmente ordenado en orden inverso), por cada comparación se puede producir un intercambio de datos (cada intercambio equivale a dos movimientos de datos en el vector), por lo que tenemos

$$\textit{total movimientos} = 2\cdot\textit{total comparaciones} = \mathtt{MAXVECTOR}\cdot(\mathtt{MAXVECTOR}-1)$$

Los resultados obtenidos con este algoritmo no son buenos. Para ordenar un vector de p.e. 100 elementos deberemos realizar 4950 comparaciones y podemos llegar a 9900 movimientos de datos en el interior del vector.

8.6.2 Ordenación por selección

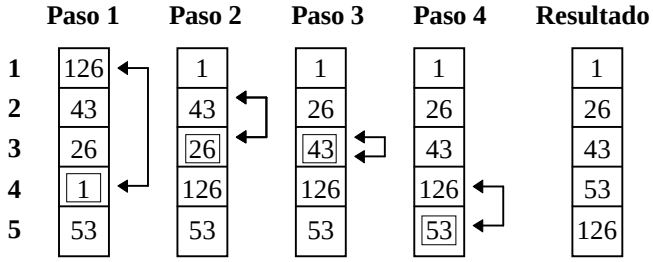
La idea en la que se basa este algoritmo de ordenación es:

- Buscar el elemento menor desde `Vector[1]` a `Vector[MAXVECTOR]` e intercambiarlo con el que está en `Vector[1]`. Ya tenemos el menor elemento del vector en `Vector[1]`.
- Repetimos el proceso, pero ahora buscamos el menor desde `Vector[2]` a `Vector[MAXVECTOR]` e intercambiarlo con `Vector[2]`. Tras esta iteración, el segundo elemento menor está situado en `Vector[2]`.
- Repetir el proceso hasta colocar bien `Vector[MAXVECTOR-1]`. Si $\forall$ j, $1 \leq j \leq$ `MAXVECTOR-1`, `Vector[j]` está bien colocado entonces también lo está `Vector[MAXVECTOR]` y el vector ya está ordenado.

Veamos como podemos usar el algoritmo anterior para ordenar un vector con 5 enteros. El vector original es:

1	**2**	**3**	**4**	**5**
126	43	26	1	113

La secuencia de pasos es:



En el paso i-ésimo se busca elemento menor desde `Vector[i]` a `Vector[MAXVECTOR]` y se intercambia este elemento con el situado en `Vector[i]`.

La implementación en Modula-2 es del algoritmo es:

```
MODULE Seleccion;
FROM IO IMPORT WrInt, WrChar, WrLn;
CONST
  MAXVECTOR = 5;
TYPE
  INDICE   = [1..MAXVECTOR];
  ELEMENTO = INTEGER;
  VECTOR   = ARRAY INDICE OF ELEMENTO;
VAR
  UnVector : VECTOR;

  PROCEDURE OrdenarSeleccion (VAR Vector : VECTOR);
  VAR
    i, j, PosicionMinimo   : INDICE;

    PROCEDURE Intercambiar (VAR a, b : ELEMENTO);
    VAR Temp : ELEMENTO;
    BEGIN
      Temp := a;
      a    := b;
      b    := Temp
    END Intercambiar;

  BEGIN
    FOR i := 1 TO MAXVECTOR-1 DO
      PosicionMinimo := i;
      FOR j := i+1 TO MAXVECTOR  DO
```

```
        IF Vector[j] < Vector[PosicionMinimo] THEN
          PosicionMinimo := j
        END
      END;
      Intercambiar (Vector[i], Vector[PosicionMinimo])
    END
  END OrdenarSeleccion;

  PROCEDURE WrVector (Vector : VECTOR);
  VAR i : INDICE;
  BEGIN
    FOR i := 1 TO MAXVECTOR DO
      WrInt (Vector[i], 0);
      WrChar (' ')
    END;
    WrLn
  END WrVector;

BEGIN
  UnVector[1] := 126;
  UnVector[2] :=  43;
  UnVector[3] :=  26;
  UnVector[4] :=   1;
  UnVector[5] :=  53;
  OrdenarSeleccion(UnVector);
  WrVector(UnVector)
END Seleccion.
```

El bucle más interno (con variable de control `j`) busca el `Vector[k]`, `i` ≤ `k` ≤ `MAXVECTOR` con contenido mínimo. El bucle más externo coloca este elemento en su posición adecuada.

Estudiemos la eficiencia de este algoritmo:

- La primera vez que se ejecuta el bucle interior se realizan `MAXVECTOR`-1 comparaciones (`j` varía desde 2 a `MAXVECTOR`).
- La segunda vez que se ejecuta el bucle interior se realizan `MAXVECTOR`-2 comparaciones (`j` varía desde 3 a `MAXVECTOR`).
- ...
- En la última iteración se realiza una comparación (`j` varía entre (`MAXVECTOR`-1)+1= `MAXVECTOR` y `MAXVECTOR`).

Sumando obtenemos:

$$
\begin{aligned}
\textit{total comparaciones} &= (\mathrm{MAXVECTOR}-1)+(\mathrm{MAXVECTOR}-2)+\cdots+2+1 \\
&= \underbrace{\mathrm{MAXVECTOR}+\mathrm{MAXVECTOR}+\cdots+\mathrm{MAXVECTOR}}_{(\mathit{MAXVECTOR}-1)\ \mathit{veces}}- \\
&\quad (1+2+\cdots+(\mathrm{MAXVECTOR}-1)) \\
&= (\mathrm{MAXVECTOR}-1)\cdot\mathrm{MAXVECTOR}- \\
&\quad (\underbrace{1}_{\mathit{primero}}+\underbrace{(\mathrm{MAXVECTOR}-1)}_{\mathit{ultimo}})\cdot\underbrace{(\mathrm{MAXVECTOR}-1)}_{\mathit{n^o\ terminos}} \\
&= \frac{(2\cdot\mathrm{MAXVECTOR}-\mathrm{MAXVECTOR})\cdot(\mathrm{MAXVECTOR}-1)}{2}= \\
&= \frac{\mathrm{MAXVECTOR}\cdot(\mathrm{MAXVECTOR}-1)}{2}
\end{aligned}
$$

En cuanto al número de movimiento de datos en el vector, tenemos que el algoritmo realiza un intercambio (dos movimientos) en cada iteración del bucle externo, y que este bucle se ejecuta `MAXVECTOR`-1 veces:

$$\textit{total movimientos} = 2 \cdot \textit{total iteraciones bucle}\ \texttt{i} = 2 \cdot (\texttt{MAXVECTOR} - 1)$$

Este algoritmo se comporta de un modo razonable en cuanto al número de movimientos de datos que realiza, pero sigue siendo poco eficiente en cuanto a comparaciones.

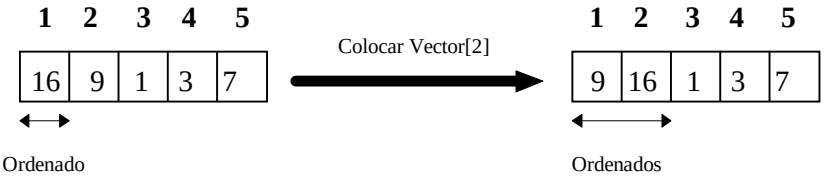
8.6.3 Ordenación por inserción

Este algoritmo se basa en la siguiente idea: si los elementos `Vector[1]` a `Vector[i-1]` están ordenados entre sí, bastará con insertar `Vector[i]` en su posición adecuada dentro de estos para que `Vector[1]` a `Vector[i]` queden ordenados.

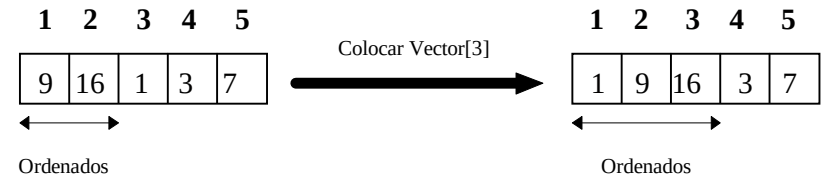
Supongamos que queremos ordenar el siguiente vector:

1	2	3	4	5
16	9	1	3	7

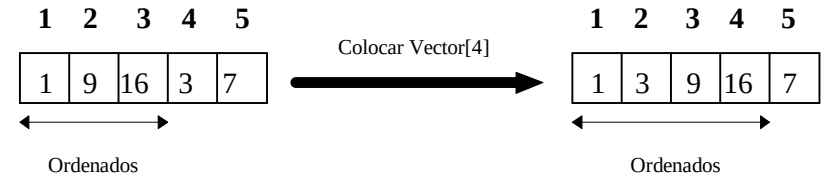
Como el vector inicial está desordenado, al principio solo podemos garantizar que `Vector[1]` a `Vector[1]` está ordenado entre sí. Si colocamos el elemento `Vector[2]` en su posición adecuada dentro de `Vector[1]` a `Vector[1]` tendremos que `Vector[1]` a `Vector[2]` está ordenado:

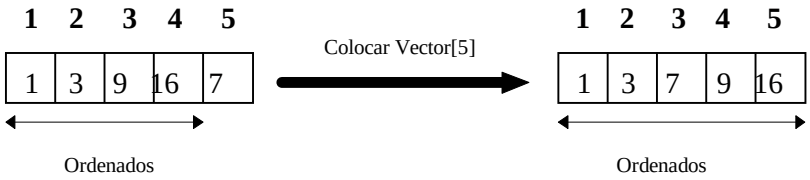


En el segundo paso, como `Vector[1]` a `Vector[2]` está ordenado, colocamos `Vector[3]` en su posición adecuada dentro de `Vector[1]` a `Vector[2]` y tendremos `Vector[1]` a `Vector[3]` ordenado:



Repetimos un par de pasos más y obtenemos el vector ordenado:



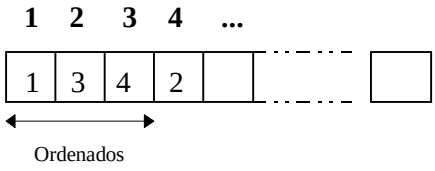


El algoritmo obtenido es:

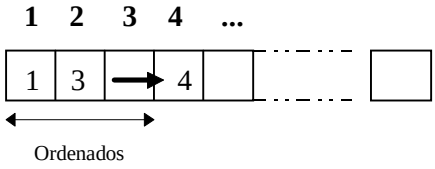
```
FOR i := 2 TO MAXVECTOR DO
  Colocar Vector[i] en su posición adecuada
  dentro de Vector[1]..Vector[i-1]
END
```

8.6.3.1 Ordenación por inserción directa

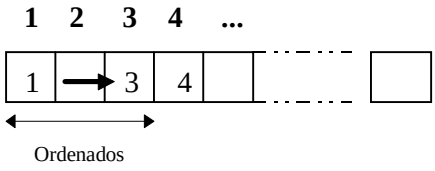
Necesitamos un método para insertar en cada paso a `Vector[i]` en su posición adecuada dentro de `Vector[1]` .. `Vector[i-1]`. Una posibilidad es ir abriendo hueco en la secuencia `Vector[1]` .. `Vector[i-1]` para encajar `Vector[i]`. Supongamos que tenemos los tres primeros elementos del vector ordenados y queremos insertar el cuarto en su posición:



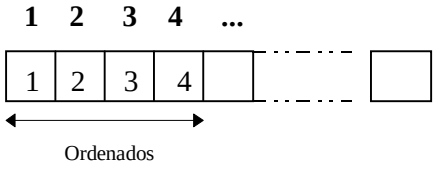
como 2 < 4 abro hueco



como 2 < 3 abro hueco



como 2 < 1 no es cierto no abro hueco y coloco 2 en el hueco, y tengo los cuatro primeros elementos ordenados:



El algoritmo resultante en Modula-2 es:

```
MODULE InserDir;
FROM IO IMPORT WrInt, WrChar, WrLn;
CONST
  MAXVECTOR = 5;
```

```
TYPE
  INDICE   = [1..MAXVECTOR];
  ELEMENTO = INTEGER;
  VECTOR   = ARRAY INDICE OF ELEMENTO;
VAR
  UnVector : VECTOR;

  PROCEDURE OrdenarInsercionDirecta (VAR Vector : VECTOR);
  TYPE
    INDICEHUECO = [0..MAXVECTOR];
  VAR
    i               : INDICE;
    j               : INDICEHUECO;
    ElementoAColocar : ELEMENTO;
  BEGIN
    FOR i := 2 TO MAXVECTOR DO
      ElementoAColocar := Vector[i];
      (* Hacer hueco *)
      j := i-1;
      WHILE (j>=1) AND (ElementoAColocar<Vector[j]) DO
        Vector[j+1] := Vector[j];
        j := j-1
      END;
      (* Colocar el elemento *)
      Vector[j+1] := ElementoAColocar
    END
  END OrdenarInsercionDirecta;

  PROCEDURE WrVector (Vector : VECTOR);
  VAR i : INDICE;
  BEGIN
    FOR i := 1 TO MAXVECTOR DO
      WrInt (Vector[i], 0);
      WrChar (' ')
    END;
    WrLn
  END WrVector;

BEGIN
  UnVector[1] := 16;
  UnVector[2] :=  9;
  UnVector[3] :=  1;
  UnVector[4] :=  3;
  UnVector[5] :=  7;
  OrdenarInsercionDirecta(UnVector);
  WrVector(UnVector)
END InserDir.
```

Analicemos la complejidad de este algoritmo. Contaremos primero el número de comparaciones del tipo `ElementoAColocar<Vector[j]` que realiza el algoritmo, en el peor caso:

- El bucle exterior (bucle `i`) se ejecuta `MAXVECTOR`-1 veces (`i` varía desde 2 a `MAXVECTOR`).
- Cuando `i`=2, el bucle `WHILE` se puede llegar a ejecutar 1 vez (`j` varía desde `i`-1=2-1=1 hasta 1 (`j`>=1)).
- Cuando `i`=3, el bucle `WHILE` se puede llegar a ejecutar 2 veces (`j` varía desde `i`-1=3-1=2 hasta 1 (`j`>=1)).
- ...
- Cuando `i`=`MAXVECTOR`, el bucle `WHILE` se puede llegar a ejecutar `MAXVECTOR`-1 veces.

El nº total de comparaciones es:

$$total\ comparaciones = 1 + 2 + 3 + \ldots + (MAXVECTOR - 1)$$

$$= \frac{\overset{1º}{1} + (\overset{ultimo}{MAXVECTOR - 1})) \cdot (\overset{nº\ de\ terminos}{MAXVECTOR - 1})}{2}$$

$$= \frac{MAXVECTOR \cdot (MAXVECTOR - 1)}{2}$$

En cuanto al número de movimientos de datos dentro del vector, en el peor caso:

- Cuando `i`=2, se puede llegar a realizar 1 movimiento para hacer hueco + otro al insertar el dato en su posición correcta = 2 movimientos.
- Cuando `i`=3, se pueden llegar a realizar 2 movimientos para hacer hueco + otro al insertar el dato en su posición correcta = 3 movimientos.
- ...
- Cuando `i`=`MAXVECTOR`, se pueden llegar a realizar `MAXVECTOR`-1 movimientos para hacer hueco + otro al insertar el dato en su posición correcta = `MAXVECTOR` movimientos.

Sumado:

$$total\ movimientos = 2 + 3 + \ldots + MAXVECTOR$$

$$= \frac{\overset{1º}{2} + \overset{ultimo}{MAXVECTOR}) \cdot (\overset{nº\ de\ terminos}{MAXVECTOR - 1})}{2}$$

8.6.3.2 Ordenación por inserción binaria

El algoritmo anterior encuentra la posición adecuada donde insertar el elemento `Vector[i]` buscando en `Vector[1]` .. `Vector[i-1]` de derecha a izquierda. Dado que `Vector[1]` .. `Vector[i-1]` ya está ordenado, podemos obtener un algoritmo de ordenación más eficiente si utilizamos una búsqueda binaria para encontrar la posición adecuada de `Vector[i]` dentro de `Vector[1]` .. `Vector[i-1]`. Un algoritmo en Modula-2 que usa esta idea es el siguiente:

```
MODULE InserBin;
FROM IO IMPORT WrInt, WrChar, WrLn;
CONST
  MAXVECTOR = 5;
TYPE
  INDICE   = [1..MAXVECTOR];
  ELEMENTO = INTEGER;
  VECTOR   = ARRAY INDICE OF ELEMENTO;
VAR
  UnVector : VECTOR;

  PROCEDURE OrdenarInsercionBinaria (VAR Vector : VECTOR);
  TYPE
    INDICEHUECO = [0..MAXVECTOR];
  VAR
    i, j                        : INDICE;
    Izquierdo, Derecho, Central : INDICEHUECO;
    ElementoAColocar            : ELEMENTO;
  BEGIN
    FOR i := 2 TO MAXVECTOR DO
      ElementoAColocar := Vector[i];
      (* Buscar posición donde insertar ElementoABuscar *)
      Izquierdo := 1;
      Derecho := i-1;
```

```
      WHILE Izquierdo <= Derecho DO
        Central := (Izquierdo+Derecho) DIV 2;
        IF ElementoAColocar < Vector[Central] THEN
          Derecho := Central-1
        ELSE
          Izquierdo := Central+1
        END
      END;
      (* Hay que insertarlo en Vector[Izquierdo] *)

      (* Hacer hueco *)
      FOR j := i-1 TO Izquierdo BY -1 DO
        Vector[j+1] := Vector[j]
      END;
      Vector[Izquierdo] := ElementoAColocar
    END
  END OrdenarInsercionBinaria;

  PROCEDURE WrVector (Vector : VECTOR);
  VAR i : INDICE;
  BEGIN
    FOR i := 1 TO MAXVECTOR DO
      WrInt (Vector[i], 0);
      WrChar (' ')
    END;
    WrLn
  END WrVector;

BEGIN
  UnVector[1] := 16;
  UnVector[2] :=  9;
  UnVector[3] :=  1;
  UnVector[4] :=  3;
  UnVector[5] :=  7;
  OrdenarInsercionBinaria(UnVector);
  WrVector(UnVector)
END InserBin.
```

Estudiemos la eficiencia del algoritmo (recordemos que solo contamos comparaciones entre elementos):

- Cuando `i`=2, se hace una búsqueda binaria en `Vector[1]`..`Vector[2-1]`, o sea, 1+*TRUNC*(*Log* $_2$ 1) = 1comparaciones en el peor caso.
- Cuando `i`=3, se hace una búsqueda binaria en `Vector[1]`..`Vector[3-1]`, o sea, 1+*TRUNC*(*Log* $_2$ 2) = 2comparaciones en el peor caso.
- ...
- Cuando `i`=`MAXVECTOR`, se hace una búsqueda binaria en `Vector[1]`..`Vector[MAXVECTOR-1]`, o sea, 1+*TRUNC*(*Log* $_2$ `MAXVECTOR-1`) comparaciones en el peor caso.

Sumando:

$$total\ comparaciones =$$

$$1+TRUNC\left[Log_2 1\right]+1+TRUNC\left[Log_2 2\right]+\ldots+1+TRUNC\left[Log_2 \mathtt{MAXVECTOR}-1\right]=$$

$$\underbrace{1+1+\cdots 1}_{MAXVECTOR-1\ veces}+\sum_{i=1}^{MAXVECTOR-1} TRUNC\left[Log_2 i\right]=$$

$$(\mathtt{MAXVECTOR}-1)+\sum_{i=1}^{MAXVECTOR-1} TRUNC\left[Log_2 i\right] <$$

$$(\mathtt{MAXVECTOR}-1)+(\mathtt{MAXVECTOR}-1)\cdot TRUNC\left[Log_2 \mathtt{MAXVECTOR}-1\right]=$$

$$(\mathtt{MAXVECTOR}-1)\cdot\left(1+TRUNC\left[Log_2 \mathtt{MAXVECTOR}-1\right]\right)$$

El número de movimientos es idéntico al del algoritmo de inserción directa:

$$total\ movimientos = \frac{(\mathtt{MAXVECTOR}+2)\cdot(\mathtt{MAXVECTOR}-1)}{2}$$

8.6.4 Consideraciones sobre los algoritmos de ordenación

Podemos ver en las siguientes dos gráficas el número de comparaciones y de movimientos que realizan en el peor caso los distintos algoritmos de ordenación estudiados para distintos tamaños del vector a ordenar:

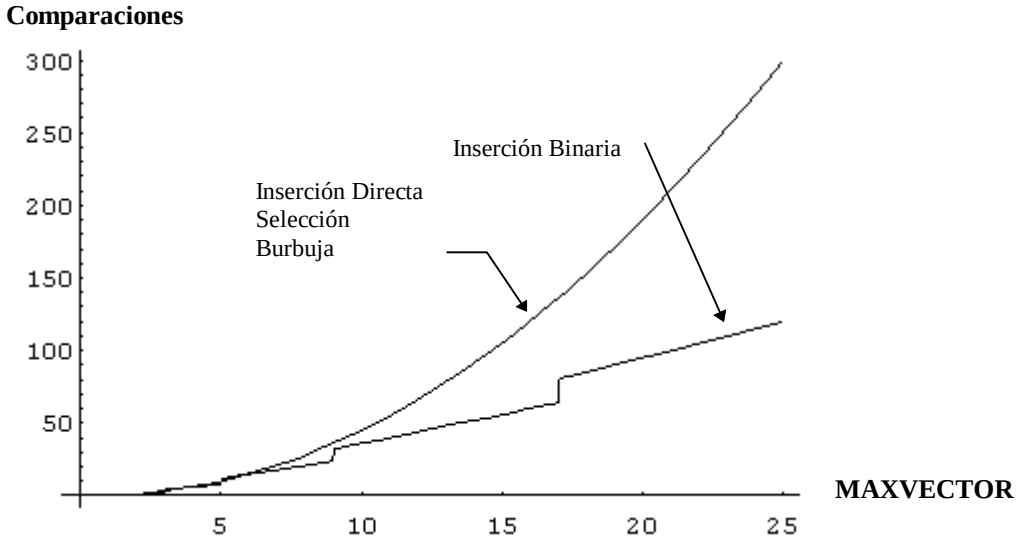


Figura 4. Comparaciones realizadas por los distintos algoritmos de ordenación, para distintos tamaños de vector.

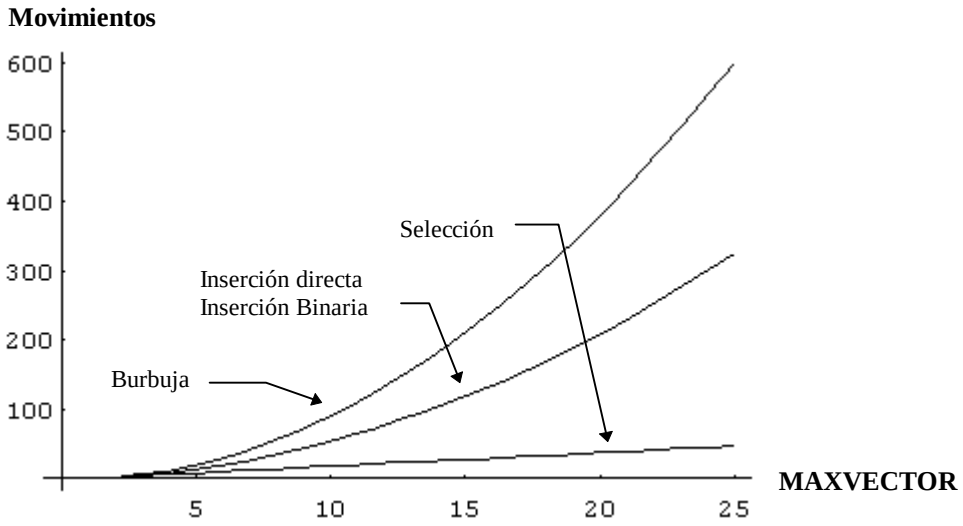


Figura 5. Movimientos realizados por los distintos algoritmos de ordenación, para distintos tamaños de vector.

Para tamaños grandes de vector (que serán los que nos interesen en casos reales), el algoritmo de ordenación por inserción binaria es el que realiza el menor número de comparaciones. Este algoritmo es el más adecuado cuando la operación de comparación de los datos que se están ordenando es compleja. En cuanto a movimientos de datos, el algoritmo de ordenación por selección obtiene los mejores resultados. Usaremos este algoritmo cuando el tamaño de los datos a ordenar sea grande. El algoritmo de ordenación por intercambio (o burbuja) obtiene los peores resultados tanto en número de movimientos como en comparaciones, por lo que no debería ser usado nunca.

Relación de problemas (Tema 8)

1. Escribe un programa que permita leer dos matrices de tamaño 10x10 de números reales del teclado, calcular la matriz producto y escribirla por pantalla.

2. Escribe un programa que permita leer una matriz de tamaño 5x6 de números reales del teclado, calcular la matriz traspuesta y escribirla por pantalla.

3. Escribe un programa que convierta un número entero positivo expresado en cualquier base natural entre 2 y 10 a una base natural entre 2 y 10. Para ello la entrada se leerá del teclado la base inicial, el número y la base destino.

4. Dadas las siguientes declaraciones:

```
TYPE
  NOMBRE             = ARRAY [0..25] OF CHAR;
  REGISTROLISTA      = RECORD
                         Lista     : ARRAY [1..5000] OF NOMBRE;
                         Sublista  : ARRAY [1..20] OF NOMBRE
                       END;
  REGISTROCATALOGO =  RECORD
                         NombreElemento: NOMBRE;
                         TipoElemento  : (Uno, Dos);
                         Subcatalogo     : REGISTROLISTA
                       END;
VAR
  Catalogo    : ARRAY [1..20] OF REGISTROCATALOGO;
  UnCatalogo  : REGISTROCATALOGO;
  UnaLista    : REGISTROLISTA;
  UnNombre    : NOMBRE;
```

¿Cuáles de las siguientes sentencias son válidas ?

1) `IF (REGISTROCATALOGO.TipoElemento = Uno) THEN ...`
2) `Catalogo[1].Subcatalogo.Lista[2] := UnCatalogo;`
3) `Catalogo[5].RegistroLista := UnaLista;`
4) `UnCatalogo.NombreElemento[2] := UnaLista.Lista[2];`
5) `Catalogo[1].Lista.Lista[1,2] := UnNombre[2];`
6) `Catalogo[2].Lista[1] := UnNombre;`
7)
```
IF Catalogo[20].TipoElemento = Dos THEN
    Catalogo[10].RegistroLista.Sublista[3,9] := 'A'
  END;
```
8) `UnCatalogo := Catalogo[5];`
9) `UnCatalogo.Subcatalogo := UnaLista;`

5. Escribe un programa que sume dos vectores de tipo base INTEGER e imprima el resultado en pantalla. Se consideran datos de entrada la dimensión de los vectores y las componentes de cada uno. Representa cada vector como un registro de dos componentes (el vector propiamente dicho y la dimensión utilizada). Declare en el programa tres subprogramas: lectura de un vector desde teclado, suma de dos vectores y escritura de un vector por pantalla. El programa debe funcionar para vectores de como máximo 10 componentes.

6. Escribe un programa que permita leer dos matrices de números reales del teclado, calcular

la matriz producto y escribirla por pantalla. El programa debe ser capaz de operar con matrices de tamaño menor o igual a 30x30. El tipo que represente la matriz debe ser un registro con tres componentes: la matriz, el número de filas y el número de columnas útiles. Escribe subprogramas para cada una de las tareas.

Nota: Detectar posibles errores en las dimensiones de las matrices.

7. Escribir un subprograma que tome como parámetro una matriz cuadrada y devuelva TRUE si es simétrica o FALSE en otro caso. Una matriz A es simétrica si $\forall i, j \quad A_{ij} = A_{ji}$

Nota: Detectar posibles errores en la dimensión de las matriz.

8. Escribe un programa que lea un texto del teclado y escriba la frecuencia de aparición de cada vocal en él (cuidado con las vocales acentuadas).

9. Escribe un programa que lea una palabra del teclado y determine si dicha palabra es un identificador válido en Modula-2.

10. Se desea efectuar operaciones de suma de cantidades enteras que se caracterizan por tener una gran cantidad de dígitos, de manera que queda excluido el uso del tipo REAL o LONGREAL. Para ello cada número se representará como una cadena de caracteres. Escriba un subprograma que realice la operación anterior.

Nota: Se considera que no se va a trabajar con números de más de 100 dígitos.

11. Escribe un programa que simule el comportamiento de una calculadora simple para aritmética fraccional expresando el resultado en forma fraccional y simplificado al máximo el resultado. Para ello se definirá el tipo `RACIONAL` como un registro de dos componentes enteras (`Numerador` y `Denominador`), y subprogramas para leer del teclado, escribir por pantalla, simplificar, sumar, restar, multiplicar, dividir y comparar números racionales.

Nota: El subprograma que simplifica un número racional debe dividir el numerador y el denominador por el máximo común divisor de ambos, y en caso de que el número racional sea negativo dejar este signo en el numerador.

12. Escribir un programa que lea dos puntos bidimensionales representados como registros y calcule la longitud del segmento que los une y la pendiente de la recta que pasa por ellos.

13. Supongamos que deseamos evaluar a un determinado número de alumnos siguiendo el criterio de que aprobará aquel que supere la nota media de la clase.

Escriba un programa que lea un número indeterminado de alumnos (como máximo 20), y las notas de tres evaluaciones, y como resultado emita un informe indicando para cada alumno las evaluaciones que tiene aprobadas y suspensas.

Ejemplo de la salida que se debe obtener.

Alumno	**Nota-1**	**Nota-2**	**Nota-3**
Juan López	Aprobado	Suspenso	Aprobado
Luis García	Suspenso	Aprobado	Aprobado
Pedro Ruiz	Aprobado	Aprobado	Aprobado

14. Definir un tipo registro variante (`FIGURA`) de modo que una variable de dicho tipo pueda almacenar un círculo, o un triángulo rectángulo o un rectángulo o bien un cuadrado. Según sea la figura se desean almacenar los siguientes datos:

- Si es un círculo, el radio.
- Si es un triángulo, la base y la altura.

- Si es un rectángulo, la base y la altura.
- Si es un cuadrado, el lado.

Escribir funciones que tomen como parámetro una `FIGURA` y calculen el área y perímetro de éstas.

15. El algoritmo de *La Criba de Eratóstenes* puede ser usado para calcular los números primos menores o iguales a un N dado:

1) Crear un conjunto llamado `Criba` que contenga todos los números naturales en el intervalo [2,N].
2) Crear un conjunto vacío y llamarlo `Primos`.
3) Repetir los siguientes pasos
    4) Tomar el primer número que quede en `Criba`.
    5) Este número será primo, por lo que se añade al conjunto `Primos`.
    6) Quitar este número y todos sus múltiplos del conjunto `Criba`.
7) Hasta que `Criba` quede vacía.
8) El conjunto `Primos` contiene todos los primos en el intervalo [2,N].

Utilizar el algoritmo anterior para escribir un programa que calcule y muestre los primos menores o iguales a 100.

16. Escribe un programa que se comporte como una calculadora que pida repetidamente un operador de conjuntos y dos operandos que sean conjuntos de letras minúsculas y que escriba el resultado de la operación. Las operaciones se expresan como caracteres, siendo válidas las siguientes:

+	Unión de conjuntos
-	Diferencia de conjuntos
*	Intersección de conjuntos

El proceso se repetirá hasta que se introduzca como código de operación el carácter ‘&’. Los operadores y el resultado se expresan como cadenas de caracteres.

Ejemplo:

Operación = *

Operando1 = azufre

Operando2 = zafio

Resultado = afz

Operación = -

Operando1 = abracadabra

Operando2 = abaco

Resultado = rd

Operación = &

FIN

17. Disponemos de un vector de 20 números enteros. Diseñar un programa que mantenga en dicho vector los números pares separados de los impares, de forma que cada uno de los subconjuntos estén ordenados entre sí de forma ascendente. Diseña otro subprograma que

ordene un vector como el anterior mediante intercalación de los dos subconjuntos.

18. Se leen dos listas de números enteros *A* y *B* de 100 y 60 elementos respectivamente. Se desea resolver mediante subprogramas las siguientes tareas:

1) Ordenar cada una de las listas *A* y *B*.
2) Crear una lista *C* que contenga la mezcla de las listas ordenadas *A* y *B* y visualizarla.
3) Localizar si el número 255 está en la lista *C*.

Notas:

- Se emplearán procedimientos y funciones siempre que sea conveniente.
- No se permite el uso de efectos laterales.
- Documentar los programas

Relación de problemas complementarios (Tema 8)

19. Se desea almacenar polinomios de coeficientes enteros y una variable real con grado máximo 25 utilizando arrays. Define el tipo adecuado y escribe subprogramas que:

1) Dado un punto y un polinomio lo evalúe en dicho punto.
2) Dados dos polinomios obtenga el polinomio suma.
3) Dado un polinomio obtenga el polinomio derivada.
4) Escriba un polinomio por pantalla.

20. La tabla:

x	3.0	3.3	3.6	4.0	4.5
$f(x)$	14.9	18.2	22.3	27.2	33.3

muestra los valores de una función f para ciertos valores de x. Se desea usar estos valores para estimar los valores de f para otros valores de x distintos, dentro del intervalo dado por la tabla. Escribe un programa que almacene la tabla anterior y que para cada valor de x escriba la correspondiente aproximación de $f(x)$.

Nota: Esta aproximación puede ser obtenida mediante el método de *interpolación lineal*. Con este método, se traza una línea entre los dos puntos de la tabla que rodean al valor de x a evaluar. Es decir si los puntos de la tabla posterior y anterior son x_a y x_b, el valor de $f(x)$ viene dado por:

$$f(x) = f(x_b) + \frac{f(x_a) - f(x_b)}{x_a - x_b} \cdot (x - x_b)$$

Supón que quieres evaluar $f(3.5)$, en ese caso los valores de x_a y x_b son x_a=3.6 y x_b=3.3.

21. Dados N valores de x y $f(x)$ la línea recta que más se acerca a estos valores (según el método de los *mínimos cuadrados*) es:

$y = a + b\,x$, donde

$$b = \frac{N \cdot \sum_{i=1}^{N} x_i \cdot f(x_i) - \sum_{i=1}^{N} x_i \cdot \sum_{i=1}^{N} f(x_i)}{N \cdot \sum_{i=1}^{N} x_i^2 - \left(\sum_{i=1}^{N} x_i\right)^2}$$

$$a = \frac{\sum_{i=1}^{N} f(x_i) - b \cdot \sum_{i=1}^{N} x_i}{N}$$

Escribe un programa que lea el número total de puntos (N) y los N pares de (x, $f(x_i)$) y calcule los valores de a y b para dichos datos. El programa debe ser válido para 50 o menos datos.

22. Escribir un programa que lea 20 números naturales, encuentre el valor máximo y lo imprima, junto con el número de veces que aparece, y las posiciones en que esto ocurre. Este proceso se debe repetir con el resto de los números hasta que no quede ningún número por tratar.

23. Escribir un programa que lea 42 números enteros en un array de dimensión 7 x 6 y realice

las siguientes operaciones:

1) Imprimir el array
2) Encontrar el elemento mayor del array
3) Indicar la posición (fila y columna) del elemento mayor
4) Si el elemento mayor está repetido, indicar cuantas veces y la posición de cada elemento repetido.

24. Escribir un programa que calcule la traspuesta de una matriz cuadrada. Utilizar la representación de los ejercicios **6** y **7**. La matriz traspuesta se obtiene cambiando filas por columnas.

Nota: Detectar posibles errores en la dimensión de la matriz.

25. Escribe un programa que lea un texto del teclado y determine si es palíndromo o no.

Nota: un texto es palíndromo si se lee igual en ambos sentidos (cuidado con los espacios en blanco, mayúsculas y minúsculas y vocales acentuadas).

"*Salta Lenin el atlas*"

"*Isaac no ronca así*"

26. Escribir un programa que lea un texto del teclado, sustituya todas las secuencias de dos o más espacios en blanco por un único espacio en blanco y escriba el texto resultante por pantalla.

27. Escribir un programa que lea un texto de teclado e imprima en pantalla un mensaje indicando cuantas veces aparecen dos letras contiguas e iguales en el mismo.

28. Se desea efectuar operaciones de resta, multiplicación y división entera y resto de cantidades enteras que se caracterizan por tener una gran cantidad de dígitos, de manera que queda excluido el uso del tipo REAL o LONGREAL. Para ello cada número se representará como una cadena de caracteres. Escriba un subprograma que realice cada una de las operaciones anteriores.

Nota: Se considera que no se va a trabajar con números de más de 100 dígitos.

Como aplicación, escriba un programa que calcule el factorial de 50.

29. Escribir un programa similar al del problema **11**, pero que opere con números complejos. Define el tipo complejo como un registro formado por dos números reales que representen la parte real e imaginaria del número. Escribe subprogramas para sumar, restar, multiplicar, dividir, calcular conjugados y comparar números complejos.

30. Escribe una función que tome como parámetro un registro que representa una fecha y devuelva TRUE si esta fecha es válida o FALSE en otro caso.

31. Un vector de registros contiene la descripción de una serie de no más de 100 personas. Cada registro contiene los campos: nombre, edad, sexo y altura. Escribir un programa que lea y almacene datos en un vector de este tipo, ordene el vector por la altura y lo muestre por pantalla.

32. Definir un tipo registro variante (`NUMERO`) de modo que una variable de dicho tipo pueda almacenar o un valor entero o un valor real. Escribir un subprograma que tome un `NUMERO` y lo escriba por pantalla. Escribir otro subprograma que permita sumar dos variables de tipo `NUMERO`. Si al menos uno de los argumentos es un número real el resultado será real, aunque si el resultado obtenido tiene parte fraccional 0 el resultado deberá ser un entero.

33. Dada la siguiente definición de tipo:

```
TYPE
```

```
CONJUNTONUMEROS =SET OF [MINIMO..MAXIMO];
```

donde `MINIMO` y `MAXIMO` son dos constantes naturales, escriba un programa que incluya un procedimiento para imprimir una variable del tipo `NUMEROCONJ`.

Ejemplo: El conjunto cuyos miembros son 3, 7, 11, 19 deberá imprimirse de la siguiente forma:

{3, 7, 11, 19}

34. Dada la siguiente definición de tipo conjunto:

```
CIFRAS   = [0..9];
CONJUNTO =SET OF CIFRAS;
```

escribe funciones que calculen:

1) La cardinalidad de un `CONJUNTO`.
2) Distancia entre dos conjuntos (menor diferencia entre sus elementos):

$$Distancia(A,B) = min\{|a-b| \mid a \in A, b \in B\}$$

35. Escribe un programa que lea un párrafo de no más de 80 letras y que tenga como salida:

1) El número de vocales que contiene.
2) El número de minúsculas, mayúsculas y dígitos.
3) Frecuencia de cada una de las letras del alfabeto.

36. Los alumnos de una clase desean celebrar una cena de confraternización un día del presente mes en el que puedan asistir todos. Se pide realizar un programa que recoja para alumno los días que le vendrían bien para ir a la cena e imprima las fechas concordantes para todos los alumnos. Los datos se introducirán por teclado

37. En una entidad bancaria el número de cuenta de un cliente se va a formar añadiendo (en la posición menos significativa) a una determinada cifra un dígito de autoverificación. Dicho dígito de autoverificación se calculará de la siguiente forma:

1) Multiplicar la posiciones impares del número por dos.
2) Sumar los dígitos no multiplicados y los resultados de los productos obtenidos en el apartado1.
3) Restar el número obtenido en el apartado 2 del número más próximo y superior a éste, que termine en cero.

El resultado será el dígito de autoverificación.

Codificar un programa que vaya aceptando números de cuenta y compruebe mediante el dígito de autoverificación si el número introducido es correcto o no. El proceso se repetirá hasta que se introduzca un cero como número de cuenta