GAP

Justo Peralta López
Dpto. Álgebra y Análisis Matemático

Resumen: *En este documento puedes encontrar una descripción de los primeros pasos en el lenguaje funcional GAP. Como podrás constatar, la sistaxis es muy parecida a otros lenguajes matemáticos como Mathematica o Maple.*

1.1. Empezar y finalizar una sesión

Si GAP ha sido correctamente instalado, solo tenemos que escribir **GAP** en el prompt de nuestro sistema operativo seguido de la tecla **Return** o **New Line**

c:\ > gap

En este momento debe aparecer en nuestro monitor el siguiente prompt

gap>

Para finalizar una sesión con GAP, simplemente tenemos que teclear **quit;**

gap> quit;

1.2. Primeros pasos

Toda expresión en GAP debe terminar en **;**, de no ser así, GAP considera que no está completada y no la evalúa.

```
gap> (9 - 7) * (5 + 6)
  > ;
  22
  gap>
```

Como se observa en el ejemplo, si no se termina una expresión con **;**, GAP espera hasta que se introduzca para evaluar, independientemente de que pulsemos **Retorno de Carro** o **Return**.

1.3. Errores sintácticos

Si cometemos un error sintáctico al introducir una expresión, GAP es incapaz de evaluarla y devolver el resultado deseado, entonces GAP devuelve un mensaje de error y la expresión debe ser modificada.

```
 gap> (9 - 7) * (5 + 6;
Syntax error: ) expected
(9 - 7) * (5 + 6;
                ^
```

Para recuperar la última expresión y modificarla podemos utilizar los cursores de la misma forma que en el **DOS**.

Algunas veces, un error sintáctico puede llevar a GAP a un ciclo roto (literalmente traducido). Esto es indicado por el prompt *brk*. Para salir de esta situación tendremos que escribir **Return;** o pulsar *Control-D*. Solo entonces volveremos al estado inicial indicado por el prompt habitual.

1.4. Constantes y operadores

En una expresión del la forma $(9-7)*(5+6)$ las constantes son $5, 6, 7$ y 9, los operadores son $+, -$ y $*$. Hay tres clases de operadores en **GAP**, operadores aritméticos, operadores de comparación y operadores lógicos. Un ejemplo de operador aplicable a números enteros es la división

```
gap> 12345/25;
  2469/5
```

Fíjese que el numerador y denominador son divididos por m.c.d. Además de este operador podemos utilizar:

Números negativos

```
gap> -3; 17 - 23;
  -3
  -6
```

Potencias

```
gap> 3^132;
  955004950796825236893190701774414011919935138974343129836853841
```

Operaciones módulo un entero

```
gap> 17 mod 3;
  2
```

La jerarquía de los operadores es la usual en cualquier lenguaje de programación, y solo puede ser modificada usando paéntesis

```
gap> (9 - 7) * 5 = 9 - 7  * 5;
   false
```

Los operadores de comparación son $=, <>, <, <=, >$ y $>=$ y se corresponden con igual, distinto, menor, menor o igual, mayor y mayor o igual respectivamente.

```
gap> 10^5 < 10^4;
   false
```

Cualquier par de objetos en GAP se pueden comparar y el resultado de cada comparación es *true o false*. Estos valores booleanos se pueden manipular utilizando operadores lógicos unitarios como $\sim$, o operadores lógicos binarios como **and** u **or**.

```
gap> not true; true and false; true or false;
   false
   false
   true
   gap> 10 > 0 and 10 < 100;
   true
```

Además de estos operadores booleanos, existen otros operadores específicos para cada tipo de objeto en GAP, ej: **in** para conjuntos. Otro tipo de constantes en GAP son:

Permutaciones

```
gap> (1,2,3);
   (1,2,3)
   gap> (1,2,3) * (1,2);
   (2,3)

gap> (1,2,3)^-1;
   (1,3,2)
   gap> 2^(1,2,3);
   3
   gap> (1,2,3)^(1,2);
   (1,3,2)
```

y caracteres

```
  gap> á';
    á'
    gap> '*';
    '*'
```

1.5. Variables y asignaciones

Una variable es simplemente un camino para referirnos a un objeto mediante un nombre o identificador. El operador de asignación es **:=** (sin espacios entre : y =).

```
gap> a:= (9 - 7) * (5 + 6);
 22
 gap> a;
 22
 gap> a * (a + 1);
 506
 gap> a:= 10;
 10
 gap> a * (a + 1);
 110
```

Después de una asignación el valor de la variable es devuelta en la siguiente linea. Si queremos que esto no suceda, sobretodo para valores muy grandes, colocaremos al final de la asignación **;**.

Un identificador puede estar formado por una secuencia de letras y dígitos. Por ejemplo *abc* y *a0bc1*. También lo es *123a*, pero no *1234* ya que puede ser confundido con un número.

En GAP ya existen algunas variables definidas, como por ejemplo los nombres de la funciones. Para no entrar en conflicto con algunas de las variables predefinidas por GAP, tendremos en cuenta que los nombres de las funciones en GAP empiezan con una letra mayúscula.

Si evaluamos la siguiente expresión

```
gap> (9 - 7) * (5 + 6);
  22
```

y olvidamos guardar el resultado en una variable, podemos hacerlo sin tener que reescribir la expresión evaluada

```
gap> a:= last;
  22
```

Es más, existen variables del tipo **last1, last2,...**.

Ejemplos de palabras reservadas o predefinidas en GAP

```
  and       do        elif      else      end       fi
  for       function  if        in        local     mod
  not       od        or        repeat    return    then
  until     while     quit
```

Ejemplos de identificadores o variables

```
a                   foo                 aLongIdentifier
hello               Hello               HELLO
x100                100x                _100
some_people_prefer_underscores_to_separate_words
WePreferMixedCaseToSeparateWords
```

1.6. Funciones

Un programa escrito en GAP es una función y son un tipo especial de objetos. Como ocurre con funciones matemáticas, son aplicadas a objetos y devuelven un nuevo objeto que dependerá de la entrada. La función factorial, por ejemplo, devuelve el factorial del entero al cual la aplicamos.

```
gap> Factorial(17);
 355687428096000
```

Los argumentos de una función deben ir entre paréntesis y separadas por comas.

```
gap> Gcd(1234, 5678);
 2
```

Hay funciones que no devuelven nada, solo producen algún efecto deseado.

```
gap> Print(1234, "\n");
 123
```

Una función se puede definir de la misma forma que una aplicación en matemáticas.

```
gap> cubed:= x -> x^3;
 function ( x ) ... end
```

y ahora la podemos aplicar

```
 gap> cubed(5);
 125
```

1.7. Listas y registros

Una lista es una colección de objetos separados por comas y encerrado entre corchetes. Ejemplo:Lista de primos

```
 gap> primes:= [2, 3, 5, 7, 11, 13, 17, 19, 23, 29];
 [ 2, 3, 5, 7, 11, 13, 17, 19, 23, 29 ]
```

Operaciones sobre listas

Append: Añade una lista de elementos al final de la lista.

```
 gap> Append(primes, [31, 37]);
    gap> primes;
    [ 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37 ]
```

Add: Añade un elemento al final de la lista.

```
gap> Add(primes, 41);
    gap> primes;
    [ 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41 ]
```

Para obtener un elemento de la lista solo hay que indicar su posición entre corchetes

```
gap> primes[7];
    17
```

Otras operaciones con listas usando ejemplos:

```
gap> Length(primes);
    13
    gap> primes[14]:= 43;
    43
    gap> primes;
    [ 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43 ]

gap> primes[20]:= 71;
    71
    gap> primes;
    [ 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43,,,,,, 71 ]
    gap> Length(primes);
    20
```

Cuando se hace referencia a una posición que no existe GAP devuelve un mensaje de error

```
gap> lll[1]:= 2;
    Error, Variable: 'lll' must have a value
    gap> lll:= [];
    [  ]
    gap> lll[1]:= 2;
    2
```

Podemos calcular la posición de un elemento en la lista usando **Position**

```
gap> Position(primes, 17);
   7
   gap> Position(primes, 20);
   false
```

No es necesario que todos los elementos de la lista sean del mismo tipo.

```
gap> lll:= [true, "This is a String",,, 3];
   [ true, "This is a String",,, 3 ]

 gap> lll[3]:= [4,5,6];; lll;
   [ true, "This is a String", [ 4, 5, 6 ],, 3 ]
   gap> lll[4]:= lll;
   [ true, "This is a String", [ 4, 5, 6 ], ~, 3 ]
```

Una cadena es un tipo especial de listas formado por caracteres y sin elementos nulos.

```
gap> s1 := ['H',á','l','l',ó',' ','w',ó','r','l','d','.'];
"Hallo world."
gap> s1 = "Hallo world.";
true
gap> s1[7];
'w'
```

Una sublista de una lista dada se puede obtener de forma sencilla como sigue

```
gap> sl := lll{ [ 1, 2, 3 ] };
[ true, "This is a String", [ 4, 5, 6 ] ]
gap> sl{ [ 2, 3 ] } := [ "New String", false ];
[ "New String", false ]
gap> sl;
[ true, "New String", false ]
```

Dos lista son iguales si son identicas componente a componente en el mismo orden.

```
gap> numbers := primes;; numbers = primes;
true
```

Ahora cambiamos el cuarto elemento de numbers

```
gap> numbers[3]:= 4;; numbers = primes;
true
```

Siguen siendo iguales ya que numbers y primes apuntan al mismo objeto, es decir, un mismo objeto puede tener varios nombres. Si queremos hacer una copia de la lista de primos de forma que sean independiente debemos usar **ShallCopy**

```
gap> primes[3]:= 5;; primes;
[ 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43,,,,,, 71 ]
gap> numbers:= ShallowCopy(primes);; numbers = primes;
true
gap> numbers[3]:= 4;; numbers = primes;
false
```

Los siguientes ejemplos crean listas de forma automática.

```
List([1,2,3]];
List((1,2)(1,2,3,4));
List([1,2,3],x->x*x));
```

1.7.1. Conjuntos

Los elementos de un conjuntos se definen como una lista ordenada

```
gap> fruits:= ["apple", "strawberry", "cherry", "plum"];
[ "apple", "strawberry", "cherry", "plum" ]
gap> IsSSortedList(fruits);
false
gap> fruits:= Set(fruits);
[ "apple", "cherry", "plum", "strawberry" ]
```

Para determinar si un elemento pertenece a una lista podemos utilizar el operador **in**

```
gap> "apple" in fruits;
true
gap> "banana" in fruits;
false
```

A un conjunto se le puede añadir elementos de forma muy similar a una lista

```
gap> AddSet(fruits, "banana");
gap> fruits;  #  The banana is inserted in the right place.
[ "apple", "banana", "cherry", "plum", "strawberry" ]
gap> AddSet(fruits, "apple");
gap> fruits;  #  fruits has not changed.
[ "apple", "banana", "cherry", "plum", "strawberry" ]
```

1.7.2. Rangos

Un rango es una lista de enteros formados por una progresión aritmética.

```
gap> [1..999999];     #  a range of almost a million numbers
[ 1 .. 999999 ]
gap> [1, 2..999999];  #  this is equivalent
[ 1 .. 999999 ]
gap> [1, 3..999999];  #  here the step is 2
[ 1, 3 .. 999999 ]
gap> Length( last );
500000
gap> [ 999999, 999997 .. 1 ];
[ 999999, 999997 .. 1 ]
```

1.7.3. ciclos y rangos

Ciclos for y while

```
gap> pp:= [ (1,3,2,6,8)(4,5,9), (1,6)(2,7,8), (1,5,7)(2,3,8,6),
>           (1,8,9)(2,3,5,6,4), (1,9,8,6,3,4,7,2)];;
gap> prod:= ();
()
gap> for p in pp do
>       prod:= prod*p;
>    od;
gap> prod;
(1,8,4,2,3,6,5,9)
```

Un ciclo for posee la siguiente sintaxis
for var **in** list **do** sentencias **od;**

```
gap> ff:= 1;
1
gap> for i in [1..15] do
>       ff:= ff * i;
>    od;
gap> ff;
1307674368000
```

Un ciclo **while** posee la siguiente sintaxis.
while *condición* **do** sentencias **od;**

```
gap> n:= 1333;;
gap> factors:= [];;
```

```
gap> for p in primes do
>       while n mod p = 0 do
>          n:= n/p;
>          Add(factors, p);
>       od;
>    od;
gap> factors;
[ 31, 43 ]
gap> n;
1
```

El siguiente ejemplo nos permite calcular los primos menores que 1000.

```
gap> primes:= [];;
gap> numbers:= [2..1000];;
gap> for p in numbers do
>       Add(primes, p);
>       for n in numbers do
>          if n mod p = 0 then
>             Unbind(numbers[n-1]);
>          fi;
>       od;
>    od;
```

La función **Unbind** nos permite borrar la entrada de una lista.

1.7.4. Operaciones con listas

Para calcular el producto de una lista de número o permutaciones

```
 gap> Product([1..15]);
1307674368000
gap> Product(pp);
(1,8,4,2,3,6,5,9)
```

Veamos como aplicar una función a todos los elementos de una lista.

```
 gap> cubed:= x -> x^3;;
gap> List([2..10], cubed);
[ 8, 27, 64, 125, 216, 343, 512, 729, 1000 ]
```

Si queremos sumar todos los cubos anteriores.

```
 gap> Sum(last) = Sum([2..10], cubed);
true
```

Tambiém podemos pasar un filtro a una lista, eliminando aquellos elementos que no verifican cierta condición.

```
gap> Filtered(primes, x-> x < 30);
[ 2, 3, 5, 7, 11, 13, 17, 19, 23, 29 ]
```

Esto último es equivalente a quedarnos con una sublista de la lista inicial.

```
gap> primes{ [1 .. 10] };
[ 2, 3, 5, 7, 11, 13, 17, 19, 23, 29 ]
```

Terminaremos mostrando si todos los elementos de una lista verifican una condición dada.

```
gap> list:= [ 1, 2, 3, 4 ];;
gap> ForAll( list, x -> x > 0 );
true
gap> ForAll( list, x -> x in primes );
false
```

1.7.5. Vectores y matrices

Un vector es una lista de elementos que pertenecen al mismo cuerpo y una matriz es una lista de vectores de la misma longitud.

```
gap> v:= [3, 6, 2, 5/2];
   [ 3, 6, 2, 5/2 ]
   gap> IsVector(v);
   true
```

Producto de un vector por un escalar y producto escalar de vectores

```
gap> 2 * v;
   [ 6, 12, 4, 5 ]
   gap> v * 1/3;
   [ 1, 2, 2/3, 5/6 ]
   gap> v * v;
   221/4  # the scalar product of 'v' with itself
```

Una matriz es una lista de vectores de igual longitud

```
gap> m:= [[1, -1, 1],
   >         [2, 0, -1],
   >         [1, 1, 1]];
   [ [ 1, -1, 1 ], [ 2, 0, -1 ], [ 1, 1, 1 ] ]
   gap> m[2][1];
   2
```

Sintácticamente una matriz es una lista de lista. Así que para hacer referencia al elemento 2 de la segunda fila y primera columna tendremos que escribir **m[2][1]**

```
gap> m:= [[1, 2, 3, 4],
   >         [5, 6, 7, 8],
   >         [9,10,11,12]];
   [ [ 1, 2, 3, 4 ], [ 5, 6, 7, 8 ], [ 9, 10, 11, 12 ] ]
   gap> PrintArray(m);
   [ [   1,   2,   3,   4 ],
     [   5,   6,   7,   8 ],
     [   9,  10,  11,  12 ] ]
   gap> [1, 0, 0, 0] * m;
   Error, Vector *: vectors must have the same length
   gap> [1, 0, 0] * m;
   [ 1, 2, 3, 4 ]
   gap> m * [1, 0, 0];
   Error, Vector *: vectors must have the same length
   gap> m * [1, 0, 0, 0];
   [ 1, 5, 9 ]
   gap> m * [0, 1, 0, 0];
   [ 2, 6, 10 ]
```

Submatrices se pueden extraer y cambiar de valor fácilmente utilizando llaves

```
gap> sm := m{ [ 1, 2 ] }{ [ 3, 4 ] };
    [ [ 3, 4 ], [ 7, 8 ] ]
    gap> sm{ [ 1, 2 ] }{ [2] } := [[1],[-1]];
    [ [ 1 ], [ -1 ] ]
    gap> sm;
    [ [ 3, 1 ], [ 7, -1 ] ]
```

1.7.6. Registros

Podemos observar como se define un registro en el siguiente ejemplo.

```
 gap> date:= rec(year:= 1997,
>              month:= "Jul",
>              day:= 14);
rec( year := 1997, month := "Jul", day := 14 )
```

Para hacer referencia a un campo de un registro hacemos

```
gap> date.year;
1997
```

Añadir nuevos campos a un registro se realiza de forma dinámica.

```
 gap> date.time:= rec(hour:= 19, minute:= 23, second:= 12);
rec( hour := 19, minute := 23, second := 12 )
gap> date;
rec( year := 1997, month := "Jul", day := 14,
  time := rec( hour := 19, minute := 23, second := 12 ) )
```

Para visualizar un registro hacemos

```
gap> Display( date );
rec(
  year := 1997,
  month := "Jul",
  day := 14,
  time := rec(
      hour := 19,
      minute := 23,
      second := 12 ) )
```

Si queremos saber que campos tiene un registro basta con escribir

```
 gap> RecNames(date);
[ "year", "month", "day", "time" ]
```

1.8. Símbolos

Los operadores y delimitadores son:

```
+     -     *     /     ^     ~     !.
=     <>    <     <=    >     >=    ![
:=    .     ..    ->    ,     ;     !{
[     ]     {     }     (     )     :
```

El símbolo # es usado para comentarios. De esta forma, la linea de código

```
if i<0 then a:=-i;else a:=i;fi;
```

es equivalente a las lineas

```
if i < 0 then    # if i is negative
  a := -i;       #   take its additive inverse
else             # otherwise
  a := i;        #   take itself
fi;
```

1.9. Palabras reservadas

Las siguientes palabras son palabras reservadas de GAP que pueden formar parte de una sentencia (además de las funciones propias de GAP).

```
and      do       elif   else    end     fi
for      function if     in      local   mod
not      od       or     repeat  return  then
until    while    quit   QUIT    break   rec
continue
```

1.10. Identificadores

Un nombre de variable está formado por letras, números y "_", y debe contener como mínimo una letra o un "_". Ejemplos de identificadores o variables son:

```
a            foo          aLongIdentifier
hello        Hello        HELLO
x100         100x        _100
some_people_prefer_underscores_to_separate_words
WePreferMixedCaseToSeparateWords
```

1.11. Variables locales y globales

Como ya sabemos, dependiendo del uso o validez de una variable, podemos hablar de variables globales y locales. En el siguiente ejemplo se definen estos dos tipos de variables.

```
g := 0;       # global variable g
x := function ( a, b, c )
  local  y;
  g := c;     # c refers to argument c of function x
  y := function ( y )
    local d, e, f;
    d := y;   # y refers to argument y of function y
```

```
    e := b;    # b refers to argument b of function x
    f := g;    # g refers to global variable g
    return d + e + f;
  end;
  return y( a ); # y refers to local y of function x
end;
```

1.12. Comparaciones

Igual	Distinto	Menor	Mayor	Menor o Igual	Mayor o Igual
=	<>	<	>	<=	>=

```
gap> 2 * 2 + 9 = Fibonacci(7);  # a comparison where the left
true                            # operand is an expression
```

1.13. Sentencias

Asignaciones, llamadas a funciones, estructuras if, while, repeat y for son sentencias. En la evaluación de una sentencia siempre se producen dos pasos: primero una compilación donde se mira si la sentencia es correcta, y una ejecución.

```
gap> if i <> 0 then k = 16/i; fi;
Syntax error: := expected
if i <> 0 then k = 16/i; fi;
                 ^
```

1.14. Asignaciones

Una asignación es de la forma *var:=expr;*, donde *var* es una variables y *expr* una expresión cualquiera. Veamos algunos ejemplos.

```
gap> data:= rec( numbers:= [ 1, 2, 3 ] );
rec( numbers := [ 1, 2, 3 ] )
gap> data.string:= "string";; data;
rec( numbers := [ 1, 2, 3 ], string := "string" )
gap> data.numbers[2]:= 4;; data;
rec( numbers := [ 1, 4, 3 ], string := "string" )
```

1.15. If

Una estructura condicional del tipo if es de la forma
if bool-expr1 then statements1 { elif bool-expr2 then statements2 } [else statements3] fi;
Veamos algunos ejemplos.

```
gap> i := 10;;
gap> if 0 < i then
>    s := 1;
>  elif i < 0 then
>    s := -1;
>  else
>    s := 0;
>  fi;
gap> s;
1    # the sign of i
```

1.16. While

Una estructura repetitiva del tipo *while* es de la forma: *while bool-expr do statements od;*

```
gap> i := 0;; s := 0;;
gap> while s <= 200 do
>    i := i + 1; s := s + i^2;
>  od;
gap> s;
204    # sum of the first i squares larger than 200
```

1.17. Repeat

Una estructura repetitiva del tipo *repeat* es de la forma:
repeat statements until bool-expr;

```
gap> i := 0;; s := 0;;
gap> repeat
>    i := i + 1; s := s + i^2;
>  until s > 200;
gap> s;
204    # sum of the first i squares larger than 200
```

1.18. For

Una estructura repetitiva del tipo *for* es de la forma
for simple-var in list-expr do statements od;

```
gap> s := 0;;
gap> for i in [1..100] do
>    s := s + i;
> od;
gap> s;
5050
```

```
gap> l := [ 1, 2, 3, 4, 5, 6 ];;
gap> for i in l do
>    Print( i, " " );
>    if i mod 2 = 0 then Add( l, 3 * i / 2 ); fi;
> od; Print( "\n" );
1 2 3 4 5 6 3 6 9 9
gap> l;
[ 1, 2, 3, 4, 5, 6, 3, 6, 9, 9 ]
```

```
gap> l := [ 1, 2, 3, 4, 5, 6 ];;
gap> for i in l do
>    Print( i, " " );
>    l := [];
> od; Print( "\n" );
1 2 3 4 5 6
gap> l;
[  ]
```

1.19. Break

La sentencia *break* produce la interrupción del ciclo desde el cual es llamado

```
gap> g := Group((1,2,3,4,5),(1,2)(3,4)(5,6));
Group([ (1,2,3,4,5), (1,2)(3,4)(5,6) ])
gap> for x in g do
> if Order(x) = 3 then
> break;
> fi; od;
gap> x;
(1,4,3)(2,6,5)
```

1.20. Funciones

Veamos a partir de ejemplos como se define una función usando GAP. La siguiente función es la función de Fibonacci.

```
gap> fib := function ( n )
>     local f1, f2, f3, i;
>     f1 := 1; f2 := 1;
>     for i in [3..n] do
>       f3 := f1 + f2;
>       f1 := f2;
>       f2 := f3;
>     od;
>     return f2;
>   end;;
gap> List( [1..10], fib );
[ 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 ]
```

La misma función de forma recursiva.

```
gap> fib := function ( n )
>     if n < 3 then
>       return 1;
>     else
>       return fib(n-1) + fib(n-2);
>     fi;
>   end;;
gap> List( [1..10], fib );
[ 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 ]
```

También podemos definir una funcíon con un numero de argumentos cualquiera de la forma

```
gap> sum := function ( arg )
>     local total, x;
>     total := 0;
>     for x in arg do
>       total := total + x;
>     od;
>     return total;
>    end;
function( arg ) ... end
gap> sum(1, 2, 3);
6
gap> sum(1, 2, 3, 4);
```

```
10
gap> sum();
0
```

1.21. Ejercicios

1. Calcular todos los números entre 1 y 10000 divisibles por 5;por 11;por 13; por 5 y 13,por 5 y 11, por 5,11 y 13.
2. Definir el código necesario para calcular el producto escalar de 2 vectores u y v, sobre un cuerpo finito F (F:=GF(2);).
3. Definir el código necesario para la unión, intersección y diferencia de conjuntos. No se puede usar ninguna función de conjuntos predefinida en GAP.
4. Calcular la longitud de la lista de número entre 1 y 2000 divisibles por 13.
5. Definir una función llamada rango(i,k,j) que devuelva el rango [i,k..j] o el rango incluido en el anterior más grande posible en el caso de que no pueda ser definido.
6. Definir las funciones necesarias para insertar un número en una lista ordenada.
7. Definir las funciones necesarias para insertar un elemento en la posición deseada de una lista.
8. Definir una función que nos devuelva una lista de elementos de la forma $[x, orden(x)]$ con x perteneciente a un grupo.
9. Dado un código representado por una lista de palabras $C := [[1,1,0,0],[0,0,1,1]]$, definir las funciones que nos devuelva un registro cuyos campos sean los parámetros indicados.
   a) Longitud de palabra código.
   b) Tamaño del código.
   c) Distancia mínima.
   d) Razón del código.
10. Definir una función que dado un código como lista y una palabra, nos devuelva la decodificación de dicha palabra atendiendo al criterio de palabra con mayor probabilidad de haber sido enviada.