GAP es un entorno de cálculo algebraico discreto. Tiene un núcleo implementado en c y dispone aparte de librerías escritas en su propio lenguaje de programación. Este lenguaje es de tipo procedural, por lo que aquellas personas que han programado en pascal, c, maple, ... no tienen problemas en adaptarse. En la página oficial de GAP (www.gap-system.org) se puede encontrar abundante información concerniente a instalación en distintas plataformas, tutoriales, manuales, paquetes. Los paquetes pasan por un proceso de arbitraje similar al de las revistas científicas. Primero son depositados (y accesibles al público en la página oficial) y si se aceptan pasan a ser parte de la distribución. Aún así muchos instaladores incluyen los paquetes depositados. GAP ha sido mantenido a lo largo de su historia por varias universidades, y se nutre de las contribuciones que hacen los usuarios en forma de paquetes para fines más específicos.

1 La línea de comandos

GAP es un lenguaje interpretado. Se pueden hacer operaciones directamente en la línea de comandos o bien se pueden leer ficheros con guiones escritos con un editor de textos. Normalmente se invoca usando el comando gap, aunque dependiendo del sistema operativo pueden haber distintas alternativas. Una vez iniciado el intérprete aparece una línea de comandos con el siguiente aspecto.

```
gap>
```

Para la edición de las entradas que queramos evaluar se pueden utilizar las flechas junto con las teclas de inicio y fin. Además existe la posibilidad de usar combinaciones de teclas similares a las de empleadas en un shell de unix (ctr-[b,f,p,n,a,e]). El tabulador sirve para completar comandos a partir del texto que hayamos introducido en ese momento. Así teclear 'Gcd+TAB' da como resultado:

```
gap> Gcd
    Gcd
    GcdCoeffs
    GcdInt
    GcdOp
    GcdRepresentation
    GcdRepresentationOp
    Gcdex
```

Para salir de la línea de comandos podemos usar el comando quit, o bien pulsar ctr-d.
El final de línea no se indica con el retorno de carro, para ello se utiliza ';'. Esto permite escribir sentencias de varias líneas.

```
gap> 1+
> 2;
3
```

La sesión de trabajo se puede almacenar en un fichero para su posterior edición usando el comando `LogTo`.

```
gap> LogTo("k:/gap4r4/pruebas/prueba");
gap> 1+1;
```

Podemos leer una secuencia de comandos (guión) con la orden `Read`.

```
gap> Read("k:/gap4r4/pruebas/uno.g");
```

Nótese que las barras son de la forma '/', incluso en windows.

A la ayuda se accede usando el signo de cierre de interrogación.

```
gap> ?LogTo
```

Las operaciones de suma y producto dependen de los argumentos que les acompañen. Así pueden representar suma de enteros, o de matrices, o de subespacios vectoriales... El producto puede significar incluso la composición de dos permutaciones, y el elevado ('^ ') la imagen de un punto por una permutación.

```
gap> [1,2,3]+[1,2];
[ 2, 4, 3 ]
gap> (1,2)*(2,5);
(1,5,2)
gap> 1^(1,2,3);
2
```

Los símbolos '=', '<', '>', '<=', '> =' y '<>' sirven para denotar igualdad, ser mayor, menor, menor o igual, mayor o igual y distinto, respectivamente.

```
gap> 2=1+1;
true
gap> [1,2]<[3];
true
gap> [1,2]>[1,3];
false
gap> 1<>1;
false
```

2 Identificadores

Si vamos a utilizar un objeto varias veces, podemos por comodidad asignarle un nombre. Esto se hace con ‘: =’.

```
gap> a:=2^10;
1024
```

Si queremos inhibir la salida, escribimos un punto y coma extra al final de la asignación.

```
gap> b:=a;;
gap>
```

Podemos visualizar las variables definidas hasta el momento de la siguiente forma.

```
gap> NamesUserGVars();
[ "a", "b" ]
```

Las variables en GAP no tienen tipo, las características del objeto al que se refieren se guardan en el propio objeto. De hecho, podemos reutilizar una misma variable para denotar distintos tipos de datos.

```
gap> a:=1;
1
gap> a:=(1,2);
(1,2)
gap> a:=[1,2];
[ 1, 2 ]
```

El identificador `last` se usa para hacer referencia a la última salida (también existen `last2` y `last3`).

```
gap> 1+3;
4
gap> last^3;
64
```

En GAP se ha establecido el convenio de escribir las variables globales y las utilizadas en las librerías empezando con mayúsculas.
Las funciones también son objetos, y se pueden definir de varias formas. Podemos utilizar un procedimiento clásico.

```
gap> f:=function(x)
> return x^2;
> end;
function( x ) ... end
gap> f(3);
9
```

Y también podemos usar notación estilo λ-cálculo.

```
gap> g:=x->x^2;
function( x ) ... end
gap> g(3);
9
```

3 Listas

Las listas en GAP se escriben, como ya hemos visto en algunos ejemplos arriba como una secuencia de elementos delimitada por corchetes. Dichas secuencias no tienen por qué ser homogéneas. Al trabajar con listas tenemos que prestar atención a algunas funciones que son 'destructivas', a saber, modifican alguno de los argumentos que se les pasa. Esto se debe en parte a que cuando uno asigna un identificador a una lista, no lo hace al objeto como tal, si no a su posición en la memoria.

```
gap> a:=[1,"a"];
[ 1, "a" ]
gap> b:=a;
[ 1, "a" ]
gap> b[2]:=3;
3
gap> a;
[ 1, 3 ]
```

Podemos usar el comando `ShallowCopy` para hacer una copia de una lista, creando un objeto nuevo.

```
gap> a:=[1,"a"];
[ 1, "a" ]
gap> b:=ShallowCopy(a);
[ 1, "a" ]
```

```
gap> b[2]:=3;
3
gap> a;
[ 1, "a" ]
```

Hay multitud de formas para definir y modificar listas, ponemos unos ejemplos a continuación.

```
gap> a:=[1,"a"];
[ 1, "a" ]
gap> Append(a,[3,4]);
gap> a;
[ 1, "a", 3, 4 ]
gap> Add(a,5);
gap> a;
[ 1, "a", 3, 4, 5 ]
gap> Concatenation(a,[8,9]);
[ 1, "a", 3, 4, 5, 8, 9 ]
gap> a;
[ 1, "a", 3, 4, 5 ]
gap> Concatenation(a,[8,9]);
[ 1, "a", 3, 4, 5, 8, 9 ]
gap> a;
[ 1, "a", 3, 4, 5 ]
gap> Filtered([1..100],IsPrime);
[ 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97 ]
gap> List(last,x->x^2);
[ 4, 9, 25, 49, 121, 169, 289, 361, 529, 841, 961, 1369, 1681, 1849, 2209, 2809, 3481, 3721,
  4489, 5041, 5329, 6241, 6889, 7921, 9409 ]
```

A veces es conveniente usar conjuntos en vez de listas, pues sus elementos se ordenan (si son comparables entre sí) y la pertenencia de un elemento es más rápida de resolver.

```
gap> a:=[1,2,1,2];;
gap> Set(a);
[ 1, 2 ]
gap> a;
```

```
[ 1, 2, 1, 2 ]
gap>  3 in a;
false
```

Hay ciertos comandos que dan como salida un conjunto.

```
gap> a:=[1,2];;
gap> b:=[2,3];;
gap> Union(a,b);
[ 1, 2, 3 ]
gap> IsSet(last);
true
gap> Intersection(a,b);
[ 2 ]
```

4 Algunos ejemplos de programación

En GAP se pueden utilizar comandos como `for`, `while`, `repeat .. until` para hacer bucles. Veamos cómo definir el factorial de distintas formas, utilizando distintos estilos de implementación.

```
f:=function(x)
    local p, i; #variable local para el producto parcial y contador

    p:=1;
    for i in [1..x] do
        p:=p*i;
    od; #final de bucle

    return p; #salida
end;
gap> f(3);
6
```

De forma recursiva...

```
f:=function(x)

    if x=0 then
        return 1;
    fi;

    return x*f(x-1);
end;
```

Usando funciones específicas de listas...

```
f:=x->Product([1..x]);
```

Calculamos ahora el primer entero perfecto y los enteros perfectos entre 1 y 100000.

```
gap> perfecto:=x->(Sum(DivisorsInt(x))=2*x);
function( x ) ... end
gap> First([1..200], perfecto);
6
gap> Filtered([1..100000],perfecto);
[ 6, 28, 496, 8128 ]
```

Veamos si todos los números entre 100 y 500 son perfectos, o si hay alguno.

```
gap> ForAll([100..500],perfecto);
false
gap> ForAny([100..500],perfecto);
true
```

Hacemos algo parecido con los amigos.

```
amigos:=function(x,y)
    local dx, dy; #divisores de x e y

    if x=y or IsPrime(x) then
        return false;
    fi; #queremos buscar parejas distintas y que no sean primos
```

```
    dx:=List(DivisorsInt(x));
    Remove(dx);; #le quitamos x
    dy:=List(DivisorsInt(y));
    Remove(dy);; #le quitamos y
    return Sum(dx)=Sum(dy);
end;
```

Para ver qué números entre 1 y 50 tienen amigos, escribimos lo siguiente.

```
gap> Filtered([1..50],x->ForAny([1..50],y->amigos(x,y)));
[ 6, 10, 12, 16, 20, 25, 26, 27, 33, 35, 38, 49 ]
```

Y si queremos saber quiénes son sus amigos, tecleamos.

```
gap> List(last,x->[x,First([1..50],y->amigos(x,y))]);
[ [ 6, 25 ], [ 10, 49 ], [ 12, 26 ], [ 16, 33 ], [ 20, 38 ], [ 25, 6 ],
  [ 26, 12 ], [ 27, 35 ], [ 33, 16 ], [ 35, 27 ], [ 38, 20 ], [ 49, 10 ] ]
```

Nótese que nos ahorramos tiempo si usamos el producto cartesiano de dos listas.

```
gap> Filtered(Cartesian([1..150],[1..150]),l->l[1]<l[2] and amigos(l[1],l[2]));
[ [ 6, 25 ], [ 10, 49 ], [ 12, 26 ], [ 16, 33 ], [ 18, 51 ], [ 18, 91 ], [ 20, 38 ], [ 27, 35 ], [ 32, 125 ],
  [ 39, 55 ], [ 40, 94 ], [ 44, 74 ], [ 44, 81 ], [ 45, 87 ], [ 48, 92 ], [ 48, 146 ], [ 51, 91 ], [ 52, 86 ],
  [ 56, 76 ], [ 56, 122 ], [ 57, 85 ], [ 63, 111 ], [ 65, 77 ], [ 69, 133 ], [ 70, 142 ], [ 74, 81 ],
  [ 76, 122 ], [ 80, 104 ], [ 80, 110 ], [ 92, 146 ], [ 93, 145 ], [ 95, 119 ], [ 95, 143 ], [ 104, 110 ],
  [ 119, 143 ] ]
```