1 Enteros

Los enteros son un tipo de dato básico en GAP. En principio los enteros pueden ser arbitrariamente grandes, el único límite es la memoria asignada para el uso de GAP.

1.1 Cociente y resto

Dados dos enteros a y b, el resto de dividir a entre b se puede calcular usando el comando `mod`.

```
gap> -3 mod 5;
2
```

Y el cociente se puede calcular de la siguiente forma.

```
gap> (-3-(-3 mod 5))/5;
-1
```

Si en GAPescribimos $-3/5$, el resultado es un racional, y si usamos el comando `Int`, obtenemos la parte entera de ese racional, que no es precisamente lo que buscamos.

```
gap> Int((-3)/5);
0
```

1.2 Máximo común divisor y mínimo común múltiplo. Coeficientes de Bézout

El máximo común divisor de dos enteros (o más) puede ser calculado con el comando `Gcd`, y su mínimo común múltiplo con el comando `Lcm`.

```
gap> Gcd(3,-5);
1
gap> Lcm(3,-5);
15
```

Si queremos conseguir los coeficientes de Bézout, podemos usar el comando `GcdRepresentation` (entre las muchas posibilidades que da GAPpara esto).

```
gap> GcdRepresentation(3,-5);
[ 2, 1 ]
gap> GcdRepresentation(10,15,18);
[ 7, -7, 2 ]
```

1.3 Ecuaciones diofánticas

Como ya sabemos, una vez resuelto el problema de encontrar los coeficientes de Bézout de un máximo común divisor de dos enteros, tenemos también solución para resolver una ecuación diofántica. Lo único que tenemos que comprobar es si el término independiente es divisible por el máximo común divisor de los coeficientes, y luego multiplicar los coeficientes de Bézout por el factor apropiado para conseguir una solución particular.

Si queremos resolver $10x + 25y = 45$, hacemos lo siguiente.

```
gap> Gcd(10,25);
5
```

Vemos si 45 es divisible por 5.

```
gap> 45 mod 5;
0
```

Calculamos 45 entre 5.

```
gap> 45/last2;
9
```

Multiplicamos el resultado por los coeficientes de Bézout.

```
gap> last*GcdRepresentation(10,25);
[ -18, 9 ]
```

Finalmente, comprobamos el resultado.

```
gap> last*[10,25];
45
```

1.4 Primos

Para factorizar un entero en producto de primos podemos usar el comando `Factors`.

```
gap> Factors(100);
[ 2, 2, 5, 5 ]
```

También podemos saber si un número es primo usando el comando `IsPrime`.

```
gap> IsPrime(10);
false
gap> IsPrime(7);
true
```

`Primes` es una lista que contiene los primos menores que mil.

```
#el tercer primo
gap> Primes[3];
5
#los primeros cuarenta primos
gap> Primes{[1..40]};
[ 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, 101, 103, 107, 109, 113, 127, 131, 137, 139, 149, 151,
  157, 163, 167, 173 ]
```

También podemos hacer un filtro de una lista para ver qué elementos en ella son primos.

```
gap> Filtered([1..300],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, 101, 103, 107, 109, 113, 127, 131, 137, 139, 149, 151,
  157, 163, 167, 173, 179, 181, 191, 193, 197, 199, 211, 223, 227, 229, 233,
  239, 241, 251, 257, 263, 269, 271, 277, 281, 283, 293 ]
```

1.5 Congruencias

Como sabemos, gracias a los coeficientes de Bézout también podemos resolver cualquier congruencia del tipo $ax \equiv b \bmod m$, con a, b y m enteros ($m \neq 0$).

Sea la congruencia $60x \equiv 90 \bmod 105$.

```
gap> Gcd(60,105);
15
```

Dividimos todo por 15.

```
gap> [60,90,105]/15;
[ 4, 6, 7 ]
```

Buscamos ahora el inverso de 4 módulo 7 (me quedo con el coeficiente de Bézout de 4).

```
gap> GcdRepresentation(4,7)[1];
2
```

La solución particular es por tanto.

```
gap> 2*6 mod 7;
5
```

Si lo que queremos es resolver un sistema de congruencias de la forma

$$\begin{aligned} x &\equiv a_1 \bmod m_1, \\ &\dots \\ x &\equiv a_n \bmod m_n, \end{aligned}$$

entonces podemos usar el comando `ChineseRem`, cuyo primer argumento es la lista de módulos y el segundo una lista (con la misma longitud) de residuos.

Así, para resolver

$$\begin{aligned} x &\equiv 3 \bmod 14, \\ x &\equiv 7 \bmod 16, \end{aligned}$$

hacemos lo siguiente.

```
gap> ChineseRem([14,16],[3,7]);
87
```

1.6 Los anillos $\mathbb{Z}_n$

La función `ZmodnZ` nos permite definir en GAPel anillo de enteros módulo el argumento entero que le pasemos.

```
gap> A:=ZmodnZ(10);
(Integers mod 10)
```

El uno de un anillo (el elemento neutro del producto) se calcula con el comando `One`.

```
gap> uno:=One(A);
ZmodnZObj( 1, 10 )
```

Podemos hacer operaciones elementales en ese anillo, como calcular inversos, sumar elementos...

```
gap> 1/(3*uno);
ZmodnZObj( 7, 10 )
gap> Int(last);
7
gap> 2*uno+6/3*uno;
ZmodnZObj( 4, 10 )
```

Si un elemento no tiene inverso, devuelve `fail`.

```
gap> 1/(2*uno);
ZmodnZObj( fail, 10 )
gap> Int(last);
fail
```

También podemos usar la función `Inverse`.

```
gap> Inverse(2*uno);
fail
gap> Inverse(3*uno);
ZmodnZObj( 7, 10 )
```

Y extraer así el conjunto de unidades de $\mathbb{Z}_{10}$.

```
gap> Filtered([1..9],n->Inverse(n*uno)<>fail);
[ 1, 3, 7, 9 ]
```

Aunque esto lo podíamos haber hecho usando la función `PrimeResidues`.

```
gap> PrimeResidues(10);
[ 1, 3, 7, 9 ]
```

O bien con

```
gap> Units(ZmodnZ(10));
<group with 1 generators>
```

Como la salida es un grupo, para ver sus elementos lo pasamos a lista y luego cada elemento lo representamos como un entero.

```
gap> List(last,Int);
[ 1, 3, 7, 9 ]
```

La función φ de Euler (función tociente) se expresa como `Phi` en GAP.

```
gap> Phi(10);
4
```

Además GAPtiene un comando para determinar si un anillo es o no un cuerpo.

```
gap> IsField(ZmodnZ(5));
true
```

2 Polinomios

Para empezar a trabajar con polinomios, tenemos que especificar las variables y qué anillo de coeficientes vamos a considerar. GAPpor defecto expande las expresiones que introducimos, a diferencia de MATHEMATICA.

```
gap> x:=Indeterminate(Rationals,"x");
x
gap> (x+1)*(x-1);
x^2-1
```

2.1 Coeficientes de un polinomio

Si queremos obtener una lista de los coeficientes de un polinomio en una variable, podemos usar lo siguiente.

```
gap> CoefficientsOfUnivariatePolynomial(x^2+x-1);
[ -1, 1, 1 ]
```

Y el polinomio líder lo obtenemos con `LeadingCoefficient`.

```
gap> LeadingCoefficient(x^2+x-1);
1
```

Definamos una función para encontrar el término líder de un polinomio respecto de una variable. En ella usamos funciones que son alternativa a las que acabamos de ver para más de una variable.

```
terminolider:=function(p,x)
    local grado;
    grado:=DegreeIndeterminate(p,x);
    return PolynomialCoefficientsOfPolynomial(p,x)[grado+1]*x^grado;
end;

gap> terminolider(x^2+x-1,x);
x^2
gap> terminolider(3*x^2+x-1,x);
3*x^2
gap> y:=Indeterminate(Rationals,"y");
y
gap> terminolider(y*x^2+y^4*x-1,x);
x^2*y
```

2.2 División de polinomios

Si el anillo de coeficientes que consideramos es un cuerpo, entonces sabemos que el anillo de polinomios sobre una sola variable es un dominio euclídeo. Por tanto, podemos usar las funciones que ya conocemos para calcular el cociente y resto de una división.

```
gap> x:=Indeterminate(Rationals,"x");
x
gap> QuotientRemainder(x^3-x+1,2*x^2-3);
[ 1/2*x, 1/2*x+1 ]
```

Si nuestro anillo de polinomios no es un dominio euclídeo, entonces no podemos usar estas funciones.

```
gap> y:=Indeterminate(Rationals,"y");
y

gap> QuotientRemainder((x^3-x+1)*(y-1),y-1);
Error, no method found! For debugging hints type ?Recovery from NoMethodFound
```

```
Error, no 2nd choice method found for ‘QuotientRemainder’ on 3 arguments calle\
d from
QuotientRemainder( DefaultRing( [ r, m ] ), r, m ) called from
<function>( <arguments> ) called from read-eval-loop
Entering break read-eval-print loop ...
you can ’quit;’ to quit to outer loop, or
you can ’return;’ to continue
brk>
```

Ahora bien, si que podemos usar la función `Quotient` que nos da el cociente, en caso de que éste pertenezca a nuestro anillo de polinomio, y `fail` en caso contrario.

```
gap> Quotient((x^3-x+1)*(y-1),y-1);
x^3-x+1

gap> Quotient(2,3);
fail
```

(Esta última instrucción viene a decir que el cociente de dos entre tres no es entero, pues considera los argumentos de la función como enteros.)

2.3 Factorización de polinomios

Si lo que queremos es factorizar polinomios, primero tenemos que definir la variable, e indicar cuál es el anillo de coeficientes para nuestros polinomios. Luego se usa `Factors` igual que antes.

```
gap> x:=Indeterminate(ZmodnZ(5),"x");
x
gap> Factors(x^2+1);
[ x+Z(5), x+Z(5)^3 ]
gap> Int(Z(5));
2
gap> Int(Z(5)^3);
3
```

Si cambiamos el anillo base, el resultado puede verse alterado.

```
gap> x:=Indeterminate(Rationals,"x");
x
gap> Factors(x^2+1);
[ x^2+1 ]

gap> x:=Indeterminate(Rationals,"x");
x
gap> Factors(x^3-1);
[ x-1, x^2+x+1 ]
gap> x:=Indeterminate(ZmodnZ(3),"x");
x
gap> Factors(x^3-1);
[ x-Z(3)^0, x-Z(3)^0, x-Z(3)^0 ]
```

Lo mismo ocurre con las raices y con el hecho de ser irreducible.

```
gap> x:=Indeterminate(ZmodnZ(3),"x");
x
gap> RootsOfUPol(x^3-1);
[ Z(3)^0, Z(3)^0, Z(3)^0 ]
gap> x:=Indeterminate(Rationals,"x");
x
gap> RootsOfUPol(x^3-1);
[ 1 ]

gap> x:=Indeterminate(ZmodnZ(3),"x");
x
gap> IsIrreducible(x^2+1);
true

gap> x:=Indeterminate(ZmodnZ(2),"x");
x
gap> IsIrreducible(x^2+1);
false
```

Veamos ahora a modo de ejemplo cómo calcular todos los polinomios irreducibles hasta un determinado grado en $\mathbb{Z}_m$. Empezamos definiendo una función que nos genere todos los polinomios hasta un determinado grado.

```
polshastagradomodm:=function(n,x,m)
    local ps;
    if (n=0) then
        return [0..(m-1)];
    fi;

    ps:=polshastagradomodm(n-1,x,m);
    return List(Cartesian(ps,List([0..(m-1)],i->i*x^n)),Sum);
end;
```

Así todos los polinomios en $\mathbb{Z}_3$ de grado menor o igual que dos son:

```
gap> polshastagradomodm(2,x,3);
[ 0*Z(3), x^2, -x^2, x, x^2+x, -x^2+x, -x, x^2-x, -x^2-x, Z(3)^0, x^2+Z(3)^0,
  -x^2+Z(3)^0, x+Z(3)^0, x^2+x+Z(3)^0, -x^2+x+Z(3)^0, -x+Z(3)^0,
  x^2-x+Z(3)^0, -x^2-x+Z(3)^0, -Z(3)^0, x^2-Z(3)^0, -x^2-Z(3)^0, x-Z(3)^0,
  x^2+x-Z(3)^0, -x^2+x-Z(3)^0, -x-Z(3)^0, x^2-x-Z(3)^0, -x^2-x-Z(3)^0 ]
```

De entre ellos podemos escoger los que son irreducibles.

```
gap> Filtered(last,IsIrreducible);
[ x, -x, x^2+Z(3)^0, x+Z(3)^0, -x^2+x+Z(3)^0, -x+Z(3)^0, -x^2-x+Z(3)^0,
  -x^2-Z(3)^0, x-Z(3)^0, x^2+x-Z(3)^0, -x-Z(3)^0, x^2-x-Z(3)^0 ]
```

Y si queremos quedarnos con un representante salvo asociados, podemos usar lo siguiente.

```
gap> Set(last,StandardAssociate);
[ x, x+Z(3)^0, x-Z(3)^0, x^2+Z(3)^0, x^2+x-Z(3)^0, x^2-x-Z(3)^0 ]
```

Para finalizar esta sección, implementamos una función que da los primos que se pueden aplicar en el criterio de Eisenstein para un polinomio en una variable.

```
eisenstein:=function(p)
    local lc,fp;
```

```
    lc:=CoefficientsOfUnivariatePolynomial(p);
    lc:=lc{[1..(Length(lc)-1)]};
    fp:=Factors(lc[1]);
    return Filtered(fp,f->(ForAll(lc,c->(c mod f=0)) and (lc[1] mod f^2=0)));
end;

gap> x:=Indeterminate(Rationals,"x");
x
gap> eisenstein(x^2+2*x-6);
[  ]
gap> eisenstein(x^2+2*x-4);
[ -2, 2 ]
```

2.4 Cociente por un ideal. Cuerpos finitos.

Intentemos calcular los divisores de cero y unidades del anillo cociente $R = \mathbb{Z}_2[x]/(x^2+1)$. Empezamos definiendo nuestra variable y el módulo.

```
gap> x:=Indeterminate(ZmodnZ(2),"x");
x

gap> modulo:=x^2+1;
x^2+Z(2)^0
```

Como cada elemento en R tiene un único representante de grado menor o igual que uno (el resto de dividir por x^2+1), podemos identificar R con el siguiente conjunto.

```
gap> elementos:=List(Cartesian([0..1],[0..1]),n->n[1]+x*n[2]);
[ 0*Z(2), x, Z(2)^0, x+Z(2)^0 ]
```

Que se lee como $\{0, x, 1, 1+x\}$. Seleccionamos aquellos elementos que son no nulos.

```
gap> elementosnonulos:=elementos{[2..4]};
[ x, Z(2)^0, x+Z(2)^0 ]
```

Así las unidades se pueden calcular de la siguiente forma.

```
gap> Filtered(elementosnonulos,n->
    ForAny(elementosnonulos,m->IsOne(EuclideanRemainder(n * m,modulo))));
[ x, Z(2)^0 ]
```

Obsérvese que hemos vuelto a utilizar `EuclideanRemainder`. La función `IsOne` sirve para determinar si un elemento en $\mathbb{Z}_2[x]$ es uno (no podemos en este caso escribir simplemente `EuclideanRemainder(n * m,modulo)=1`).

Los divisores de cero no nulos, se calculan de forma análoga.

```
gap> Filtered(elementosnonulos,
n->ForAny(elementosnonulos,
m->IsZero(EuclideanRemainder(n * m,modulo))))
[ x+Z(2)^0 ]
```

3 Combinatoria

Si queremos calcular el número de listas (ordenadas) de r tomados a partir de n elementos iniciales, podemos definir la siguiente función.

```
permutaciones:=function(n,r)
    if (r<=n) then
        return Factorial(n)/Factorial(n-r);
    fi;
end;
```

El conjunto de éstas se puede obtener con `Arrangements`. También podemos usar `NrArrangements` para conocer cuántas hay, evitando así definir la función `permutaciones`.

```
gap> permutaciones(3,2);
6
gap> Arrangements([1,2,3],2);
[ [ 1, 2 ], [ 1, 3 ], [ 2, 1 ], [ 2, 3 ], [ 3, 1 ], [ 3, 2 ] ]
gap> Length(last);
6
gap> NrArrangements([1..3],2);
6
```

Cuando no importa el orden, a saber, buscamos subconjuntos de r elementos de un conjunto de n, usamos `Binomial`.

```
gap> permutaciones(6,3)/Factorial(3);
20
gap> Binomial(6,3);
20
```

Y podemos ver cuáles son con el comando `Combinations`.

```
gap> Combinations([1..6],3);
[ [ 1, 2, 3 ], [ 1, 2, 4 ], [ 1, 2, 5 ], [ 1, 2, 6 ], [ 1, 3, 4 ],
  [ 1, 3, 5 ], [ 1, 3, 6 ], [ 1, 4, 5 ], [ 1, 4, 6 ], [ 1, 5, 6 ],
  [ 2, 3, 4 ], [ 2, 3, 5 ], [ 2, 3, 6 ], [ 2, 4, 5 ], [ 2, 4, 6 ],
  [ 2, 5, 6 ], [ 3, 4, 5 ], [ 3, 4, 6 ], [ 3, 5, 6 ], [ 4, 5, 6 ] ]
gap> Length(last);
20
gap> NrCombinations([1..6],3);
20
```

Las permutaciones de una lista se pueden construir con `Permutations`.

```
gap> PermutationsList([1..3]);
[ [ 1, 2, 3 ], [ 1, 3, 2 ], [ 2, 1, 3 ], [ 2, 3, 1 ], [ 3, 1, 2 ],
  [ 3, 2, 1 ] ]
gap> PermutationsList([11,12,2,3]);
[ [ 2, 3, 11, 12 ], [ 2, 3, 12, 11 ], [ 2, 11, 3, 12 ], [ 2, 11, 12, 3 ],
  [ 2, 12, 3, 11 ], [ 2, 12, 11, 3 ], [ 3, 2, 11, 12 ], [ 3, 2, 12, 11 ],
  [ 3, 11, 2, 12 ], [ 3, 11, 12, 2 ], [ 3, 12, 2, 11 ], [ 3, 12, 11, 2 ],
  [ 11, 2, 3, 12 ], [ 11, 2, 12, 3 ], [ 11, 3, 2, 12 ], [ 11, 3, 12, 2 ],
  [ 11, 12, 2, 3 ], [ 11, 12, 3, 2 ], [ 12, 2, 3, 11 ], [ 12, 2, 11, 3 ],
  [ 12, 3, 2, 11 ], [ 12, 3, 11, 2 ], [ 12, 11, 2, 3 ], [ 12, 11, 3, 2 ] ]
gap> Length(last);
24
```

El comando `Tuples` me construye todas las listas con un número dado de elementos tomados de una lista dada.

```
gap> Tuples([1..2],4);
```

```
[ [ 1, 1, 1, 1 ], [ 1, 1, 1, 2 ], [ 1, 1, 2, 1 ], [ 1, 1, 2, 2 ],
  [ 1, 2, 1, 1 ], [ 1, 2, 1, 2 ], [ 1, 2, 2, 1 ], [ 1, 2, 2, 2 ],
  [ 2, 1, 1, 1 ], [ 2, 1, 1, 2 ], [ 2, 1, 2, 1 ], [ 2, 1, 2, 2 ],
  [ 2, 2, 1, 1 ], [ 2, 2, 1, 2 ], [ 2, 2, 2, 1 ], [ 2, 2, 2, 2 ] ]
gap> Length(last);
16
```

El comando `PartitionsSet` se utiliza para calcular el número de particiones de un conjunto dado.

```
gap> PartitionsSet([1..3]);
[ [ [ 1 ], [ 2 ], [ 3 ] ], [ [ 1 ], [ 2, 3 ] ], [ [ 1, 2 ], [ 3 ] ],
  [ [ 1, 2, 3 ] ], [ [ 1, 3 ], [ 2 ] ] ]
```

Si especificamos un segundo argumento, calcula el número de particiones con cardinal exactamente ese argumento.

```
gap> PartitionsSet([1..5],2);
[ [ [ 1 ], [ 2, 3, 4, 5 ] ], [ [ 1, 2 ], [ 3, 4, 5 ] ],
  [ [ 1, 2, 3 ], [ 4, 5 ] ], [ [ 1, 2, 3, 4 ], [ 5 ] ],
  [ [ 1, 2, 3, 5 ], [ 4 ] ], [ [ 1, 2, 4 ], [ 3, 5 ] ],
  [ [ 1, 2, 4, 5 ], [ 3 ] ], [ [ 1, 2, 5 ], [ 3, 4 ] ],
  [ [ 1, 3 ], [ 2, 4, 5 ] ], [ [ 1, 3, 4 ], [ 2, 5 ] ],
  [ [ 1, 3, 4, 5 ], [ 2 ] ], [ [ 1, 3, 5 ], [ 2, 4 ] ],
  [ [ 1, 4 ], [ 2, 3, 5 ] ], [ [ 1, 4, 5 ], [ 2, 3 ] ],
  [ [ 1, 5 ], [ 2, 3, 4 ] ] ]
gap> NrPartitionsSet([1..5],2);
15
```

El comando `Partitions` devuelve el número de formas posibles de sumar el argumento entero que pasemos.

```
gap> Partitions(5);
[ [ 1, 1, 1, 1, 1 ], [ 2, 1, 1, 1 ], [ 2, 2, 1 ], [ 3, 1, 1 ],
[ 3, 2 ], [ 4, 1 ], [ 5 ] ]
```

Podemos calcular cuántas de estas tienen una longitud determinada.

```
gap> Partitions(5,2);
[ [ 3, 2 ], [ 4, 1 ] ]
```

Si buscamos partiones ordenadas, hacemos lo siguiente

```
gap> OrderedPartitions(5);
[ [ 1, 1, 1, 1, 1 ], [ 1, 1, 1, 2 ], [ 1, 1, 2, 1 ], [ 1, 1, 3 ],
  [ 1, 2, 1, 1 ], [ 1, 2, 2 ], [ 1, 3, 1 ], [ 1, 4 ], [ 2, 1, 1, 1 ],
  [ 2, 1, 2 ], [ 2, 2, 1 ], [ 2, 3 ], [ 3, 1, 1 ], [ 3, 2 ], [ 4, 1 ], [ 5 ] ]
```

y el cardinal de

```
gap> OrderedPartitions(5,2);
[ [ 1, 4 ], [ 2, 3 ], [ 3, 2 ], [ 4, 1 ] ]
```

nos dice el número de soluciones enteras positivas de $x+y=5$.

```
gap> Binomial(6,1)-2;
4
```

(quitamos 2 por `[0,5]` y `[5,0]`).

4 Grafos

Vamos a representar un grafo como una lista con dos componentes: la primera es una lista con los vértices, y la segunda una lista de lados. Por simplicidad, vamos cada lado será una lista con dos vértices, despreciando así la posibilidad de etiquetarlos.

Vamos a ilustrar cómo definir una función que calcule las componentes conexas de un grafo. Para ello usamos una función auxiliar que tome una lista de listas, y que si alguna de ellas interseca con otra, las una. El proceso parará cuando todas las listas sean disjuntas entre sí. Éstas contendrán entonces los vértices de cada una de las componentes conexas de nuestro grafo.

```
junta:=function(ll)
    local l,n,m,l1,l2,bucle;

    l:=Set(ll);
    n:=Length(l);
    bucle:=Filtered(Cartesian([1..n],[1..n]),x->x[1]<x[2]);
    m:=First(bucle,x->Intersection(l[x[1]],l[x[2]])<>[]);
    if (m<>fail) then
        l1:=l[m[1]];
```

```
        l2:=l[m[2]];
        l:=Difference(l,[l1,l2]);
        return junta(Union(l,[Union(l1,l2)]));
    fi;
    return l;

end;

componentesConexas:=function(l)
    return junta(Union(Set(l[1],v->[v]),l[2]));
end;
```

Veamos un ejemplo.

```
gap> componentesConexas([[1,2,3,4,5],[[1,2],[2,3],[1,5]]]);
[ [ 1, 2, 3, 5 ], [ 4 ] ]
```

A continuación mostramos cómo calcular la matriz de adyacencia de un grafo, y utilizamos esta función para determinar el número de caminos de longitud dada entre dos vértices.

```
matrizAdyacenciaDir:=function(l)
    local n;

    n:=Length(l[1]);
    return List([1..n],
        i->List([1..n],j->Length(Filtered(l[2],lado->lado=[i,j]))));
end;

matrizAdyacencia:=function(l)
    local n;

    n:=Length(l[1]);
    return List([1..n],i->List([1..n],
        j->Length(Filtered(l[2],lado->Set(lado)=Set([i,j])))));
```

```
end;

numeroCaminos:=function(i,j,l,n)
    local m;

    m:=matrizAdyacencia(l)^n;
    return (m[i][j]);
end;
```

Veamos un ejemplo.

```
gap> matrizAdyacencia([[1,2,3,4,5],[[1,2],[2,3],[1,5]]]);

[ [ 0, 1, 0, 0, 1 ], [ 1, 0, 1, 0, 0 ], [ 0, 1, 0, 0, 0 ],
[ 0, 0, 0, 0, 0 ],  [ 1, 0, 0, 0, 0 ] ]
```

Por último implementamos una función para calcular el polinomio cromático de un grafo. Para ello usamos los grafos auxiliares resultantes de quitar un lado de un grafo, y de contraer dicho lado en un vértice.

```
quitaLado:=function(l,g)
    return [g[1],Difference(g[2],[l])];
end;

identificaVertices:=function(i,j,g)
    local v,l;

    v:=Difference(g[1],[j]);
    l:=ShallowCopy(g[2]);
    for k in [1..Length(l)] do
        if (j in l[k]) then
            l[k]:=Union(Difference(l[k],[j]),[i]);
        fi;
    od;
    l:=Set(l,e->Set(e));
    l:=Filtered(l,e->Length(e)>1);
```

```
    return [v,l];
end;

polinomioCromatico:=function(g,x)
    local ge,gec,l;

    l:=g[2];
    if (l=[]) then
        return x^(Length(g[1]));
    fi;

    ge:=quitaLado(l[1],g);
    gec:=identificaVertices(l[1][1],l[1][2],ge);

    return polinomioCromatico(ge,x)-polinomioCromatico(gec,x);
end;
```

Para ver un ejemplo, definimos primero una variable x sobre los racionales.

```
gap> x:=Indeterminate(Rationals,"x");
x
gap> g1:=[[1,2,3],[[1,2],[2,3],[1,3]]];
[ [ 1, 2, 3 ], [ [ 1, 2 ], [ 2, 3 ], [ 1, 3 ] ] ]
gap> polinomioCromatico(g1,x);
x^3-3*x^2+2*x
gap> Factors(last);
[ x-2, x-1, x ]
```

Con lo que el número cromático de este grafo es 3 (el primer entero no negativo en el que no se anula su polinomio cromático).