Tema 1

gcdex(x,y) → devuelve los coeficientes de la identidad de Bézout y el mcd que aparece en el tercer argumento de salida

```
gcdex(250,11) → [4, -9, 1]
```

Reducción por la izquierda

```
lreduce(f,[a,b,c])=f(f(a,b),c)

b7(x,y):=x·x+y$
lreduce(b7, [1,4,0,1,5]) = 3785
```

cf(x/11) → nos devuelve la secuencia de cocientes en el algoritmo de Euclides

```
cf(250 / 11) → [2, 3, 1, 27]
```

Cociente

```
quotient(x,y)
quotient(231,17) = 13
```

Resto

```
remainder(x,y)
remainder(231,17) = 10
```

Factorización

```
factor(11016) = 2³3⁴17
```

Primos

```
primep(17) = true
```

producto de matrices:

```
[x,y] = matrix([70/5,q]).
        matrix([0,1],[1,-3]).
        matrix([0,1],[1,-4]).
        matrix([0,1],[1,-2]) ;
```

Máximo común divisor

```
mcd(x,y)
```

Mínimo común múltiplo

```
lcm(x,y)
lcm(60,46) = 1380
```

Operador mod

```
mod(231,17) -> 10
if n > 0 -> remainder(n,m) = mod(n,m)
```

Operador Euler-Fermat

```
totient(504) -> 144
```

Factorizar

```
factor(504)->2³3²7
```

Teorema de Euler-Fermat

$$x \equiv a^x (mod\ m)$$

```
power_mod(15,39,37) -> 8
power_mod(a, x, m)  ==  mod(a^x,m)
```

Inverso modular

```
inv_mod(a,n)
inv_mod(5, 17) = 3
```

Teorema chino del resto

determina las soluciones de un sistema de congruencias con solución o devuelve false si no tiene solución. El sistema que hemos resuelto en el ejemplo anterior se escribiría:

```
chinese(coefs, mods)
chinese([1,2,3],[3,5,7]) = 52
```

Ejercicios

Resuelva, si es posible, la siguiente ecuación diofántica

$$27x + 6y = 9$$

```
gcdex(27,6) = [1,-4, 3]
9/3 = 3
3 · gcde(27,6) = [3, -12, 9]
27/3 = 9
6/3 = 2
```

La solución es:

$$x = 3 + 2q$$
$$y = -12 - 9q$$

Resuelva la ecuación en congruencias

$$44^{2001}x \equiv 9(mod\ 25)$$

```
totient(25) = 20
quotient(2001, 20) = 100 // 4·100+1 = 2001 -> a⁴ === 1
inv_mod(44,25) = 4
mod(9·4, 25) = 11 // x = 11 (mod 25)
```

La solución a la ecuación en congruencia es 11 mod 25

Estudia si el siguiente sistema de congruencias tiene solución y, en caso afirmativo, resuélvelo.

$$\begin{cases} 7x \equiv 5(mod\ 15) \\ x + 3 \equiv -1(mod\ 21) \end{cases}$$

```
inv_mod(7,15) = 13 // x = 5(mod 15)
                                    //  x = -4 (mod 21)

chinese([5,-4], [15,21]) = 80
lcm(15,21) = 105 // La solcución es 80 módulo 1053
```

Halla la representación decimal de los siguientes números expresados en las bases indicadas:

La funcion b2 convierte un numero en base 2, y de dos digitos, a base 10

a. 10011101/2

b. 1231/7

```
a)
b2(x,y):=2·x+y$
b2(1,1);
# usando lreduce podemos convertir cualquier numero en base 2
lreduce(b2,[1,0,0,1,1,1,0,1]); // 157
b)
```

```
b7(x,y)=7*x+y$
lreduce(b7,[1,2,3,1]);
```

Halla la representación en las bases 2,8 y 11 de los siguientes números expresados en base decimal:

a. 237

b. 2002

```
base(b,N,res):=if N=0 then res
    else base(b,quotient(N,b),cons(remainder(N,b),res))$

base(2, 2002,[]); // [1,1,1,1,1,0,1,0,0,1,0]
base(8, 2002, []); // [3,7,2,2]
base(11, 2002, []); // [1,5,6,0]
```

Halla el valor de x∈N e y∈N para que se verifiquen las siguientes igualdades:

a. 331/x = 106/11

```
e1:3·x^2+3·x+1$
e2:11^2+6; // 127
solve(e1=e2, x); // [x=6,x=-7]
```

b. 274/8 = y/2

```
b8(x,y):=8·x+y$
lreduce(b8, [2,7,4]); // 188
base(2,188,[]); // [1,0,1,1,1,1,0,0]
```

Define de forma recursiva, con recursión de cola, la función suma(n) que calcula la suma de los números naturales del 1 a *n*

```
suma(n,s):=if n=0 then s else suma(n-1, n+s)$
trace(suma);
suma(5,0);
untrace(suma);
suma(100,0);
sum(k,k,1,100);
100·(100+1)/2;
```

En cada uno de los siguientes apartados, expresa el mcd(a,b) como combinación lineal de a y b

a. a=250, b=111

```
// usamos la forma vectorial para obtener directamente los co
// identidad de bezout
mcdex(n,m):=bezout_vect([n,1,0],[m,0,1])$
bezout_vect(u,v):= if v[1] > u[1] then bezout_vect(v,u)
    else if v[1]=0 then u
    else bezout_vect(u-v,v)$

mcdex(250,111); // [1,4,-9]
-9·111+4·250; // 1
//el operador gcdex da el mismo resultado
gcdex(250, 111); //[4,-9,1]
//cf -> nos da la secuencia de cocientes del algoritmo de euc
cf(250/111); //[2,3,1,27]
//podemos obtener la identidad de bezout con el producto de m
matrix([0,1],[1,-27]).
matrix([0,1],[1,-1]).
matrix([0,1],[1,-3]).
matrix([0,1],[1,-2]).
```

Estudia si las siguientes ecuaciones tienen solución y en tal caso, encuentra todas las soluciones.

a. 42x+312y=834

```
cf(312,42); //[7,2,3]
mcdex(312,42); //[6,-2,15]
312·(-2)+42·15; //6
// la ecuacion del apartado a tiene solucion puesto que 6 div
834/6; //139
matrix([139,k]).
matrix([0,1],[1,-3]).
matrix([0,1],[1,-2]).
matrix([0,1],[1,-7]).
matrix([312],[42]);
// 834, por lo tanto, la solucion general la obtenemos con el
// de las cuatro primeras matrices

matrix([139,k]).
matrix([0,1],[1,-3]).
matrix([0,1],[1,-2]).
matrix([0,1],[1,-7]).
// (7k-278 2085-52k)
// es decir, x = 2085-52·k, y = 7·k-278
42·(2085-52·k)+312·(7·k-278),expand: //834
mcdex(42,312); // [6,15,-2]
[x,y]=(1/6)·(834·[15,-2]+q·[-312,42]),expand:
//[x,y]=[2085-52q, 7q-278]
```

Para tender un tramo de vía de 122 m. se dispone de barras de 30 m. y de 16 m. de largo. ¿Es posible cubrir el tramo utilizando solamente ese tipo de barras? Si es posible, determina cuántas barras de cada longitud se necesitan para cubrir los 122m.

Tenemos que resolver la ecuacion 30x+16y=122, en donde x es el numero de tramos de longitud 30 e y el numero de tramos de longitud y

```
// 30x+16y=122
gcdcex(30,16); //[-1,2,2]
```

```
30·(-1)+16·2; // 2
sol3: (1/2)·(122·[-1,2]+q·[-16,30]),expand; //[-8q-61, 15q+12
load(fourier_elim)$
fourier_elim([sol3[1]>0, sol3[2]>0],[q]),numer;
// [-8.133 < q, q < -7.625] por lo tanto, q debe ser igual a
[x,y]=ev(sol3,q=-8); //[x,y]=[3,2]
//utilizando la forma matricial:
cf(30/16); //[1,1,7]
gcd(30,16) // 2
solm3: matrix([122/2,k]).
            matrix([0,1],[1,-7]).
            matrix([0,1],[1,-1]).
            matrix([0,1],[1,-1]);
// (8k-61   122-15k)
```

Enviamos por correo dos tipos de paquetes A y B. Por enviar los del tipo A nos cobran 15 céntimos de euro más que por los del tipo B. Sabiendo que hemos enviado más paquetes del tipo B que del tipo A, que en total hemos enviado 12 paquetes y que nos han cobrado un total de 13 euros con 20 céntimos, ¿cuántos hemos enviado de cada tipo y qué nos han cobrado por cada uno?

```
// 15A + 12x = 1320
gcdex(15,12); //[1,-1,3]
sol4: (1/3)·(1320·[1,-1]+q·[-12,15]),expand; //[440-4q, 5q-44
//Dado que A debe ser una cantidad entre 0 y 6, hallamos los
//valores de q para que se de esta restriccion
load(fourier_elim)$
fourier_elim([sol4[1]<6,sol4[1]>0],[q]),numer;
// [108.5<q, q<110]
// por lo tanto, necesariamente q = 109
A:ev(sol4[1],q=109); // 4
B: 12-A; // 8
x: ev(sol4[2], q=109); // 105
y: x+15; //120
```

```
// comprobamos la solucion obtenida
A·y+B·x; // 1320
// resolvemos la ecuacion diofantica usando la forma matricia

cf(15/12);
gcd(15,12);
// [1,4]
// 3
matrix([1320/3,k]).
matrix([0,1],[1,-4]).
matrix([0,1],[1,-1]);
// (440-4k  5k-440)
```

Resuelve la ecuación diofántica $4x+6y+7z=12$ siguiendo el siguiente procedimiento

a. Aplica el cambio de variable $u=2x+3y$ y resuelve, con ayuda de Maxima, la ecuación diofántica en u y z obtenida.

b. Para cualquier solución uu obtenida en el apartado anterior, resuelve con ayuda de Maxima, la ecuación diofántica $2x+3y = u$ en x e y.

c. Razona la posibilidad de utilizar los pasos anteriores para resolver cualquier ecuación de 3 o más incógnitas; ¿qué otros cambios de variable podríamos haber utilizado?

```
// Tras hacer el cambio de variable, obtenemos la ecuacion
// 2u+7z = 12, en donde u = 2x+3y
gcdex(2,7); // [-3,1,1]
[u,z]: 12·[-3,1]+q·[-7,2],expand; //[-7q-36, 2q+12]
//la solucion en u es u=-7q-36, es decir 2x+3y=-7q-36, resolv
gcdex(2,3); // [-1,1,1]
[x,y]:(-7·q-36)·[-1,1]+k·[-3,2],expand; // [7q-3k+36, -7q+2k-
4·x+6·y+7·z,expand; //12
```

Hemos definido la función mcdex utilizando las diferencias sucesivas en lugar de los cocientes, tal y como aparece en la descripción del algoritmo de Euclides: da una definición recursiva (con recursión de cola) para mcdex en la que se utilicen los cocientes sucesivos.

```
mcdex_c(n,m):=bezout_vect_c([n,1,0],[m,0,1])$
bezout_vect_c(u,v):= if v[1]>u[1] then bezout_vect_c(v,u)
                                        else if v[1]=0 th
                                        else bezout_vect_
trace(bezout_vect_c);
mcdex_c(250,111);
```

Da una definición recursiva de la función

$$bezout\_mat(n,m) = \begin{pmatrix} s & t \\ \pm m/d & \mp n/d \end{pmatrix} (1)$$

en donde *n*·*s*+*m*·*t*=*d*=mcd(*n*,*m*) es la identidad de Bezout para n y m.

```
bezout_mat(n,m,s):= if m>n then bezout_mat(m,n,s)
                                    else if m=0 then s
                                    else bezout_mat(m, re
                                            matrix([0,1],
gcd(500,222) // 2
bezout_mat(500, 222, ident(2));
```

COMANDOS MÁXIMA

ECUACIONES DIOFÁNTICAS

312x + 42y = 834

gcdex(312,42);

[-2,15,6]

Esta función nos devuelve la identidad de Bezout, es decir, que si multiplicamos el 312 por -2 y el 15 por 42 nos da el máximo común divisor, que es 6. El resto del procedimiento tenemos que realizarlo aparte.

RESOLUCIÓN DE CONGRUENCIAS

Congruencias lineales (incluyendo exponentes grandes)

6x = -2 (mod 5)

inv_modC6,5);

1

Esta función nos calcula el inverso de un número respecto a un módulo.

x 2 1* -2 (mod 5)

mod (-2,5);

3

Gracias a esa función acabamos de resolver la congruencia. Es decir, calculando el inverso podemos "aislar" la x y con el operador mod podemos calcular el resultado de la congruencia (3+5m).

SISTEMAS DE CONGRUENCIAS LINEALES

x ≡ 2 -2 (mod 3)

2x ≡ 1 (mod 7)

x ≡ 3 (mod 8)

Primero tenemos que "aislar" las x, usando los operadores que hemos visto anteriormente

```
inv_mod(2,7);
        4
```

x ≡ -2 (mod 3)

x ≡ 4 (mod 7)

x ≡ 3 (mod 8)

```
chinese ([-2, 4, 31],[3, 7, 8]);
        67
lcm(3,7,8);
        168
```

Como podemos comprobar, Máxima nos resuelve el resultado (67+168m). En el primer campo del operador chinese introducimos lo que hay a la derecha de la igualdad y en el segundo operando introducimos los módulos. Luego hay que calcular el mínimo común múltiplo de los módulos.

PASAR DE UNAS BASES A OTRAS

Para pasar de base X a decimal

Primero tenemos que definir una función

```
base(b,N,res):= if N=0 then res
        else base(b,quotient(N,b), cons(remainder(N,b),res));
base(b,N,res):= if N=0 then res else
base(b, quotient (N, b), cons (remainder (N, b), res))
```

Ahora introducimos la base, el número a convertir y los corchetes.

```
base(7, 3785, []);
```

[1,4,0,1,5]

Pasar de decimal a base X

Primero tenemos que definir otra función auxiliar. En este ejemplo vamos a hacerlo también con base 7

b7(x, y): = 7x+y;

b7(x,y):= 7x+y

Ahora usamos el operador lreduce, introducimos en los corchetes los dígitos del número a convertir y llamamos a la función que acabamos de definir

lreduce(b7, [1,4,0,1,53);

3785

-Resolver polinomios:

(%i1) pol: x^2+3*x+2$
(%i2) **solve**(pol=0,x);

[x=−2,x=−1]

-Evaluar funciones:

(%i1) f(x):=x^2+3*x+2$
(%i2) f(10);

132

-Funciones recursivas:

En Maxima podemos utilizar if-then-else para definir funciones por casos o ramas. Esta misma estructura se puede utilizar para definir funciones recursivas si la función que estamos definiendo se llama a sí misma. Como vemos en este ejemplo, podemos trasladar literalmente la definición recursiva del factorial al lenguaje de Maxima.

(%i1) fact(n):= **if** n=0 **then** 1 **else** n*fact(n-1)$
(%i2) fact(5);

120120

No obstante, el operador factorial está predefinido en Maxima con el mismo símbolo que utilizamos en matemáticas.

(%i3) 5!;

120

-Binomio de Newton

El operador expand elimina los paréntesis en cualquier expresión para obtener su forma expandida y en cierto sentido simplificada. En particular, podremos ver el resultado de aplicar la fórmula de Newton si lo aplicamos a la potencia de un binomio.

```
(%i1) expand((x-2)^7);
                    x7−14x6+84x5−280x4+560x3−672x2+448x−128
```

-Función factorial recursiva

En este ejemplo, vamos a definir la función factorial con recursión de cola. El operador tiene dos argumentos, de forma que en el segundo se acumula la salida en cada paso recursivo. De esta forma, fact_tail(n,1)=n!fact_tail(n,1)=n!.

```
(%i1) fact_tail(n,s):= if n=0 then s else fact_tail(n-1,n*s)$
(%i2) fact_tail(5,1);
                                    120120
```

Una opción muy interesante de Maxima es la que da la posibilidad de ver la "traza" de los operadores definidos por el usuario: trace(<op>) provocará que a partir de ese momento la salida del operador <op> muestre los cálculos intermedios.

```
(%i3) trace(fact_tail)$
(%i4) fact_tail(5,1);
1" Enter "fact_tail" "[5,1]
 .2" Enter "fact_tail" "[4,5]
 ..3" Enter "fact_tail" "[3,20]
 ...4" Enter "fact_tail" "[2,60]
 ....5" Enter "fact_tail" "[1,120]
 .....6" Enter "fact_tail" "[0,120]
 .....6" Exit  "fact_tail" "120
 ....5" Exit  "fact_tail" "120
 ...4" Exit  "fact_tail" "120
 ..3" Exit  "fact_tail" "120
 .2" Exit  "fact_tail" "120
1" Exit  "fact_tail" 120
                                    120120
```

Como podemos ver el factorial de 5 se calcula durante el proceso de evaluación y en la fase de salida no se realiza ningún cálculo.

- Función forma recursiva de los números combinatorios:

La siguiente función define de forma recursiva de los números combinatorios:

```
(%i1) binom(n,k):= binom_tail(k,n-k+1,1,1)$
(%i2) binom_tail(k,d,c,s):= if k=1 then s*d/c else
                    binom_tail(k-1,d+1,c+1,s*d/c)$
(%i3) binom(10,5);
                                    252
```

La variable k funciona como un contador que determina el número de factores d y c del numerador y denominador respectivamente que tenemos que añadir. Además, cada resultado parcial que se acumula en la variable ss es un número combinatorio y, por lo tanto, es un número natural.

```
(%i4) trace(binom_tail)$
(%i5) binom(7,3);
1 Introducir binom_tail [3,5,1,1]
 .2 Introducir binom_tail [2,6,2,5]
 ..3 Introducir binom_tail [1,7,3,15]
 ..3 Salir binom_tail 35
 ...
                                    35
```

Naturalmente, Maxima incluye un operador primitivo para el cálculo de los números combinatorios.

```
(%i4) binomial(10,5);
                                    252
```

-Cociente y Resto

Para determinar cociente y el resto con Maxima, usamos los operadores quotient y remainder respectivamente.

```
(%i1) quotient(231,17);
                                   1313
(%i2) remainder(231,17);
                                    10
```

-Sumatorio:

Vemos en este ejemplo como calcular la siguiente suma en Maxima:

$$\sum_{k=1}^{100} \frac{1}{k} = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{100}$$

(%i1) **sum**(1/k,k,1,100);

$$\frac{14466636279520351160221518043104131447711}{2788815009188499086581352357412492142272}$$

(%i2) **float**(%);

5.187377517639621

En este ejemplo, hemos usado la variable %, que representa al resultado del último cálculo efectuado. Además, observamos que por defecto, el programa realiza los cálculos de forma exacta o simbólica y si queremos obtener los resultados aproximados debemos pedirlo explícitamente; esto podemos hacerlo con el operador float o con la opción numer.

(%i3) **sum**(1/k,k,1,100),numer;

5.187377517639621

-Factorizar(descomponer)

(%i1) **factor**(11016);

$$2^3\ 3^4\ 17$$

-Numeros primos

(%i2) **primep**(17);

true

(%i3) **primep**(22);

false

-Mínimo común múltiplo

(%i1) **lcm**(60,46);

1380

-Máximo común divisor

Disponemos igualmente de un operador que nos devuelve el máximo común divisor de dos números.

(%i4) **gcd**(30,12);

6

Vamos a calcular el máximo común divisor de 30 y 12 usando la definición

(%i1) D1: **divisors**(30);

{1,2,3,5,6,10,15,30}

```
(%i2) D2: divisors(12);
                                {1,2,3,4,6,12}
```

Con el operador intersect podemos determinar la intersección de los dos conjuntos, es decir, los divisores comunes.

```
(%i3) intersect(D1,D2);
                                  {1,2,3,6}
```

Por lo tanto, el máximo de todos los divisores es 6: mcd(30,12)=6

-Def recursiva más simple de mcd

Con el lema anterior podemos dar la definición recursiva más simple de mcdmcd:

```
(%i1) mcd(m,n):= if m = n then m
      else if m > n then mcd(m
      - n,n)
      else mcd(m,n - m)$ (%i2) mcd(33,21);
                                     3
```

Recordemos que ya habíamos visto que Maxima incluye el operador gcd.

-Ejemplo ec diofántica vectorial

La forma vectorial que hemos presentado en el ejemplo anterior, permite adaptar fácilmente el procedimiento que definimos para hallar el máximo común divisor con Maxima basado en las diferencias sucesivas.

```
(%i1) mcdex(n,m):= bezout_vect([n,1,0],[m,0,1])$
(%i2) bezout_vect(u,v):= if v[1] > u[1] then bezout_vect(v,u)
   else if v[1]=0 then u
   else bezout_vect(u-v,v)$ (%i3) mcdex(250,111);
                                  [1,4,−9]
```

También disponemos de varios operadores relacionados con los procesos anteriores. Por ejemplo, el operador gcdex devuelve los coeficientes de la identidad de Bézout y el máximo máximo común divisor, que aparece en el tercer argumento de la salida:

```
(%i3) gcdex(250,111);
                                  [4,−9,1]
```

4=x0, -9=y0, 1=mcd

Para ayudarnos a verificar los pasos intermedios, podemos usar el operador cf, que nos devuelve la secuencia de cocientes en el algoritmo de Euclides si lo aplicamos a la fracción determinada por los dos números:

(%i4) **cf**(250/111);

[2,3,1,27]

-Ejemlo ec.diofantica matrices

$$145x + 65y = 70$$

Vamos a completar los cálculos con ayuda de Maxima. En primer lugar, vamos a comprobar que hemos obtenido correctamente la secuencia de cocientes en el algoritmo de Euclides y que hemos calculado correctamente el máximo común divisor.

(%i1) **cf**(145/65);

[2,4,3]

(%i2) **gcd**(145,65);

5

(%i3) 70/5;

14

Y finalmente obtenemos la solución general usando el producto de matrices:

(%i4) [x,y] = **matrix**([70/5,q]).
matrix([0,1],[1,-3]).
matrix([0,1],[1,-4]).
matrix([0,1],[1,-2]) ;

(x,y)=(13q−56,126−29q)

-Calcular numero congruente(resto)

El operador mod(n,m) de Maxima determina el número
entre 0 y m−1 congruente con n módulo m, incluso aunque n sea negativo.

(%i1) **mod**(231,17)

10

(%i2) (231-10)/17

13

(%i3) **mod**(-231,17)

7

(%i4) (-231-7)/17

−14

Si n es positivo, remainder(n,m)=mod(n,m), pero no ocurre lo mismo si el primer argumento es negativo, por lo que no debemos confundir los dos operadores.

-Teorema de Euler

La función de Euler se cálcula en Maxima con el operador totient

(%i1) **totient**(504);

144144

(%i2) **factor**(504);

2332723327

(%i3) (2^3-2^2)*(3^2-3)*(7-1);

144

-Ejemplo congruencia

$$x \equiv 15^{39} \pmod{37}$$

(%i1) **power_mod**(15,39,37);

8

(%i2) **mod**(15^39,37);

8

-Inverso de un número??????????(no está bien)

$$\begin{pmatrix} 17 \\ 0 \end{pmatrix} - 2\begin{pmatrix} 6 \\ 1 \end{pmatrix} = \begin{pmatrix} 5 \\ -2 \end{pmatrix}$$
$$\begin{pmatrix} 6 \\ 1 \end{pmatrix} - \begin{pmatrix} 5 \\ -2 \end{pmatrix} = \begin{pmatrix} 1 \\ 3 \end{pmatrix} = \begin{pmatrix} \text{m.c.d.} \\ \text{Inverso} \end{pmatrix}$$

El operador inv_mod(a,m) de Maxima calcula el inverso de a módulo m

(%i1) **inv_mod**(5,17)

7

(%i2) **mod**(5*7,17)

1

-Sistemas de congruencias

El operador

chinese(coefs,mods)

determina las soluciones de un sistema de congruencias con solución o devuelve false si no tiene solución. El sistema que hemos resuelto en el ejemplo anterior se escribiría:

(%i1) **chinese**([1,2,3],[3,5,7);

52

Otro ejemplos con módulos que no son coprimos dos a dos:

(%i2) **chinese**([14,20,34],[15,21,35]);

104

(%i3) **chinese**([13,20,33],[15,21,35]);

false

-Encriptación

Mark e Iván se van a comunicar con mensajes encriptados con el método RSA en un entorno determinado por el número $m = 143$. Para recibir mensajes encriptados, Mark dispone de dos claves:

$$\text{Clave pública de Mark} = e = 53, \qquad \text{Clave privada de Mark} = d = 77$$

La clave pública sirve para encriptar el mensaje y debe ser conocida por cualquiera que quiera enviarle un mensaje encriptado. Sin embargo, la clave privada, que sirve para desencriptar, solo debe ser conocida por Mark. Si Iván quiere enviar el mensaje formado por la letra K, en primer lugar deberemos convertir ese mensaje en un número. Esta conversión se hace utilizando las tablas de códigos ASCII o Unicode. En este ejemplo, utilizaremos números decimales de dos dígitos, aunque para una tabla completa tendríamos que recurrir a tres dígitos decimales. La siguiente tabla muestra parte de los códigos ASCII:

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
32	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
P	Q	R	S	T	U	V	W	X	Y	Z	0	1	2	3	4
80	81	82	83	84	85	86	87	88	89	90	48	49	50	51	52
5	6	7	8	9	.	?	!	@							
53	54	55	56	57	46	63	23	54							

Por lo tanto, dado que el código ASCII de la letra K es 75, para encriptar el mensaje destinado a Mark, Iván calcula el resto de dividir 75^53 entre 143:

(%i1) **power_mod**(75,53,143);

69

Cuando Mark recibe el mensaje 69, tiene que desencriptarlo y lo hace calculando el resto de dividir 69^77entre 143:

(%i2) **power_mod**(69,77,143);

75

En este entorno de comunicación segura, Iván dispone de sus propias claves publica y privada:

Clave pública de Iván=e=47, Clave privada de Iván=d=23

De esta forma, si Mark quiere enviarle el mensaje S, que en ASCII se corresponde con el número 83, le enviará el resto de dividir 83^47entre 143:

(%i3) **power_mod**(83,47,143);

8

E Iván lo desencriptará calculando el resto de dividir 8^23 entre 143:
(%i4) **power_mod**(8,23,143);

83

-Determinación de claves

Volviendo al ejemplo inicial para ver cómo hemos elegido los números:

$$m = 11 \cdot 13 = 143$$
$$\phi(m) = (11-1)(13-1) = 120$$

Los números 53 y 47, elegidos como claves privadas en el ejemplo, son coprimos con 120 y por eso ha sido posible calcular sus inversos módulo 120, lo que determina los pares de claves:

(%i4) priv_mark: 77$
pub_mark: **inv_mod**(priv_mark,120);

53

(%i5) priv_ivan: 23$
pub_ivan: **inv_mod**(priv_ivan,120);

47

-Desencriptación

En primer lugar para construir el sistema de encriptado siguiendo el método RSA, necesitamos convertir los mensajes en números. Tal y como hemos dicho, esto se hace habitualmente con las tablas de códigos ASCII o Unicode. Por ejemplo, el mensaje "HOLA MUNDO!" se traduciría como:

72797665327785786879 23

A continuación el mensaje se divide en bloques con el mismo número de dígitos, de forma que cada bloque se encriptará por separado. Además, el número máximo que aparezca en un bloque deberá ser menor que el número mm que elijamos para determinar las claves. Por ejemplo, si queremos tomar bloques de cuatro dígitos, tendríamos que encriptar y enviar los siguientes números:

a1=7279, a2=7665, a3=3277, a4=8578, a5=6879, a6=2300

Y para encriptar tendríamos que considerar un número m mayor que 10^4y que esté dado por el producto de dos números primos. Por ejemplo:

m=79·127=10033

Las claves podrán ser cualquier número coprimo con $\phi(m)=(79-1)(127-1)=9828=22\cdot33\cdot7\cdot13$. Por ejemplo, podemos elegir e=817 como clave pública y d=409 sería la correspondiente clave privada:

```
(%i6) privada: 409$
publica: inv_mod(privada,9828);
                                    817
```

Vamos a definir en Maxima las funciones que nos permiten encriptar y desencriptar mensajes:

```
(%i7) encriptar(m):= power_mod(m,publica,10033)$
(%i8) desencriptar(m):= power_mod(m,privada,10033)$
```

De esta forma, encriptamos el mensaje "HOLA MUNDO!" con:

```
(%i9) mensaje: [7279,7665,3277,8578,6879,2300]$
(%i10) map(encriptar,mensaje);
                    [5720,4681,8471,2751,3465,8809]
```

Y comprobamos que al desencriptar obtenemos la misma lista de números:

```
(%i11) map(desencriptar,[5720,4681,8471,2751,3465,8809]);
                    [7279,7665,3277,8578,6879,2300]
```

Maxima incluye un operador primitivo para el cálculo de los números combinatorios.
binomial(x,y);
Suma en maxima donde 1/k es la funcion, k es la letra, 1 donde empieza y 100 donde termina.
sum(1/k,k,1,100);
El operador expand elimina los paréntesis en cualquier expresión para obtener su forma expandida
y en cierto sentido simplificada.
(%i1) expand((x-2)^7); --devuelve
x7−14x6+84x5−280x4+560x3−672x2+448x−128
Para determinar cociente y el resto con Maxima, usamos los operadores quotient y remainder respectivamente.
(%i1) quotient(231,17); -- cociente
(%i2) remainder(231,17); -- resto
Factorización de numeros:
factor(x);
Devuelve true si es primo, false si no lo es:
primep(x);
Devuelve los divisores de un numero:
divisors(x);
Intersección de dos conjuntos:
intersect(D1,D2);
MCD de dos numeros:
gcd(x,y);
MCM de dos numeros:
lcm(x,y);
El operador gcdex devuelve los coeficientes de la identidad de Bézout y el máximo máximo común divisor,
que aparece en el tercer argumento de la salida:
gcdex(250,111);
[4,−9,1]
Nos devuelve la secuencia de cocientes en el algoritmo de Euclides
si lo aplicamos a la fracción determinada por los dos números:
cf(250/111);
[2,3,1,27]
Se utiliza para representar una matriz:
matrix([x,y])
CONGRUENCIA
El operador mod(n,m) de Maxima determina el número entre 0 y m−1 congruente con n módulo m, incluso aunque n sea negativo.
mod(231,17)
10

La función de Euler se calcula en Maxima con el operador totient
totient(x);
El teorema de Euler-Fermat es parte de los cálculos realizados por el operador power_mod de Maxima.
(%i1) power_mod(15,39,37); -- donde 15 es el numero, 39 es la potencia y 37 el modulo
8
(%i2) mod(15^39,37);
8
Calcula el inverso de a módulo m
(%i1) inv_mod(x,y) -- x es el numero e y el modulo
determina las soluciones de un sistema de congruencias con solución o devuelve false si no tiene solución.
El sistema que hemos resuelto en el ejemplo anterior se escribiría:
chinese([1,2,3],[3,5,7); -- primero escribimos los coeficientes y después el módulo asociado a cada uno

PRUEBA MAXIMA

MACORÍS DECENA GIMÉNEZ

EJERCICIO 1

Convierte el número 573A047C357B dado en base 16 al correspondien

```
b16(x,y):=16·x+y$
lreduce(b16,[5,7,3,A,0,4,7,C,3,5,7,B]), expand;
```

$$65536\,C+B+4294967296\,A+95863744509296$$

```
65536·12+11+4294967296·10+95863744509296;
```

$$95906694968699$$

```
base(b,N,res):= if N=0 then res
    else base(b,quotient(N,b),cons(remainder(N,b),res))$
base(16, 95906694968699, []);
```

$$[5,7,3,10,0,4,7,12,3,5,7,11]$$

EJERCICIO 2

Determina la solución general de la siguiente ecuación y comprueba qu

7345623x + 7395261y = 1906584

```
gcdex(7345623,7395261);
```

$$[923253,-917056,3]$$

```
quotient(1906584,3);
```

$$635528$$

```
(%o6)·(%o7);
```

$$[586753132584,-582814765568,1906584]$$

SOLUCIÓN PARCIAL: X = 586753132584 ; Y = −582814765568

```
586753132584·7345623−582814765568·7395261;
```

$$1906584$$

```
quotient(7345623, 3);
```

$$2448541$$

```
quotient(7395261, 3);
```

$$2465087$$

SOLUCIÓN: X = 586753132584 + 2465087K ; Y = 582814765568 − 24

```
(7345623·(586753132584 + 2465087))+(7395261·(−582814765568 − 2
```

$$1906584$$

EJERCICIO 3:
Resuelve si es posible el siguiente sistema de congruencias:
$13x \equiv 2275 \pmod{61373}$
$357x \equiv 847 \pmod{7319}$
$x \equiv 479 \pmod{5817}$

```
inv_mod(13,61373);
```

false

```
inv_mod(357,7319);
```

7278

LA PRIMERA CONGRUENCIA EQUIVALE A
$x == 175 \pmod{4721}$

LA SEGUNDA CONGRUENCIA EQUIVALE A

```
inv_mod(357, 7319);
```

7278

```
847·(%o16);
```

6164466

```
mod((%o17), 7319);
```

1868

$x == 1869 \pmod{7319}$

```
chinese([175, 1869, 479], [4721, 7319, 5817]);
lcm(4721, 7319, 5817);
```

135391548029
200994795183

$x == 135391548029 \pmod{200994795183}$

MATEMÁTICA DISCRETA TEMA 1

// 1ºA Ingeniería Software

EJERCICIO 1

a) Calcule d=mcd(3572,83430)

```
d:gcd(3572,83430);
```

Solución: mcd(3572,83430)=2

b) Halle los valores que verifican la combinación lineal

3572*m+83430*n=d

Para resolver esta combinación lineal, utilizaremos la Identidad de Bézout:

```
gcdex(3572,83430);
```

Hallamos que la solución particular de esta combinación lineal es: m=-17354 ; n=743 ; d=2

EJERCICIO 2

a)Comprueba si se puede aplicar el T. Euler

4^169 en Z203

Para comprobar si se puede aplicar el teorema, debemos comprobar si mcd(4,203)=1

```
gcd(4,203);
```

De esta forma, sabemos que se puede aplicar el teorema

b)Calcula φ(203)

totient(203);

Solución: φ(203)=168

c)Halle el resto de 4^169 en Z203

power_mod(4,169,203);

Solución: el resto de 4^169 en Z203 es 4, es decir, 4^169 = 4 (mod 203)

EJERCICIO 3

Comprueba si tiene solución y resuelve el siguiente sistema de congruencias

168x=24(mod220)

56x=40(mod68)

Para comprobar si tiene solución, primero debemos despejar las x:

inv_mod(168,220);

Como no existe el inverso, utilizamos el método de ecuaciones diofánticas:

168x=24+220m --> 168x-220m=24 --> si -m=y --> 168x+220y=24 --> Resolvemos

d:gcd(168,220);

mod(24,4);

Tiene solución la ecuación porque 4 es divisor de 24

coef:firstn(gcdex(168,220),2);

t:24/d;

(coef·t)+([220,−168]/d)·k, expand;

Obtenemos que x=55k-102, es decir, x=-102(mod55)

mod(−102,55);

Por lo tanto, x=8(mod55)

Despejamos la siguiente congruencia:

```
inv_mod(56,68);
```

Tampoco se puede calcular el inverso, así que resolvemos la ecuación diofántica: 56x=40+68m --> 56x-68m=40 --> si -m=y --> 56x+68y=40

```
d:gcd(56,68);
```

```
mod(40,d);
```

Tiene solución la ecuación porque 4 es divisor de 40

```
coef:firstn(gcdex(56,68),2);
```

```
t:40/d;
```

```
(coef·t)+([68,−56]/d)·k, expand;
```

Obtenemos que x=17k-60, es decir, x=-60(mod17)

```
mod(−60,17);
```

Por lo tanto, x=8(mod17)

Y tenemos las dos congruencias despejadas: x=8(mod55) ; x=8(mod17)

Comprobamos si tiene solución si mcd(55,17) es divisor de 8-8

```
gcd(55,17);
```

```
mod(8−8,1);
```

El sistema tiene solución porque 1 es divisor de 0

```
sol1:chinese([8,8],[55,17]);
```

```
sol2:lcm(55,17);
```

```
solfinal:sol1+sol2·k, expand;
```

La solución es 935*k+8, le damos el valor k=0 y obtenemos que x=8

EJERCICIO 4

123456798x428=0(mod7)

```
mod(1234567980428,7);
```

```
mod(1000,7);
```

6+6x=0(mod7) -->6x=-6(mod7)

```
inv_mod(6,7);
```

```
mod(6·(−6),7);
```

Solución: x=6

Comprobamos:

```
mod(1234567986428,7);
```

TEMA 1: ARITMÉTICA

Instrucciones básicas

quotient(,) → cociente de una división
remainder(,) == mod(,) (cuando algún número es negativo) → resto
primep( ) → Si es primo o no
factor( ) → factorizar
next_prime( ) → siguiente número primo
prev_prime( ) → anterior número primo
gcd(,) → máximo común divisor (mcd)
lcm(,) → mínimo común múltiplo (mcm)
gcdex(a,b)→ Identidad de Bézout (am+bn=d) → devuelve: [m,n,d]
cf(/) → sucesión de cocientes (Euler) → devuelve: [, , ...]
firstn(,2) → guardar dos primeras cifras de un vector

A. ENTERA

Ec. diofánticas (ax+by=c)

EJEMPLO: 64x + 84y = 32

```
d:gcd(64,84);
```

4

```
mod(32,4);
```

0

tiene solución, porque 4 es divisor de 32

```
coef:firstn(gcdex(64,84),2);
```

$[4,-3]$

```
t:32/d;
```

8

```
coef·t;
```

$[32,-24]$

Solución particular: x=32 ; y=-24

```
sol:(coef·t)+([84,-64]/d)·k, expand;
```

$[21\,k+32,-(16\,k)-24]$

Solución general: x=21*k+32 ; y=-(16*k)-24

Si tuviese inecuaciones --> solve(sol[1]),float;solve(sol[2]),float

Factorial

EJEMPLO: 4!

```
fact(n):=if n=0 then 1 else n·fact(n−1)$
trace(fact)$
fact(4);
```

```
1 Enter fact [4]
 .2 Enter fact [3]
 ..3 Enter fact [2]
 ...4 Enter fact [1]
 ....5 Enter fact [0]
 ....5 Exit  fact 1
 ...4 Exit  fact 1
 ..3 Exit  fact 2
 .2 Exit  fact 6
1 Exit  fact 24          24
```

A base 10

EJEMPLO: 1231(7

```
b7(x,y):=7·x+y$
lreduce(b7,[1,2,3,1]);
```

463

A otra base

EJEMPLO: 237(2

```
base(b,N,res):=if N=0 then res else base(b,quotient(N,b),cons(remainder(N,b),res))$
base(2,237,[]);
```

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

A. MODULAR

Congruencias lineales (ax=b mod m) por inverso

EJEMPLO: 3x≡1(mod11)

```
d:gcd(3,11);
```

1

```
mod(1,1);
```

0

Tiene solución porque 1 es divisor de 1, y tiene 1 solución en base al T. Brahmagupta

```
inv_mod(3,11);
```

4

```
mod(4·1,11);
```

4

Por tanto nos queda esta expresión: x=4+11k --> le damos valores a k para obtener 1 solución

```
x1:mod(4+11·0,11);
```

4

Solución: x=4

Congruencias lineales (ax=b mod m) por ec. diofántica

EJEMPLO: 64x+11≡43(mod84)

64x+11≡43(mod84) --> 64x+11=43 +84m --> 64x-84m=32 --> -m=y --> 64x+84y=32

```
d:gcd(64,84);
```

4

```
mod(32,4);
```

0

Tiene solución porque 4 es divisor de 32, en total tiene 4 soluciones

```
coef:firstn(gcdex(64,84),2);
```

$$[4,-3]$$

```
sol:(coef·t)+([84,-64]/d)·k, expand;
```

$$[21\,k+32,-(16\,k)-24]$$

Solo utilizamos la solución de x=21*k+32 --> damos valores a k para obtener las 4 soluciones

```
x1:mod(21·0+32,84);
```

32

```
x2:mod(21·1+32,84);
```

53

```
x3:mod(21·(−1)+32,84);
```

11

```
x4:mod(21·2+32,84);
```

74

Soluciones: x=11, 32, 53, 74

Módulos (a^b=x mod m)

EJEMPLO 1: 100^101≡x (mod7)

```
power_mod(100,101,7);
```

4

Solución x=4

EJEMPLO 2: 2^11·3^13≡x(mod7)

```
a:power_mod(2,11,7);
```

4

```
b:power_mod(3,13,7);
```

3

```
mod(a·b,7);
```

5

Solución x=5

NOTA: Se puede utilizar el T. Euler (aφ(m)=1 mod m -->φ(m) --> totient(m))

T. Chino del resto (2 congruencias)

EJEMPLO: x = 4 (mod 7)
x = 3 (mod 8)

```
sol1:chinese([4,3],[7,8]);
```

11

```
d:gcd(7,8);
```

1

```
sol2:(7·8/d);
```

56

```
solfinal:sol1+sol2·k, expand;
```

```
56 k+ 11
```

T. Chino del resto (3 o más congruencias)

EJEMPLO: x = 5(mod6)
x = 3(mod10)
x = 8(mod15)

```
sol1:chinese([5,3,8],[6,10,15]);
```

```
23
```

```
sol2:lcm(6,10,15);
```

```
30
```

```
solfinal:sol1+sol2·k, expand;
```

```
30 k + 23
```

Le damos el menor valor posible a k --> k=0 --> Solución: x=23

Restos potenciales

makelist(mod(a^k,m),k,0,n) → (n+1 es el número de restos potenciales que quieres hallar)

Tema 2

Comprobar conjuntos

```
elementp(a,{a}); // true
subsetp({a},{a}) // true
A: {-2,-1,0,1,2};
B: {0,1,2};
C: {-1,0,1};
union(A,B,C);
intersection(A,B,C);
symmdifference(A,union(B,C)); //Muestra la diferencias entre 

f:{[0,1],[1,2],[2,4],[3,8],[4,8],[5,4],[6,2]};
Dominio_f:map(first,f); // {0,1,2,3,4,5,6}
is(cardinality(f)=cardinality(Dominio_f));
//Si es true, es una función
```

Permutaciones

```
perm3: permutations({1,2,3});
```

Cardinalidad

```
cardinality(perms3);
```

Comprobar cardinalidad

```
is(cardinality(perms3));
```

n-permutaciones

```
distintos(l):= is(l[1]#l[2])$
distintos([1,1]);
```

```
distintos([2,1]);
```

Coeficiente multinomial

```
cadinality(perm222)=multinomial_coeff(2,2,2)
```

Binomial

```
binomial(6,3);
```

Combinación con repetición

CR(3,7) de $x_1 + x_2 + x_3 = 7$

```
suma_list(l):=lreduce("+",l)$
suma_list([2,3,1,4]);
suma7Q:=is(suma_list(l)=7)$
A:{0,1,2,3,4,5,6,7};
comb_rep37:=subset(cartesian_product(A,A,A), suma7Q);
cardinality(comb_rep37);
binomial(3+7-1,7);
```

Stirling

```
partes74: set_partitions({1,2,3,4,5,6,7},4)$
cardinality(partes74)=stirling2(7,4);
```

Función generadora

```
gen: sum(z^n,n,1,10)*sum(z^n,n,2,7)*sum(z^n,n,1,30),expand;
coeff(gen,z,30);
gen1: sum(z^n,n,1,10)*sum(z^n,n,2,7)*sum(z^n,n,1,inf)$
coeff(gen1(z),z,30);
taylor(gen1,z,0,35);
coeff(%,z,30);
```

Ecuación de recurrencia

```
load(solve_rec)$
ecrec: 18*a[n]-21*a[n-1]+8*a[n-2]-a[n-3]$
solve_rec(ecrec,a[n]);
solve_rec(ecrec,a[n],a[0]=1,a[1]=5/6,a[2]=17/36);
```

Recurrencias lineales no homogéneas

```
load(solve_rec)$
solve_rec(a[n] = 3*a[n-1] - 2*a[n-2] - 2^(n-1), a[n], a[0]=5,
```

Ejercicio combinatoria

Vamos a considerar que las 9 personas son los números 1, 2, 3, 4, 5, 6, 7, 8, 9 y que 1 y 2 no pueden estar juntos en el comité

Primero, vamos a generar todos los posibles comités de cuatro personas. Es decir, los subconjuntos de cardinal 4:

```
card4(s):=is(cardinality(s)=4)$
set1:powerset({1,2,3,4,5,6,7,8,9})$
comites4:subset(set1,card4); //El número total de comités es
cardinality(comites4);
binomial(9,4); // 126
//Ahora vamos a quitar aquellos comités en los que están 1 y
//usamos el predicado subsetp
no12(s):=not subsetp({1,2},s)$
sol7:subset(comites4,no12);
//Vemos que el número de comités coincide con el calculado en
//dos formas, usando el principio del complementario y distin
cardinality(sol7);
binomial(9,4)-binomial(7,2);
binomial(7,4)+2·binomial(7,3);
```

Ejercicio ordenación

El reparto de habitaciones lo podemos realizar colocando de forma ordenada a los siete amigos, 1,2,3,4,5,6,7 y repartiéndole las 7 llaves de las tres habitaciones. Si las habitaciones son A, B y C, tendríamos las siguientes llaves: [A,A,B,B,C,C,C]. Por tanto, lo que estamos determinando son las permutaciones de esta lista:

```
sol13: permut(["A","A","B","B","C","C","C"]);
// Los posibles repartos son las permutaciones con repetición
length(sol13);
7!/(2!·2!·3!);
// También hemos calculado los repartos de forma secuencial y
// combinaciones. En este caso, es independiente el orden en
//repartamos cada habitación:
binomial(7,3)·binomial(4,2)·binomial(2,2);
binomial(7,2)·binomial(5,3)·binomial(2,2);
binomial(7,2)·binomial(5,2)·binomial(3,3);
```

Primos menores que 100

Para calcular los primos menores que 100, lo hacemos usando el complementario, es decir, a los 100 números le quitamos el 1, los múltiplos de 2, los múltiplos de 3, los múltiplos de 5 y los últiplos de 7. Pero tenemos que usar el principio de inclusión-exclusión, ya que hay números que son, por ejemplo, múltiplos de 2 y de 3 al vez, los múltiplos de 6.

Si p es primo, los múltiplos de p menores que 100 son floor(100/p)-1, ya que en floor(100/p) estaríamos contando a p, que sí es primo.

Si m no es primo, los múltiplos de m menores que 100 son floor(100/m).

```
sol18: (100-1)-(floor(100/2)-1)-(floor(100/3)-1)-(floor(100/5
+floor(100/6)+floor(100/10)+floor(100/14)+floor(100/15)+floor
-floor(100/30)-floor(100/42)-floor(100/70)-floor(100/105)
+floor(100/210);
// 25
cien: setify(makelist(k,k,1,100))$
primos100: subset(cien,primep);
```

```
cardinality(primos100);
//{2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71,73
// 25
```

Funciones generadoras

Vamos a resolver el ejercicio utilizando funciones generadoras. En cada factor de la siguiente función, los exponentes son los posibles que valores que pueden tomar las variables x1, x2, x3 y x4. Como vemos, podemos usar infinito como límite superior

```
gen: sum(z^k,k,1,inf)*sum(z^k,k,3,inf)*sum(z^k,k,0,8)*sum(z^k
```

Dado que hemos introducido sumas infinitas, la expansión la hacemos con el operador taylor, indicando el grado máximo que queremos obtener

```
genexp: taylor(gen,z,0,29)$
```

Podemos mirar el coeficiente en la expresión o pedirle que nos lo diga:

```
coeff(genexp,z,29); // 1197
```

Usando las propiedades de las sumas infinitas, hemos transformado el producto de los cuatro factores en un producto de solo dos factores, lo que facilita el cálculo a mano del coeficiente de z^29:

```
binomial(28,3)-binomial(21,3)-binomial(19,3)+binomial(12,3);
```

Finalmente, vamos a generar exhaustivamente las soluciones, tal y como hicimos en un ejercicio anterior:

```
suma29(l):= is(l[1]+l[2]+l[3]+l[4]=29)$
sol21: subset(cartesian_product(
{1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,
{3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,2
{0,1,2,3,4,5,6,7,8},
{0,1,2,3,4,5,6}),suma29)$
```

```
cardinality(sol21);
//1197
```

Número de desarreglos de 10 objetos

```
n:10$ sum((-1)^j*n!/j!,j,2,n); // 1334961
desarreglo(l):= not is(l[1]=1 or l[2]=2 or l[3]=3 or l[4]=4 o
desarreglo([5,1,4,3,2]); // true
desarreglo([5,1,3,2,4]); // false
des5: subset(permutations({1,2,3,4,5}),desarreglo);
// {[2,1,4,5,3],[2,1,5,3,4],....[5,4,2,1,3],[5,4,2,3,1]}

cardinality(des5); // 44
n:5$ sum((-1)^j*n!/j!,j,2,n); // 44
```

Aplicaciones sobreyectivas del conjunto de días {L,M,X,J,V,S,D} al conjunto de asignaturas {CC,MD,FP,FF}

En primer lugar, hacemos una partición del conjunto de días en cuatro partes, cada una de ellas corresponderá a los días en los que se estudiará una de las asignaturas.

```
part74: set_partitions({"L","M","X","J","V","S","D"},4);
```

El número de particiones es S(7,4), en donde S corresponde a los números de Stirling de segunda especie

```
cardinality(part74); // 350
stirling2(7,4); // 350
```

"part74" es un conjunto, y por lo tanto, el orden de las partes es indiferente. Para definir las aplicaciones sobreyectivas, debemos asignar cada parte a una asignatura. Si establecemos el orden [CC,MD,FP,FF], esto corresponde a generar todas la permutaciones de cada una de las 4 partes de las distintas

particiones. Esto lo hacemos con la siguiente secuencia de operadores, aunque no podemos visualizar las salidas por su excesiva longitud

```
Ipart74: listify(part74)$
IIpart74: map(listify,Ipart74)$
permpart74: map(permut,IIpart74)$
horarios: lreduce(append, permpart74)$
length(horarios); //8400
4!·stirling2(7,4); //8400
```

Fórmula explícita para las sucesiones

$U_0 = 1, U_1 = 2$ y $U_n = -2U_{n-1} + 3U_{n-2}$ para todo n ≥ 2

```
kill(all)$
load(solve_rec)$
ecrec32a:u[n]+2·u[n-1]-3·u[n-2];
solve_rec(ecrec32a,u[n]);
solve_rec(ecrec32a,u[n],u[0]=1,u[1]=2); // u_n=5/4 - (-3)^n/4
```

Vamos a resolver la recurrencia siguiendo los pasos del método de la ecuación característica. En primer lugar, factorizamos el polinomio que determina la ecuación característica:

```
factor(r^2+2·r-3); // (r-1)(r+3)
```

Con las soluciones r=1, r=-3, ambas con multiplicidad 1, obtenemos el esquema de la solución de la recurrencia

```
u[n]:=A+B·(-3)^n // u_n=A+B(-3)^n
```

Y determinamos A y B usando los terminos iniciales de la sucesión

```
solve([u[0]=1,u[1]=2],[A,B]); //[[A=5/4,B=-1/4]]
```

Fórmula explícita para las sucesiones

$U_0 = 1, U_1 = 3$ y $U_n = -4U_{n-1} + 4U_{n-2}$ para todo n ≥ 2

```
kill(all)$
load(solve_rec)$
ecrec32a:u[n]-4·u[n-1]+4·u[n-2];
solve_rec(ecrec32a,u[n]);
solve_rec(ecrec32a,u[n],u[0]=1,u[1]=3); // u_n=(n/2 + 1)·2^n
```

Con la ecuación característica:

```
factor(r^2-4·r+4); // (r-2)^2
u[n]:=(A·n+B)·2^n; // u_n: (A·n+B)·2^n
solve([u[0]=1,u[1]=3],[A,B]); // [[A=1/2, B=1]]
```

Establece si son verdaderas o falsas las siguientes relaciones:

- $a \in \{a\}$
- $\{a\} \in \{a\}$
- $\{a, b\} \in \{a, \{a, b\}\}$
- $a \subseteq \{a\}$
- $\{a\} \subseteq \{a\}$
- $\{a, b\} \subseteq \{a, \{a, b\}\}$

```
elementp(a,{a}) //true
elementp({a},{a}); //false
elementp({a,b},{a,{a,b}}//true
```

```
subsetp(a,{a});//false -> error
subsetp({a},{a});//true
subsetp({a,b},{a,{a,b}}};//false
elementp({a},{a,{a}}});//true
```

2. Sean los conjuntos $A_1 = \{-2, -1, 0, 1, 2\}$, $A_2 = \{0, 1, 2\}$, $A_3 = \{-1, 0, 1\}$ y sea el conjunto de índices $I = \{1, 2, 3\}$.
   - Determina los siguientes conjuntos: (a) $\bigcup_{i \in I} A_i$; (b) $\bigcap_{i \in I} A_i$
   - Tomando $\mathbb{Z}$ como conjunto universal, determina: (c) $\bigcup_{i \in I} \overline{A_i}$; (d) $\bigcap_{i \in I} \overline{A_i}$

```
A1={-2,-1,0,1,2};
A2={0,1,2};
A3:{-1,0,1};
union(A1,A2,A3);
intersection(A1,A2,A3);
```

Da un contraejemplo que demuestre que la siguiente igualdad no es válida para cualesquiera conjuntos A,B y C:

$$A - (B - C) = (A - B) - C$$

```
kill(all);
A:{1,2,3};
B:{1,2,4};
C:{2,3,4,5};
is(union(intersection(A,B),C)=intersection(A,union(B,C))); //
union(intersection(A,B,C); //{1,2,3,4,5}
intersection(A,union(B,C)); //{1,2,3}
//Damos un contraejemplo
A:{1,2};
B:{3};
C:{2,4};
```

```
is(setdifference(A,setdifference(B,C))=setdifference(setdiffe
//false
```

Diferencia simetrica $\triangle$

Da un contraejemplo para demostrar que no se verifican la igualdad $A\triangle(B \cup C) = (A\triangle B) \cup (A\triangle C)$

```
A:{1,2,3};
B:{2,4};
C:{3,4};
symmdifference(A,union(B,C));//{1,4}
union(symmdifference(A,B),symmdifference(A,C));//{1,2,3,4}
```

Consideramos la siguiente relación:

$$R = \{(0,1),(1,2),(2,4),(3,8),(4,8),(5,4),(6,2)\}$$

a. Escribe la matriz de adyacencia de la relación.

b. Estudia si la relación es una función y, en tal caso, determina su dominio, su codominio, su rango y estudia si es inyectiva y sobreyectiva.

```
f:{[0,1],[1,2],[2,4],[3,8],[4,8],[5,4],[6,2]};
Dominio_f:map(first,f); // {0,1,2,3,4,5,6}
//Dado que el dominio esta determinado a partir de la relacio
//que efectivamente es una funcion, basta observar que el car
//relacion coincide con el cardinal de su dominio y por lo ta
//del dominio tiene una unica imagen
is(cardinality(f)=cardinality(Dominio_f));// true
Imagen_f:map(second,f); //{1,2,4,8}
//dado que la imagen estan determinada a partir de la relacio
//considerar que coincide con el codominio y por lo tanto es
//dado que el dominio y codominio son conj, finitos y el card
//es esctricamente mayor que el cardinal de la imagen, por ta
```

¿Cuántas permutaciones de las letras ABCDEFABCDEF contienen las letras DEFDEF juntas en cualquier orden?

```
conj1: permutations(["A","B","C","D","E","F"]);
cardinality(conj1);
sec1:permut(["A","B","C",x]);
length(sec1);
sec2:makelist(ev(sec12,x=y),y,permut(["D","E","F"]));
sec22:lreduce(append,sec2); //juntamos las listas
solej1:map(flatten,sec22); //aplanamos las listas
length(solej1);
cardinality(setify(solej1));
```

Consideramos el número 29338848000=$2^8 \cdot 3^5 \cdot 5^3 \cdot 7^3 \cdot 11$

```
// a) ¿Cuántos divisores positivos tiene este número?

solej3: divisors(29338848000);
factor(29338848000);
cardinality(solej3);
exponentes: cartesian_product({0,1,2,3,4,5,6,7,8},{0,1,2,3,4,
cardinality(exponentes);
solej32: makeset(2^x*3^y*5^z*7^u*11^v,[x,y,z,u,v],exponentes)
is(solej3=solej32);

// b) ¿Cuántos son múltiplos de 99?
mul99(x):=is(mod(x,99)=0)$
solej3b: subset(solej3,mul99)$
cardinality(solej3b);
exponentesb: cartesian_product({0,1,2,3,4,5,6,7,8},{2,3,4,5},
solej32b: makeset(2^x*3^y*5^z*7^u*11^v,[x,y,z,u,v],exponentes
cardinality(solej32b);
is(solej3b=solej32b);
```

1. De un grupo de 9 personas se quiere elegir un comité con 4 miembros, pero hay dos personas que no podrían estar juntas en él ¿de cuántas formas se puede constituir el comité?

```
Vamos a considerar que las 9 personas son los números 1, 2, 3
que 1 y 2 no pueden estar juntos en el comité Primero, vamos
posibles comités de cuatro personas. Es decir, los subconjunt
card4(s):=is(cardinality(s)=4)$
set1: powerset({1,2,3,4,5,6,7,8,9})$
comites4: subset(set1,card4);
cardinality(comites4);
binomial(9,4);

Ahora vamos a quitar aquellos comités en los que están 1 y 2.
el predicado subsetp
no12(s):=not subsetp({1,2},s)$
sol7: subset(comites4,no12);
cardinality(sol7);
binomial(9,4)-binomial(7,2);
binomial(7,4)+2*binomial(7,3);
```

Se dispone de una gran cantidad de bolas rojas, azules y verdes. ¿De cuántas formas se puede seleccionar nueve bolas si se debe tener al menos una de cada color?

```
Cada extracción está determinada por una terna ordenada, [r,a
tales que, r es el número de bolas rojas, a es el número de b
 número de bolas verdes. Si queremos calcular todas las extra
 buscaríamos las ternas tales que r+a+v=9, en donde, r, a, v
 {0,1,2,3,4,5,6,7,8,9}

 suma9(l):= is(l[1]+l[2]+l[3]=9)$
 A: {0,1,2,3,4,5,6,7,8,9}$
 extr9: subset(cartesian_product(A,A,A),suma9);
 cardinality(extr9);
 binomial(3+9-1,9);
 B:{1,2,3,4,5,6,7,8,9}$
```

```
sol10: subset(cartesian_product(B,B,B),suma9);
cardinality(sol10);
binomial(3+6-1,6);
```

Siete amigos llegan a un hotel y sólo hay disponibles dos habitaciones dobles y una triple. ¿De cuántas maneras pueden repartirse?

```
El reparto de habitaciones lo podemos realizar colocando de f
siete amigos, 1,2,3,4,5,6,7 y repartiéndole las 7 llaves de l
Si las habitaciones son A, B y C, tendríamos las siguientes l
Por tanto, lo que estamos determinando son las permutaciones

sol13: permut(["A","A","B","B","C","C","C"]);
length(sol13);
7!/(2!*2!*3!);
binomial(7,3)*binomial(4,2)*binomial(2,2);
binomial(7,2)*binomial(5,3)*binomial(2,2);
binomial(7,2)*binomial(5,2)*binomial(3,3);
```

Halla cuántos enteros no superiores a 100 son primos.

```
Para calcular los primos menores que 100, lo hacemos usando e
a los 100 números le quitamos el 1, los múltiplos de 2, los m
múltiplos de 5 y los múltiplos de 7. Pero tenemos que usar el
inclusión-exclusión, ya que hay números que son, por ejemplo,
al vez, los múltiplos de 6.
Si p es primo, los múltiplos de p menores que 100 son floor(1
floor(100/p) estaríamos contando a p, que sí es primo.
Si m no es primo, los múltiplos de m menores que 100 son floo

sol18: (100-1)-(floor(100/2)-1)-(floor(100/3)-1)-(floor(100/5
+floor(100/6)+floor(100/10)+floor(100/14)+floor(100/15)+floor
-floor(100/30)-floor(100/42)-floor(100/70)-floor(100/105)
```

```
+floor(100/210);

cien: setify(makelist(k,k,1,100))$
primos100: subset(cien,primep);
cardinality(primos100);
```

Calcula el número de soluciones de enteros no negativos tiene la ecuación x1+x2+x3+x4=29, ¿Cuántas de ellas satisfacen x1>0,x2>2,x3<9,x4<7?

```
Vamos a resolver el ejercicio utilizando funciones generadora
siguiente función, los exponentes son los posibles que valore
variables x1, x2, x3 y x4. Como vemos, podemos usar infinito
gen: sum(z^k,k,1,inf)*sum(z^k,k,3,inf)*sum(z^k,k,0,8)*sum(z^k
genexp: taylor(gen,z,0,29)$
coeff(genexp,z,29);
binomial(28,3)-binomial(21,3)-binomial(19,3)+binomial(12,3);
binomial(28,3)-binomial(21,3)-binomial(19,3)+binomial(12,3);
sol21: subset(cartesian_product(
{1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,
{3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,2
{0,1,2,3,4,5,6,7,8},
{0,1,2,3,4,5,6}),suma29)$
cardinality(sol21);
```

¿De cuántas formas se pueden disponer los números 1,2,...,101,2,...,10 para que ninguno ocupe su posición natural?

```
Según se demuestra en los apuntes, el número de desarreglos d

n:10$ sum((-1)^j*n!/j!,j,2,n);
Vamos a construir los "desarreglos" de cinco números, ya que
es excesiva
desarreglo(l):= not is(l[1]=1 or l[2]=2 or l[3]=3 or l[4]=4 o
desarreglo([5,1,4,3,2]);
desarreglo([5,1,3,2,4]);
```

```
des5: subset(permutations({1,2,3,4,5}),desarreglo);
cardinality(des5);
n:5$ sum((-1)^j*n!/j!,j,2,n);
```

Un alumno de primer curso se va a examinar de cuatro asignaturas: MD, CC, FP y FF. Dispone de los siete días de una semana durante los cuales repasará todas las asignaturas dedicando cada día al estudio de una única asignatura (sin descansar ningún día). ¿Y si consideramos que los días son distintos?

```
La solución del apartado b son las el número de aplicaciones
de días {L,M,X,J,V,S,D} al conjunto de asignaturas {CC,MD,FP,
En primer lugar, hacemos una partición del conjunto de días e
cada una de ellas corresponderá a los días en los que se estu

part74: set_partitions({"L","M","X","J","V","S","D"},4);
El número de particiones es S(7,4), en donde S corresponde a
de segunda especie.
cardinality(part74);
stirling2(7,4);

"part74" es un conjunto, y por lo tanto, el orden de las part
definir las aplicaciones sobreyectivas, debemos asignar cada
Si establecemos el orden [CC,MD,FP,FF], esto corresponde a ge
de cada una de las 4 partes de las distintas particiones. Est
siguiente secuencia de operadores, aunque no podemos visualiz
su excesiva longitud.
En primer lugar, convertimos el conjunto de las particiones e
lpart74: listify(part74)$
llpart74: map(listify,lpart74)$
permpart74: map(permut,llpart74)$
horarios: lreduce(append,permpart74)$
length(horarios);
horarios[278];
Teniendo en cuenta el orden fijado, esto significa que dedica
el martes y viernes a Discreta, el domingo a Programación y e
a Física.
```

```
4!*stirling2(7,4);
length(horarios);
```

1. Una empresa de telecomunicaciones desea instalar en Málaga 210 antenas de telefonía móvil y 600 antenas parabólicas de televisión. La ciudad de Málaga se divide en 40 sectores. Para que no queden zonas sin cobertura es necesario que en cada sector haya un mínimo de 10 antenas de televisión y 4 de telefonía. Por otra parte, las ordenanzas municipales impiden colocar más de 7 antenas de telefonía en cada sector. Determina el número de formas distintas de colocar las antenas cumpliendo las restricciones anteriores, sabiendo que las antenas de televisión son indistinguibles entre sí y las de telefonía son también indistinguibles entre sí, pero obviamente se distinguen unas antenas de un tipo de las de otro.

```
kill(all)$
gen31: sum(z^j,j,0,3)^40,expand$
coeff(gen31,z,50);
```

$$u_0 = 1, u_1 = 2 y u_n = -2u_{n-1} + 3u_{n-2}\ para\ todo\ n \geq 2$$

```
kill(all)$
load(solve_rec)$
ecrec32a: u[n]+2*u[n-1]-3*u[n-2];
solve_rec(ecrec32a,u[n]);
solve_rec(ecrec32a,u[n],u[0]=1,u[1]=2);
factor(r^2+2*r-3);
u[n]:= A+B*(-3)^n;
solve([u[0]=1,u[1]=2],[A,B]);
```

MÁXIMA TEMA 2

Los conjuntos se definen en Maxima tal y como lo hemos hecho antes, delimintando sus elementos entre llaves.

```
(%i1) A: {2,3,5,5,3,-1,2,1};
```

$$\{-1,1,2,3,5\}$$

En este ejemplo podemos observar que Maxima elimina los elementos repetidos aunque nosotros los hayamos escrito. También observamos que ha escrito los elementos, numéricos en este caso, de forma ordenada

También podemos definir conjuntos con el operador set(), que será útil en combinación con otros operadores.

```
(%i2) set(2,2,3,1,5);
```

$$\{1,2,3,5\}$$

Si queremos definir un conjunto de cadenas de caracteres (strings), debemos delimitarlas por comillas, en caso contrario, Maxima lo interpretará como un parámetro. En el siguiente ejemplo vamos a utilizar el parametro xx y también la cadena de caracteres "x".

```
(%i3) B: set(x,"x");
```

$$\{x,x\}$$

Aparentemente, el conjunto está formado por un único elemento repetido. Pero si ahora le damos un valor al parámetro xx, vemos que el conjunto está formado por la cadena 'x' y el valor que tome el parámetro xx.

```
(%i4) ev(B,x=1);
```

$$\{1,x\}$$

Cuando trabajamos con conjuntos grandes, puede ser útil el operador elementp que analiza si un elemento pertenece o no a un conjunto.

```
(%i5) C: {0,1,-1,3,5,-7};
```

$$\{-7,-1,0,1,3,5\}$$

```
(%i6) elementp(3,C);
```

$$\text{true}$$

```
(%i7) elementp(2,C);
```

$$\text{false}$$

En Maxima, disponemos de los operadores subsetp y setequalp para analizar la relación de inclusión o igualdad de dos conjuntos,

```
(%i1) A: {-2,-1,0,2,4,6}$
      B: {-1,0,2}$
      C: {-1,0,1}$
      D: {0,-1,-2,2,4,6}$
(%i2) subsetp(B,A);
```

```
                                    true
(%i3) subsetp(C,A);
                                    false
(%i4) setequalp(A,D);
                                    true
```

El conjunto vacío lo podemos introducir con un par de llaves o con set()set().

```
(%i5) setequalp({},set());
                                    true
(%i6) subsetp({},A);
                                    True
```

UNIÓN

En `Maxima`, la unión de dos conjuntos se determina con el operador `union`.

```
(%i1) A: {-1,1,2,3,5}$
   B: {2,3}$
(%i2) union(A,B)
                                {−1,1,2,3,5}
```

INTERSECCIÓN

El operador de `Maxima` para calcular la intersección de dos conjuntos es `intersection`.

```
(%i1) A: {-1,1,2,3,5}$
      B: {2,4,5}$
(%i2) intersection(A,B);
                                    {2,5}
```

Y `setdifference` determina la diferencia de dos conjuntos

```
(%i3) setdifference({1, 2, 8, 9}-{2, 6, 7});
                                  {1,8,9}
(%i4) setdifference({2, 6, 7}-{1, 2, 8, 9});
                                  {6,7}
```

DETERMINAR CONJUNTOS Y PRIMOS

Podemos definir conjuntos por comprensión en `Maxima` de dos formas, con los operadores `makeset` y `subset`. El operador `makeset` tiene tres argumentos: el primero es una expresión que determina los elementos del conjunto a partir de unos parámetros; el segundo argumento es la lista de los parámetros que se

usan en el argumento anterior y el tercer argumento es una lista con los distintos valores que toman los parámetros para construir los elementos del conjunto.

```
(%i1) makeset(j/k,
              [j,k],
              [[1,a],[1,b],[2,b],[3,c]]);
```

$$\{1a,1b,2b,3c\}$$

```
(%i2) makeset(x^2,[x],{[2],[-2],[3]});
```

$$\{4,9\}$$

El operador `subset` es más parecido a la definición matemática de la construcción por compresión. Ese operado tiene dos argumentos: el primero es un conjunto previamente definido y el segundo es una propiedad, es decir, un operador de `Maxima` que actúe sobre un argumento y cuya salida sea `true` o `false`. Por ejemplo, el operador `evenp` es uno de estos operadores; aplicado a un número, nos devuelve `true` o `false` según el número sea par o impar:

```
(%i3) evenp(4);
```

$$\text{true}$$

```
(%i4) evenp(5);
```

$$\text{false}$$

Vamos a utilizar este operador para determinar el subconjunto de los números pares de un conjunto de números.

```
(%i5) A: {1,27,8,9,12}$
(%i6) subset(A,evenp);
```

$$\{8,12\}$$

En `Maxima` hay varios operadores que podemos usar como propiedades para definir subconjuntos, pero también podemos definir otras propiedades con el operador is()is() cuya sintaxis vemos en el siguiente ejemplo en el que definimos una propiedad que analiza si un número es o no múltiplo de 3.

```
(%i7) prop(x):= is(remainder(x,3)=0)$
(%i8) subset(A,prop);
```

$$\{9,12,27\}$$

Ya hemos dicho anteriormente que `Maxima` trabaja internamente con los conjuntos como si fueran listas, es decir, utilizando algún orden intrínseco de los elementos. De hecho, para definir y trabajar con nuestros propios conjuntos también es preferible utilizar la estructura de lista, es decir, con el operador `makelist` en lugar de `makeset` y con el operador `setify` que convierte listas en conjuntos.

```
(%i9) C: makelist(k^2,k,-5,5);
```

$$[25,16,9,4,1,0,1,4,9,16,25]$$

```
(%i10) setify(C);
```

$$\{0,1,4,9,16,25\}$$

Vemos otro ejemplo, en el que además hacemos uso de la propiedad `primep`, para determinar los números primos menores que 50.

```
(%i11) D: setify(makelist(i,i,1,50))$
(%i12) subset(D,primep);
```

$$\{2,3,5,7,11,13,17,19,23,29,31,37,41,43,47\}$$

El operador de `Maxima` para calcular la el conjunto de las partes o conjunto potencia es `powerset`.

```
(%i1) powerset({0,1,2,3});
```

$$\{\{\},\{0\},\{0,1\},\{0,1,2\},\{0,1,2,3\},\{0,1,3\},\{0,2\},\{0,2,3\},\{0,3\},\{1\},\{1,2\},\{1,2,3\},\{1,3\},\{2\},\{2,3\},\{3\}\}$$

PERMUTACIONES

En `Maxima`, disponemos de algunos operadores relacionados con los modelos de recuento que vamos a estudiar en este tema. Por ejemplo, el operador `permutations` permite obtener el conjunto de todas las permutaciones de los elementos de un conjunto.

```
(%i1) perm3: permutations({1,2,3});
```

$$\{[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]\}$$

Naturalmente, el resultado es un conjunto de listas. En este caso, estamos obteniendo permutaciones de todos los elementos del conjunto y por lo tanto, el número total coincide con el factorial del cardinal del conjunto.

```
(%i2) is(cardinality(perm3)=3!);
```

$$\text{True}$$

Utilizando los operadores para trabajar con conjuntos que hemos aprendido, también podemos obtener variaciones. Por ejemplo, las 2-permutaciones de $A=\{1,2,3\}$ son los elementos de $A\times A$ con las dos componentes distintas y por lo tanto, podemos obtener este conjunto como subconjunto del producto cartesiano.

```
(%i3) distintos(l):= is(l[1]#l[2])$
```

```
(%i4) distintos([1,1]);
```

False

```
(%i5) distintos([2,1]);
```

True

```
(%i6) perm23:
subset(cartesian_product({1,2,3},{1,2,3}),distintos)
;
```

$$\{[1,2],[1,3],[2,1],[2,3],[3,1],[3,2]\}$$

El cardinal de este conjunto es igual a $3\cdot 2=6$3·2=6:

```
(%i7) cardinality(perm23);
```

PERMUTACIONES GENERALIZADAS

En las permutaciones generalizas puede haber elementos repetidos, lo que significa que estamos permutando elementos de una lista. De hecho, el operador `permutations` de `Maxima` se puede aplicar a una lista y en ese caso, nos genera las permutaciones generalizadas.

```
(%i1) perm222: permutations([1,1,2,2,3,3]);
```

$$\{[1,1,2,2,3,3],[1,1,2,3,2,3],\ldots,[3,3,2,1,2,1],[3,3,2,2,1,1]\}$$

La expresión que determina el cardinal de este conjunto se denomina coeficiente multinomial y esa denominación es la que da nombre al operador en `Maxima`.

$$P(n;n_1,n_2,\ldots,n_r)=multinomial\_coeff(n_1,n_2,\ldots,n_r)=n!n_1!\cdot n_2!\cdots$$

```
(%i2) cardinality(perm222)=multinomial_coeff(2,2,2)

=6!/(2!*2!*2!);
```

$$90=90=90$$

COMNINACIONES

Ya habíamos visto que `binomial` es el operador de `Maxima` que determina los número combinatorios: $binomial(n,m)=\binom{n}{m}$. Pero, ¿cómo determinamos los subconjuntos de un conjunto con un determinado número de elementos? Nuevamente, podemos recurrir a los operadores que hemos aprendido hasta ahora y, concretamente, a la forma de determinar subconjuntos definidos por una propiedad. Si queremos obtener los subconjuntos con 3 elementos, podemos usar la siguiente propiedad:

```
(%i1) card3(x):=is(cardinality(x)=3)$
```

De esta forma, los subconjuntos con 3 elementos del conjunto $\{1,2,3,4,5,6\}$ se pueden determinar como sigue:

```
(%i2) subconj63:
subset(powerset({1,2,3,4,5,6}),card3);
```

$$\{\{1,2,3\},\{1,2,4\},\{1,2,5\},\{1,2,6\},\{1,3,4\},\{1,3,5\},\{1,3,6\},\{1,4,5\},\{1,4,6\},\{1,5,6\},\{2,3,4\},\{2,3,5\},\{2,3,6\},\{2,4,5\},\{2,4,6\},\{2,5,6\},\{3,4,5\},\{3,4,6\},\{3,5,6\},\{4,5,6\}\}$$

Naturalmente, el cardinal de este conjunto coincide con (63):

```
(%i3) cardinality(subconj63)=binomial(6,3);
```

$$20=20$$

COMBINACIONES CON REPETICIÓN

No disponemos de un operador en `Maxima` que genere las combinaciones con repetición, pero no es difícil hacerlo con los operadores que hemos aprendido hasta ahora. Concretamente, usaremos las soluciones de las ecuaciones lineales como ejemplo básico.

Por ejemplo, las combinaciones con repetición $CR(3,7)$ son las soluciones positivas de $x_1+x_2+x_3=7$ y por lo tanto, necesitamos generar listas de tres números positivos que sumen 7. Para que el proceso sea fácilmente generalizable, definiremos los operadores sin tener en cuenta la longitud de las misma. Empezamos por definir un operador que determina la suma de los elementos de una lista.

```
(%i1) suma_list(l):=lreduce("+",l)$
(%i2) suma_list([2,3,1,4]);
```

$$9$$

Para seleccionar las listas adecuadas, definimos una propiedad sobre listas, en este caso, que la suma sea igual a 7:

```
(%i3) suma7Q(l):=is(suma_list(l)=7)$
```

Ya podemos construir el subconjunto de las combinaciones con repetición.

```
(%i4) A: {0,1,2,3,4,5,6,7};
(%i5)
comb_rep37:=subset(cartesian_product(A,A,A),suma7Q);
```

{[0,0,7],[0,1,6],[0,2,5],[0,3,4],[0,4,3],[0,5,2],[0,6,1],[0,7,0],[1,0,6],[1,1,5],[1,2,4],[1,3,3],[1,4,2],[1,5,1],[1,6,0],[2,0,5],[2,1,4],[2,2,3],[2,3,2],[2,4,1],[2,5,0],[3,0,4],[3,1,3],[3,2,2],[3,3,1],[3,4,0],[4,0,3],[4,1,2],[4,2,1],[4,3,0],[5,0,2],[5,1,1],[5,2,0],[6,0,1],[6,1,0],[7,0,0]}

```
(%i6) cardinality(comb_rep37);
```

36

```
(%i7) binomial(3+7-1,7);
```

36

NUMEROS STIRLING

En `Maxima` disponemos tanto del operador que calcula los números de Stirling de segunda especie, `stirling2`, como del operador que determina las particiones de un conjunto con un determinado número de partes, `set_partitions`.

```
(%i1) partes74: set_partitions({1,2,3,4,5,6,7},4)$
(%i2) cardinality(partes74)=stirling2(7,4);
```

350=350

FUNCIONES GENERADORAS

Naturalmente, si utilizamos `Maxima`, no necesitaremos realizar las transformaciones del ejemplo anterior. Por ejemplo, vamos a determinar el número de soluciones de la ecuación $x_1+x_2+x_3=30$x1+x2+x3=30 teniendo en cuenta que $1\geq x_1\geq 10$1≥x1≥10, $2\geq x_2\geq 7$2≥x2≥7, $1\geq x_3$1≥x3. Vamos a hacerlo de dos formas. En la primera vamos a utilizar que, dado que la suma de los tres sumandos es 30, ninguno de ellos puede ser mayor que 30, es decir, $x_3\geq 30$x3≥30. De esta forma, la función generadora de este problema es el siguiente polinomio:

```
(%i1) gen:
sum(z^n,n,1,10)*sum(z^n,n,2,7)*sum(z^n,n,1,30),expand;
```

Z^47+3z^46+6z^45+10z^44+15z^43+21z^42+27z^41+33z^40+39z^39+45z^38+50z^37+54z^36+57z^35+59z^34++60z^33+60z^32+60z^31+60z^30+60z^29+60z^28+60z^27++60z^26+60z^25+60z^24+60z^23+60z^22+60z^1+60z^20++60z^19+

$$60z^{18}+59z^{17}+57z^{16}+54z^{15}+50z^{14}+45z^{13}++39z^{12}+33z^{11}+27z^{10}+21z^{9}+15z^{8}+10z^{7}+6z^{6}+3z^{5}+z^{4}$$

Aunque podemos ver fácilmente cual es el coeficiente de $z^{30}z30$, también podemos determinarlo con el operador `coeff`. Hay que tener en cuenta que para que el resultado sea correcto, debemos aplicar este operador a una expresión polinómica en su forma expandida, tal y como estamos haciendo en este ejemplo.

```
(%i2) coeff(gen,z,30);
```

$$60$$

La función generadora anterior solo nos sirve para encontrar el numero de soluciones de $x_1+x_2+x_3=mx1+x2+x3=m$ si $m\geq 30m\geq30$, ya que hemos añadido la restricción $x_3\geq 30x3\geq30$. Si queremos hallar la función generadora que podamos utilizar para cualquier valor de mm con las restricciones que habíamos indicado al principio, $1\geq x_1\geq 101\geq x1\geq10$, $2\geq x_2\geq 72\geq x2\geq7$, $1\geq x_31\geq x3$., tendremos que usar series:

```
(%i3) gen1:
sum(z^n,n,1,10)*sum(z^n,n,2,7)*sum(z^n,n,1,inf)$
```

En este caso, no podemos recurrir directamente el operador `coeff`, ya que la expresión no está expandida:

```
(%i4) coeff(gen1(z),z,30);
```

$$0$$

La alternativa a la opción `expand` cuando trabajamos con series es el operador `taylor`, que determina el polinomio de Taylor de la función generadora hasta el grado que deseemos y que en este caso coincide con la forma expandida truncada en ese grado:

```
(%i5) taylor(gen1,z,0,35);
```

$$z^{4}+3z^{5}+6z^{6}+10z^{7}+15z^{8}+21z^{9}+27z^{10}+33z^{11}+39z^{12}+45z^{13}+50z^{14}++54z^{15}+57z^{16}+59z^{17}+60z^{18}+60z^{19}+60z^{20}+60z^{21}+60z^{22}+60z^{23}++60z^{24}+60z^{25}+60z^{26}+60z^{27}+60z^{28}+60z^{29}+60z^{30}+60z^{31}+60z^{32}++60z^{33}+60z^{34}+60z^{35}+...$$

```
(%i6) coeff(%,z,30);
```

$$60$$

ECUACIONES EN RECURRENCIA

En este primer ejemplo con Maxima vamos a utilizar el paquete solve_rec, que resuelve directamente ecuaciones en recurrencia lineales (homogéneas o no).

```
(%i1) load(solve_rec)$
```

Primero introducimos la ecuación, en donde la sucesión a determinar se escribe como una lista. Si escribimos solo una expresión involucrando elementos de la lista, Maxima entiende que la expresión está igualada a 0. Por ejemplo, la ecuación en recurrencia de orden 3 dada por $18a_n - 21a_{n-1} + 8a_{n-2} - a_{n-3} = 0$ se introduce como sigue:

```
(%i2) ecrec: 18*a[n]-21*a[n-1]+8*a[n-2]-a[n-3]$
```

El operador que resuelve la ecuación es solve_rec y toma como argumentos la ecuación y la incógnita.

```
(%i3) solve_rec(ecrec,a[n]);
```

$$a[n] = \frac{\%k_3 * n + \%k_2}{3^n} + \frac{\%k_1}{2^n}$$

En este caso, nos devuelve la solución general en función de tres parámetros, %k1%k1, %k2%k2 y %k3%k3, ya que la ecuación es de orden 3.

También podemos añadir, como argumentos, las condiciones iniciales de la ecuación y, en tal caso, nos devolverá la solución particular correspondiente.

```
(%i1)
solve_rec(ecrec,a[n],a[0]=1,a[1]=5/6,a[2]=17/36);
```

$$a[n] = \frac{n}{3^n} + \frac{1}{2^n}$$

CON EC.CARACTERÍSTICA

En este ejemplo, vamos resolver una ecuación en recurrencia usando la ecuación característica. Concretamente, vamos a resolver la ecuación que determina la sucesión de Fibonacci:

$$f[n] - f[n-1] - f[n-2] = 0, \quad f[0] = 0, \quad f[1] = 1.$$

La ecuación característica es $r^2 - r - 1 = 0$:

```
(%i1) eccar: r^2-r-1$
(%i2) solve(eccar,r);
```

$$r = -\frac{\sqrt{5}-1}{2}, \quad r = \frac{\sqrt{5}+1}{2}$$

Por lo tanto, el esquema de la solución de la ecuación en recurrencia es:

```
(%i3) sol: A*(-(sqrt(5)-1)/2)^n+B*((sqrt(5)+1)/2)^n$
```

Para hallar los valores de A y B evaluamos esta expresión con $n = 0$ y $n = 1$ e igualamos los resultados a los correspondientes valores iniciales:

```
(%i4) ec1: ev(sol,n=0)=0$
(%i5) ec2: ev(sol,n=1)=1$
```

El operador `solve` nos da los valores de A y B:

```
(%i6) solve([ec1,ec2],[A,B]);
```

$$A = \frac{-1}{\sqrt{5}}, \quad B = \frac{1}{\sqrt{5}}$$

Por lo tanto, la solución de la ecuación en recurrencia que nos da la forma explícita de la sucesión de Fibonacci:

$$a[n] = \frac{-1}{\sqrt{5}} \cdot \left(\frac{-\sqrt{5}+1}{2} \right)^n + \frac{1}{\sqrt{5}} \cdot \left(\frac{\sqrt{5}+1}{2} \right)^n$$

El resultado obtenido con el operador `solve_rec` es naturalmente el mismo.

```
(%i7) load(solve_rec)$
(%i8) solve_rec(f[n]-f[n-1]-f[n-2]=0, f[n],f[0]=0,
f[1]=1.);
```

$$f[n] = \frac{(\sqrt{5}+1)^n}{\sqrt{5} \cdot 2^n} - \frac{(\sqrt{5}-1)^n(-1)^n}{\sqrt{5} \cdot 2^n}$$

EJEMPLO

$a_0 = 5, a_1 = 6, a_n = 3a_{n-1} - 2a_{n-2} - 2^{n-1}$

Dado que $2^n = 1 \cdot 2^n$, la ecuación característica de esta recurrencia es

$$(r^2 - 3r + 2)(r-2)^{0+1} = (r^2 - 3r + 2)(r-2) = 0$$

```
(%i1) load(solve_rec)$
(%i2) solve_rec(a[n] = 3*a[n-1] - 2*a[n-2] - 2^(n-
1),a[n], a[0]=5,a[1]=6);
```

$$a[n] = -n \cdot 2^n + 3 \cdot 2^n + 2$$

```
→   /*Relations using binary predicates*/;

(%i1) R(x, y):= is(remainder(x−y, 3)=0)$

(%i2) R(−1, 3);
(%o2) false

(%i3) R(2, 5);
(%o3) true

(%i4) R(5, 2);
(%o4) true

→   /*Relations as a set of pairs*/;

(%i5) S: {[1, 2], [2, 4], [3, 2], [4, 1]}$

(%i6) Sp(x, y):= elementp([x, y], S)$

(%i7) Sp(2, 4);
(%o7) true

(%i8) Sp(1, 4);
(%o8) false

(%i9) Sp(1, 2);
(%o9) true
```

```
→   /*Boolean sum*/;

(%i10) infix("+b")$

(%i11) m +b n := m+ n−m·n$

(%i12) 1 +b 1;
(%o12) 1

→   /*Matrix sum*/;

(%i13) matrix([1, 0], [0, 1]) +b matrix([1, 0], [1, 0]);
(%o13) ⎛1  0⎞
       ⎝1  1⎠

→   /*Matrix product*/;

(%i14) matrix_element_add: lambda([[x]], lreduce ("+b", x))$

(%i15) matrix([1, 0, 1], [0, 1, 0]).matrix([0, 1], [1, 0], [0, 0]);
(%o15) ⎛0  1⎞
       ⎝1  0⎠

(%i16) matrix([0, 1], [1, 0]).matrix([0, 0], [1, 0]);
(%o16) ⎛1  0⎞
       ⎝0  0⎠
```

→ /*Properties of a relation*/;

(%i17) S: {[1, 2], [2, 4], [3, 2], [4, 1]}$

(%i18) MS: genmatrix(lambda([i, j], if elementp([i, j], S) then 1 else 0), 4, 4);

MS
$$\begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix}$$

→ /*The matrix is not symmetric*/;

(%i19) is(MS=transpose(MS));

(%o19) false

→ /*Identity matrix*/;

(%i20) ident(4);

(%o20)
$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

→ /*The matrix is not reflexive*/;

(%i21) is(%=%·MS);

(%o21) false

→ /*Square matrix with all elements equal to ones*/;

(%i22) ones(n):= genmatrix(lambda([i, j], 1), n, n)$

(%i23) ones(4);

$$(\%o23)\ \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

→ /*The matrix is antisymmetric*/;

(%i24) MS·transpose(MS);

$$(\%o24)\ \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

(%i25) is(%=%·ident(4));

(%o25) true

```
→  /*The matrix is not transitive*/;

(%i26) infix ("+b")$

(%i27) m +b n:= m+n−m·n$

(%i28) matrix_element_add: lambda([[x]], lreduce("+b", x))$

(%i29) MS.MS;
```

$$(\%o29)\begin{pmatrix}0&0&0&1\\1&0&0&0\\0&0&0&1\\0&1&0&0\end{pmatrix}$$

```
(%i30) is(%=%·MS);
(%o30) false

→  /*The matrix is not connected*/;

(%i31) is(MS +b transpose(MS)=ones(4));
(%o31) false

→  /*Equivalence classes*/;

(%i32) conj: setify(makelist(i, i, −10, 10));
conj  {−10,−9,−8,−7,−6,−5,−4,−3,−2,−1,0,1,2,3,4,5,6,7,8,9,10}

(%i33) R(x, y) := is(remainder(x−y, 3)=0)$

(%i34) equiv_classes(conj, R);
(%o34) {{−10,−7,−4,−1,2,5,8},{−9,−6,−3,0,3,6,9},{−8,−5,−2,1,4,7,10}}

(%i35) warshall(m):= warshall_aux(m, length(m), 1)$

(%i36) warshall_aux(m, n, k):= if k=n+1 then m else warshall_aux(genmatrix(lambda([i, j], max(m[i][j], min(m[i][k], m[k][j]))), n, n), n, k+1)$

(%i37) M0: matrix([1, 0, 0, 1],
                  [0, 0, 0, 1],
                  [1, 0, 0, 0],
                  [0, 1, 0, 0])$

(%i38) trace(warshall_aux)$
```

(%i39) **warshall(M0);**

1 Enter warshall_aux $\left[\begin{pmatrix}1&0&0&1\\0&0&0&1\\1&0&0&0\\0&1&0&0\end{pmatrix},4,1\right]$

.2 Enter warshall_aux $\left[\begin{pmatrix}1&0&0&1\\0&0&0&1\\1&0&0&1\\0&1&0&0\end{pmatrix},4,2\right]$

..3 Enter warshall_aux $\left[\begin{pmatrix}1&0&0&1\\0&0&0&1\\1&0&0&1\\0&1&0&1\end{pmatrix},4,3\right]$

...4 Enter warshall_aux $\left[\begin{pmatrix}1&0&0&1\\0&0&0&1\\1&0&0&1\\0&1&0&1\end{pmatrix},4,4\right]$

....5 Enter warshall_aux $\left[\begin{pmatrix}1&1&0&1\\0&1&0&1\\1&1&0&1\\0&1&0&1\end{pmatrix},4,5\right]$

....5 Exit warshall_aux $\begin{pmatrix} 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 \end{pmatrix}$

...4 Exit warshall_aux $\begin{pmatrix} 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 \end{pmatrix}$

..3 Exit warshall_aux $\begin{pmatrix} 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 \end{pmatrix}$

.2 Exit warshall_aux $\begin{pmatrix} 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 \end{pmatrix}$

1 Exit warshall_aux $\begin{pmatrix} 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 \end{pmatrix}$

(%o39) $\begin{pmatrix} 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 \end{pmatrix}$

TEMA 2: CONJUNTOS, FUNCIONES, RECUENTO Y RECURRENCIA

RECUENTO

1 Variaciones

EJEMPLOS: n=4 ; k=2

1.1 Con repetición

```
4^2;
```

16

1.2 Sin repetición

```
load(functs)$
permutation(4,2);
```

12

2 Combinaciones

EJEMPLOS: n=4 ; k=2

2.1 Con repetición

Estructura: binomial(n+k-1,k)

```
binomial(4+2-1,2);
```

10

2.2 Sin repetición

```
binomial(4,2);
```

6

3 Permutaciones

EJEMPLOS: n=4

3.1 Con repetición

(número de veces que aparece cada letra: c-->1 ; a-->2; s-->1)

```
multinomial_coeff(2,1,1);
```

12

```
permutations([c,a,s,a]);
```

```
{[a,a,c,s],[a,a,s,c],[a,c,a,s],[a,c,s,a],[a,s,a,c],[a,s,c,a]
,[c,a,a,s],[c,a,s,a],[c,s,a,a],[s,a,a,c],[s,a,c,a],[s,c,a,a]}
```

3.2 Sin repetición

```
4!;
```

24

```
permutations({c,a,s,a});
```

{[a,c,s],[a,s,c],[c,a,s],[c,s,a],[s,a,c],[s,c,a]}

4 Números de Stirling

EJEMPLO: n=3; k=2 --> Si las cajas fuesen iguales

4.1 Opción 1

```
stirling2(3,2);
```

3

4.2 Opción 2 (calculando las particiones de un conjunto formadas por dos conjuntos)

```
partes32:set_partitions({1,2,3},2);
```

$$\left\{\{\{1\},\{2,3\}\},\{\{1,2\},\{3\}\},\{\{1,3\},\{2\}\}\right\}$$

```
cardinality(partes32);
```

3

EJEMPLO: n=3; k=2 --> Si las cajas fuesen distintas (se está calculando el número de funciones sobreyectivas de 3 elementos de A en 2 elementos de B)

```
factorial(2)·stirling2(3,2);
```

6

5 Funciones generatrices

5.1 Ejemplo 1

f(x):=x1+x2+x3=5

OPCIÓN 1 (0<=xi<=10):

```
gen1:sum(x^n,n,0,10)^3,expand;
```

$$x^{30}+3x^{29}+6x^{28}+10x^{27}+15x^{26}+21x^{25}+28x^{24}+36x^{23}+45x^{22}+55x^{21}+66x^{20}+75x^{19}+82x^{18}+87x^{17}+90x^{16}+91x^{15}+90x^{14}+87x^{13}+82x^{12}+75x^{11}+66x^{10}+55x^{9}+45x^{8}+36x^{7}+28x^{6}+21x^{5}+15x^{4}+10x^{3}+6x^{2}+3x+1$$

```
coeff(gen1,x,5);
```

$$21$$

OPCIÓN 2 (0>=xi)

```
gen2:sum(x^n,n,0,inf)^3;
```

$$\left(\sum_{n=0}^{\infty} x^{n}\right)^{3}$$

```
tay2:taylor(gen2,x,0,10);
```

$$1+3\,x+6\,x^{2}+10\,x^{3}+15\,x^{4}+21\,x^{5}+28\,x^{6}+36\,x^{7}+45\,x^{8}+55\,x^{9}+66\,x^{10}+\ldots$$

```
coeff(tay2,x,5);
```

$$21$$

5.2 Ejemplo 2

f(x): x1+x2+x3+x4=24 ; x2,x3 -->nºpares ; x4>=6 ;
2<=x3<10

```
gen3:sum(x^(2·n),n,0,inf)^2·sum(x^n,n,6,inf)·sum(x^n,n,2,9)$
```

```
tay3:taylor(gen3,x,0,30);
```

$$x^{8}+2\,x^{9}+5\,x^{10}+8\,x^{11}+14\,x^{12}+20\,x^{13}+30\,x^{14}+40\,x^{15}+54\,x^{16}+68\,x^{17}+86\,x^{18}+104\,x^{19}+126\,x^{20}+148\,x^{21}+174\,x^{22}+200\,x^{23}+230\,x^{24}+260\,x^{25}+294\,x^{26}+328\,x^{27}+366\,x^{28}+404\,x^{29}+446\,x^{30}+\ldots$$

```
coeff(tay3,x,24);
```

$$230$$

RECURRENCIA

EJEMPLO:
a_(n+4) − 5a_(n+3) + 6a_(n+2) + 4a_(n+1) − 8a_(n) = 0
a_0 =0 ; a_1 =−9 ; a_2 =−1 ; a_3 =21

1 Opción 1

```
eccar1:x^4−5·x^3+6·x^2+4·x−8$
```

NOTA: eccar=ecuación característica

```
solve(eccar1,x);
```

$$[x=-1,x=2]$$

Como el polinomio es de grado 4 y hemos obtenido 2 soluciones, significa que están repetidas. Por tanto calculamos su multiplicidad:

```
multiplicities;
```

$$[1,3]$$

Significa que el (-1) tiene multiplicidad 1 y el 2 multiplicidad 3, es decir: eccar1 = (x+1)(x-2)(x-2)(x-2) = (x+1)^1 * (x-2)^3

```
sol:(A·n^2+B·n+C)·2^n+D·(−1)^n$
ec1:ev(sol,n=0)=0$
ec2:ev(sol,n=1)=−9$
ec3:ev(sol,n=2)=−1$
ec4:ev(sol,n=3)=21$
solve([ec1,ec2,ec3,ec4],[A,B,C,D]);
```

$$[[A=1,B=-1,C=-3,D=3]]$$

```
e:first(%);
```

$$[A=1,B=-1,C=-3,D=3]$$

```
ev(sol,e),simplify;
```

$$(n^2-n-3)\,2^n+3\,(-1)^n$$

2 Opción 2 -->función predeterminada de wxMaxima

a_(n+4) − 5a_(n+3) + 6a_(n+2) + 4a_(n+1) − 8a_(n) = 0
a_0 =0 ; a_1 =−9 ; a_2 =−1 ; a_3 =21

```
load(solve_rec)$
```

ESTRUCTURA: solve_rec(función entera, "a[n]",condiciones)

```
solve_rec(a[n+4]−5·a[n+3]+6·a[n+2]+4·a[n+1]−8·a[n]=0,a[n],a[0]=0,a[1]=−9, a[2]=
```

$$a_n=(n^2-n-3)\,2^n+3\,(-1)^n$$

EXTRAS

1 Ejercicio1

Hallar cuantos números enteros no superiores a 100 son primos:

1-Enumerar los 100 primeros números:

```
num100:setify(makelist(i,i,1,100));
```

```
{1 ,2 ,3 ,4 ,5 ,6 ,7 ,8 ,9 ,10 ,11 ,12 ,13 ,14 ,15 ,16 ,17 ,18 ,19 ,20 ,21
,22 ,23 ,24 ,25 ,26 ,27 ,28 ,29 ,30 ,31 ,32 ,33 ,34 ,35 ,36 ,37 ,38 ,39 ,40 ,41 ,
42 ,43 ,44 ,45 ,46 ,47 ,48 ,49 ,50 ,51 ,52 ,53 ,54 ,55 ,56 ,57 ,58 ,59 ,60 ,61 ,
62 ,63 ,64 ,65 ,66 ,67 ,68 ,69 ,70 ,71 ,72 ,73 ,74 ,75 ,76 ,77 ,78 ,79 ,80 ,81 ,
82 ,83 ,84 ,85 ,86 ,87 ,88 ,89 ,90 ,91 ,92 ,93 ,94 ,95 ,96 ,97 ,98 ,99 ,100 }
```

2-Enumerar los primos entre esos 100 números:

```
primos:subset(num100,primep);
```

{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}

3-Contar cuántos primos hay:

```
cardinality(primos);
```

25

2 Ejercicio 2

Número de enteros positivos menores que 600 que son coprimos con 600:

1-Enumerar los 100 primeros números:

```
num600:setify(makelist(i,i,1,600))$
```

2-Definir la función para hallar los coprimos:

```
cop600(x):=is(gcd(x,600)=1);
```

cop600(x):=is(gcd(x,600)=1)

3-Enumerar los coprimos entre los 600 números:

```
coprimos:subset(num600,cop600)$
```

4-Contar cuántos coprimos hay:

```
cardinality(coprimos);
```

160

Tema 3

INICIO

```
kill(all)$
load(graphs)$
la2max1(lst,p,s):=
    if lst=[] then s else
    (h:first(lst), if h<p then la2max1(rest(lst),p,s) else
                               la2max1(rest(lst),p,cons([p,h
la2max2(lst,c,s):= if lst=[] then s else
la2max2(rest(lst),c+1,append(la2max1(first(lst),c,[]),s))$
la2max(la):= la2max2(la,1,[])$
crear_grafo(g):= block([l:length(g),i],
                create_graph(makelist(i,i,1,l),la2max(g)
                  ))$
```

Paquete

```
load(graphs)$
```

Crear grafo

```
ej2a: [[2,3,4,7],[1,3,4,5],[1,2,4,6,7],[1,2,3,5,6],[2,4,6,7],
```

Planaridad

```
is_planar(ej2a)
```

Bipartito

```
bipartite(ej2a)
```

Obtener grafo isomorfo

```
max_clique(ej2a);
```

Dibujar grafo aleatorio

```
draw_graph(random_graph(e,.g));
siendo e el número de vértices y g el grado máximo
```

Dibujar grafo declarado

```
draw_graph(gr2a,
    vertex_size=4,show_id=true,redraw=true,
    fixed_vertices=[1,7,6,5,4,3,2]);
```

Dibujar grafo con aristas coloreadas

```
draw_graph(gr2a,
    vertex_size=4,show_id=true,redraw=true,
    fixed_vertices=[1,7,6,5,4,3,2],
    show_edges=[[1,2],[1,3],[1,4],[1,7],[2,3],[2,4],[2,5],[3,
);
```

Número cromático

```
chromatic_number(gr4);
```

Obtener la coloración

```
vc4: vertex_coloring(gr4);
```

Obtener lista coloración

```
col4: coloracion(vc4);
```

Dibujar grafo con vértices coloreados

```
draw_graph(gr4,
    vertex_size=4,show_id=true,redraw=true,
    fixed_vertices=[1,2,5,6,8,7,3,4],
    vertex_partition=col4);
```

Grafo coloreado completo

```
ej4:[[2,3,4],[1,5,6,8],[1,4,7],[1,3,5],[2,4,6,7,8],[2,5,8],[3
gr4: crear_grafo(ej4);
chromatic_number(gr4);
vc4: vertex_coloring(gr4);
coloracion(vc) := makelist(colorX(k,vc[2],[]),k,1,vc[1])$
colorX(k,ls,s):= if ls=[] then s
     else if first(ls)[2]=k then colorX(k,rest(ls),cons(first
     else colorX(k,rest(ls),s)$
col4: coloracion(vc4);
draw_graph(gr4,
    vertex_size=4,show_id=true,redraw=true,
    fixed_vertices=[1,2,5,6,8,7,3,4],
    vertex_partition=col4);
```

Grafo de lista de adyacencia

```
gpr: from_adjacency_matrix(mat);
```

Grafo con aristas con peso

```
gr13:create_graph([1,2,3,4,5,6,7,8],[
        [[1,2],3],
        [[1,3],5],
        [[1,4],6],
        [[2,3],1],
        [[2,5],5],
        [[2,6],1],
```

```
        [[3,4],2],
        [[3,6],2],
        [[4,6],1],
        [[4,7],2],
        [[5,6],2],
        [[5,8],3],
        [[6,7],4],
        [[6,8],8],
        [[7,8],1]
        ])$
draw_graph(gr13,vertex_size=4,
           show_id=true, show_weight=true);
mst13: minimum_spanning_tree(gr13);
draw_graph(gr13,vertex_size=4,
    show_id=true,show_weight=true,
    show_edges=edges(mst13));
shortest_weighted_path(1,2,gr13);
```

grafo definido mediante su lista de adyacencia

```
gr:[[7,6,5,2],[7,1],[6],[5],[7,4,1],[7,3,1],[6,5,2,1]];
//[[7,6,5,2],[7,1],[6],[5],[7,4,1],[7,3,1],[6,5,2,1]]
la2max1(lst,p,s):=
    if lst=[] then s else
    (h:first(lst), if h<p then la2max1(rest(lst),p,s) else
                               la2max1(rest(lst),p,cons([p,h
la2max2(lst,c,s):= if lst=[] then s else
la2max2(rest(lst),c+1,append(la2max1(first(lst),c,[]),s))$
la2max(la):= la2max2(la,1,[])$
la2max(gr);
//[[6,7],[5,7],[4,5],[3,6],[2,7],[1,2],[1,5],[1,6],[1,7]]
load(graphs)$
grmax: create_graph([1,2,3,4,5,6,7],
    [[6,7],[5,7],[4,5],[3,6],[2,7],[1,2],[1,5],[1,6],[1,7]]);
print_graph(grmax)
crear_grafo(g):= block([l:length(g),i],
                create_graph(makelist(i,i,1,l),la2max(g)
                  ))$
```

```
grmax: crear_grafo(gr)$
print_graph(grmax); // devuelve la lista de adyancencia
draw_graph(grmax,
    vertex_size=4,show_id=true,redraw=true,
    fixed_vertices=[2,7,6,3,4,5,1]); //Pinta el gráfico
```

Grafo completo

```
complete_graph(9); // 9 es el número de vértices
```

Grafo completo bipartito

```
complete_bipartite_graph(6,12);
```

Comprobar Bipartito

```
ej10: [[2,4,5,8],[1,3,6,7],[2,4,5,8],[1,3,6,7],[1,3,6,7],[2,4
    [2,4,5,8],[1,3,6,7]];
grej10: crear_grafo(ej10);
bipartition(grej10);
```

Ejercicio

a	b	c	d	e	f
b	a	b	a	b	a
d	c	d	c	d	c
f	e	f	e	f	e

```
kill(all)$
load(graphs)$
la2max1(lst,p,s):=
    if lst=[] then s else
    (h:first(lst), if h<p then la2max1(rest(lst),p,s) else
                               la2max1(rest(lst),p,cons([p,h
```

```
la2max2(lst,c,s):= if lst=[] then s else
la2max2(rest(lst),c+1,append(la2max1(first(lst),c,[]),s))$
la2max(la):= la2max2(la,1,[])$
crear_grafo(g):= block([l:length(g),i],
                create_graph(makelist(i,i,1,l),la2max(g)
                  ))$

//Creamos el array
ej4:[[2,4,6],[1,3,5],[2,4,6],[1,3,5],[2,4,6],[1,3,5]]
//Creamos el grafo
grej4: crear_grafo(ej4);
is_bipartite(grej4); //Comprobamos si es bipartito
draw_graph(grej4);
draw_graph(minimum_spanning_tree (grej4));
draw_graph(grej4,
    vertex_size=4,show_id=true,redraw=true,
    show_edges=edges(minimum_spanning_tree (grej4)),
    fixed_vertices=[6,1,3,2,4,5]);
```

Grado grafos:

```
ej7a: [[2,3,5,6],[1,3,4,6],[1,2,4,5],[2,3,5,6],[1,3,4,6],[1,2
gr7a: crear_grafo(ej7a);
degree_sequence(gr7a)
```

Obtener isomorfismo

```
ej7a: [[2,3,5,6],[1,3,4,6],[1,2,4,5],[2,3,5,6],[1,3,4,6],[1,2
gr7a: crear_grafo(ej7a);
degree_sequence(gr7a)
isomorphism(gr7a,gr7b);
```

Obtener matriz de adyacencia (Cond de Ore)

```
ej9:[[2,3,5,6],[1,3,4,6],[1,2,4,5],[2,3,7],[1,3,7],[1,2,7],[4
grej9: crear_grafo(ej9);
amgr9: adjacency_matrix(grej9);
```

```
//Ver si cumple la condición de Ore
genmatrix(lambda([i,j],if amgr9[i][j]=0 then
        sum(amgr9[i,k],k,1,7)+sum(amgr9[k,j],k,1,7) else 7),
```

Ciclo hamilton

```
ej9:[[2,3,5,6],[1,3,4,6],[1,2,4,5],[2,3,7],[1,3,7],[1,2,7],[4
grej9: crear_grafo(ej9);
hamilton_cycle(grej9);
draw_graph(grej9,
    vertex_size=4,show_id=true,redraw=true,
    show_edges=vertices_to_path(hamilton_cycle(grej9)),
    fixed_vertices=[1,2,4,3,5,7,6]);
```

EULERIANO

- Si hay vértices de grao impar, el grafo no es euleriano
- Si hay más de dos vertices de grafo impar, tampoco es semieuleriano
- Hay cuatro vértices de grado 3, que es estrictamente menor que 7/2, por lo que no se verifica la condición de Dirac y no podemos concluir que sea hamiltoniano.

Grafos

Antes de hacer cualquier ejercicio de máxima con grafos tendremos que añadir el paquete de grafos

```
load(graphs)$
```

Vértices = números naturales
Aristas = listas de longitud 2

Definir un grafo:

create_graph()

Definimos un grafo a partir de la lista de vértices y su lista de aristas.
1º número de vértices
2º lista de listas de longitud 2 que definen las aristas del grafo

```
g1: create_graph(5, [[1,2],[1,3],[2,3],[0,4]]);
```

Lista de adyacencia de un grafo (escritas como columa):

```
print_graph(g1);
```

Representación matricial de un grafo:

```
mg1: adjacency_matrix(g1);
```

Para dibujar un grafo usamos el comando:

```
draw_graph(g1);
```

Para que aparezca el número que corresponde a cada vértice aumentamos el tamaño de los vértices (vertex_sixe) y mostramos el número del vértice (show_id=true)

```
draw_graph(g1, vertex_size=5, show_id=true);
```

Si queremos decir nosotros el nombre que queremos asignar a cada vértice lo escribimos en el primer argumento del comando create_graph()

```
g2: create_graph([1,2,3,4,5,6,7,8], [[1,2], [1,6], [1,8],[1,3],[2,4],[2,5],[2,7],[3,4],[3,5],[3,7],[4,6],[4,8],[5,6],
        [5,8],[6,7],[7,8]]);

print_graph(g2);

draw_graph(g2, vertex_size=4, show_id=true);
```

Para crear un grafo dada su matriz de adyacencia cargamos la matriz (nombrando los vértices empezando por el cero) y usamos el comando (from_adjacency_matrix)

```
mat: matrix([0,0,1,1,1],[0,0,1,0,1],[1,1,0,0,0],[1,0,0,0,0],[1,1,0,0,0]);

g3: from_adjacency_matrix(mat);

print_graph(g3);

draw_graph(g3, vertex_size=4, show_id=true);
```

1 *Propiedades*

1.1 Grados

Para ver la secuencia de grados de los vértices usamos el comando (degree_sequence())

```
degree_sequence(g1);

degree_sequence(g2);

degree_sequence(g3);
```

1.2 Conexo

Analizar si un grafo es conexo (is_connected())

```
is_connected(g1);

is_connected(g2);

is_connected(g3);
```

Saber las componentes conexas de un grafo (connected_components())

```
connected_components(g1);

connected_components(g2);

connected_components(g3);
```

1.3 Bipartido

Para saber si un grafo es bipartito o no (is_bipartite())

```
is_bipartite(g1);

is_bipartite(g2);

is_bipartite(g3);
```

Para saber las dos particiones (bipartition()) -> Si no es bipartito devuelve lista vacía

```
bipartition(g1);

bipartition(g2);

bipartition(g3);
```

1.4 Hamiltoniano

Para saber si un grafo es hamiltoniano o no y obtener su ciclo en caso de tenerlo usamos el comando (hamilton_cycle())

```
k33: complete_bipartite_graph(3,3);

hk33: hamilton_cycle(k33);
```

Para dibujar el ciclo de hamilton usamos el comando (show_edges) en el operador (draw_graph) y que máxima resalte el ciclo de hamilton

```
draw_graph(k33, show_edges=vertices_to_cycle(hk33));
```

1.5 Planaridad

Para estudiar si un grafo es plano o no usamos el comando (is_planar())

```
g5: create_graph(7, [[0,1],[0,5],[1,2],[1,4],[2,3], [0,3], [2,5],[3,6],[4,5],[4,6],[5,6]])$

is_planar(g5);
```

Si queremos forzar a máxima a dibujar el grafo en su forma plana podemos añadir el comando (planar_embedding) en la función (draw_graph)

```
draw_graph(complete_graph(4), redraw=true, program=planar_embedding);
```

1.6 Coloración

Para calcular el número cromático de un grafo usamos el comando (chromatic_number())

```
gcol: create_graph([1,2,3,4,5,6,7,8],[[3, 4],[4, 8],[2,5],[1,8],
        [5,6],[7,8],[4,7],[2,6],[1,4],[3,7],[2,7],[6,8],[2,3],[3,5],[1,6],[1,5]])$

chromatic_number(gcol);
```

Para saber los colores que se asocian a cada vértice usamos el comando (vertex_coloring())

```
vertex_coloring(gcol);
```

```
color 1: 4,6
color 2: 3,8
color 3: 1,2
color 4: 5,7
```

Además, si queremos ver los vértices coloreados dibujados usamos el comando (vertex_partition) en la función (draw_graph)

```
draw_graph(gcol, vertex_size=4, show_id=true, vertex_partition=[[4,6],[3,8],[1,2],[5,7]]);
```

2 Tipos

2.1 Completos

Para dibujar un grafo completo usamos el comando (draw_graph) y en el primer argumento (complete_graph())

```
draw_graph(complete_graph(7), vertex_size=3, show_id=true);
```

2.2 Ciclos

Para dibujar un grafo completo usamos el comando (draw_graph) y en el primer argumento (cycle_graph())

```
draw_graph(cycle_graph(8), vertex_size=3, show_id=true);
```

2.3 Rueda

Para dibujar un grafo completo usamos el comando (draw_graph) y en el primer argumento (wheel_graph())
En los grafos rueda n tienen n+1 vértices

```
draw_graph(wheel_graph(8), vertex_size=3, show_id=true);
```

2.4 n-Cubos

Para dibujar un grafo completo usamos el comando (draw_graph) y en el primer argumento (cube_graph())

```
draw_graph(cube_graph(4), vertex_size=3, show_id=true);
```

2.5 Bipartito completo

Para dibujar un grafo completo usamos el comando (draw_graph) y en el primer argumento (complete_bipartite_graph(,))
En el paréntesis escribimos el número de vértices que hay en cada partición

```
draw_graph(complete_bipartite_graph(4,3), vertex_size=3, show_id=true);
```

3 Árboles

Para ver si un grafo es un árbol o no usamos el comando (is_tree())

```
is_tree(cube_graph(3));
```

3.1 Árbol generador

Para determinar el árbol generador de un grafo usamos el comando (minimum_spanning_tree()))

```
gt: create_graph([1,2,3,4,5,6,7],[[1,2],[1,6],[2,3],[2,4],[2,6],[2,7],[3,4],[4,5],[4,7],
       [5,6],[5,7],[6,7]]);
```

3.2 Árbol generador minimo

```
sgt: minimum_spanning_tree(gt);
```

Usamos la función (edges()) para mostrar las aristas que forman el árbol generador

```
draw_graph(gt, vertex_size=4, show_id=true, show_edges=edges(sgt));
```

4 *Grafos ponderados*

Para definir un grafo ponderado usamos el mismo comando que para cualquier grafo (create_graph) pero cuando escribamos las aristas de dicho grado, habrá dos elementos: el primero será la arista en si y el segundo el peso de dicha arista

```
grp: create_graph([1,2,3,4,5,6], [[[1,2],2],[[1,3],3],[[2,4],5],[[2,5],2],[[3,5],5],
    [[4,5],1],[[4,6],4],[[5,6],2]]);
```

Para que al dibujar el grafo nos muestre los pesos usamos el comando (show_weight=true)

```
draw_graph(grp, vertex_size=3, show_id=true, show_weight=true);
```

4.1 Dikjstra

Para calcular el camino de longitud minima entre dos vértices ponderados usamos el comando (shortest_weighted_path(v2,v1,g)) donde v1 y v2 son los vértices y g el grafo del que queremos hallar el camino

```
dijkgrp: shortest_weighted_path(1,4,grp);
```

Si queremos visualizar ese camino usamos la función (show_edges=) en la función de dibujar el grafo

```
draw_graph(grp, vertex_size=3, show_id=true, show_weigh=true, show_edges=vertices_to_path(dijkgrp[2]));
```

4.2 Árbol generador mínimo

Usamos el mismo comando que en el caso de los árboles generadores

```
mstgrp:  minimum_spanning_tree(grp);
```

```
draw_graph(grp, vertex_size=4, show_id=true, show_weight=true, show_edges=edges(mstgrp));
```

CÓDIGOS MÁXIMA TEMA 3

Dado que una relación es un conjunto, podemos utilizar los operadores de `Maxima`, que ya conocemos, para definir nuevas relaciones. Por ejemplo, el operador `is` permite definir relaciones usando predicados binarios:

```
(%i1) R(x,y) := is(remainder(x-y,3)=0)$
```

$R(x,y)$ es verdadero o "x está relacionado con y por R" si 3 divide a $x-y$

```
(%i2) R(-1,3);
```

false

```
(%i3) R(2,5);
```

true

También podemos definir las relaciones como un conjunto de pares.

```
(%i4) S: {[1,2],[2,4],[3,2],[4,1]}$
```

El operador `elementp` permite convertir la definición anterior en un predicado:

```
(%i5) Sp(x,y):= elementp([x,y],S)$ (%i6) Sp(2,4);
```

true

```
(%i7) Sp(1,4);
```

false

E incluso construir la matriz de adyacencia:

```
(%i8) M_S: genmatrix(lambda([i,j],
    if elementp([i,j],S) then 1
    else 0), 4,4);
```

$$\begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix}$$

En Maxima, podemos trabajar con las constantes booleanas `true` y `false`, y el sistema dispone de varios operadores sobre estos valores. Sin embargo, es más simple y práctico trabajar con los números 0 y 1 y definir las operaciones booleanas sobre ellos. Solo necesitamos definir la "suma booleana", ya que el "producto booleano" coincide con el producto numérico. Vamos a utilizar la secuencia "+b" escrita de forma infija para representar la suma booleana:

```
(%i1) infix("+b ")$ (%i2) m +b n := m+n-m*n$
```

De esta forma, podremos usar el operador "+b" también para operar matrices booleanas elemento a elemento.

```
(%i3) 1 +b 1;
```

$$1$$

```
(%i4) matrix([1,0],[0,1]) +b matrix([1,0],[1,0]);
```

$$\begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}$$

También podemos redefinir el producto estándar de matrices numéricas para que opere con la suma y producto booleano tal y como hemos visto anteriormente. Para ello, Maxima dispone de dos constantes con las que podemos establecer cual debe ser la suma y cual el producto en el producto de matrices:
con `matrix_element_add` podemos establecer la suma y con `matrix_element_mult` el producto. En este caso, solo necesitamos cambiar la suma, puesto que sí estamos usando el producto numérico.

```
(%i5) matrix_element_add:
        lambda ([[x]], lreduce ("+b ", x))$
```

A partir de aquí, el operador producto matricial (representado por un punto bajo) se comportará como el producto de matrices booleanas. Obsérvese que hemos tenido que recurrir al operador `lreduce`, puesto que "+b" se ha definido para dos argumentos y en el producto matricial se aplicará a una cantidad arbitraria de sumandos.

```
(%i6) matrix([1,0,1],[0,1,0]).
    matrix([0,1],[1,0],[0,0]);
```

$$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

Vamos a estudiar las propiedades de la siguiente relación.

```
(%i1)  S: {[1,2],[2,4],[3,2],[4,1]}$
(%i2)  MS: genmatrix(lambda([i,j],
                if elementp([i,j],S) then 1 else 0),
                4,4);
```

$$\begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix}$$

Por ejemplo, la relación no es simétrica, puesto que no lo es la matriz:

```
(%i3) is(MS=transpose(MS));
```

$$\text{false}\text{false}$$

Como hemos visto anteriormente, para estudiar algunas propiedades, necesitamos algunas matrices auxiliares. Por ejemplo, la matriz identidad, cuyos elementos en la diagonal principal son iguales a 1 y el resto son iguales a 0:

```
(%i4) ident(4);
```

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

También necesitamos las matrices cuadradas con todos lo elementos iguales a 1, que podemos definir fácilmente:

```
(%i5) unos(n) := genmatrix(lambda([i,j],1),n,n)$
(%i6) unos(4);
```

$$\begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

La relación SS es antisimétrica, ya que el producto booleano de M_SMS por M_{tS}MSt elemento a elemento, es menor que la matriz diagonal.

```
(%i7) MS*transpose(MS);
```

$$\begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

```
(%i8) is(%=%*ident(4));
```

truetrue

Para comprobar la propiedad transitiva, tenemos que usar el producto matricial considerando la suma y el producto booleano.

```
(%i9) infix("+b")$
(%i10) m +b n := m+n-m*n$
(%i11) matrix_element_add:
        lambda ([[x]], lreduce ("+b", x))$
(%i12) MS.MS;
```

$$\begin{pmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{pmatrix}$$

Por lo tanto, la relación SS no es transitiva.

```
(%i13) is(%=%*MS);
```

falsefalse

En estas comprobaciones hemos tenido en cuenta que el producto booleano coincide con la función mínimo y
que $\min(a,b)$=amin(a,b)=a si y solamente si $a \leq b$a≤b.

Finalmente, también podemos comprobar que la relación no es conexa.

```
(%i14) is(MS +b transpose(MS)=unos(4));
```

false

El operador equiv_classes(A,R)equiv_classes(A,R) determina las clases de equivalencia en el conjunto AA dadas por la relación RR definida con un predicado.

```
(%i1) conj: setify(makelist(i,i,-10,10));
```

$$\{-10,-9,-8,-7,-6,-5,-4,-3,-2,-1,0,1,2,3,4,5,6,7,8,9,10\}\{-10,-9,-8,-7,-6,-5,-4,-3,-2,-1,0,1,2,3,4,5,6,7,8,9,10\}$$

```
(%i2) R(x,y) := is(remainder(x-y,3)=0)$
(%i3) equiv_classes(conj,R);
```

$$\{\{-10,-7,-4,-1,2,5,8\},\{-9,-6,-3,0,3,6,9\},\{-8,-5,-2,1,4,7,10\}\}$$

Aunque Maxima no incluye ningún operador o paquete específico para el algoritmo de Warshall, no es difícil escribir una función recursiva a partir de la descripción del algoritmo.

```
(%i1) warshall(m):= warshall_aux(m,length(m),1)$
(%i2) warshall_aux(m,n,k):= if k=n+1 then m else
          warshall_aux (genmatrix(lambda([i,j],

max(m[i][j],min(m[i][k],m[k][j]))),
                              n,n),n,k+1)$
```

Para utilizar recursión de cola, recurrimos a un operador auxiliar con dos argumentos adicionales, el tamaño de la matriz y un contador para las iteraciones del algoritmo. Hemos utilizado el operador $\mathrm{genmatrix(a[i,j],k,m)}$genmatrix(a[i,j],k,m), que construye una matriz de tamaño $k \times m$k×m con el elemento $\mathrm{a[i,j]}$a[i,j] en la posición $\mathrm{(i,j)}$(i,j). Para describir los elementos de la matriz, recurrimos al operador $\mathrm{lambda([i,j],<expresión\ en\ i,\ j>)}$lambda([i,j],<expresión en i, j>), que permite dar la expresión que corresponde a cada posición sin necesidad de asignarle un nombre. De esta forma, es fácil observar que si $\mathrm{m=M=W_0}$m=M=W0, entonces $\mathrm{warshall_aux(m,,k)=W_k}$warshall_aux(m,,k)=Wk.

Vamos a verificar el ejemplo que hemos hecho más arriba, "trazando" el operador auxiliar para verificar las etapas intermedias.

```
(%i3) M0: matrix([1,0,0,1],
                 [0,0,0,1],
                 [1,0,0,0],
                 [0,1,0,0])$
(%i4) trace(warshall_aux)$
(%i15)  warshall(M0);
1 Introducir warshall_aux [matrix(
                [1,     0,      0,      1],
                [0,     0,      0,      1],
                [1,     0,      0,      0],
                [0,     1,      0,      0]
        ),4,1]
 .2 Introducir warshall_aux [matrix(
                [1,     0,      0,      1],
                [0,     0,      0,      1],
                [1,     0,      0,      1],
```

```
                [0,     1,      0,      0]
        ),4,2]
 ..3 Introducir warshall_aux [matrix(
                [1,     0,      0,      1],
                [0,     0,      0,      1],
                [1,     0,      0,      1],
                [0,     1,      0,      1]
        ),4,3]
 ...4 Introducir warshall_aux [matrix(
                [1,     0,      0,      1],
                [0,     0,      0,      1],
                [1,     0,      0,      1],
                [0,     1,      0,      1]
        ),4,4]

 ....5 Introducir warshall_aux [matrix(
                [1,     1,      0,      1],
                [0,     1,      0,      1],
                [1,     1,      0,      1],
                [0,     1,      0,      1]
        ),4,5]
 ....5 Salir  warshall_aux matrix(
                [1,     1,      0,      1],
                [0,     1,      0,      1],
                [1,     1,      0,      1],
                [0,     1,      0,      1]
...............
        )
```

$$\begin{pmatrix} 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 \end{pmatrix}$$

GRAFOS

Maxima dispone de un paquete con operadores específicos para trabajar con grafos. Podemos trabajar con grafos simples, dirigidos o no dirigidos, y también con grafos ponderados. Internamente los grafos son representados por sus listas de adyacencia, aunque también podemos trabajar con la matriz de adyacencia. Los vértices son identificados con números naturales y las aristas son representadas por listas de longitud dos. Se pueden asignar etiquetas a vértices y se pueden asignar pesos a las aristas para definir grafos ponderados. El operador `create_graph` sirve para definir una grafo a partir de su lista de vértices y su lista de aristas. Este operador admite varias sintaxis; en la más simple, escribimos un primer argumento con un número positivo nn, que será el número de vértices y los vértices serán identificados con los números 0, 1, 2,…, $n-1$. El segundo argumento es una lista de listas de longitud 2 que definen las aristas del grafo.

```
(%i1) load(graphs)$
(%i2) g1: create_graph(5,[[1,2],[1,3],[2,3],[0,4]]);
(%o2) GRAPH(5 vertices, 4 edges)
```

Como vemos, la salida no muestra nada, solo nos da la información del grafo que hemos definido, concretamente el número de vértices y el número de aristas. Para ver su representación como lista de adyacencia usamos `print_graph`:

```
(%i3) print_graph(g1);
Graph on 5 vertices with 4 edges.
Adjacencies:
  4 :  0
  3 :  2  1
  2 :  3  1
  1 :  3  2
  0 :  4
(%o3)   done
```

Obsérvese que la muestra como columna, no como fila. También podemos obtener la representación mediante la matriz de adyacencia del grafo:

```
(%i4) mg1: adjacency_matrix(g1);
```

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

A lo largo del tema iremos viendo distintos operadores para estudiar los grafos y obtener información sobre ellos. Por ejemplo, en la sección anterior, hemos definido la noción de grado de un vértice y la lista de adyacencia no permite visualizar ese grado, pero un elemento importante del grafo es lo que se llama la *secuencia gráfica*, la lista de los grados de todos los vértices ordenada de forma creciente:

```
(%i5) degree_sequence(g1);
(%o15)  [1,1,2,2,2]
```

Finalmente, el operador `draw_graph` implementa un algoritmo para construir una representación gráfica.

```
(%i6) draw_graph(g1,vertex_size=4,show_id=true);
```

El operador `draw_graph` dispone de varias opciones para configurar y mejorar la representación gráfica. En este caso, hemos utilizado "show_id==true" para que se muestre la etiqueta de cada vértice y "vertex_size=4=4" para aumentar el tamaño del círculo que los encierra.

También podemos crear un grafo a partir de su matriz de adyacencia.

```
(%i7) mat:
matrix([0,0,1,1,1],[0,0,1,0,1],[1,1,0,0,0],
            [1,0,0,0,0],[1,1,0,0,0])$
(%i7) g2: from_adjacency_matrix(mat);
(%o7) GRAPH(5 vertices, 5 edges)
(%i8) print_graph(g2);
Graph on 5 vertices with 5 edges.
Adjacencies:
  4 :  1  0
  3 :  0
  2 :  1  0
  1 :  4  2
  0 :  4  3  2
(%o8) done
```

Como vemos, los vértices se etiquetan con los números consecutivos desde el 0, siguiendo el orden dado por las filas y columnas.

En `Maxima` disponemos de un predicado que analiza si un grafo es bipartito, `is_bipartite` y un operador que determina las dos partes de un grafo si es bipartito, `bipartition`.

- El grafo C_6C6 (ciclo de longitud 6) es bipartito:

$$V_1=\{a_1,a_3,a_5\},V_2=\{a_2,a_4,a_6\}V1=\{a1,a3,a5\},V2=\{a2,a4,a6\}$$

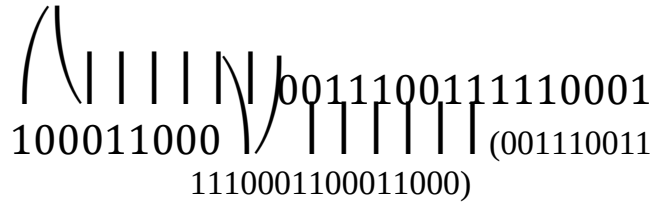
```
(%i1) load(graphs)$
(%i2) is_bipartite(cycle_graph(6));
(%o2) true
(%i3) bipartition(cycle_graph(6));
(%o3) [[4,2,0],[5,3,1]]
```

- El grafo S_5S5 (estrella de 5 puntas) también es bipartito:

$$V_1=\{a_0\},V_2=\{a_1,a_2,a_3,a_4,a_5\}V1=\{a0\},V2=\{a1,a2,a3,a4,a5\}$$

- Sin embargo, el grafo W_6W6 (rueda de 7 vértices) no es bipartito.
- `(%i5) is_bipartite(wheel_graph(6));`
- `(%o5) false`
- `(%i6) bipartition(wheel_graph(6));`
  `(%o6) []`

El grafo **bipartito completo** $K_{m,n}$Km,n es el grafo con $m+n$m+n vértices en donde el conjunto de vértices está partido en dos conjuntos V_1V1 con mm vértices y V_2V2 con nn vértices, de tal forma que cada vértice de V_1V1 está conectado con cada vértice de V_2V2. Vemos abajo el grafo bipartito completo $K_{3,2}$K3,2:



Para escribir la matriz de adyacencia, hemos ordenado los vértices según su subíndice, los dos primeros corresponden a una parte de los vértices y los tres últimos a la otra parte. De esta forma, la matriz ha quedado formada por cuatro cajas que contienen o bien unos o bien ceros.

El operador `complete_bipartite_graph` define en `Maxima` operadores bipartitos completos del tamaño que deseemos. La siguiente línea siguiente mostrará el grafo del ejemplo anterior.

```
(%i1) load(graphs)$
(%i2) draw_graph(complete_bipartite_graph(3,2),
          vertex_size=3,show_id=true);
```

En Maxima, el operador is_conneceted analiza si un grafo es o no conexo y connected_components nos devuelve las componentes conexas.

```
(%i1) load(graphs)$ (%i2) g1:
create_graph(5,[[1,2],[1,3],[2,3],[0,4]])$ (%i3)
is_connected(g1);
        (%o3) false (%i4) connected_components(g1);
(%o4) [[2,1,3],[0,4]] (%i5) g2:
create_graph([1,2,3,4,5,6,7,8],
        [[1,2], [1,6], [1,8],[1,3],
[2,4],[2,5],[2,7],[3,4],[3,5],[3,7],
[4,6],[4,8],[5,6],[5,8],[6,7],[7,8]])$ (%i6)
is_connected(g2); (%o6) true (%i7)
connected_components(g2);
        (%o7) [[3,8,6,7,5,4,2,1]]
```

En Maxima, el operador is_tree analiza si un grafo es o no árbol.

```
(%i1) load(graphs)$ (%i2) is_tree(cube_graph(3));
(%o2) false (%i3) arb:
create_graph(8,[[0,5],[0,6],[1,3],[2,4],[2,6],[2,7],
[3,6]])$
        (%i4) is_tree(arb); (%o4) true
```

Podemos determinar árboles generadores de un grafo, concretamente aquellos que contienen el menor número de aristas; para ello usamos el operador minimun_spanning_tree.x

```
(%i5) gt: create_graph([1,2,3,4,5,6,7],
[[1,2],[1,6],[2,3], [2,4], [2,6], [2,7], [3,4],
[4,5], [4,7], [5,6], [5,7], [6,7]])$ (%i6) sgt:
minimum_spanning_tree(gt)$
        (%i7) draw_graph(gt,vertex_size=4,
show_id=true,show_edges=edges(sgt)); (%o8) done
```

En el código anterior hemos usado la opción show_edges=edges(sgt) para que se resalten las aristas del subgrafo sgt, es decir, del subárbol generador minimal.

En Maxima, disponemos del operador isomorphism que analiza si dos grafos son isomorfos y, en tal, caso determina un isomorfismo. Los tres ejemplos que mostramos a continuación tienen la misma secuencia gráfica, pero solo dos de ellos son isomorfos.

```
(%i1) load(graphs)$ (%i2) g4:
create_graph([1,2,3,4,5],[[1,4],[1,5],[2,3],
[2,4],[2,5],[3,4]])$
        degree_sequence(g4); (%o2) [2,2,2,3,3] (%i3)
g5: create_graph([1,2,3,4,5],[[1,2],[1,3],[1,5],
[2,3],[3,4],[4,5]])$
        degree_sequence(g5); (%o3) [2,2,2,3,3] (%i4)
g6: create_graph([1,2,3,4,5],[[1,4],[1,5],[2,4],
[2,5],[3,4],[3,5]])$
        degree_sequence(g6); (%o4) [2,2,2,3,3] (%i5)
isomorphism(g4,g5); (%o5) [4->3,5->5,3->2,1->4,2->1]
        (%i6) isomorphism(g4,g6); (%o6) []
```

El operador `hamilton_cycle` determina si un grafo es o no Hamiltoniano y en tal caso calcula el ciclo de Hamilton.

```
(%i1) load(graphs)$ (%i2) k33:
complete_bipartite_graph(3,3)$ (%i3) hk33:
hamilton_cycle(k33);
                  (%o3) [0,5,2,4,1,3,0]
```

Utilizando la opción `show_edges` del operador `draw_graph` podemos ver el dibujo de un grafo en el que se resalte un determinado camino, por ejemplo el ciclo de Hamilton de un grafo hamiltoniano.

```
(%i4)
draw_graph(k33,show_edges=vertices_to_cycle(hk33));
(%o4) done
```

También podemos determinar el camino de Hamilton contenido en un grafo semihamiltoniano. Por ejemplo, sabemos que el grafo $K_{3,4}$K3,4 no es hamiltoniano, pero sí es semihamiltoniano.

```
(%i5) k34:complete_bipartite_graph(3,4)$
                  hamilton_cycle(k34); (%o5) [] (%i6)
hk34: hamilton_path(k34); (%o6) [6,2,5,1,4,0,3]
```

En este caso, el operador `vertices_to_path` en la opción `show_edges` también nos permite visualizar el camino de Hamilton.

```
(%i7)
draw_graph(k34,show_edges=vertices_to_path(hk34));
(%o7) done
```

En Maxima, podemos analizar la planaridad de un grafo utilizanado el operador is_planar.

```
(%i1) load(graphs)$
(%i2) g9: create_graph(7, [[0,1],[0,5],[1,2],[1,4],
        [2,3], [0,3],
[2,5],[3,6],[4,5],[4,6],[5,6]])$
        is_planar(g9);
(%o2)   false
```

El algoritmo básico que utiliza Maxima para generar la representación gráfica de un grafo, no produce siempre una representación plana. Sin embargo, también permite elegir entre varios programas para determinar la posición final de los vértices en la representación gráfica; por ejemplo, planar_embedding fuerza la representación plana en la "mayoría" de las situaciones.

```
(%i3) draw_graph(complete_graph(4),redraw=true,
        program=planar_embedding);
(%o3)   done
```

En Maxima, disponemos de operadores que nos determinan el número cromático de un grafo y calculan una coloración óptima.

```
(%i1) load(graphs)$
(%i2) gcol:create_graph([1,2,3,4,5,6,7,8],[[3,
4],[4, 8],

[2,5],[1,8],[5,6],[7,8],[4,7],[2,6],[1,4],[3,7],
         [2,7],[6,8],[2,3],[3,5],[1,6],[1,5]])$
(%i3) chromatic_number(gcol);
(%o4)   4
```

Por otra parte, vertex_coloring, también devuelve el número cromático pero incluyendo una coloración con ese número de colores. Los colores están representados por números naturales

```
(%i5) vertex_coloring(gcol);
(%o5)
[4,[[8,2],[7,4],[6,1],[5,4],[4,1],[3,2],[2,3],[1,3]]
]
```

Es decir, los vértices 4 y6 están coloreados con el color 1, los vértices 3 y 8 con el color 2, los vértices 1 y 2 con el color 3 y los vértices 5 y7 están con el color 4. Podemos visualizar el grafo con diferentes colores usando la opción vertex_partition.

```
(%i6) draw_graph(gcol,vertex_size=4,show_id=true,

vertex_partition=[[1,2],[3,8],[4,6],[5,7]]);
(%o72)  done
```

```
→ /*Graphs*/;

(%i1) load(graphs)$

(%i2) g1: create_graph(5, [[1, 2], [1, 3], [2, 3], [0, 4]]);
g1    GRAPH(5 vertices, 4 edges)

→ /*Adjacency list*/;

(%i3) print_graph(g1);
Graph on 5 vertices with 4 edges.
Adjacencies:
 4 : 0
 3 : 2  1
 2 : 3  1
 1 : 3  2
 0 : 4
(%o3) done
```

→ /*Adjacency matrix*/;

(%i4) mg1: adjacency_matrix(g1);

mg1 =

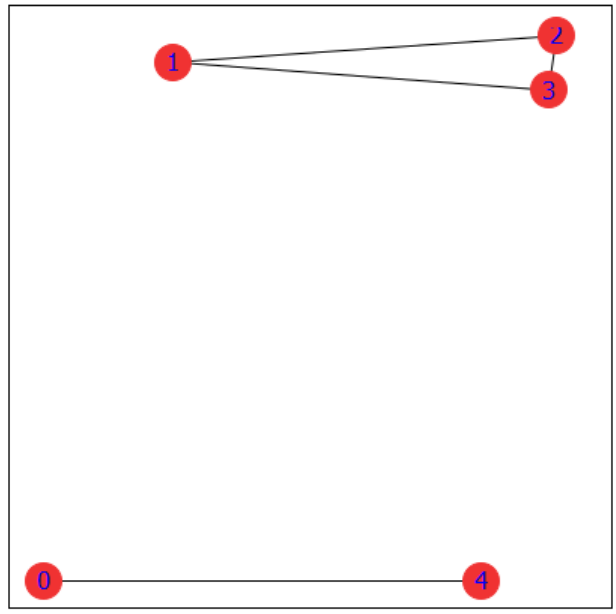
$$\begin{pmatrix} 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \end{pmatrix}$$

(%i5) degree_sequence(g1);

(%o5) [1,1,2,2,2]

(%i6) draw_graph(g1, vertex_size=4, show_id=true);

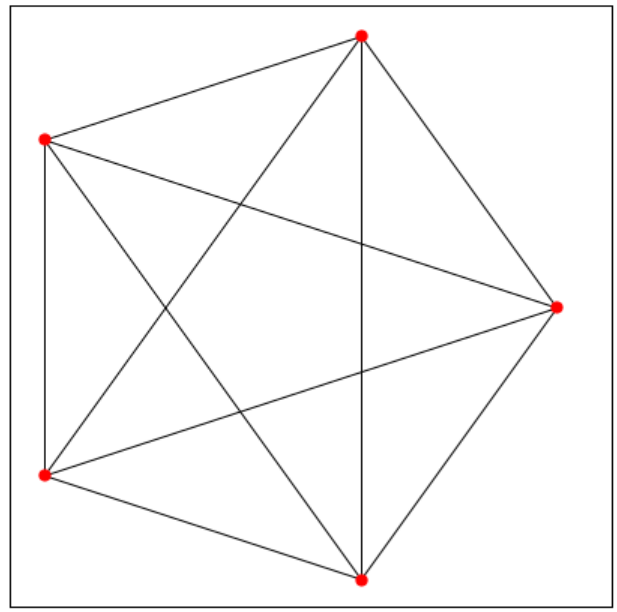
(%t6)



(%o6) done

```
→  /*Graph from its adjacemcy matrix*/;

(%i7) mat: matrix([0,0,1,1,1],[0,0,1,0,1],[1,1,0,0,0],[1,0,0,0,0],[1,1,0,0,0])$

(%i8) g2: from_adjacency_matrix(mat);
g2    GRAPH(5 vertices, 5 edges)

(%i9) print_graph(g2);
Graph on 5 vertices with 5 edges.
Adjacencies:
 4 : 1 0
 3 : 0
 2 : 1 0
 1 : 4 2
 0 : 4 3 2
(%o9) done

→  /*Complete graph*/;

(%i10) load(graphs)$

(%i11) k5: complete_graph(5)$

(%i12) draw_graph(k5);
```

(%t12)



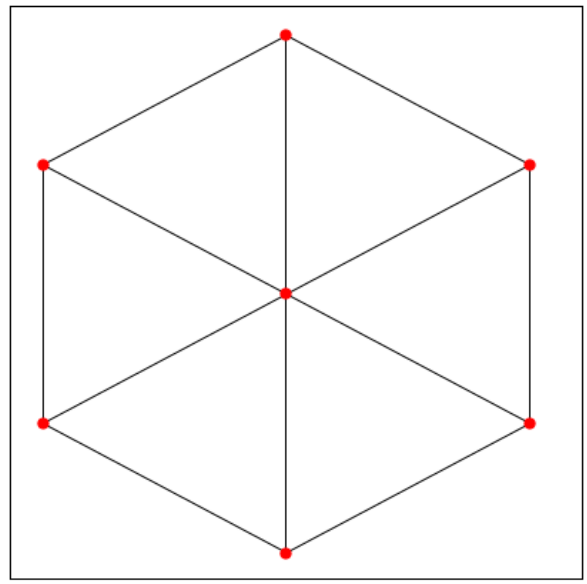
```
(%o12) done
```

→ /*Wheel graph*/;

(%i13) w6: wheel_graph(6)$

(%i14) draw_graph(w6);

(%t14)



(%o14) done

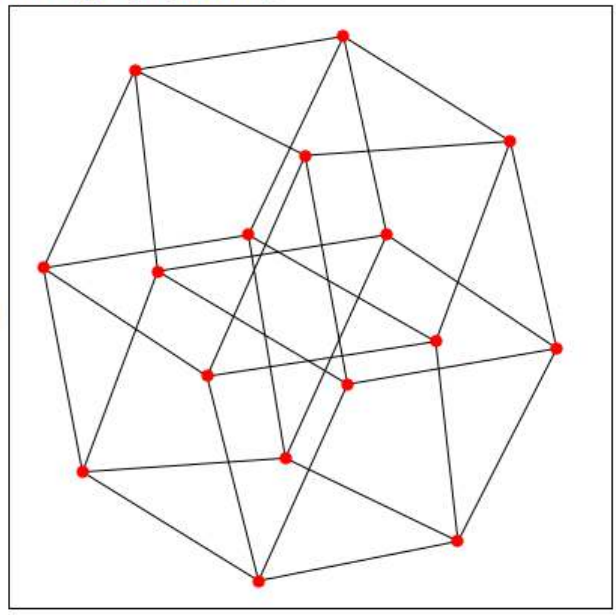
→ /*Cube graph*/;

(%i15) load(graphs)$

(%i16) cube4: cube_graph(4)$

(%i17) draw_graph(cube4);

(%t17)



(%o17) done

→ /*Bipartite graph*/;

(%i18) load(graphs)$

(%i19) is_bipartite(cycle_graph(6));

(%o19) true

(%i20) bipartition(cycle_graph(6));

(%o20) [[4,2,0],[5,3,1]]

(%i21) is_bipartite(wheel_graph(6));

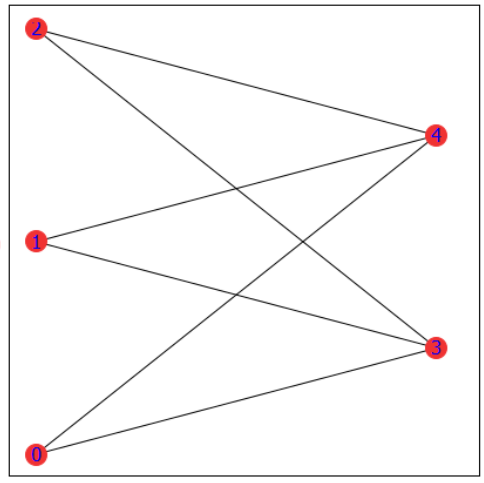
(%o21) false

(%i22) bipartition(wheel_graph(6));

(%o22) []

```
→  /*Complete bipartite graph*/;

(%i23) load(graphs)$

(%i24) draw_graph(complete_bipartite_graph(3,2), vertex_size=3, show_id=true);
```

(%t24)




```
(%o24) done

→  /*Connected graph*/;

(%i25) load(graphs);
(%o25) C:/maxima-5.47.0/share/maxima/5.47.0/share/graphs/graphs.mac

(%i26) g1: create_graph(5,[[1,2],[1,3],[2,3],[0,4]])$

(%i27) is_connected(g1);
(%o27) false

(%i28) connected_components(g1);
(%o28) [[2,1,3],[0,4]]
```

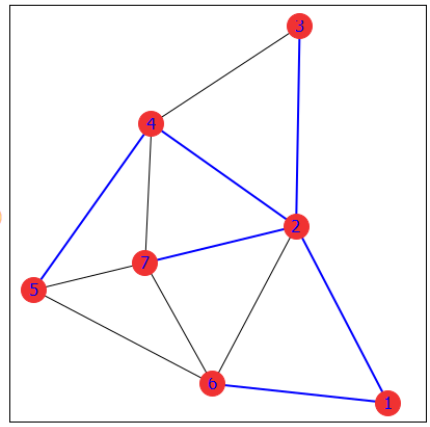
```
→  /*Tree or not tree*/;
```

```
(%i29) load(graphs);
(%o29) C:/maxima-5.47.0/share/maxima/5.47.0/share/graphs/graphs.mac

(%i30) is_tree(cube_graph(3));
(%o30) false

(%i31) arb: create_graph(8,[[0,5],[0,6],[1,3],[2,4],[2,6],[2,7],[3,6]]);
arb    GRAPH(8 vertices, 7 edges)

(%i32) is_tree(arb);
(%o32) true
```

```
→  /*Minimum spanning tree*/;
```

```
(%i33) gt: create_graph([1,2,3,4,5,6,7],[[1,2],[1,6],[2,3],[2,4],[2,6],[2,7],[3,4],[4,5],[4,7],[5,6],[5,7],[6,7]]);
gt     GRAPH(7 vertices, 12 edges)

(%i34) sgt: minimum_spanning_tree(gt);
sgt    GRAPH(7 vertices, 6 edges)

(%i35) draw_graph(gt, vertex_size=4, show_id=true, show_edges=edges(sgt));
```

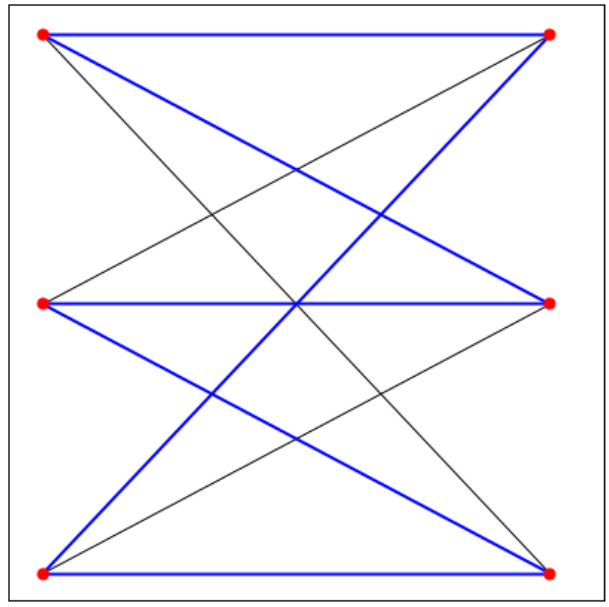
(%t35)




```
(%o35) done
```

```
→ /*Isomorphism*/;

(%i36) g4: create_graph([1,2,3,4,5],[[1,4],[1,5],[2,3],[2,4],[2,5],[3,4]]);
g4 GRAPH(5 vertices, 6 edges)

(%i37) g5: create_graph([1,2,3,4,5],[[1,2],[1,3],[1,5],[2,3],[3,4],[4,5]]);
g5 GRAPH(5 vertices, 6 edges)

(%i38) g6: create_graph([1,2,3,4,5],[[1,4],[1,5],[2,4],[2,5],[3,4],[3,5]]);
g6 GRAPH(5 vertices, 6 edges)

(%i39) degree_sequence(g4);
(%o39) [2,2,2,3,3]

(%i40) degree_sequence(g5);
(%o40) [2,2,2,3,3]

(%i41) degree_sequence(g6);
(%o41) [2,2,2,3,3]

(%i42) isomorphism(g4,g5);
(%o42) [4→3,5→5,3→2,1→4,2→1]

(%i43) isomorphism(g4,g6);
(%o43) []
```

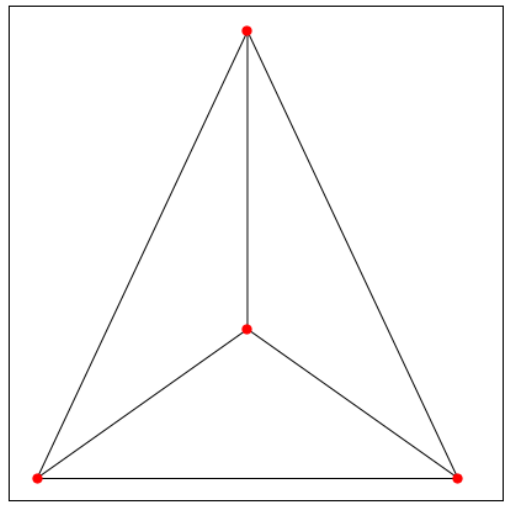
```
→    /*Hamiltonian cycle*/;

(%i44) load(graphs)$

(%i45) k33: complete_bipartite_graph(3,3)$

(%i46) hk33: hamilton_cycle(k33);
hk33  [0,5,2,4,1,3,0]

(%i48) draw_graph(k33,show_edges=vertices_to_cycle(hk33));
```



```
(%o48) done
```

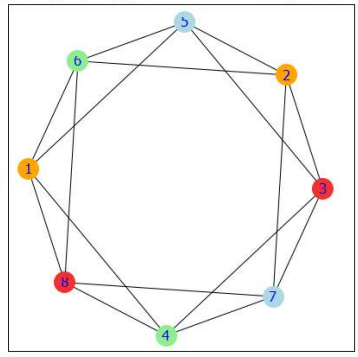
```
→ /*Planarity*/;

(%i1) load(graphs);
(%o1) C:/maxima-5.47.0/share/maxima/5.47.0/share/graphs/graphs.mac

(%i2) g9: create_graph(7,[[0,1],[0,5],[1,2],[1,4],[2,3],[0,3],[2,5],[3,6],[4,5],[4,6],[5,6]]);
g9    GRAPH(7 vertices, 11 edges)

(%i3) is_planar(g9);
(%o3) false

(%i4) draw_graph(complete_graph(4), redraw=true, program=planar_embedding);
```

(%t4)



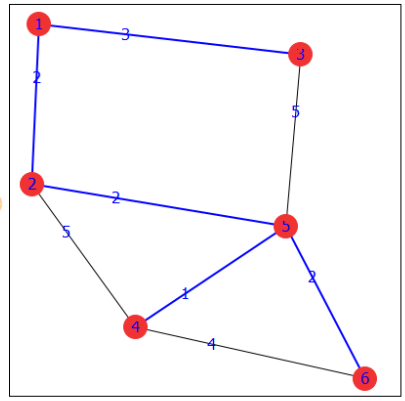
```
(%o4) done

→ /*Graph couloring*/;

(%i5) gcol: create_graph([1,2,3,4,5,6,7,8],[[3,4],[4,8],[2,5],[1,8],[5,6],[7,8],[4,7],[2,6],[1,4],[3,7],[2,7],[6,8],[2,3],[3,5],[1,6],[1,5]]);
gcol  GRAPH(8 vertices, 16 edges)

(%i6) chromatic_number(gcol);
(%o6) 4

(%i7) vertex_coloring(gcol);
(%o7) [4,[[8,2],[7,4],[6,1],[5,4],[4,1],[3,2],[2,3],[1,3]]]

(%i8) draw_graph(gcol, vertex_size=4, show_id=true, vertex_partition=[[3,8],[5,7],[4,6],[1,2]]);
```

(%t8)




```
(%o8) done
```

```
→  /*Minimal spanning tree in a weighted graph*/;

(%i9)  load(graphs)$

(%i10) grp: create_graph([1,2,3,4,5,6],[[[1,2],2],[[1,3],3],[[2,4],5],[[2,5],2],[[3,5],5],[[4,5],1],[[4,6],4],[[5,6],2]])$

(%i11) mstgrp: minimum_spanning_tree(grp)$

(%i12) draw_graph(grp, vertex_size=4, show_id=true, show_weight=true, show_edges=edges(mstgrp));
```



(%t12)

(%o12) done

TEMA 3: RELACIONES Y GRAFOS

SIEMPRE se debe cargar el siguiente paquete para trabajar con grafos:

```
load(graphs)$
```

Representación de grafos simples

1 Definir una grafo a partir de su lista de vértices y su lista de aristas

(nº vértices,[aristas]) --> empezar a numerar los vértices desde 0

([nombre de cada vértice],[aristas]) --> numerar los vértices dependiendo de cómo se hayan nombrado

```
g1: create_graph(3,[[0,1],[0,2],[1,2]]);
```

GRAPH(3 vertices, 3 edges)

2 Lista de adyacencia

```
print_graph(g1);
```

Graph on 3 vertices with 3 edges.
Adjacencies:
2 : 1 0
1 : 2 0
0 : 2 1 done

3 Matriz de adyacencia

```
mg1: adjacency_matrix(g1);
```

$$\begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix}$$

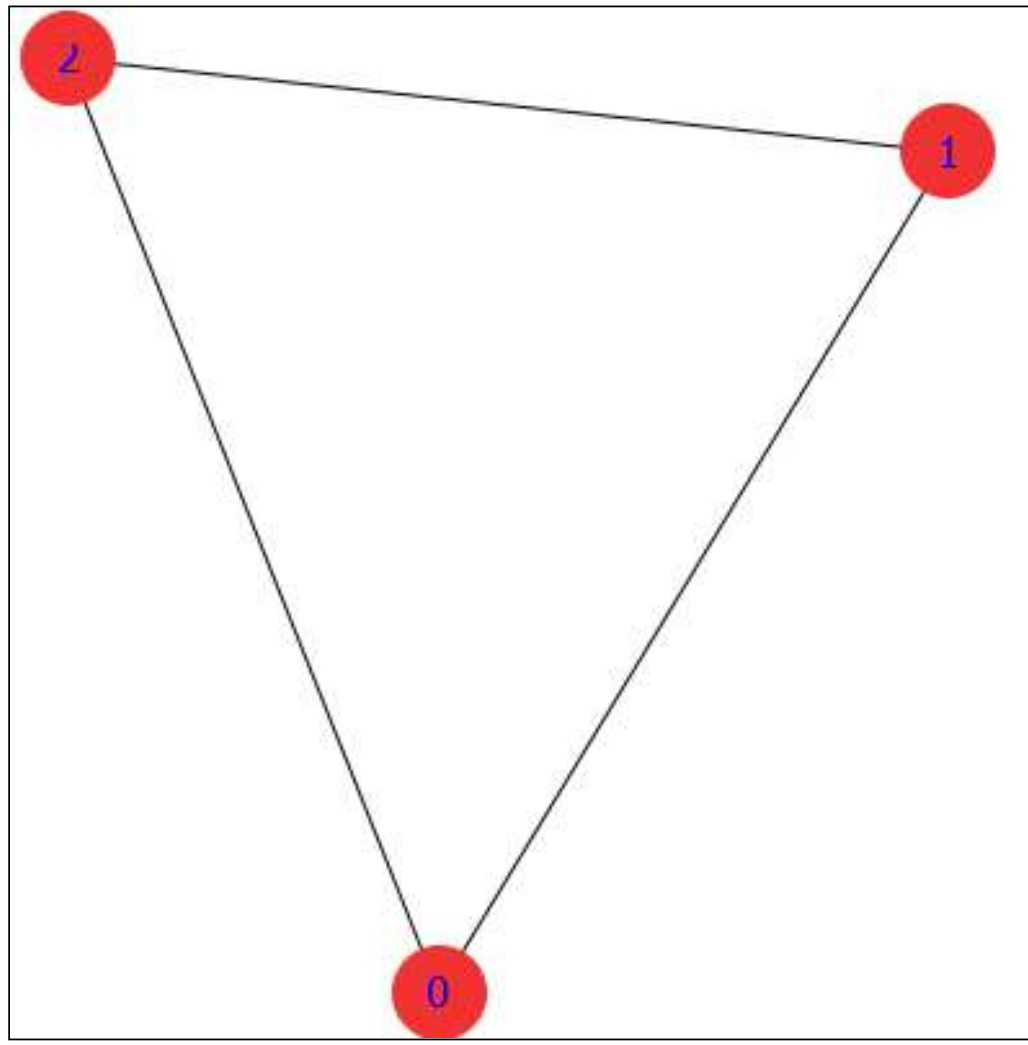
4 Secuencia gráfica (lista creciente de grados)

```
degree_sequence(g1);
```

[2,2,2]

5 Representación gráfica

```
draw_graph(g1,vertex_size=6,show_id=true);
```



done

6 Crear un grafo a partir de la matriz de adyacencia

```
mat: matrix([0,0,1],[0,0,1],[1,1,0]);
```

$$\begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix}$$

```
g2: from_adjacency_matrix(mat);
```

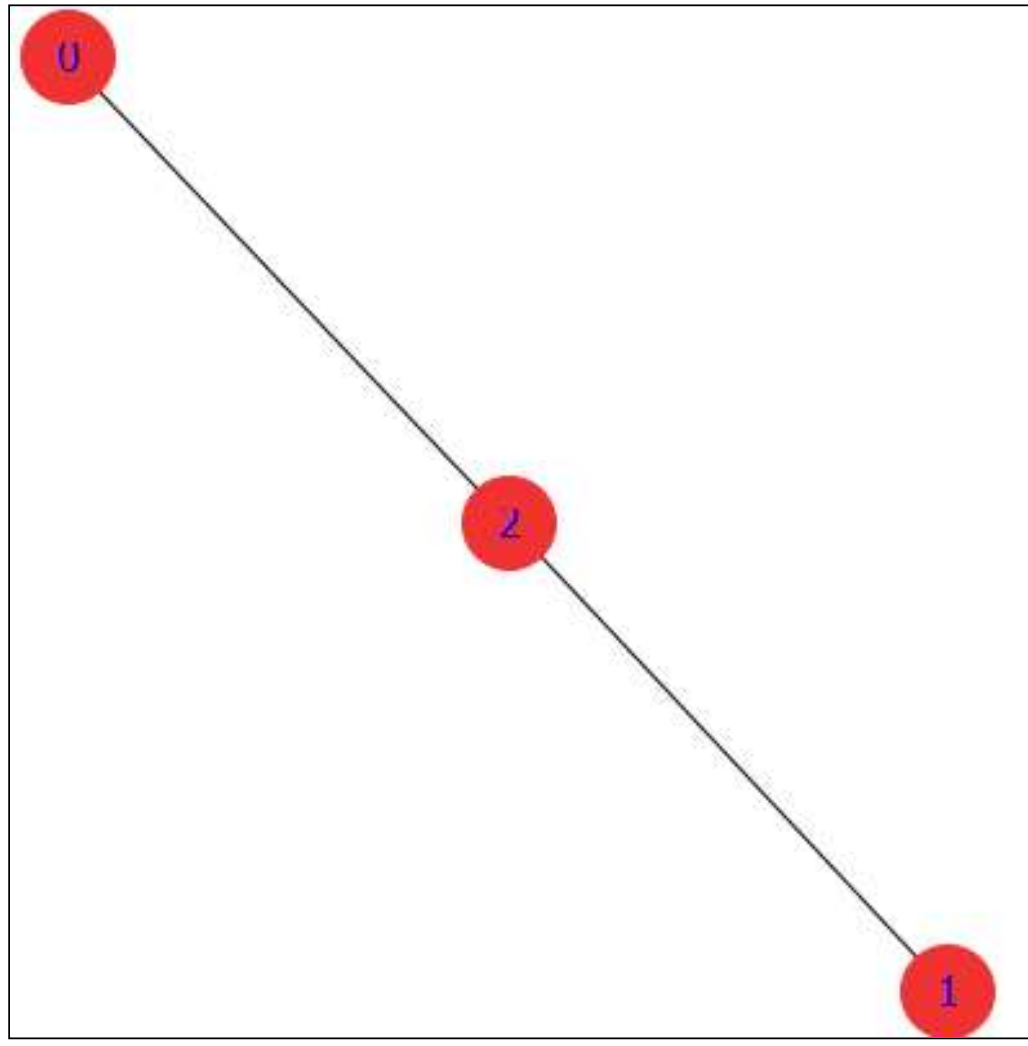
GRAPH(3 vertices, 2 edges)

```
print_graph(g2);
```

Graph on 3 vertices with 2 edges.
Adjacencies:
2 : 1 0

```
1 :  2
0 :  2     done
```

```
draw_graph(g2,vertex_size=6,show_id=true);
```



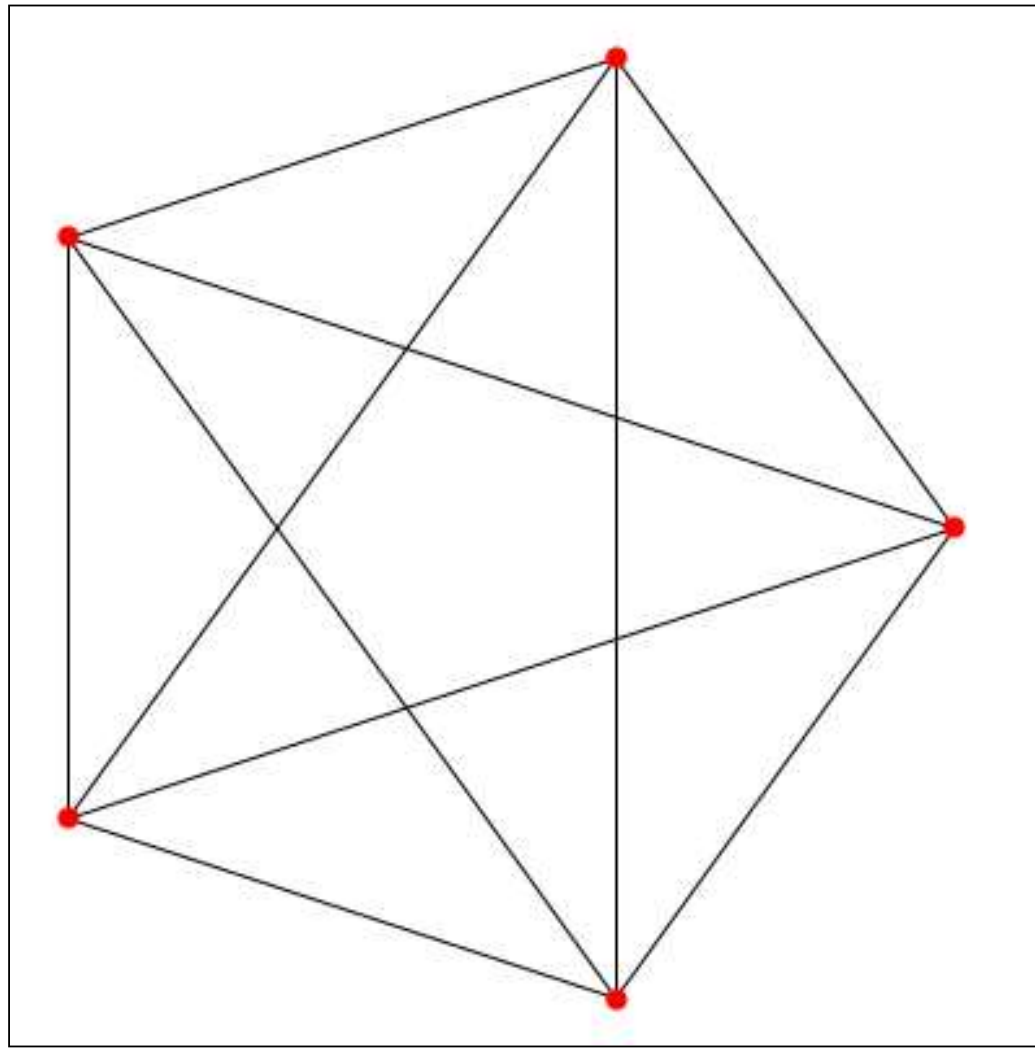
done

Tipos de grafos

1 Grafos completos

Ejemplo: k5

```
k5: complete_graph(5)$
draw_graph(k5);
```

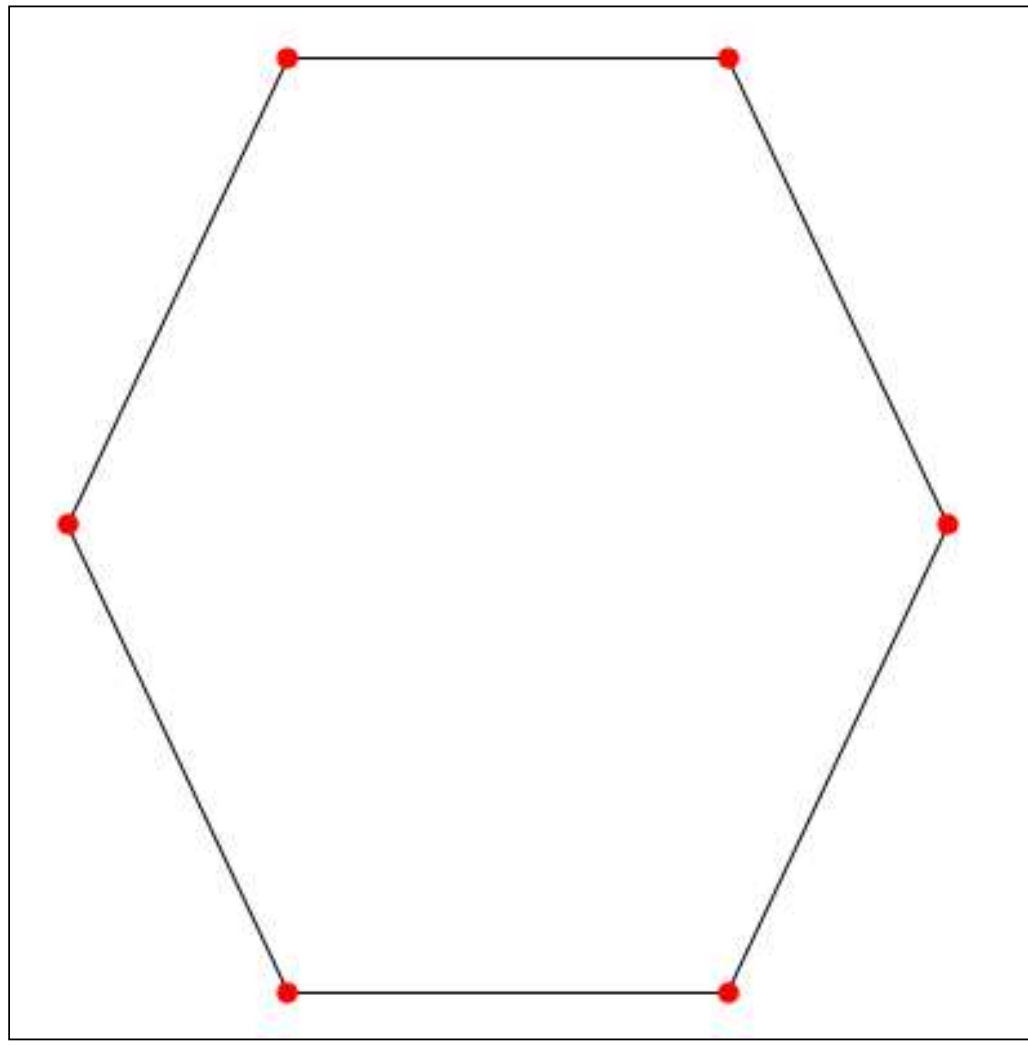


done

2 Grafos ciclo

Ejemplo: ciclo 6 (hexágono)

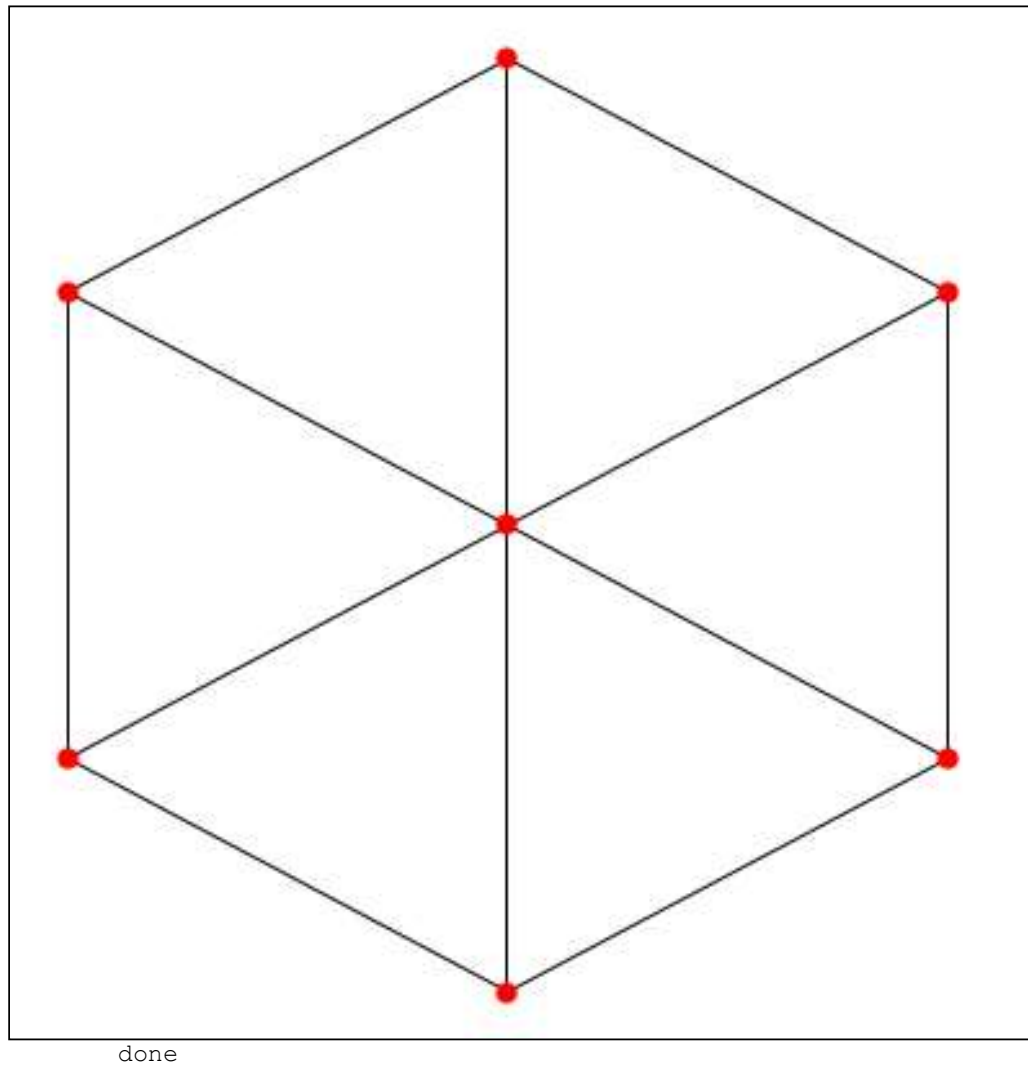
```
c6: cycle_graph(6)$
draw_graph(c6);
```



done

3 Rueda

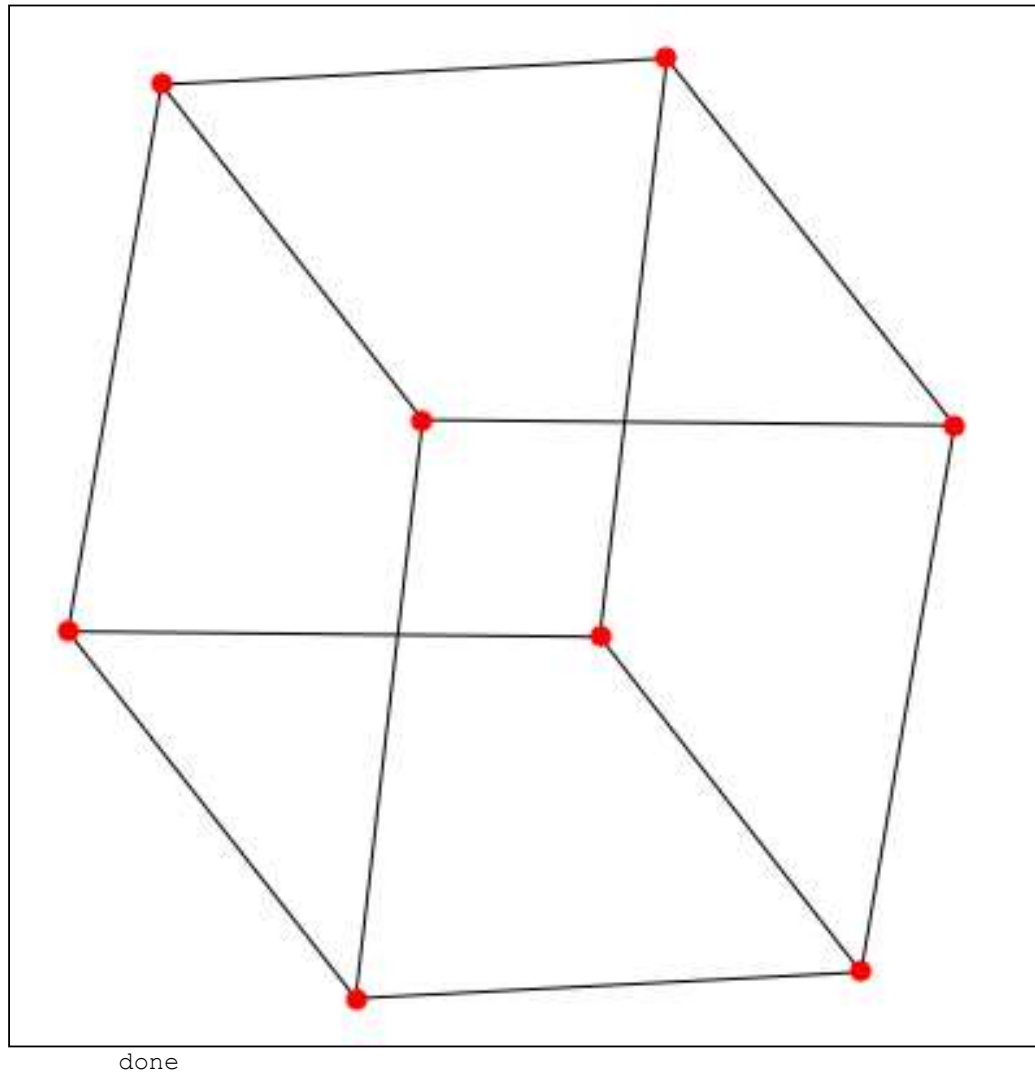
```
w6: wheel_graph(6)$
draw_graph(w6);
```



done

4 n-cubo

```
cube3: cube_graph(3)$
draw_graph(cube3);
```



done

Grafo bipartito

1 Determinar si es bipartito

```
is_bipartite(cycle_graph(6));
```

true

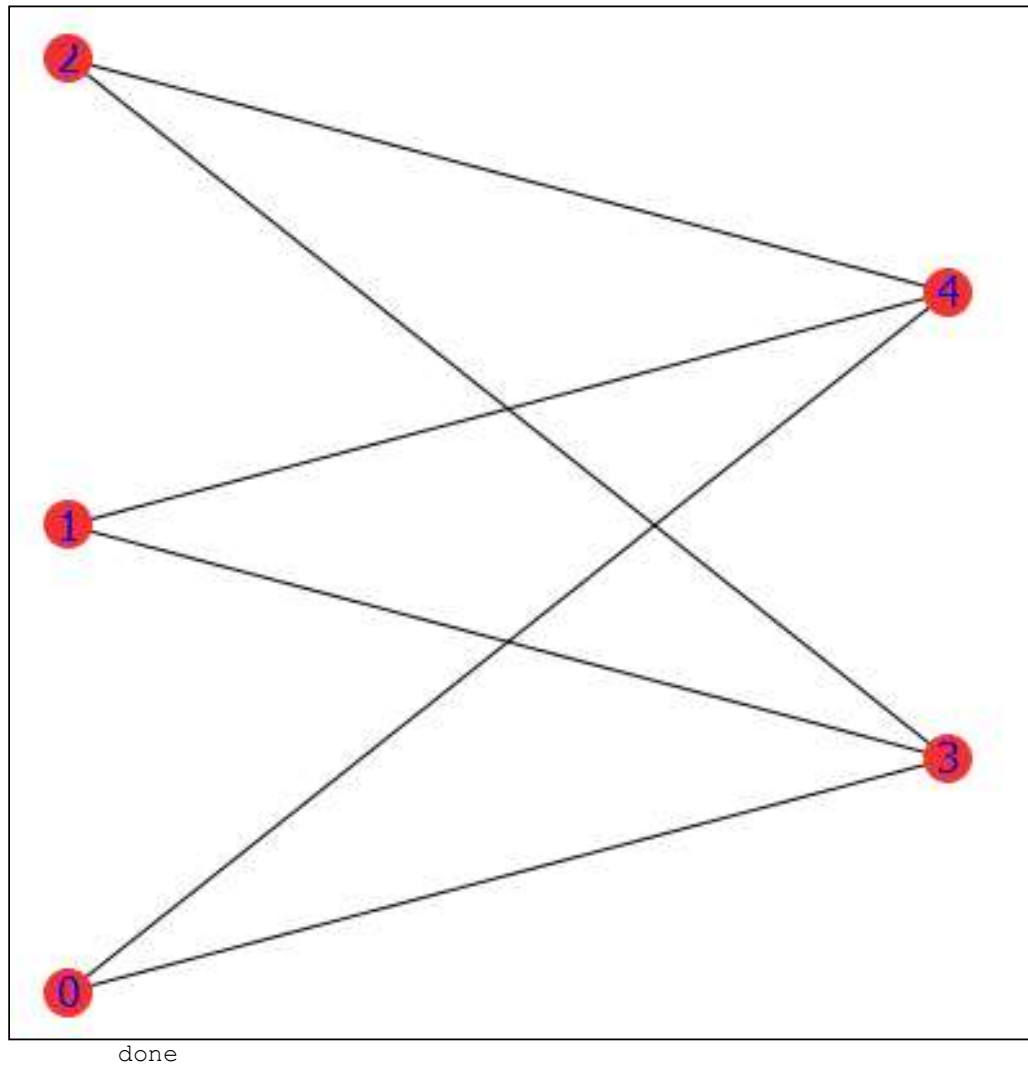
2 Hallar V1 y V2

```
bipartition(cycle_graph(6));
```

[[4,2,0],[5,3,1]]

3 Representar

```
draw_graph(complete_bipartite_graph(3,2),vertex_size=3,show_id=true);
```



done

Grafo conexo

1 Determinar si es conexo

Ejemplo 1:

```
g1: create_graph(5,[[1,2],[1,3],[2,3],[0,4]])$
is_connected(g1);
```

false

Ejemplo 2:

```
g2: create_graph([1,2,3,4,5,6,7,8],[[1,2], [1,6], [1,8],[1,3], [2,4],[2,5],[2,7],[3,4],[3,5],[
is_connected(g2);
```

true

2 Hallar componentes conexas

```
connected_components(g1);
```

```
[[2 ,1 ,3 ],[0 ,4 ]]
```

```
connected_components(g2);
```

```
[[3,8,6,7,5,4,2,1]]
```

Árbol

1 Determinar si es árbol

Ejemplo 1:

```
is_tree(cube_graph(3));
```

```
false
```

Ejemplo 2:

```
arb: create_graph(8,[[0,5],[0,6],[1,3],[2,4],[2,6],[2,7],[3,6]])$
```

```
is_tree(arb);
```

```
true
```

2 Hallar árbol generador de un grafo

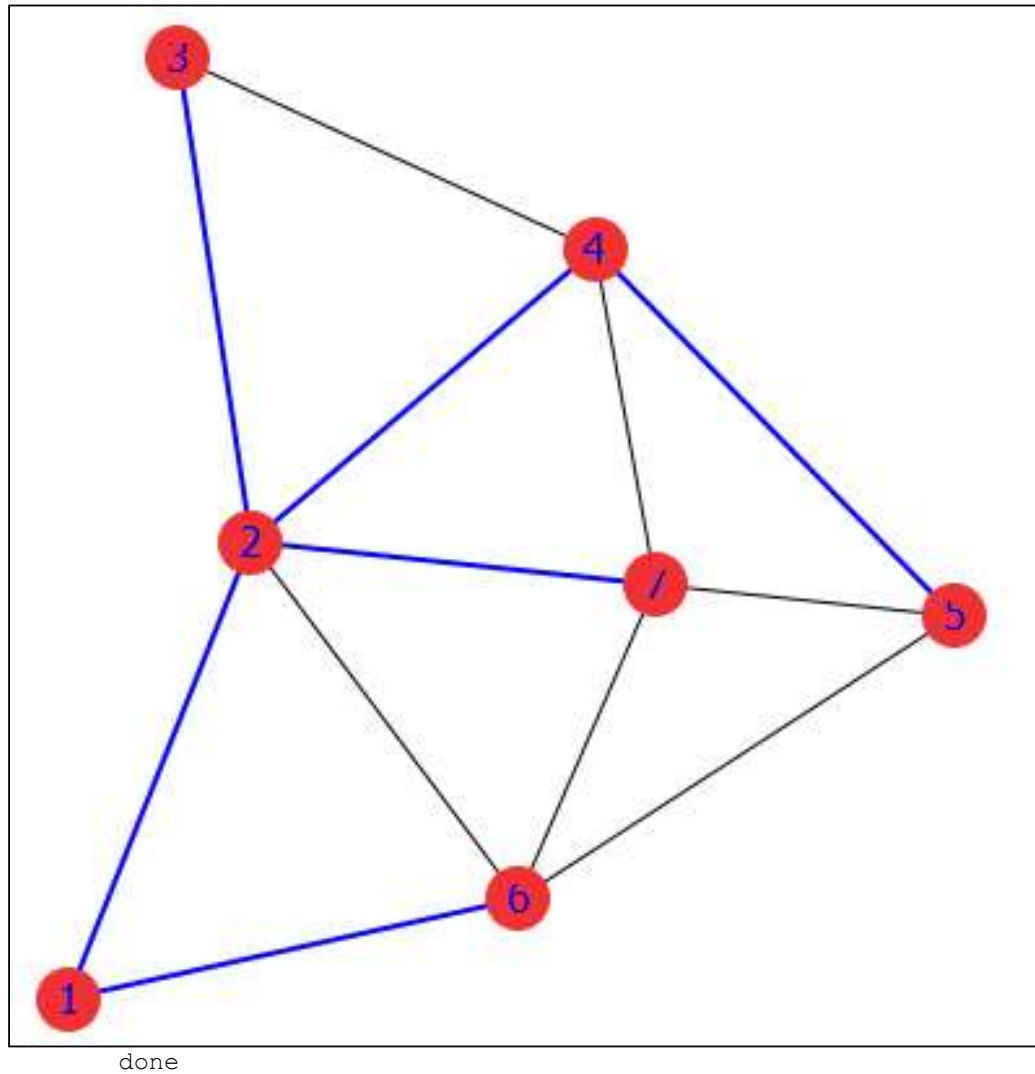
Ejemplo:

```
gt: create_graph([1,2,3,4,5,6,7], [[1,2],[1,6],[2,3], [2,4], [2,6], [2,7], [3,4], [4,5], [4,7],
```

```
sgt: minimum_spanning_tree(gt)$
```

Usamos "show_edges=edges(sgt)" para que resalten las
aristas del subgrafo sgt (subárbol generador minimal)

```
draw_graph(gt,vertex_size=4, show_id=true,show_edges=edges(sgt));
```



done

Grafo Hamiltoniano

1 Determinar si existe ciclo de Hamilton

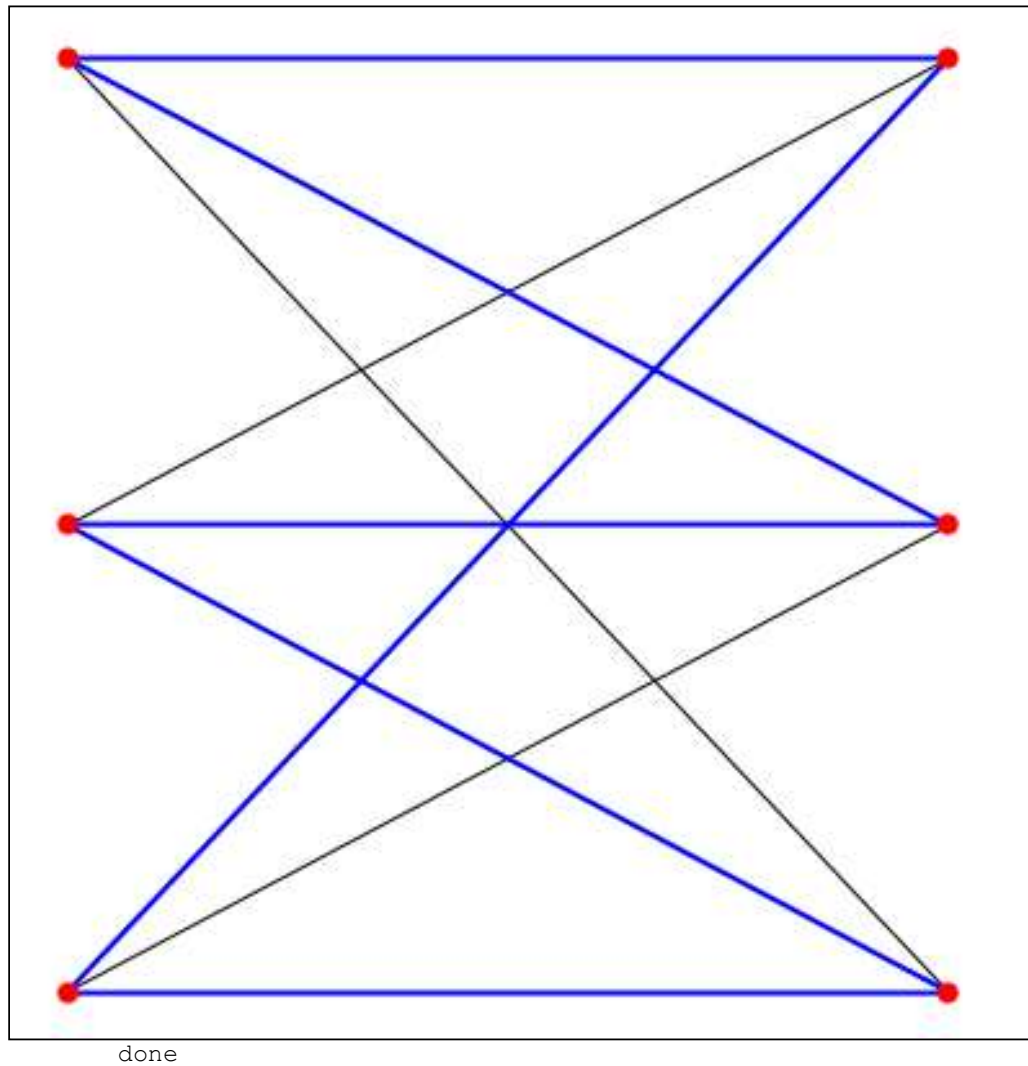
Ejemplo 1 (existe ciclo):

```
k33: complete_bipartite_graph(3,3)$
hk33: hamilton_cycle(k33);
```

[0,5,2,4,1,3,0]

2 Representar ciclo de Hamilton

```
draw_graph(k33,show_edges=vertices_to_cycle(hk33));
```



done

3 Determinar si existe camino de Hamilton

Ejemplo 2 (no existe ciclo):

```
k34:complete_bipartite_graph(3,4)$
hamilton_cycle(k34);
```

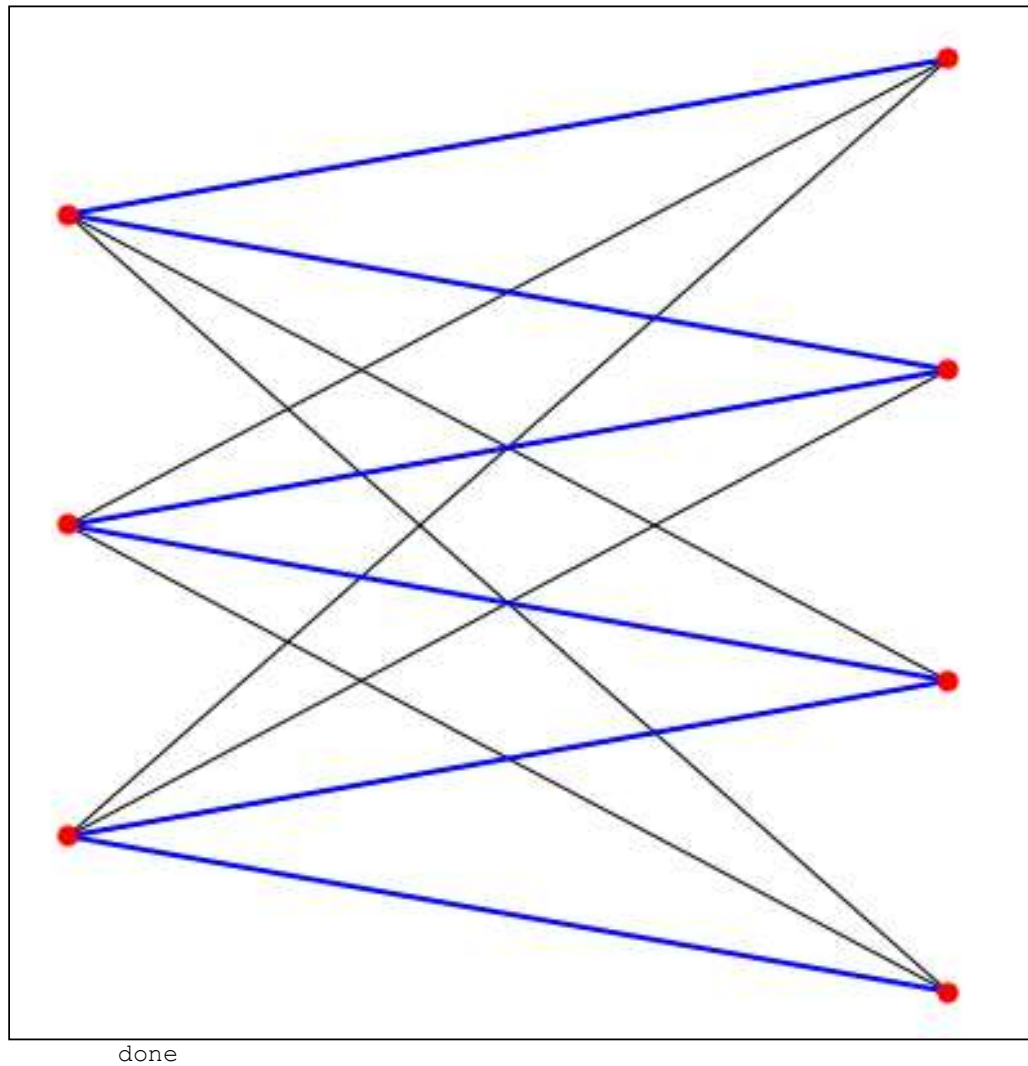
[ ]

```
hk34: hamilton_path(k34);
```

[6,2,5,1,4,0,3]

4 Representar camino de Hamilton

```
draw_graph(k34,show_edges=vertices_to_path(hk34));
```



done

Planaridad de un grafo

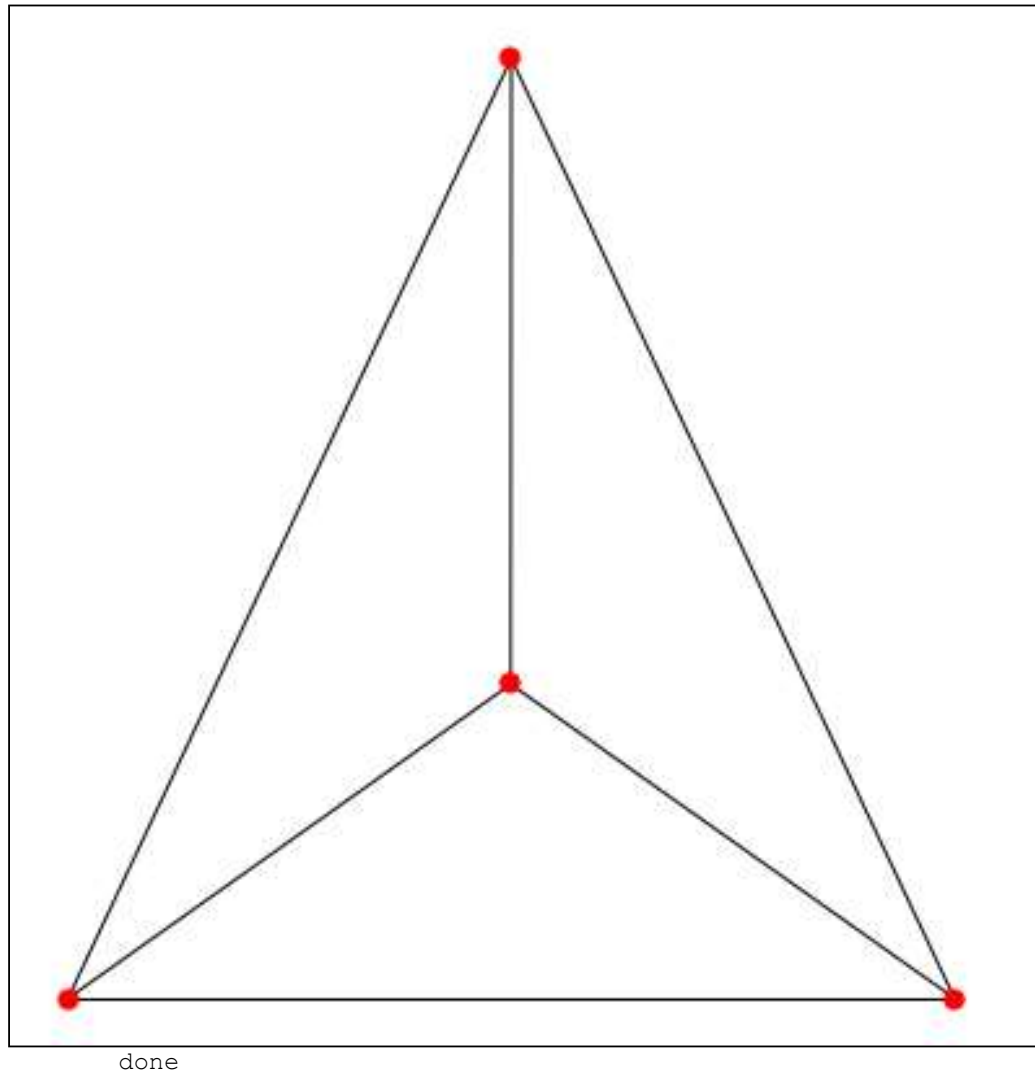
1 Determinar si es plano

```
g9: create_graph(7, [[0,1],[0,5],[1,2],[1,4], [2,3], [0,3], [2,5],[3,6],[4,5],[4,6],[5,6]])$
is_planar(g9);
```

false

2 Representar (no siempre funciona)

```
draw_graph(complete_graph(4),redraw=true, program=planar_embedding);
```



done

Coloración de grafos

1 Hallar número cromático

```
gcol:create_graph([1,2,3,4,5,6,7,8],[[3, 4],[4, 8], [2,5],[1,8],[5,6],[7,8],[4,7],[2,6],[1,4],
chromatic_number(gcol);
```

4

2 Asociar colores a los vértices

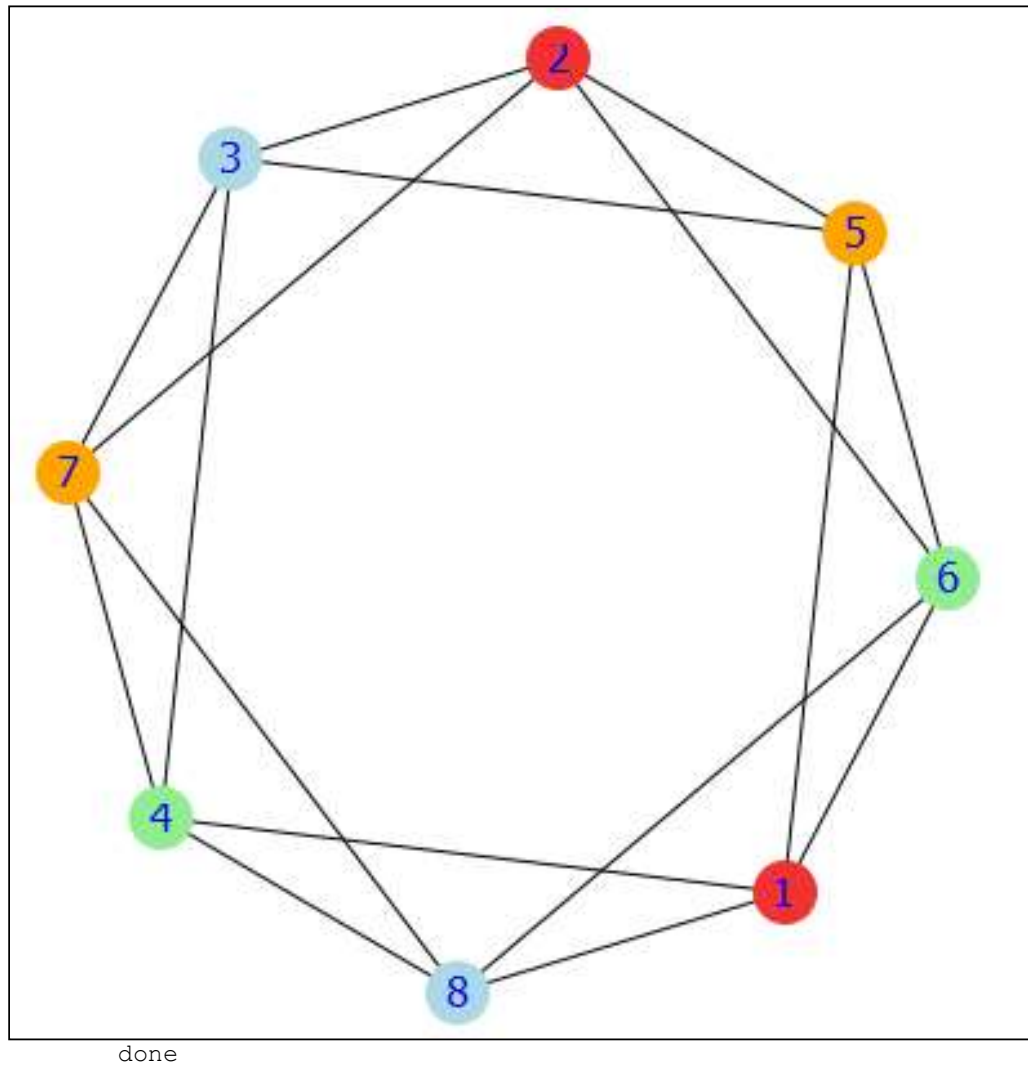
Estructura: [nº cromático,
[[vétice,color],[vétice,color],[vértice,color],...]]

```
vertex_coloring(gcol);
```

$$\left[4,[[8,2],[7,4],[6,1],[5,4],[4,1],[3,2],[2,3],[1,3]]\right]$$

3 Representar

```
draw_graph(gcol,vertex_size=4,show_id=true, vertex_partition=[[1,2],[3,8],[4,6],[5,7
```



done

Isomorfismo

Ejemplo 1:

```
g4: create_graph([1,2,3,4,5],[[1,4],[1,5],[2,3], [2,4],[2,5],[3,4]])$
degree_sequence(g4);
```

[2,2,2,3,3]

```
g5: create_graph([1,2,3,4,5],[[1,2],[1,3],[1,5], [2,3],[3,4],[4,5]])$
degree_sequence(g5);
```

[2,2,2,3,3]

```
isomorphism(g4,g5);
```

[4→3,5→5,3→2,1→4,2→1]

Ejemplo 2:

```
g6: create_graph([1,2,3,4,5],[[1,4],[1,5],[2,4], [2,5],[3,4],[3,5]])$
degree_sequence(g6);
```

[2 ,2 ,2 ,3 ,3]

```
isomorphism(g4,g6);
```

[ ]

Grafos ponderados

1 Definir el grafo ponderado

Estructura: ([nombre de los vértices],[[[arista],peso],[[arista],peso],...])

Ejemplo:

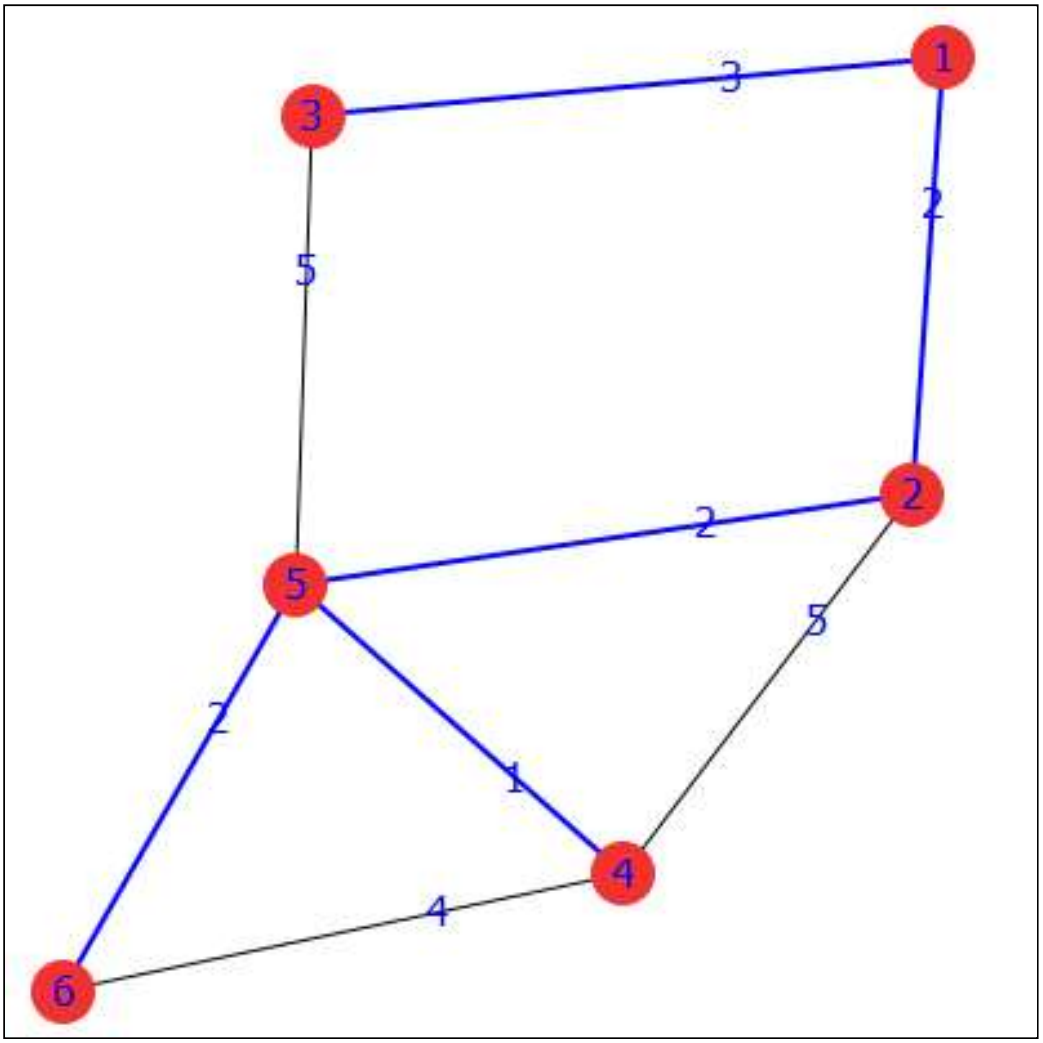
```
grp:create_graph([1,2,3,4,5,6],[ [[1,2],2], [[1,3],3], [[2,4],5], [[2,5],2], [[3,5],5], [[4,5],
```

2 Calcular árbol generador minimal

```
mstgrp: minimum_spanning_tree(grp)$
```

3 Representar

```
draw_graph(grp,vertex_size=4,show_id=true, show_weight=true, show_edges=edge
```



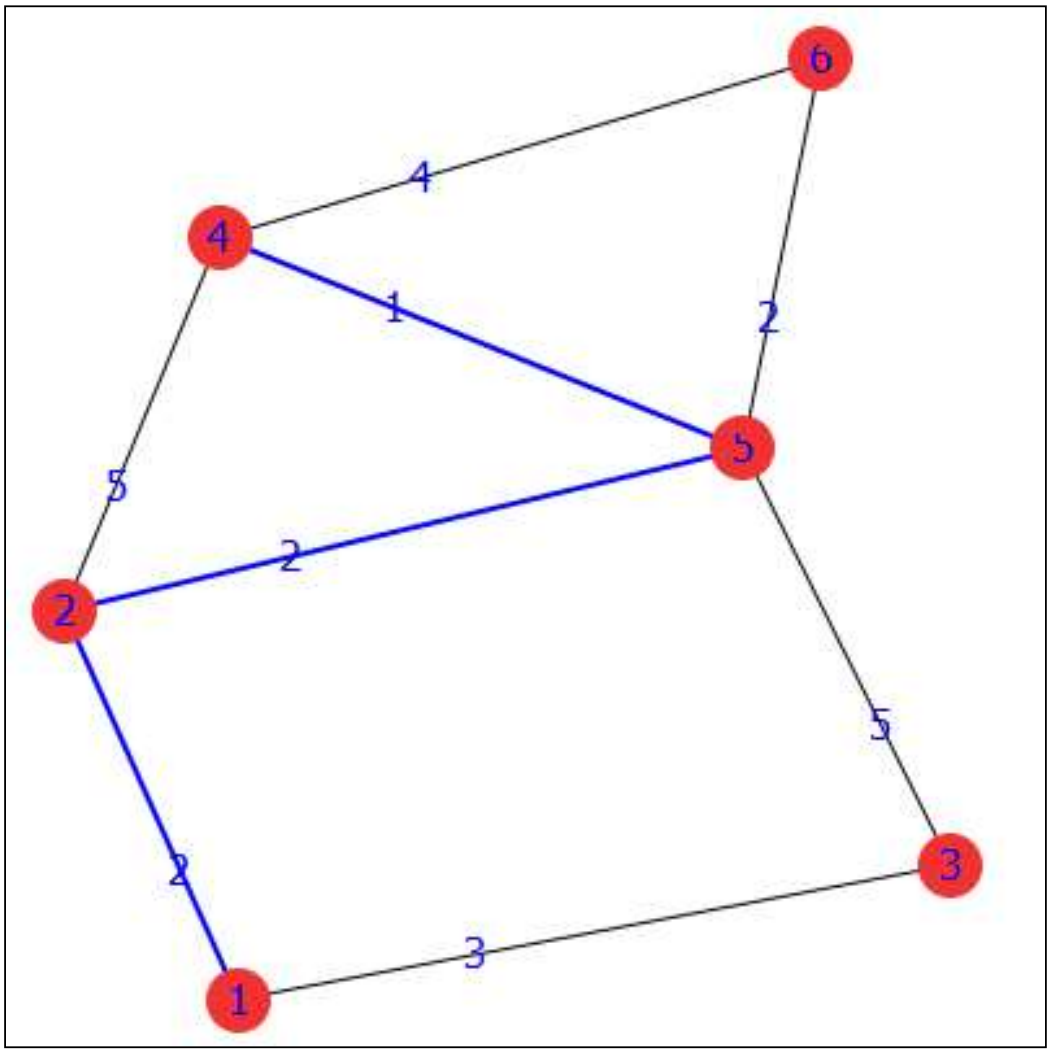
done

4 Algoritmo de Dijkstra

```
dijkgrp: shortest_weighted_path(1,4,grp);
```

[5,[1,2,5,4]]

```
draw_graph(grp,vertex_size=4,show_id=true,show_weight=true,show_edges=vertice
```



done