```prolog
% Ejercicio 1

progenitor(angel,elena).
progenitor(maite,elena).

progenitor(angel1,angel).
progenitor(teresa,angel).
progenitor(angel1,baby).
progenitor(teresa,baby).
progenitor(angel1,ramon).
progenitor(teresa,ramon).

progenitor(ramon,ramonin).

hombre(angel).
hombre(angel1).
hombre(ramon).

mujer(maite).
mujer(teresa).


% X es padre de Y
padre(X,Y):-  progenitor(X,Y), hombre(X).

% X es madre de Y
madre(X,Y):- progenitor(X,Y), mujer(X).

% X e Y son hermanos/as
hermanos(X,Y):-
   madre(Z,X),
   madre(Z,Y),
   padre(T,X),
   padre(T,Y).

% X es el tio de Y
tio(X,Y) :- hermanos(X,Z), progenitor(Z,Y).

%X e Y son primos
primos(X,Y):- progenitor(Z,X), tio(Z,Y).

% X es antepasado de Y
antepasado(X,Y) :- progenitor(X,Y).
antepasado(X,Y):- progenitor(X,Z), antepasado(Z,Y).
```

```
%X es descendiente de Y
descendiente(X,Y):-antepasado(Y,X).

%X e Y son parientes (creo que esto esta mal)
pariente(X,Y) :- descenciente(X,Y).
pariente(X,Y) :- antepasado(X,Y).
pariente(X,Y) :- hermanos(X,Y)
pariente(X,Y) :- tio(X,Y)
pariente(X,Y) :- primos(X,Y)


% Ejercicio 2
arista(a,b).
arista(b,c).

camino(X,Y) :-  arista(X,Y).
camino(X,Y) :- arista(X,Z) , camino(Z,Y).
camino(X,Y) :- arista(Y,Z),  camino(Z,Y).
(También se puede poner camino(X,Y) :- camino(Y,X). en vez de la última)

% Ejercicio 3

arista(a,b).
arista(b,c).
arista(c,d).

% tiene en cuenta el orden
cam(X,Y,[]) :- arista(X,Y).
cam(X,Y,[Z|Zs]):-arista(X,Z), cam(Z,Y,Zs).


% Ejercicio 4
```

a) {p(X, f(Y)) = p(Z,X)}
   Descomposición
   {X=Z, f(Y)=X}
   Ligadura 1
   {X=Z, f(Y)=Z}

b) { f(X, g(X)) = f(g(Z), Y) }
   Descomposición
   { X = g(Z) , g(X) = Y }
   Ligadura 1
   { X = g(Z), g(g(Z)) = Y }

c) { f(X, g(X), g(X,Y)) = f(g(Z), g(W), g(W, W)) }
Descomposición
{ X = g(Z), g(X) = g(W), g(X,Y) = g(W,W) }
Descomposición 3
{ X = g(Z), X = W, X = W, Y = W }
Re orden
{ X = g(Z), X = W, Y = W }
Ligadura 2
{ X = g(Z), X = W, Y = X }
Re orden
{X = g(Z), X = W, X = Y }

d) { f(g(a),X,g(X)) = f(Y,g(h),Y)}
Descomposición
{ g(a) = Y, X =g(h), g(X) = Y }
Ligadura 1
{ g(a) = Y, X = g(h), g(X) = g(a) }
Ligadura 2
{ g(a) = Y, X = g(h), g(g(h)) = g(a) }
Descomposición 3
{ Y = g(a), X = g(h), g(h) = a }

```
% Ejercicio 5
% Funciones auxiliares que uso en este ejercicio
% Concatena dos listas
	conc([],B,B).
	conc([X|D],B,[X|E]) :- conc(D,B,E).

% Usa la función concatenar para ir añadiendo la lista al revés
	invierte([],[]).
	invierte([H|T],L):- invierte(T,R), conc(R,[H],L).

% Compara uno a uno los elementos de dos listas
	igualdad([],[]).
	igualdad([X|Xs],[Y|Ys]) :- X=Y, igualdad(Xs, Ys).

% Ejercicio 5.a
% Dadas dos listas, invierte una y las compara
	inversa(X,Y) :- invierte(Y,B), igualdad(X,B).
```

%Ejercicio 5.b
%Ser prefijo? Entiendo que dadas dos listas una es la parte inicial de la otra.
% Uso una función muy parecida a la de igualdad, pero cuando una de las listas es vacía devolvemos True porque si llega a esa posición es que la que es [] era el prefijo de la otra

```
pre([],_).
pre(_,[]).
pre([X|Xs],[Y|Ys]) :- X=Y, pre(Xs, Ys).
```

% Ejercicio 5.c
% Ser sufijo.
% Voy a invertir las listas y ver si son prefijos.

```
suf([],_).
suf(_,[]).
suf(Xs, Ys) :- invierte(Xs,Fs), invierte(Ys,Gs), pre(Fs,Gs).
```

% Ejercicio 6
% Ejercicio 6.a: Pertenece un elemento a la lista
% Si el primer elemento de una lista, es el mismo que queremos saber si % pertenece a la lista entonces True, sino eliminamos el primer
% elemento de la lista y seguimos comparendo.

```
pertenece(X,[Y|_]) :- X=Y.
pertenece(X,[_|Ys]) :- pertenece(X,Ys).
```

% Ejercicio 6.b: La segunda lista es equivalente a eliminar de la primera
% lista todas las apariciones del elemento dado.

% Primero hacer una función que elimine un elemento dado de una
% lista. Si la lista es vacía se deja como está, si el primer elemento de la % lista es el mismo que el elemento no se añade a la lista que quiero
% que devuelva, en caso contrario se añade.

% elim(lista larga, elemento, lista sin el elemento).

```
elim([],_,_).
elim([X|Xs],Y,B) :- X=Y, elim(Xs, Y, B).
elim([X|Xs],Y,[X|B]) :- elim(Xs, Y, B).
```

% Primero elimino el elemento de la lista y luego uso la función igualdad.

% ej6b(listalarga, elemento, lista corta).

```
ej6b(Xs, X, Ys) :- elim(Xs,X,B), igualdad(B, Ys).
```

% Ejercicio 6.c: La segunda lista es equivalente a eliminar de la primera
% lista la primera aparición del elemento dado.
% Primero creo la función que elimina de una lista la primera aparición de un elemento dado.
% Si el primer elemento de la lista es el que queremos eliminar devuelve la lista sin el primer elemento, en caso contrario añade el elemento a la otra lista.

% elim1(elemento, lista con elemento, lista sin elemento).

```
elim1(X, [X|Xs], Xs).
elim1(X, [Y|Ys], [Y|Zs]) :- elim1(X, Ys, Zs).
```

% Usar la función elim1 y luego la función igualdad.
%ej6c(Lista larga, elemento que queremos eliminar, lista corta):

```
ej6c(Xs, X, Ys) :- elim1(X, Xs, B), igualdad(B,Ys).
```

% Ejercicio 7
% La función coge el último elemento que se repite.
% Caso base: Solo queda un elemento en la lista
% Casos recursivos: si los dos primeros elementos de la lista dada son
% iguales eliminamos el primero y no ponemos nada en la lista que
% devuelve.
% Si los dos primeros elementos de la lista son distintos, pongo en la
% segunda lista el primer elemento.

```
sinDuplicados([X],[X]).
sinDuplicados([X,X|Xs], Ys) :- sinDuplicados([X|Xs],Ys),!.
sinDuplicados([X,Y|Xs],[X|Ys]) :- sinDuplicados([Y|Xs], Ys),!.
```

% Ejercicio 8
% Operador ++
% Caso base: cuando una de las listas es vacía devuelve la otra lista
% Caso recursivo: voy añadiendo los elementos de la primera lista hasta % llegar al caso base.

```
masmas([],B,B).
masmas([X|D],B,[X|E]) :- masmas(D,B,E).
```

%concat Haskell
% Dada una lista de listas, va cogiendo la primera lista y usa la función
% anterior para ir añadiéndola a una lista de elementos.

```
concat([],_).
concat([X|D],C) :- masmas(X,B,C), concat(D,B).
```

% Ejercicio 9

```
Arbol =:= vacioB |nodoB( Raiz, Arbol,Arbol)
```

% A =nodoB(4,nodoB(2,nodoB(1,vacioB,vacioB),nodoB(3,vacioB,vacioB))
,nodoB(6,nodoB(5,vacioB,vacioB),nodoB(7,vacioB,vacioB))).

```
conc([],B,B).
conc([X|D],B,[X|E]) :- conc(D,B,E).

conc3([],[],[],[]).
conc3(A,B,C,D) :- conc(A,B,D1), conc(D1,C,D).
```

% Tienen las tres la misma estructura, pero cuando concatenan, lo hacen de forma
% distinta.

```
preorden(vacioB,[]).
preorden(nodoB(X,Iz,Dr),Xs) :-
        preorden(Iz,Is), preorden(Dr,Ds), conc3([X],Is,Ds,Xs).

inorden(vacioB, []).
inorden(nodoB(X,Iz,Dr),Xs) :-
        inorden(Iz,Is),inorden(Dr,Ds), conc3(Is,[X],Ds,Xs).

postorden(vacioB, []).
postorden(nodoB(X,Iz,Dr),Xs) :-
        postorden(Iz, Is), postorden(Dr,Ds), conc3(Is, Ds, [X],Xs).
```

% Ejercicio 10: No funcionan
% Función take
% Cuando el contador es cero devuelve [], mientras el contador sea
% distinto de cero añade un elemento a la lista que tiene que devolver.

```
take(_,[],[]).
take(cero,_,[]).
take(s(n),[X|Xs],[X|Ys]) :- take(n,Xs,Ys).
```

% Función drop
% Cuando el contador es cero devuelve lo que queda de la lista,
% mientras el contador es distinto de cero elimina el primer elemento
% de la lista.

```
drop(_,[],[]).
drop(cero,B,B).
drop(s(n),[_|Xs],Zs):-  drop(s(n), Xs, Zs).
```

% Función splitat
% Cuando el contador es cero, devuelve como prefijo [] y como sufijo lo % que queda de la lista dada.
% En el caso recursivo vamos añadiendo los n primeros elementos de la % lista dada a la lista prefijo y no cambiamos la lista sufijo.

% splitat(numero, lista entera, prefijo,sufijo).

```
splitat(_,[],[],[]).
splitat(cero,X,[],X).
splitat(s(n),[X|Xs],[X|Ys],B) :- splitat(n,Xs,Ys,B).
```

% Ejercicio 11: He usado las mismas ideas que en el ejercicio anterior
% pero en este caso si funcionan.

% Función take

```
taKe(_,[],[]).
taKe(0,_,[]).
taKe(N,[X|Xs],[X|Ys]) :- N1 is N-1,taKe(N1,Xs,Ys).
```

% Función drop

```
droP(_,[],[]).
droP(0,B,B).
droP(N,[_|Xs],Zs):- N1 is N-1, droP(N1, Xs, Zs).
```

% splitAt

```
splitAt(_,[],[],[]).
splitAt(0,X,[],X).
splitAt(N,[X|Xs],[X|Ys],B) :- N1 is N-1,splitAt(N1,Xs,Ys,B).
```

% Ejercicio 12:
% máximo común divisor de dos naturales.
% Caso base: si hay un 0 y un elemento positivo X, devuelve X.
% Casos recursivos: Cuando X >= Y , devuelve el máximo común divisor % de X-Y e Y.
% Cuando X<Y, devuelve el máximo común divisor de Y-X y X.

```
gcd(0, X, X):- X > 0, !.
gcd(X, Y, Z):- X >= Y, X1 is X-Y, gcd(X1,Y,Z).
gcd(X, Y, Z):- X < Y, X1 is Y-X, gcd(X1,X,Z).
```

% Ejercicio 14:
% Factorial de un numero
% Caso base: En cero el factorial es 1
% Caso recursivo, si X>0, devolvemos el factorial de X-1, y a Y le multiplicamos X.

```
fact(0,1).
fact(X,Y) :- X>0, X1 is X-1, fact(X1, Y1),Y is Y1*X,!.
```

% Sumatorio de los elementos de una lista
% Coge el primer elemento de la lista y lo va sumando a B, hasta llegar a la lista de un
% solo elemento y suma ese elemento.

```
suml([X],X).
suml([Y|Ys], B) :- suml(Ys,B1),B is Y+B1,!.
```

% Elemento máximo de una lista
% Primero he creado la función que devuelve el máximo entre dos elementos

```
max(X,Y,Y) :- X<Y.
max(X,Y,X) :- X>=Y.
```

% Si la lista es de un solo elemento devuelve ese elemento. Cuando la lista tiene mas de % un elemento coge el primer elemento y lo compara con Y, devuelve la función
% recursiva sin el primer elemento de la lista y con Y' = máximo{X,Y}

```
maxl([X],X).
maxl([X|Xs],Y) :- maxl(Xs, Z), max(X,Z,Y).
```

% Calcular el producto escalar de los elementos de dos listas
% Usa el mismo método que en la función suml, pero en vez de añadir un elemento de
% una lista, sumo el elemento X*Y siendo X e Y los primeros elementos de dos listas
% dadas.

```
prod([X],[Y],Z) :- Z is X*Y.
prod([X|Xs],[Y|Ys],Z) :- prod(Xs, Ys,Z1),Z is X*Y+Z1,!.
```

% Calcular la suma de matrices
% Suma los elementos de dos listas
% Crea una lista con los elementos X+Y, siendo X e Y los elementos de dos listas dadas

```
	sum([X],[Y],[Z]) :- Z is X+Y.
	sum([X|Xs],[Y|Ys],[Z|Zs]) :- sum(Xs,Ys,Zs),Z is X+Y,!.
```

% Suma de matrices
% Dadas dos listas de listas, usa la función anterior en los elementos de estas

```
	summ([X],[Y],[Z]) :- sum(X,Y,Z).
	summ([X|Xs],[Y|Ys],[Z|Zs]) :- summ(Xs,Ys,Zs), sum(X,Y,Z).
```

% Ejercicio 15:

% Idea: Cojo el primer elemento de una lista y pongo los elementos más pequeños en % una lista a la izq y los más grandes a la derecha sin ningún orden. Luego hago lo mismo % en las listas. Y lo uno todo en una lista.

% Función que dada una lista devuelve la lista de elementos
% mas pequeños y la lista de elementos más grandes y la lista con el primer elemento

```
	mayorMenor(_,[],[],[]).
	mayorMenor(X,[Y|Xs],[Y|Me],Ma) :- X>=Y, mayorMenor(X,Xs,Me,Ma).
	mayorMenor(X,[Y|Xs],Me,[Y|Ma]) :- X<Y, mayorMenor(X,Xs,Me,Ma).

	conc([],B,B).
	conc([X|D],B,[X|E]) :- conc(D,B,E).

	conc3([],[],[],[]).
	conc3(A,B,C,D) :- conc(A,B,D1), conc(D1,C,D).
```

% Dada una lista, coge el primer elemento de la lista y divide la lista en los elementos % que son más menores que el primer elemento y los que son mayores. Hace lo mismo % en esas dos listas y por último concatena de forma que la lista de elementos menores % vaya a la izquierda, el elemento en medio y por ultimo la lista de elementos mayores.

```
	quicksort([],[]).
	quicksort([X|Xs],Ys) :-
		mayorMenor(X, Xs, Me,Ma),quicksort(Me,Me1), quicksort(Ma,Ma1),
		conc3(Me1,[X],Ma1,Ys).
```

```
% Ejercicio 16:
% ArbolB =:= vacioB |nodoB( Raiz, ArbolB,ArbolB)
% A = nodoB(4,nodoB(2,nodoB(1,vacioB,vacioB),nodoB(3,vacioB,vacioB)),
% nodoB(6,nodoB(5,vacioB,vacioB),nodoB(7,vacioB,vacioB))).

% Buscar un elemento
% Caso base, si X es un nodo devuelve True. Si no busca en el árbol Iz y si no en Dr.
	buscarArbol(X, nodoB(X,_,_)).
	buscarArbol(X, nodoB(_,Iz,_)) :- buscarArbol(X,Iz).
	buscarArbol(X, nodoB(_,_,Dr)) :- buscarArbol(X,Dr).

% Añadir un elemento
% Casos base: Si es un árbol vacío devuelve nodoB(X,vacio,vacio). Si el elemento ya
% está no lo pone otra vez.
% Casos recursivos, si el nodo con el que estamos comparando es mayor que X
% añadimos  X en la parte izquierda del árbol. En el otro caso, añadimos X a la parte
% derecha del árbol.
	anadir(X,vacioB,nodoB(X,vacioB,vacioB)).
	anadir(X,nodoB(X,A1,A2),nodoB(X,A1,A2)).
	anadir(X,nodoB(Y,A1,A2),nodoB(Y,A1N,A2)) :- X<Y, anadir(X,A1,A1N).
	anadir(X,nodoB(Y,A1,A2),nodoB(Y,A1,A2N)) :- X>Y, anadir(X,A2,A2N).

% Eliminar un elemento
% pasar de Arbol a lista
	conc([],B,B).
	conc([X|D],B,[X|E]) :- conc(D,B,E).

	conc3([],[],[],[]).
	conc3(A,B,C,D) :- conc(A,B,D1), conc(D1,C,D).

	inorden(vacioB, []).
	inorden(nodoB(X,Iz,Dr),Xs) :- inorden(Iz,Is),inorden(Dr,Ds), conc3(Is,[X],Ds,Xs).

% eliminar el elemento de la lista
	elimLista(_,[],[]).
	elimLista(X,[X|Xs],Xs) :- elimLista(X,Xs,Xs).
	elimLista(X,[Y|Xs],[Y|Ys]) :- elimLista(X,Xs,Ys).

% Crear Arbol a partir de lista
	crearArbol([X],nodoB(X, vacioB, vacioB)).
	crearArbol([X|Xs],A) :-  crearArbol(Xs,A1), anadir(X,A1,A).

%Función que elimina un elemento de un árbol
	elimArbol(X,A,A1) :- inorden(A,L), elimLista(X,L,L1), crearArbol(L1,A1).
```