**Entre com um número e retorne se ele é maior ou menor que 100**

maiorQueCem(X) :- write(Entre com o número:'),

read(X),

(

(X>100, write('O número é maior que cem'));

(X<100, write('O número é menor que cem'))

).

**Entre com uma nota de aluno e descubra se ele foi aprovado, reprovado ou está em recuperação, lembrando que o máximo que o aluno pode tirar é 10. Se a sua nota for maior ou igual a 7 e menor que 10, ele está aprovado, se a nota for maior ou 5 e menor ou igual a 6.9, ele está em recuperação e se a nota for menor que 4.9, ele está reprovado.**

situacaoAluno (Aluno):- nota(Aluno, Nota), (( Nota >= 7.0 , Nota < 10.0, write('Aprovado'));

( Nota >= 5.0, Nota < 6.9, write('Recuperação'));

( Nota >= 0.0, Nota < 4.9, write('Reprovado'));).

**Saber se um número é membro de uma lista:**

membro(X, [X|\_]).

membro(X,[\_|R]) :- membro(X,R).

**Remove um único elemento**

remove(X, [X|C], C).

remove(X, [Y|Z], [Y|P]) :- remove(X, Z, P).

**Remove todos os elementos**

remover(\_, [], []).

remover(X, [X|C], R) :- remover(X, C, R).

remover(X, [Y|C], [Y|Z]) :- remover(X, C, Z).

**Conta quantos elementos existem em uma lista:**

contagem([], 0).

contagem([\_|T], N) :- contagem(T, N1), N is N1+1.

**Soma todos os elementos de uma lista:**

soma([], 0).

soma([X|W], S):- soma(W, S1), S is S1 + X.

**Multiplicação de todos os elementos da lista:**

multiplicacao([], 0).

multiplicacao(L, P) :- produto(L,P) .

produto([], 1).

produto([X|W], S) :- produto(W, S1), S is S1 \* X.

**Obter o primeiro elemento de uma lista:**

p\_elemento([X|T], X).

**Obter o último elemento de uma lista:**

u\_elemento([X], X) :- !. u\_elemento([\_|T], L) :- u\_elemento(T, L).

**Mergear 02 listas distintas:**

merge([], [], []).

merge(H, [], H).

merge([], H, H).

merge([H|T], L, [H|Z]) :- merge(T, L, Z).

**Lista N números em ordem decrescente de N para 1**

liste(0, [0]).

liste(X, [X|Y]) :- X>0, Z is X-1, liste(Z, Y).

**Lista N números em ordem crescente de 1 para N**

% Predicado para gerar uma lista de números de 1 até N

gerar\_lista\_ate(N, Lista) :-

gerar\_lista\_ate\_aux(1, N, Lista).

% Caso base: quando o primeiro número atinge N, paramos a recursão

gerar\_lista\_ate\_aux(N, N, [N]).

% Caso recursivo: incrementamos o número atual e continuamos a recursão

gerar\_lista\_ate\_aux(Current, N, [Current|Resto]) :-

Current < N,

Next is Current + 1,

gerar\_lista\_ate\_aux(Next, N, Resto).

**Junte duas listas:**

% Predicado para juntar duas listas

juntar\_listas([], L, L). % Se a primeira lista for vazia, a segunda lista é a resposta

juntar\_listas([X|L1], L2, [X|L3]) :- juntar\_listas(L1, L2, L3). % Recursivamente juntar o restante da primeira lista com a segunda lista

**Juntar duas listas mas a segunda vem na frente**

% Predicado para concatenar duas listas, onde a segunda lista vem na frente concatena\_segunda\_na\_frente(L1, L2, Concatenada) :- juntar\_listas (L2, L1, Concatenada).

**Juntar duas listas intercalando os elementos**

% Predicado para concatenar duas listas intercalando os elementos

concatenar\_intercalando([], L, L).

concatenar\_intercalando(L, [], L).

concatenar\_intercalando([X | L1], [Y | L2], [X, Y | Resultado]) :-

concatenar\_intercalando(L1, L2, Resultado).

**Inserir um elemento em uma lista (no inicio)**

% Predicado para inserir um elemento no início de uma lista

inserir\_inicio(Elemento, Lista, [Elemento|Lista]).

**Inserir um elemento em uma lista (final)**

% Predicado para inserir um elemento no final de uma lista

inserir\_final(Elemento, [], [Elemento]). % Caso base: se a lista for vazia, o resultado é uma lista contendo apenas o Elemento inserido

inserir\_final(Elemento, [X|Resto], [X|NovaLista]) :- inserir\_final(Elemento, Resto, NovaLista).

**Ordenar uma lista de forma crescente**

% Predicado principal para ordenar uma lista utilizando insertion sort

insertion\_sort(List, Sorted) :-

insertion\_sort\_acc(List, [], Sorted).

% Caso base: quando a lista não ordenada está vazia, o resultado é a lista ordenada acumulada

insertion\_sort\_acc([], Acc, Acc).

% Caso recursivo: insere o primeiro elemento da lista não ordenada na lista ordenada acumulada

insertion\_sort\_acc([Head|Tail], Acc, Sorted) :-

insert(Head, Acc, NewAcc),

insertion\_sort\_acc(Tail, NewAcc, Sorted).

% Predicado para inserir um elemento em uma lista ordenada

insert(X, [], [X]).

insert(X, [Y|Tail], [X,Y|Tail]) :-

X =< Y.

insert(X, [Y|Tail], [Y|Result]) :-

X > Y,

insert(X, Tail, Result).

**Ordena uma lista de forma decrescente**

% Predicado principal para ordenar uma lista em ordem decrescente utilizando insertion sort

insertion\_sort\_desc(List, Sorted) :-

insertion\_sort\_desc\_acc(List, [], Sorted).

% Caso base: quando a lista não ordenada está vazia, o resultado é a lista ordenada acumulada

insertion\_sort\_desc\_acc([], Acc, Acc).

% Caso recursivo: insere o primeiro elemento da lista não ordenada na lista ordenada acumulada

insertion\_sort\_desc\_acc([Head|Tail], Acc, Sorted) :-

insert\_desc(Head, Acc, NewAcc),

insertion\_sort\_desc\_acc(Tail, NewAcc, Sorted).

% Predicado para inserir um elemento em uma lista ordenada de forma decrescente

insert\_desc(X, [], [X]).

insert\_desc(X, [Y|Tail], [X,Y|Tail]) :-

X >= Y.

insert\_desc(X, [Y|Tail], [Y|Result]) :-

X < Y,

insert\_desc(X, Tail, Result).

**Separa números pares e impares**

% Predicado principal para separar números pares e ímpares em uma lista de listas

split\_even\_odd\_list([], [[], []]).

split\_even\_odd\_list([X|Xs], [[X|Evens], Odds]) :-

X mod 2 =:= 0,

split\_even\_odd\_list(Xs, [Evens, Odds]).

split\_even\_odd\_list([X|Xs], [Evens, [X|Odds]]) :-

X mod 2 =\= 0,

split\_even\_odd\_list(Xs, [Evens, Odds]).

**Elimina repetições de números consecultivos**

group\_equal([], []).

group\_equal([X], [[X]]).

group\_equal([X, X | T], [[X | Group] | Resto]) :-

group\_equal([X | T], [[X | Group] | Resto]).

group\_equal([X, Y | T], [[X] | Resto]) :-

X \= Y,

group\_equal([Y | T], Resto).

Exemplo:

?- group\_equal([1, 1, 2, 3, 3, 3, 4, 1, 1, 1], Result).

Result = [[1], [2], [3], [4], [1]].

**Agrupar elementos iguais consecutivos em uma lista**

% Predicado para agrupar elementos iguais consecutivos em uma lista

group\_equal([], []).

group\_equal([X], [[X]]).

group\_equal([X, X | T], [[X, X | Group] | Resto]) :-

group\_equal([X | T], [[X | Group] | Resto]).

group\_equal([X, Y | T], [[X] | Resto]) :-

X \= Y,

group\_equal([Y | T], Resto).

Exemplo:

?- group\_equal([1, 1, 2, 3, 3, 3, 4, 1, 1, 1], Result).

Result = [[1, 1], [2], [3, 3, 3], [4], [1, 1, 1]].

**Divide lista**

divide([],\_,[],[]).

divide(L, 0, [], L):- !.

divide(L, N, \_, \_) :-

length(L, Len),

N > Len, !, fail.

divide([X|R],N,[X|L1],L2):-

M is N-1,

divide(R,M,L1,L2).

**Conta o número de repetições de cada elemento agrupando em uma lista**

% Predicado para codificar uma lista em uma lista de listas

codifique([], []).

codifique([X | Resto], [[N, X] | Codificado]) :-

contar\_ocorrencias([X | Resto], X, N, SemXs),

codifique(SemXs, Codificado).

% Predicado para contar o número de ocorrências consecutivas de um elemento na lista

contar\_ocorrencias([], \_, 0, []).

contar\_ocorrencias([X | Resto], X, N, SemXs) :-

contar\_ocorrencias(Resto, X, N1, SemXs),

N is N1 + 1.

contar\_ocorrencias([Y | Resto], X, 0, [Y | Resto]) :-

X \= Y.

**Testar se dois conjuntos C1 e C2 representados por listas são iguais (a ordem dos elementos não importa).**

% Predicado para verificar se dois conjuntos são iguais (ordem dos elementos não importa)

iguais(C1, C2) :-

permutation(C1, C2). % C1 é uma permutação de C2

% O predicado permutation/2 é definido recursivamente:

permutation([], []).

permutation([X|Xs], Ys) :-

remove(X, Ys, Zs), % Remove X de Ys, obtendo Zs

permutation(Xs, Zs). % Verifica se Xs é uma permutação de Zs

remove(X, [X|C], C).

remove(X, [Y|Z], [Y|P]) :- remove(X, Z, P).

**Concatena e ordena uma Lista de Listas:**

mescleLL([], []).

mescleLL([L], L) :- !.

mescleLL([L1,L2|Resto], Merged) :-

merge\_sorted\_lists(L1, L2, MergedTemp),

mescleLL([MergedTemp|Resto], Merged).

% Predicado para mesclar duas listas ordenadas

merge\_sorted\_lists([], L, L).

merge\_sorted\_lists(L, [], L) :- L \= [].

merge\_sorted\_lists([X|Xs], [Y|Ys], [X|Merged]) :-

X =< Y,

merge\_sorted\_lists(Xs, [Y|Ys], Merged).

merge\_sorted\_lists([X|Xs], [Y|Ys], [Y|Merged]) :-

X > Y,

merge\_sorted\_lists([X|Xs], Ys, Merged).

Exemplo:

?- mescleLL([[4,7],[1,3,6],[2,5,8]],L).

X = [1,2,3,4,5,6,7,8].

**Concatenar uma lista de listas**

concatenar\_lista\_de\_listas([], []).

concatenar\_lista\_de\_listas([Lista|ListasRestantes], Concatenada) :-

concatenar\_lista\_de\_listas(ListasRestantes, ConcatenadaRestante),

acrescentar(Lista, ConcatenadaRestante, Concatenada).

acrescentar([], Lista, Lista).

% Caso recursivo: concatenar duas listas, decompondo a primeira lista e construindo a lista concatenada.

acrescentar([X|Xs], Lista2, [X|Resultado]) :-

acrescentar(Xs, Lista2, Resultado).

Exemplo:

?- concatenar\_lista\_de\_listas([[a],[b,c],[d,e,f]], Concatenada).

Concatenada = [a, b, c, d, e, f].

**decodifica uma lista de listas onde cada sublista contém um par [N, A] indicando quantas vezes o átomo A deve ser repetido em L**

decodifica(LL, L) :-

decodifica(LL, [], L).

decodifica([], Acc, Acc).

decodifica([[N, A]|Resto], Acc, L) :-

replicar\_elemento(A, N, Elementos),

append(Acc, Elementos, Acc1),

decodifica(Resto, Acc1, L).

replicar\_elemento(\_, 0, []).

replicar\_elemento(A, N, [A|Resto]) :-

N > 0,

N1 is N - 1,

replicar\_elemento(A, N1, Resto).

Exemplo:

?- decodifica([[3,a],[2,b],[2,a],[1,d]], L).

L = [a, a, a, b, b, a, a, d].

**União de dois conjuntos**

% Verifica se um elemento pertence a uma lista

pertence(X, [X|\_]).

pertence(X, [\_|T]) :- pertence(X, T).

% Remove elementos duplicados de uma lista

sem\_repeticoes([], []).

sem\_repeticoes([H|T], L) :-

pertence(H, T),

!,

sem\_repeticoes(T, L).

sem\_repeticoes([H|T], [H|L]) :-

sem\_repeticoes(T, L).

% Predicado para a união de dois conjuntos

uniao([], Conjunto, Conjunto).

uniao([X|Conj1], Conj2, Resultado) :-

pertence(X, Conj2),

!,

uniao(Conj1, Conj2, Resultado).

uniao([X|Conj1], Conj2, [X|Resultado]) :-

uniao(Conj1, Conj2, Resultado).

**Diferença dos conjuntos**

% Verifica se X está na lista L

member(X, [X|\_]).

member(X, [\_|T]) :- member(X, T).

% Remove um elemento X da lista L, resultando em L1

removerr(\_, [], []).

removerr(X, [X|T], T) :- !.

removerr(X, [H|T], [H|T1]) :- removerr(X, T, T1).

% Calcula a diferença entre duas listas A e B

difference([], \_, []).

difference([H|T], B, Diff) :-

member(H, B), % Se H está em B, não inclui H na diferença

!,

difference(T, B, Diff).

difference([H|T], B, [H|Diff]) :-

difference(T, B, Diff).

**Interseção de dois conjuntos**

% Verifica se X está na lista L

my\_member(X, [X|\_]).

my\_member(X, [\_|T]) :-

my\_member(X, T).

% Calcula a interseção entre duas listas A e B

my\_intersection([], \_, []).

my\_intersection([H|T], B, [H|Result]) :-

my\_member(H, B),

!,

my\_intersection(T, B, Result).

my\_intersection([\_|T], B, Result) :-

my\_intersection(T, B, Result).