%Сумма элементов вверх

sum\_el([], 0) :- !.

sum\_el([H|T], Sum) :- sum\_el(T, SumT), Sum is H + SumT.

% Сумма элементов вниз

% sum\_el\_down(+List, -Sum)

% Sum - это сумма всех элементов списка List, используя аккумулятор.

sum\_el\_down(List, Sum) :-

sum\_el\_down(List, 0, Sum).

% sum\_el\_down(+List, +CurSum, -Sum)

% Вспомогательный предикат для суммы элементов вниз.

% CurSum - текущая сумма, которая обновляется на каждом шаге.

sum\_el\_down([], Sum, Sum) :- !.

sum\_el\_down([H|T], CurSum, Sum) :-

NewSum is CurSum + H,

sum\_el\_down(T, NewSum, Sum).

% Предикат maxList(+List, Ind), определяющий индекс элемента списка, имеющего максимальную сумму цифр.

%Сумма цифр числа, рекурсия вверх

summ\_cifr(0,0):-!.

summ\_cifr(X,SummCifr):-X1 is X // 10, Ost is X mod 10,

summ\_cifr(X1, SummCifr1), SummCifr is SummCifr1 + Ost.

%Число с максимальной суммой цифр в списке

max\_summ\_cifr\_el([A], A) :- !.

max\_summ\_cifr\_el([CurEl|Tail], Res1) :-

summ\_cifr(CurEl,SCCurEl), max\_summ\_cifr\_el(Tail, Res1),

summ\_cifr(Res1, ScRes1), SCCurEl < ScRes1, !.

max\_summ\_cifr\_el([CurEl|\_], CurEl).

%Номер заданного элемента в списке

list\_el\_numb(List,El,Numb) :- list\_el\_numb(List,El,Numb,1).

list\_el\_numb([],\_,\_,\_) :- !,fail.

list\_el\_numb([El|\_],El,Numb,Numb).

list\_el\_numb([\_|T],El,Numb,Ind) :- NewInd is Ind + 1,

list\_el\_numb(T,El,Numb,NewInd).

% Индекс элемента списка, имеющего максимальную сумму цифр.

maxList(List,Numb):- max\_summ\_cifr\_el(List, X), list\_el\_numb(List,X,Numb).

% Предикат fib(+N,?A), проверяющий является ли число А числом Фибоначчи с

% номером N или находящий число с таким номером и записывающим его в A.

fib(1,1):-!. %рекурсия вверх

fib(2,1):-!.

fib(N,X):-N1 is N-1, N2 is N-2, fib(N1,X1), fib(N2,X2), X is X1+X2.

fib1(N,X):-fib2(1,1,2,N,X). %рекурсия вниз

fib2(\_,B,N,N,B):-!.

fib2(A,B,I,N,X):- I1 is I+1, C is A+B, fib2(B,C,I1,N,X).

% Факториал (перестановки) рекурсия вверх

fact(0,1).

fact(N,X):- N>0, N1 is N-1, fact(N1,X1), X is X1\*N.

%Факториал (перестановки) рекурсия вниз

fact2(0,X,X).

fact2(N,A,X):- N>0,

N1 is N-1, A1 is A\*N, fact2(N1,A1,X1), X is X1.

call\_fact2(N,X):-fact2(N,1,X).

%Простое число

pr(2):-!.

pr(X):-pr1(X,2).

pr1(X,X):-!.

pr1(X,I):- Y is X mod I, Y\=0, I1 is I+1, pr1(X,I1).

%Наибольший простой делитель

n\_pr(N,X):-npr(N,N,X).

npr(N,I,I):- Y is N mod I, Y=0, pr(I),!.

npr(N,I,X):- I1 is I-1, npr(N,I1,X).

% НОД (наибольший общий делитель)

nod(A,0,A):-!.

nod(\_,0,\_):-!,fail.

nod(A,B,C):-Ost is A mod B, nod(B,Ost,C).

%Взаимно простые числа с числом A (их колво от 1 до A)

euler(A,N):-euler(1,A,0,N).

euler(A,A,N,N):-!.

euler(I,A,Count,N):-nod(I,A,1),!, K is Count+1, I1 is I+1,euler(I1,A,K,N).

euler(I,A,Count,N):-I1 is I+1,euler(I1,A,Count,N).

%Сумма всех делителей числа A, кроме самого числа A

pr\_sd(A,N):-pr\_sd(A,1,N,0).

pr\_sd(A,A,N,N):-!.

pr\_sd(A,I,N,Sum):- Ost is A mod I, Ost=0, !, S is Sum+I, I1 is I+1,

pr\_sd(A,I1,N,S).

pr\_sd(A,I,N,Sum):-I1 is I+1, pr\_sd(A,I1,N,Sum).

%Сумма элементов списка, рекурсия вверх

sum\_el([], 0) :- !.

sum\_el([H|T],Sum) :- sum\_el(T, SumT), Sum is H + SumT.

%Сумма элементов списка, рекурсия вниз

sum\_el\_down(List, Sum) :- sum\_el\_down(List, 0, Sum).

sum\_el\_down([], Sum, Sum) :- !.

sum\_el\_down([H|T],CurSum, Sum) :- NewSum is CurSum + H,

sum\_el\_down(T, NewSum, Sum).

%Сумма простых элементов списка

sum\_pr\_el\_down(List, Sum) :- sum\_pr\_el\_down(List, 0, Sum).

sum\_pr\_el\_down([], Sum, Sum) :- !.

sum\_pr\_el\_down([H|T],CurSum, Sum) :- pr(H), NewSum is CurSum + H,

sum\_pr\_el\_down(T, NewSum, Sum), !.

sum\_pr\_el\_down([\_|T],CurSum, Sum) :-

sum\_pr\_el\_down(T, CurSum, Sum).

%Максимальный элемент в списке

max\_in\_list([H|T],Imax):-max\_in\_list(T,H,1,2,Imax).

max\_in\_list([],\_,Cur,\_,Cur):-!.

max\_in\_list([H|T],Max,Cur,Ind,Imax):-H>Max,Ind1 is Ind+1,max\_in\_list(T,H,Ind,Ind1,Imax),!.

max\_in\_list([\_|T],Max,Cur,Ind,Imax):-Ind1 is Ind+1,max\_in\_list(T,Max,Cur,Ind1,Imax).

% Длина строки (считывает строку в список символов A и возвращает ее длину N)

read\_str(A,N):-get0(X),r\_str(X,A,[],N,0).

r\_str(10,A,A,N,N):-!.

r\_str(X,A,B,N,K):-K1 is K+1,append(B,[X],B1),get0(X1),r\_str(X1,A,B1,N,K1).

%Количество вхождений каждого элемента из списка List в список A

count\_elems(\_,[],[]):-!.

count\_elems(A,[H|T],[Cur|Tail]):-count\_el(H,A,Cur),count\_elems(A,T,Tail).

%Количество вхождений элемента El в список List

count\_el(El,List,Count):-count\_el(El,List,Count,0).

count\_el(\_,[],Count,Count):-!.

count\_el(El,[El|T],Count,Cur):-Cur1 is Cur+1, count\_el(El,T,Count,Cur1),!.

count\_el(El,[\_|T],Count,Cur):-count\_el(El,T,Count,Cur).

% Количество слов в строке, представленной списком символов A

get\_word([],[],[]):-!.

get\_word(A,Word,A2):-get\_word(A,[],Word,A2).

get\_word([],Word,Word,[]).

get\_word([32|T],Word,Word,T):-!.

get\_word([H|T],W,Word,A2):-append(W,[H],W1),get\_word(T,W1,Word,A2).

count\_words(A,K):-count\_words(A,0,K).

count\_words([],K,K):-!.

count\_words(A,I,K):-skip\_space(A,A1),get\_word(A1,Word,A2),Word \=[],I1 is I+1,count\_words(A2,I1,K),!.

count\_words(\_,K,K).

% Построить предикат, который выводит на экран все слова длины 6 над

% алфавитом [a,b,c,d,e,f], в которых три буквы a, две буквы b.

generate\_word(Word) :-

Word = [\_, \_, \_, \_, \_, \_],

permutation([a, a, a, b, b, X], Word),

member(X, [c, d, e, f]),

write(Word), nl.

% Построить предикат, который выводит на экран все слова длины 6, в

% которых первые три буквы любые из [a,b,c,d,e] без повторов, остальные

% буквы [v,w,x,y,z] возможно с повторами. Определение возможных символов

% для первых трех и последних трех позиций

first\_three([a,b,c,d,e]).

last\_three([v,w,x,y,z]).

% Генерация и печать всех допустимых слов

generate\_words:-

first\_three(FirstThree),

last\_three(LastThree),

permute(FirstThree, PermutedFirstThree),

select\_three(PermutedFirstThree, Char1, Char2, Char3),

member(Char4, LastThree),

member(Char5, LastThree),

member(Char6, LastThree),

Word = [Char1, Char2, Char3, Char4, Char5, Char6],

write(Word), nl,

fail.

generate\_words.

% Перестановка элементов списка

permute([], []).

permute(L, [H|T]) :-

select(H, L, R),

permute(R, T).

% Выбор трех уникальных символов из списка

select\_three([H1, H2, H3|\_], H1, H2, H3).

% Функция, которая строит все слова длины k, содержащие 3 буквы A на

% заданном алфавите.

in\_list([El|\_],El).

in\_list([\_|T],El):-in\_list(T,El).

razm\_povt(\_,0,Razm,Razm):-!.

razm\_povt(Alphabet,NCur,RazmCur,Razm):-in\_list(Alphabet,El),NNew is NCur-1,razm\_povt(Alphabet,NNew,[El|RazmCur],Razm).

make\_pos\_list(K, K, []):-!.

make\_pos\_list(K, CurPos, [NewPos|TailPos]) :- NewPos is CurPos + 1, make\_pos\_list(K, NewPos, TailPos).

make\_3a\_empty\_word(K, K, \_, []):-!.

make\_3a\_empty\_word(K, CurIndex, [NewIndex|PosTail], [a|Tail]) :-

NewIndex is CurIndex + 1, make\_3a\_empty\_word(K, NewIndex, PosTail, Tail),!.

make\_3a\_empty\_word(K, CurIndex, PosList, [\_|Tail]) :-

NewIndex is CurIndex + 1, make\_3a\_empty\_word(K, NewIndex, PosList, Tail).

build\_word([],[],\_):-!.

build\_word([a|WordTail],[X|WordEmpty3aTail],RestWord) :-

nonvar(X),build\_word(WordTail,WordEmpty3aTail,RestWord),!.

build\_word([Y|WordTail],[X|WordEmpty3aTail],[Y|RestWordTail]) :-

var(X),build\_word(WordTail,WordEmpty3aTail,RestWordTail).

build\_3a\_words\_of\_k(Alphabet,K,Word) :- make\_pos\_list(K, 0, PosList),

sochet(Pos\_a\_List, PosList, 3), make\_3a\_empty\_word(K, 0, Pos\_a\_List, WordEmpty3a), Alphabet = [a|NewAlphabet],

M is K - 3, razm\_povt(NewAlphabet, M, [], RestWord), build\_word(Word, WordEmpty3a, RestWord).

varia(0,\_,[]).

varia(N,L,[H|Varia]):-N>0,N1 is N-1,delete(H,L,Rest),varia(N1,Rest,Varia).

% Сочетания без повторений

% comb(+List, +K, -Combination)

comb(\_, 0, []).

comb([H|T], K, [H|Comb]) :-

K > 0,

K1 is K - 1,

comb(T, K1, Comb).

comb([\_|T], K, Comb) :-

K > 0,

comb(T, K, Comb).

% Сочетания с повторениями

% comb\_with\_replacement(+List, +K, -Combination)

comb\_with\_replacement(\_, 0, []).

comb\_with\_replacement([H|T], K, [H|Comb]) :-

K > 0,

K1 is K - 1,

comb\_with\_replacement([H|T], K1, Comb).

comb\_with\_replacement([\_|T], K, Comb) :-

K > 0,

comb\_with\_replacement(T, K, Comb).

% Размещения без повторений

% arrange(+List, +K, -Arrangement)

arrange(List, K, Arrangement) :-

comb(List, K, Comb),

permutation(Comb, Arrangement).

% Размещения с повторениями

% arrange\_with\_replacement(+List, +K, -Arrangement)

arrange\_with\_replacement(\_, 0, []).

arrange\_with\_replacement(List, K, [H|T]) :-

K > 0,

K1 is K - 1,

member(H, List),

arrange\_with\_replacement(List, K1, T).

% Перестановки без повторений

% permute(+List, -Permutation)

permute([], []).

permute(L, [H|T]) :-

select(H, L, R),

permute(R, T).

% Перестановки с повторениями

% permute\_with\_replacement(+List, +Length, -Permutation)

permute\_with\_replacement(\_, 0, []).

permute\_with\_replacement(List, Length, [H|T]) :-

Length > 0,

Length1 is Length - 1,

member(H, List),

permute\_with\_replacement(List, Length1, T).

% Подмножества

% subsets(+List, -Subset)

subsets([], []).

subsets([H|T], [H|Subset]) :-

subsets(T, Subset).

subsets([\_|T], Subset) :-

subsets(T, Subset).

%Общее для графов

in\_list1([El|\_],El):-!.

in\_list1([\_|T],El):-in\_list1(T,El).

in\_list([El|\_],El).

in\_list([\_|T],El):-in\_list(T,El).

append1([],X,X):-!.

append1([H|T],X,[H |Z]):-append1(T,X,Z).

check\_vertex([V1|\_],V1):-!.

check\_vertex([\_|T],V1):-check\_vertex(T,V1).

get\_edge(V,[V1,V2]):-write("Edge"),nl,read\_str(X),name(V1,X),check\_vertex(V,V1),

read\_str(Y),name(V2,Y),check\_vertex(V,V2).

%Гамильтонов цикл (неориентир)

gamilton:-get\_graph\_edges(V,E),gamilton(V,E).

gamilton(V,E):-b\_a\_r(V,W,Way),way\_check(Way,E),write(Way).

way\_check([H|T],E):-append1([H|T],[H],Way),w\_c(Way,E).

w\_c([\_],\_):-!.

w\_c([V1,V2|T],E):-in\_list1(E,[V1,V2]),w\_c([V2|T],E).

in\_list\_exlude([El|T],El,T).

in\_list\_exlude([H|T],El,[H|Tail]):-in\_list\_exlude(T,El,Tail).

b\_a\_r([],Perm1,Perm1):-!.

b\_a\_r(A,Perm,Perm1):-in\_list\_exlude(A,El,A1),b\_a\_r(A1,[El|Perm],Perm1).

%Эйлеров цикл (неориентир граф)

euler:-get\_graph\_edges(V,E),euler(V,E).

euler(V,E):-b\_a\_r(E,W,Way),way\_edge\_check(Way),write(Way).

way\_edge\_check([H|T]):-append1([H|T],[H],Way),w\_e\_c(Way).

w\_e\_c([\_]):-!.

w\_e\_c([[\_,X]|[[X|Y]|Tail]]):-w\_e\_c([[X|Y]|Tail]).

%Эйлеров цикл (ориентированный граф)

euler\_N:-get\_graph\_edges(V,E),euler\_N(V,E).

euler\_N(V,E):-b\_a\_r(E,W,Way),way\_edge\_check\_N(Way),write(Way).

way\_edge\_check\_N([H|T]):-append1([H|T],[H],Way),w\_e\_c\_N(Way).

w\_e\_c\_N([\_]):-!.

w\_e\_c\_N([[\_,X]|[[X,Y]|Tail]]):-w\_e\_c\_N([[X,Y]|Tail]).

w\_e\_c\_N([[\_,X]|[[Y,X]|Tail]]):-w\_e\_c\_N([[X,Y]|Tail]).

%Раскраска графа

raskras(K):-get\_graph\_edges(V,E),make\_ar(K,C),raskras(V,E,C,[],Itog),write(Itog).

make\_ar(0,[]):-!.

make\_ar(K,[K|Tail]):-K1 is K-1,make\_ar(K1,Tail).

raskras([],\_,C,Itog,Itog):-!.

raskras([V|Vtail],E,C,Ras,Itog):-

in\_list(C,Color),check\_color(V,E,Color,Ras),append1(Ras,[[V,Color]],Ras1),

raskras(Vtail,E,C,Ras1,Itog).

check\_color(\_,\_,\_,[]):-!.

check\_color(V,E,C1,[[\_,C2]|Tail]):-C1\=C2,check\_color(V,E,C1,Tail),!.

check\_color(V,E,Col,[[Ver,Col]|Tail]):-

not(in\_list1(E,[V,Ver])),not(in\_list1(E,[Ver,V])),

check\_color(V,E,Col,Tail).

%Поиск пути между двумя вершинами графа

make\_way:-

get\_graph\_edges(V,E),write("Otkuda"),nl,read\_str(X),name(I,X),

write("Kuda"),nl,read\_str(Y),name(S,Y),make\_way(V,E,I,S,Way),

write\_way(Way).

make\_way(V,E,I,S,Way):-in\_list\_exlude(V,I,Tail),make\_way(Tail,E,I,S,[I],Way).

make\_way(\_,\_,S,S,Way,Way):-!.

make\_way(V,E,I,S,Cur\_Way,Way):- in\_list\_exlude(V,X,Tail),in\_list1(E,[I,X]), append1(Cur\_Way,[X],C\_W),make\_way(Tail,E,X,S,C\_W,Way).

write\_way([V]):-write(" "),write(V),write("."),!.

write\_way([V|Tail]):-write(" "),write(V),write(" -"),write\_way(Tail).

list\_len([],0):-!.

list\_len([X|T],L):-list\_len(T,L1),L is L1+1.

%Кратчайший путь между двумя вершинами графа

short\_way:-get\_graph\_edges(V,E),write("Otkuda"),nl,read\_str(X),name(I,X),

write("Kuda"),nl,read\_str(Y),name(S,Y),list\_len(V,Len),

short\_way(V,E,I,S,Way,Len+1),write\_way(Way).

short\_way(V,E,I,S,Way,Len):-short\_way(V,E,I,S,Way,\_,Len).

short\_way(V,E,I,S,Way,\_,Len):-

make\_way(V,E,I,S,Cur\_Way1),list\_len(Cur\_Way1,L),L<Len,!,short\_way(V,E,I,S,Way,Cur\_Way1,L).

short\_way(\_,\_,\_,\_,Way,Way,\_).

Размещения без повторений

static void generatePermutations(ArrayList<Integer> list, int k, ArrayList<Integer> permutation) {

if (permutation.size() == k) {

System.out.println(permutation);

return;

}

for (int i = 0; i < list.size(); i++) {

int current = list.get(i);

if (!permutation.contains(current)) {

ArrayList<Integer> newPermutation = new ArrayList<>(permutation);

newPermutation.add(current);

// Рекурсивно вызываем метод для построения оставшихся элементов перестановки

generatePermutations(list, k, newPermutation);

}

}

}

Размещения с повторениями

public static void permutationsWithRepetitions(List<Integer> list, int k) {

int[] indexes = new int[k];

permutationUtil(list, k, indexes, 0);

}

private static void permutationUtil(List<Integer> list, int k, int[] indexes, int level) {

if (level == k) {

for (int i = 0; i < k; i++) {

System.out.print(list.get(indexes[i]) + " ");

}

System.out.println();

return;

}

for (int i = 0; i < list.size(); i++) {

indexes[level] = i;

permutationUtil(list, k, indexes, level + 1);

}

}

% Факториал (перестановки) рекурсия вверх

fact(0,1).

fact(N,X):- N>0, N1 is N-1, fact(N1,X1), X is X1\*N.

%Факториал (перестановки) рекурсия вниз

fact2(0,X,X).

fact2(N,A,X):- N>0,

N1 is N-1, A1 is A\*N, fact2(N1,A1,X1), X is X1.

call\_fact2(N,X):-fact2(N,1,X).

%in\_list(+InputList:List,?El:atom)

%True if El is in InputList

in\_list([El|\_],El).

in\_list([\_|T],El):-in\_list(T,El).