Домашна 1

Данило Најков 206033

Задача 1

За да ја решам оваа задача дефинирав помошен предикат ***izbrisi_i_vrati_posleden(X,Y,Z)***, кој има 3 аргументи:

- X - листа
- Y - листа без последниот елемент
- Z - последниот елемент од влезната листа

Целта на овој предикат е да се се отсрани последниот елемент од листата за потоа заедно со отсранувањето на првиот да се извршува рекурзивно ***neparen_polindrom***.

При првичното рекурзивно изминување се кратат елементите по ред од иницијалната листа се додека не се стигне до еден единствен елемент. Кога ке се дојде до последниот елемент во листата се повикува правилото ***izbrisi_i_vrati_posleden([Posleden], [], Posleden)*** каде се зачувува послениот елемент во Posleden а истиот се брише од листата без последниот. При враќањето назад елементот X во секоја итерација се “залепува од напред” на листите, при што бидејќи во едната листа фали последниот елемент, се добива листа без тој.

```
izbrisi_i_vrati_posleden([Posleden], [], Posleden).
izbrisi_i_vrati_posleden([X|Dr], [X|Po], Posleden) :-
    izbrisi_i_vrati_posleden(Dr, Po, Posleden).
```

Предикатот ***neparen_polindrom*** користејќи го претходниот предикат го отсранува последниот елемент на листата и дополнително и првиот, проверува дали се исти и продолжува рекурзивно. Нема потреба да се проверува парноста бидејќи во еден чекор од рекурзијата се кратат 2 елементи, а дефинирано е само правило ***neparen_polindrom([_])*** кое дозволува на крај да има само 1 елемент. Со тоа секогаш би се добиле 2к+1 елементи што е непарен број. Дополнително додадов и cut оператор за да се намали бројот на повикувања - не менува ништо во однос на логиката.

```
neparen_polindrom([_]).
neparen_polindrom([X|L]) :-
    izbrisi_i_vrati_posleden(L, LnoPos, Pos),
    X == Pos,
    !,
    neparen_polindrom(LnoPos).
```

Задача 2

За оваа задача дефинирав повеќе предикати

podniza_so_dolzina(L1,L2,N)

```
podniza_so_dolzina(_,[],0).
podniza_so_dolzina([X|L],[X|LR],N) :-
    podniza_so_dolzina(L, LR, M),
    N is M+1.
```

Функцјата на овој предикат е да врати подлиста со почеток од првиот елемент која е долга n елементи. Работи со тоа што рекурзивно се бројат елементите се дур се стигне до n=0 и потоа при враќањето од рекурзијата се конкатанираат во резултантната листа.

ista_podniza(L1,L2)

```
ista_podniza(_,[]).
ista_podniza([X|L],[Y|PL]) :-
    X==Y,
    ista_podniza(L,PL).
```

Овој предикат има функција да провери дали листата 2 е подниза од листата 1 почнувајќи од првиот елемент. Краен чекор во рекурзијата е кога ќе се поминат сите елементи од листа 2 и тие се еднакви па ќе врати true, во било кој друг случај ќе врати false

pojavuvanja_vo_niza(L,PL,N)

```
pojavuvanja_vo_niza([],_,0).
pojavuvanja_vo_niza([X|L],PL,Broj) :-
    ista_podniza([X|L],PL), pojavuvanja_vo_niza(L,PL,BrojN), Broj is BrojN + 1;
    pojavuvanja_vo_niza(L,PL,Broj).
```

Овој предикат има за задача да го изброи бројот на појавувања на листата PL во листата L. Се користи претходно дефинираниот предикат **ista_podniza** за да се провери дали поднизата се појавува и со OR се дефинирани 2 случаеви:

- Низата се појавува - зголеми го бројачот
- Низата не се појавува - провери за следниот елемент од L со рекурзијата

Ова ќе заврши кога ќе се поминат сите елементи од L

maks_el(X,XNiza,Y, YNiza, Z, ZNiza)

```
maks_el(X, XNiza, Y, _, Z, ZNiza) :-
   X>Y, Z=X, ZNiza=XNiza.
maks_el(X, _, Y, YNiza, Z, ZNiza) :-
   X=<Y, Z=Y, ZNiza=YNiza.
```

Едноставен предикат кој прима број X,Y ги споредува и најголемиот го става во Z заедно со низата од X или Y која ја става во ZNiza

naj_podniza_rec(LF,N,L,MaxBrojPojav, MaksNiza)

```
naj_podniza_rec([],_,M,_,M).
naj_podniza_rec([X|LF],N,L,MaxBrojPojav, MaksNiza) :-
   podniza_so_dolzina([X|LF],CheckNiza,N),
   pojavuvanja_vo_niza([X|LF],CheckNiza,OutBroj),
   maks_el(OutBroj, CheckNiza, MaxBrojPojav, MaksNiza, MaksRecBroj, MaksRecNiza),
   naj_podniza_rec(LF,N,L,MaksRecBroj, MaksRecNiza).
naj_podniza_rec([X|LF],N,L,_, MaksNiza) :-
   not(podniza_so_dolzina([X|LF],_,N)),
   naj_podniza_rec([],_,L,_,MaksNiza).
```

Главната идеја на овој предикат е да ги помине сите можни низи со дадената должина, да ги изброи и да ја врати максималната. Најпрвин се зема низа со должина N преку **podniza_so_dolzina,** се проверува колку пати се појавува таа подниза во целосната низа - **pojavuvanja_vo_niza,** па доколку бројот е поголем од максималниот (се проверува во **maks_el**) се проверува за следната можна подниза со должина N без првиот елемент и се зачувува најдолгата низа до сега во MaksRecNiza. Кога сме на крај од низата **podniza_so_dolzina** може да врати false, односно да не најде низа со N членови. Затоа е додадено последното правило кое проверува и на некој начин ја завршува рекурзијата.

naj_podniza(LF,N,L)

```
naj_podniza(LF,N,L) :-
        naj_podniza_rec(LF,N,L,0,L).
```

Едноставен “wrapper” предикат кои не ги прикажува иницијалните max вредности од **naj_podniza_rec**.

Задача 3

За случајот на листа со 2 елемента го дефинирав предикатот
```
proveri(L) :-
   [X,Y]=L,
   X<Y.
```
Кој проверува дали навистина се само 2 елемента и ако се дали првиот е помал.

За случајот на 3 или повеќе елемента
```
proveri_nazad([_]).
proveri_nazad([_,_]).
proveri_nazad([X,Y,Z|L]) :-
   Y>X,
   Y>Z,
   proveri_nazad([Z|L]).

proveri(L) :-
   [_,_,_|_]=L,
   proveri_nazad(L).
```

Го дефинирав предикатот со тоа што најпрвин проверувам дали навистина се 3 или повеќе елементи со ***[_,_,_|_]=L***, (3 _ за мин 3 елементи и |_ за повеќе) и потоа се повикува ***proveri_nazad*** кој проверува дека средниот елемент (Y) е најголем од соседните и повикува рекурзивно за следните три заедно со последниот елемент Z. Ако се поминат сите елементи - останат 1 или 2 (првите 2 дефинирани правила) ке врати true.

Задача 4

За оваа задача ги искористив и предикатите **brisi_prvo** и **dolzina** од аудиториските вежби. Дополнително го дефинирав помошниот предикат **zalepi_odnapred(X,L,LR)** кој прима листа од листи во L и на секоја листа му го додава елементот X од напред.
```
zalepi_odnapred(_,[],[]).
zalepi_odnapred(X, [P|L], [Temp|LR]) :-
   zalepi_odnapred(X,L,LR),
   Temp = [X|P].
```

Главната логика на задачата е сместена во предикатот **perm_test(L,LFull,LRes),** кој за секој елемент во листата **L** го трга тој елемент од целата листа **LFull** преку предикатот **brisi_prvo**, и рекурзивно се повикува за останатиот дел од листата. Така при вракањето од рекурзијата се собираат сите можни пермутации за тој дел од скратената листа. Понатаму уште еднаш се повикува **perm_test,** но не за скратената листа, туку за останатите случаеви каде пермутациите почнуваат со други елементи од листата (на некој

начин е слично на while циклус во императивните јазици за секој елемент во листата да е почетен). На крај од предикатот се собираат резултатите од рекурзивното повикување + повикувањето за следниот елемент во една листа со предикатот **dodadi**. Вака се генерираат сите пермутации со должина <=LD, каде LD е должината на input листата.

```
perm_test([],_,[[]]).
perm_test([X|L],LFull,NovRez) :-
    brisi_prvo(X,LFull,LSkrateno),
    perm_test(LSkrateno, LSkrateno, LSkratenoRez),
    zalepi_odnapred(X,LSkratenoRez,TempRezRec),
    perm_test(L,LFull,DopL),
    dodadi(TempRezRec,DopL,NovRez).
```

На крај уште останува да се исфилтрираат преостанатите пермутации кои што се со помала должина од input листата. За ова го дефинирав предикатот **filtriraj_dolzina**

```
filtriraj_dolzina([],_,[]).
filtriraj_dolzina([X|L], XD, [X|Rez]) :-
    proveri_dolzina(X,XD),
    filtriraj_dolzina(L,XD,Rez).
filtriraj_dolzina([X|L], XD, Rez) :-
    not(proveri_dolzina(X,XD)),
    filtriraj_dolzina(L,XD,Rez).
```

Овој предикат при враќањето назад од рекурзијата за залепува листата X на резултатот само ако X е со должина иста како input должината XD.

На крај сето ова е споено во еден предикат, кој првин ја мери должината на листата, ги наоѓа сите пермутации, па ги филтрира само тие со еднаква должина како input листата.

```
permutacii(L1,LR) :-
    izmeri_dolzina(L1,L1D),
    perm_test(L1,L1,LRez),
    filtriraj_dolzina(LRez,L1D,LR).
```

Задача 5

Главната идеја ми беше да го имплементирам собирањето, а потоа сите други операции да одат преку него

Собирање

За да се реши проблемот на листи со различна должина го дефинирав предикатот израмни, кој ја зема пократката листа и ѝ додава нули на почеток за да бидат со иста должина. За тоа има три случаеви, кога првата е подолга, втората е подолга или се исти:

```
izramni(L1,L2,L1R,L2R) :-
    dolzina(L1,N1),
    dolzina(L2,N2),
    N1>N2,
    Razlika is N1-N2,
    izramni_rec(L2,L2R,Razlika),
    L1R=L1,
    !.

izramni(L1,L2,L1R,L2R) :-
    dolzina(L1,N1),
    dolzina(L2,N2),
    N1<N2,
    Razlika is N2-N1,
    izramni_rec(L1,L1R,Razlika),
    L2R=L2,
    !.

izramni(L1,L2,L1R,L2R) :-
    dolzina(L1,N1),
    dolzina(L2,N2),
    N1=N2,
    L1R=L1,
    L2R=L2,
    !.
```

izramni_rec е едноставен предикат кој при backtracking додава нули на почетокот на резултантната листата.

```
izramni_rec(L,L,0).
izramni_rec(L,[0|LR],N) :-
    M is N-1,
    izramni_rec(L,LR,M).
```

Потоа ги дефинирав предикатите soberi_rec, soberi_dirty и rezultat_dodeli

```
soberi_dirty(L1,L2,[Carry|LR]) :-
    izramni(L1,L2,L1R,L2R),
    soberi_rec(L1R,L2R,LR, Carry).
```

soberi_dirty, ги израмнува листите и го повикува предикатот **soberi_rec.**
soberi_rec рекурзивно при backtracking ги собира вредностите на истите позиции на листите L1 и L2, како и Carry од претходното собирање во рекурзијата. Тука се повикува предикатот **rezultat_dodeli** кој според резултатот од збирот ке додели нови вредности за новата позичија во резултантната листа, како и carry за следното рекурзивно враќање наназад.

```
soberi_rec([],[],[], 0).
soberi_rec([X|L1],[Y|L2],[Z|LR], CarryM) :-
   soberi_rec(L1,L2,LR, Carry),
   Res is X + Y + Carry,
   rezultat_dodeli(Res, Z, CarryM).

rezultat_dodeli(0, 0, 0).
rezultat_dodeli(1, 1, 0).
rezultat_dodeli(2, 0, 1).
rezultat_dodeli(3, 1, 1).
```

На крај може да се случи при целото собирање да остане 1 како carry, што треба да се стави најнапред во резултатот и при тоа да се зголеми должината на листата. Затоа е дефинирано **sobiranje** кој повикува **soberi_dirty** со резултатот го повикува **chuvaj_carry** за да додаде 1 на почетокот доколку carry е 1.

```
sobiranje(L1,L2,LRez) :-
   soberi_dirty(L1,L2,[X|LR]),
   chuvaj_carry(X,LR,LRez).

chuvaj_carry(0,LRez,LRez).
chuvaj_carry(1,LRez,[1|LRez]).
```

Одземање

Идејата кај одземањето ми беше наместо да се пресметува одново се, да се најде вториот комплемент на вториот бинарен број и да се користи собирањето.

Комплементот може многу едноставно да се имплементира. Едноставно ги променува вредностите, 1->0 и 0->1 за секоја цифра.

```
komplement([0],[1]).
komplement([1],[0]).
```

```
komplement([X|L],[T|LR]) :-
   komplement(L,LR),
   obratno(X,T).

obratno(0,1).
obratno(1,0).
```

Пред да го направиме одземањето треба да провериме дали првиот број е поголем од вториот. За ова го дефинирав предикатот **pogolem_broj.**

```
pogolem_broj([1|L],[0|_], [1|L]).
pogolem_broj([0|_],[1|L], [1|L]).
pogolem_broj([], [], []).
pogolem_broj([X|L1],[X|L2], [X|LR]) :- pogolem_broj(L1, L2, LR).
```

При собирањето на вториот комплемент може да се случи да се појави 1 на почеток како и повеќе нули кои би требало да се тргнат во резултатот. За тоа ги дефинирав предикатите **izbrishi_preceeding_nuli_odzemanje** кој ги брише сите нули дур не види 1, и **trgni_kec_odzemanje** кој според должините на input листите проверува дали е потребно да се тргне 1 од почетокот на резултатот.

```
izbrishi_preceeding_nuli_odzemanje([1|L],[1|L]).
izbrishi_preceeding_nuli_odzemanje([0],[0]).
izbrishi_preceeding_nuli_odzemanje([_|L],LRez) :-
   izbrishi_preceeding_nuli_odzemanje(L,LRez).

trgni_kec_odzemanje(D1,D2,[_|L], L) :- D1>D2.
trgni_kec_odzemanje(D1,D2,L, L) :- D1=D2.
```

Сега може да го дефинираме одземањето:

- Израмни ги листите
- Провери дека првиот број е поголем, ако не е врати 0
- Најди комплемент на вториот
- Додади 1 во бинарно на вториот број за да се најде втор комплемент
- Собери ги првиот број со вториот - како втор комплемент
- Тргни вишок 1 и повеќе нули пред вистинскиот број

```
odzemanje(L1,L2,LRez) :-
   izramni(L1,L2,L1R,L2R),
   pogolem_broj(L1R,L2R,L1R),
   komplement(L2R,L2RK),
   sobiranje(L2RK,[1],L2RK2), %vtor komplement
   sobiranje(L1R,L2RK2, LRezTemp),
   dolzina(L1,N1),
```

```
    dolzina(LRezTemp,N2),
    trgni_kec_odzemanje(N2,N1,LRezTemp, LRezTempSkrateno),
    izbrishi_preceeding_nuli_odzemanje(LRezTempSkrateno,LRez),!.

odzemanje(L1,L2,LRez) :-
    izramni(L1,L2,L1R,L2R),
    pogolem_broj(L1R,L2R,L2R),
    LRez=[0].
```

Множење

Множење е всушност собирање на истиот број m пати, па лесно може да се имплементира во пролог:

```
mnozenje(L1,[1],L1).
mnozenje(_,[0],[0]).
mnozenje(L1,L2,LR) :-
    odzemanje(L2,[1],LMinus),
    mnozenje(L1,LMinus,LP),
    sobiranje(L1,LP,LR),!.
```

Идејата е го намалуваме множителот L2 за еден се додека не стигне нула, па во backtracking, при секое враќање се собира L1 во резултатот.

Делење

Делењето е многу слично како множењето, само што тука идејата е да го одземаме дељеникот од делителот, се додека дељеникот е поголем. Вака на некој начин гледаме колку пати може да го собере дељеникот во делителот - што е еквивалентно на цело бројно делење

```
delenje(L,L,[1]).
delenje(L1,L2,[0]) :- izramni(L1,L2,L1R,L2R),pogolem_broj(L1R,L2R,L2R).
delenje(L1,L2,LR) :-
    odzemanje(L1,L2,LOdzemeno),
    delenje(LOdzemeno,L2,LM),
    sobiranje(LM,[1],LR),!.
```

Задача 6

За оваа задача најпрвин имплементирав на некој начин индексирање, така што може да се најде елемент при даден x и y индекс:

```
element_na_indeks_smeni_dimenzija(0,0,[X|_],X).
element_na_indeks_smeni_dimenzija(0,I2,[_|L],E) :-
    M is I2-1, element_na_indeks_smeni_dimenzija(0,M,L,E).

element_na_indeks(0,I2,[X|_],E) :- element_na_indeks_smeni_dimenzija(0,I2,X,E).
element_na_indeks(I1,I2,[_|L],E) :-
    M is I1-1, element_na_indeks(M,I2,L,E).
```

element_na_indeks пребарува во првата димензија се дур индексот не стане 0 (го намалува за 1), а **element_na_indeks_smeni_dimenzija**, пребарува во другата димензија на сличен начин.

За да го најдеме резултатот од множењето на даден индекс во резултантната матрица, го дефинирав **calc_multiply_na_indeks.** При разгледување на проблемот, сфатив дека нема потреба да се транспонира матрицата, само треба да ги земеме елементите што одговараат на на индексот на редиците x1 и x2, каде (x1,x2) е индексот на елементот во резултантната матрица. Ова се прави се додека бројачот (се зголемува при секое повикување) не стигне до должината на матрицата, за да се поминат сите елементи во таа редица.

```
calc_multiply_na_indeks(_,_,_,0,M,M).
calc_multiply_na_indeks(L,I1,I2, E, Brojac, Dolzina) :-
    B is Brojac+1,
    calc_multiply_na_indeks(L,I1,I2, K, B, Dolzina),
    element_na_indeks(I1,Brojac,L,E1),
    element_na_indeks(I2,Brojac,L,E2),
    E is K + E1*E2.
```

На крај само треба да се пресмета ова за секој индекс во резултантната матрица. Ова е имплементирано во **presmetaj_rec_2d** и **presmetaj_rec_1d** и идејата за како се поминуваат сите вредности за индексите е скоро идентична како за **element_na_indeks,** при што се оди до дефинираната должина на матрицата. **presmetaj** е само “wrapper” предикат кој ја пресметува должината и повикува **presmetaj_rec_2d.**

```
presmetaj_rec_1d(M,_,_,M,[]).
presmetaj_rec_1d(I1,I2,L, D, [El|R]) :-
    IR is I1 + 1,
    presmetaj_rec_1d(IR, I2, L, D, R),
    calc_multiply_na_indeks(L, I1, I2, El, 0, D).
```

```
presmetaj_rec_2d(M,_,M,[]).
presmetaj_rec_2d(I,L,D, [Rez|R]) :-
    IR is I + 1,
    presmetaj_rec_2d(IR, L, D, R),
    presmetaj_rec_1d(0,I,L,D,Rez).

presmetaj(M,R) :-
    dolzina(M,D),
    presmetaj_rec_2d(0,M,D,R).
```

Задача 7

За оваа задаца најпрвин дефинирав **get_lista_so_dolzini**, што како прв аргумент зема листа од листи, а како втор враќа листа од должините на сите тие под-листи. Cut операторот е ставен за да не се враќа и да бара други решенија, во случај да fail во друг предикат.

```
get_lista_so_dolzini([],[]).
get_lista_so_dolzini([X|L],[D|LD]) :-
    get_lista_so_dolzini(L,LD),
    dolzina(X,D),
    !.
```

Целата логика за сортирање опишана во задачата е ставена во **sortiraj_dve.** Првин ги споредува должините, па ако се исти ги споредува елементите еден по еден. Па ако и тие се исти става flag 1 во аргументот Ista. Ова е потребно за потоа да не се зачуваат двата елементи ако се исти (според задачата).

```
sortiraj_dve(X,Y,XD,YD,X,XD,Y,YD,0) :- XD>YD.
sortiraj_dve(X,Y,XD,YD,Y,YD,X,XD,0) :- YD>XD.
sortiraj_dve([E1|X],[E2|Y],XD,XD,[E1|X],XD,[E2|Y],XD,0) :- E1>E2.
sortiraj_dve([E1|X],[E2|Y],XD,XD,[E2|Y],XD,[E1|X],XD,0) :- E1<E2.
sortiraj_dve([],[],_,_,[],L,[],L,1).
sortiraj_dve([E1|X],[E1|Y],XD,XD,[E1|RP],XD,[E1|RS],XD,Ista) :-
    sortiraj_dve(X,Y,XD,XD,RP,XD,RS,XD,Ista).
```

sortiraj_1iteracija е едно рекурзивно поминување низ листата и сортирање на секој пар елементи. При back-propagation, се додаваат елементите (листите) од напред според логиката во **sortiraj_dve.** Ако елементите се исти, само еден се додава.

```
sortiraj_1iteracija([X|[]],[XD|[]], [X], [XD]).
sortiraj_1iteracija([X,Y|L], [XD,YD|LD], [B|RL], [BD|RD]) :-
```

```
    sortiraj_dve(X, Y, XD, YD, B, BD, S, SD,Ista),
    Ista==0,
    sortiraj_1iteracija([S|L], [SD|LD], RL, RD),
    !.
sortiraj_1iteracija([X,Y|L], [XD,YD|LD], [B|RL], [BD|RD]) :-
    sortiraj_dve(X, Y, XD, YD, B, BD, _, _,Ista),
    Ista==1,
    sortiraj_1iteracija(L, LD, RL, RD),
    !.
```

За да се провери дали листата од листи е сортирана, го напшав **dali_sortirana**, кој проверува за секој пар елементи, дали се според правилата од **sortiraj_dve.**

```
dali_sortirana([_|[]],_).
dali_sortirana([X,Y|L], [XD,YD|LRD]) :-
    sortiraj_dve(X,Y,XD,YD,R,_,_,_,Ista),
    X=R,
    Ista=0,
    dali_sortirana([Y|L],[YD|LRD]),
    !.
```

На крај се е споено во **transform**, кое што всушност повторува **sortiraj_1iteracija**, се додека листата не е сортирана.

```
transform(L,LR) :-
    get_lista_so_dolzini(L,LD),
    sortiraj_1iteracija(L,LD,LRTemp,LRD),
    dali_sortirana(LRTemp, LRD),
    LR=LRTemp.
transform(L,LR) :-
    get_lista_so_dolzini(L,LD),
    sortiraj_1iteracija(L,LD,LRTemp,_),
    transform(LRTemp,LR).
```

Задача 8

Главната идеја е да се направи една помошна листа која во 1D ќе прикажува за секој елемент колку пати се појавил претходно. Според тоа би се бришеле елементи од резултантната листа. Од аудиториските вежби ги искористив предикатите **e_lista, izramni, dodadi**
Дефинирав предикат **deep_dolzina**, кој гледа колку елементи има во една листа, но тие елементи да не се други листи. На некој начин гледа колку елементи би имало во израмнетата листа.

```
deep_dolzina([],0).
deep_dolzina([X|O],N) :-
   not(e_lista(X)),
   deep_dolzina(O,N1), N is N1+1.
deep_dolzina([X|O],N) :-
   e_lista(X),
   deep_dolzina(O,N1),
   deep_dolzina(X,N2), N is N1+N2.
```

За да се пресмета бројот на појавување на секој елемент, го напишав предикатот **broj_pojavuvanja_element,** кој гледа за даден елемент колку пати се појавува во дадена листа и зголемува бројач, кој се враќа како резултат. **broj_pojavuvanja** дава листа, каде овој број е пресметан за секој елемент во израмнетата листа.

```
broj_pojavuvanja_element(_,[],0).
broj_pojavuvanja_element(A,[A|L],Broj) :-
   broj_pojavuvanja_element(A,L,BrOut),
   Broj is BrOut+1.
broj_pojavuvanja_element(A,[_|L],Broj) :-
   broj_pojavuvanja_element(A,L,Broj).

broj_pojavuvanja([],_,[]).
broj_pojavuvanja([X|L], FullL, [TempBroj|OutArr]) :-
   broj_pojavuvanja_element(X,FullL,FullBroj),
   broj_pojavuvanja_element(X,L,MinusBroj),
   TempBroj is FullBroj - MinusBroj,
   broj_pojavuvanja(L,FullL, OutArr).
```

Одкога ќе ги извлечеме овие информации, може да се премине на самото бришење на елементи. За тоа го дефинирав **brishi** кој проверува, ако елементот не е листа, дали во листата со појавување се јавува парен број пати. Ако да, не го залепува во резултантната листа при backtracking. Доколку елементот е листа, најпрвин се повикува рекурзивно на таа листа. А потоа и на остатокот на елементите. Но тука во листата на појавувања треба да ги рипнеме елементите што се во под-листата. Тоа го прави **skrati_od_lista** која прима аргумент листа, и колку треба елементи треба да тргне од главата на листата.

```
skrati_od_lista(L,0,L).
skrati_od_lista([_|L],N,LR) :-
   NR is N-1,
   skrati_od_lista(L,NR,LR).

brishi([],[],_).
brishi([X|L],[X|LRez],[P|LPojavuvanja]) :-
   not(e_lista(X)),
```

```
    1 is P mod 2,
    brishi(L,LRez,LPojavuvanja).

brishi([X|L],LRez,[P|LPojavuvanja]) :-
    not(e_lista(X)),
    0 is P mod 2,
    brishi(L,LRez,LPojavuvanja).

brishi([X|L],[OutRez|SlRez],LPojavuvanja) :-
    e_lista(X),
    deep_dolzina(X,XDolzina),
    brishi(X,OutRez,LPojavuvanja),
    skrati_od_lista(LPojavuvanja, XDolzina, LPojavuvanjaNew),
    brishi(L,SlRez,LPojavuvanjaNew).
```

На крај **brisi_sekoe_vtoro** ги извршува сите овие предикати, со тоа што првин, ја израмнува листата и генерира листа на појавување, па потоа ги брише елементите според задачата.

```
brisi_sekoe_vtoro(L,LR) :-
    izramni(L,LIzramneta),
    broj_pojavuvanja(LIzramneta,LIzramneta,LPojavuvanja),
    brishi(L,LR,LPojavuvanja),!.
```