ДОМАШНА ЗАДАЧА 1

ЛОГИЧКО ПРОГРАМИРАЊЕ
РАБОТА СО ЛИСТИ ВО PROLOG

Марија Вецовска 185008

Задача 1. (10 поени) Да се напише предикат во PROLOG neparen_palindrom(L) кој ќе провери дали дадена непразна листа има непарен број на елементи и претставува палиндром.

1. odd(X)-Предикат за проверка дали елементот X е непарен

```
%Основен случај
odd(X) :- X==1.
```

%Во секој рекурзивен повик како аргумент се задава Y = X-2, за евентуално да се дојде до елементарниот случај, ако X е непарен број, или пак да се добие неточно на проверката Y>0(кога Y ќе има вредност 0) ако X е парен број.

```
odd(X) :- Y is X-2, Y>0, odd(Y), !.
```

2. neparen_palindrom(L)- Проверува дали листата L има непарен број на елементи и претставува палиндром.

```
neparen_palindrom(L) :- reverse(L2, L),  neparen_palindrom(L, L2, 0).
```

3. neparen_palindrom(L, L2, X) - Предикатот испишува 'yes' ако L1 и L2 се идентични и тие имаат непарен број на елементи.

%Основен случај, кога двете листи успешно се испразниле и бројачот е непарен број(значи во листите имало непарен број на елементи).

```
neparen_palindrom([], [], X):- odd(X), write(yes).
```

%Проверка дали првите елементи од две листи се исти, зголемување на бројачот за еден и рекурзивен повик на neparen_palindrom со остатокот од листите и зголемениот бројач.

```
neparen_palindrom([H1|O1],[H2|O2], X) :- H1==H2, Y is X+1,
    neparen_palindrom(O1, O2, Y).
```

Задача 2. (10 поени) Да се напише предикат во PROLOG naj_podniza(L1,N,L2) кој ќе ја најде поднизата L2 со должина N која се појавува најмногу пати во влезната листа од атомични елементи L1.

1. prefix(L, L1) - предикат кој проверува дали листата L1 е префикс на листата L. Се' додека L1 има елементи, ги споредува со елементите од L. Ако за секој елемент од L1 има соодветен ист елемент од L, L1 ќе се испразни и ќе дојде до основниот случај.

```
prefix(_, []).
prefix([H|L], [H|L1]):- prefix(L, L1), !.
```

2. zemi_podniza(L, X, L1) - во променливата L1 ја враќа листата која ги содржи првите X елементи од L.

Во секоја повик бројачот се намалува за 1 и на L1 и се додава еден елемент од почетокот на L, се' додека брочачот не дојде до 0 - поминале X повици.

```
zemi_podniza(_, 0, []).
zemi_podniza([H1|L], X, [H1|L1]):- Y is X-1, zemi_podniza(L, Y, L1).
```

3. `broj_pojavuvanja_podniza(L, L1, X)` – во X го брои бројот на појавувања на L1 во L.
Во еден повик, предикатот прво ги споредува првите елементи на L и L1,
Ако се исти повикува **prefix(L, L1)** за низите(проверува дали L1 без првиот елемент е префикс на L без првиот елемент).
Ако ова е точно, бројачот X се зголемува за 1 и повторно се повикува broj_pojavuvanja_podniza, за L без првиот елемент, и за L1(со првиот). Ако е неточно-истото, но бројачот не се зголемува.
Броењето завршува кога L ќе се испразни.

```
broj_pojavuvanja_podniza([], _, 0).
broj_pojavuvanja_podniza([H|L],[H|L1], X) :- prefix(L, L1),
    broj_pojavuvanja_podniza(L, [H|L1], Y), X is Y+1, !.
broj_pojavuvanja_podniza([H|L],[H1|L1], X):-broj_pojavuvanja_podniza(L,
[H1|L1], X).
```

4. `naj_podniza(L1,N,L2)` – во L2 ја враќа листата со должина N која се појавува најмногу пати во листата L1.

```
naj_podniza(L1,N,L2) :-  naj_podniza(L1,N,L2,P).
```

5. naj_podniza(L1,N,L2, P2) – во L2 ја враќа листата со должина N која се појавува најмногу пати во листата L1, P2 е бр на појавување на L2 во L1.

%Во еден повик, **`zemi_podniza([H|L1], N, Lnova)`**, во Lnova ја враќа листата со првите N елементи од [H|L1].
broj_pojavuvanja_podniza([H|L1], Lnova, P) – во P враќа колку пати Lnova се појавува во [H|L1].
naj_podniza(L1,N,L2,P2) – во L2 ја враќа листата со должина N која се појавува најмногу пати во L1.
Ако P2 >= P, значи L2 се појавува повеќе пати во [H|L1] од Lnova, па таа се проследува во naj_podniza([H|L1], N, L2,P2).

```
naj_podniza([H|L1], N, L2,P2) :- naj_podniza(L1,N,L2,P2),
    zemi_podniza([H|L1], N, Lnova),
    broj_pojavuvanja_podniza([H|L1], Lnova, P), P2>=P, !.
```

%Инаку, се проследува листата со првите N елементи од [H|L1].

```
naj_podniza([H|L1], N, L2,P2) :-
    zemi_podniza([H|L1], N, L2),
    broj_pojavuvanja_podniza([H|L1], L2, P2).
```

Задача 3. (10 поени) Да се напише предикат во PROLOG proveri(L) кој за дадена листа ќе го проверува следното:
- доколку листата има помалку од два елементи предикатот да враќа no.
- доколку има два елементи, ако вториот елемент е поголем од првиот да врати yes, во спротивен случај да врати no.
- доколку листата има повеќе од два елементи предикатот да врати yes ако елементите го исполнуваат следното правило: вториот елемент да биде поголем од првиот, третиот

елемент да биде помал од вториот, четвртиот да биде поголем од третиот и така натаму до крајот на листата (наизменично да се менуваат поголемо па помало).

%Ги проверува елементарните случаеви за да врати не ако листата има помалку од 2 елементи.
```
proveri([]):-write(no).
proveri([_]):-write(no).
```

1. proveri(L) – Предикатот проверува дали листата L ги исполнува барањата наведени погоре.
%Го повикува предикатот `proveriPogolem(H, L)`, за да провери дали првиот е поголем од вториот елемент во L.
```
proveri([H|L]) :- proveriPogolem(H, L).
```

1. proveriPogolem(H1, [H2|L]) - проверува дали елементот H1 е помал од H2, се повикува за споредба на првиот и вториот, третиот и четвртиот, петиот и шестиот...
```
proveriPogolem(_, []):- write(yes),!.
proveriPogolem(H1, [H2|L]) :- H1 < H2, proveriPomal(H2, L), !.
proveriPogolem(H1, [H2|L]):- write(no), !.
```

2. proveriPomal(H1, [H2|L]) - проверува дали елементот H1 е поголем од H2, се повикува за споредба на вториот и третиот, четвртиот и петиот....
```
proveriPomal(_, []):- write(yes), !.
proveriPomal(H1, [H2|L]) :- H1 > H2, proveriPogolem(H2, L), !.
proveriPomal(H1, [H2|L]) :- write(no), !.
```

Задача 4. (10 поени) Да се напише предикат во PROLOG permutacii(L1,L2) кој ќе ги најде сите пермутации на елементите на листата L1 и ќе ги врати како подлисти на листата L2.

1. dodadi_element_na_pozicija(X, L, P, L1) - ја додава променливата X на позиција P во листата L и новата листа ја враќа во L1.
Ова го постигнува т.ш. P го намалува за 1 и вади еден елемент од почеток на L во секоја повик. Кога P = 0, L1 ја добива вредноста на L, но со X на почеток, а потоа се додаватат останатите елементи при враќање наназад.

```
dodadi_element_na_pozicija(X, L, 0, [X|L]).
dodadi_element_na_pozicija(X, [H|L], P, [H|L1]) :- P1 is P-1,
    dodadi_element_na_pozicija(X, L, P1, L1), !.
```

2. dodadi_element_na_sekoja_pozicija(X, L, L1) – за секоја можна позиција P во L, во L1 додава елемент – листата L со X додадено на таа позиција.
dodadi_element_na_sekoja_pozicija(X, L, P, L1) - се преоптоварува така што се додава бројот на можни позиции-**P**.

```
dodadi_element_na_sekoja_pozicija(X, L, L1) :- length(L, P),
    dodadi_element_na_sekoja_pozicija(X, L, P, L1).
```

%Во секој повик, на L1 се додава L трансформирано(променливата E) со предикатот погоре(за една позиција), и се додава на L1. Потоа се повикува истото за помала позиција.

```
dodadi_element_na_sekoja_pozicija(X, L, 0, [L1]) :-
    dodadi_element_na_pozicija(X, L, 0, L1).
dodadi_element_na_sekoja_pozicija(X, L, P, [E|L1]) :-
    dodadi_element_na_pozicija(X, L, P, E),
    P2 is P-1, dodadi_element_na_sekoja_pozicija(X, L, P2, L1), !.
```

3. dodadi_element_na_sekoja_pozicija_lista(X,L, L1) – за секоја подлиста од L и променливата X го повикува погорниот предикат и резултатот го враќа во L1.
`dodadi_element_na_sekoja_pozicija(X, H, H1)` – Во H1 ги враќа сите листи кои произлегле од H со додавање на X.
`dodadi_element_na_sekoja_pozicija_lista(X, L, L2)` – Во L2 враќа листа со сите елементи на L со додадено X на секоја позиција.
`append(H1, L2, L1)` – ги спојува H1 и L во H1.

```
dodadi_element_na_sekoja_pozicija_lista(_, [], []).
dodadi_element_na_sekoja_pozicija_lista(X, [H|L], L1) :-
    dodadi_element_na_sekoja_pozicija(X, H, H1),
    dodadi_element_na_sekoja_pozicija_lista(X, L, L2),
    append(H1, L2, L1).
```

4.permutacii(L, L2) – Ги дава сите пермутации на листата L во листата L2.
При секој рекурзивен повик се добива листа на сите пермутации на моменталната L(постфикс на L со кој се повикува предикатот) во L1.
Потоа со `dodadi_element_na_sekoja_pozicija_lista(H, L1, L2), L1` се трансформира во листа на пермутации L2 со плус еден елемент - H.

```
permutacii([E], [[E]]).
permutacii([H|L], L2) :- permutacii(L, L1),
    dodadi_element_na_sekoja_pozicija_lista(H, L1, L2).
```

Задача 5. (10 поени) Да се напишат предикати во PROLOG со кои ќе се реализираат аритметичките операции на собирање, одземање, множење и делење на бинарни броеви. Еден бинарен број се претставува како листа од 0 и 1 и големината на бројот е неограничена. Притоа се работи само со позитивни броеви односно доколку резултатот од одземањето е негативен број истиот се заменува со 0.

1. dodaj_nuli(L1, N, L2) – на листата L1 и додава N нули на почеток и резултатот го враќа во L2.
Овој предикат е потребен за да се средат листите пред операциите.

```
dodaj_nuli(L1, 0, L1).
dodaj_nuli(L1, N, [0|L2]) :- N1 is N-1, dodaj_nuli(L1, N1, L2), !.
```

2. izramni_listi(L1, L2, K1, K2) - на пократката листа од L1 или L2 и додава нули на почеток за листите да имаат иста должина. Резултантните листи се враќаат во К1 и К2.

%Во X is Len1-Len2 се зачувува разликата од должините на L1 и L2. Ако X>=0, значи L1 е подолга од L2 и треба да и се додадат X нули на почеток на L2 за да листите да имаат иста должина.

```
izramni_listi(L1, L2, L1, K2) :- length(L1, Len1), length(L2, Len2),
    X is Len1-Len2, X>=0, dodaj_nuli(L2, X, K2), !.
```

%Ако X<0(се подразбира), значи L1 е пократка од L2 и треба да и се додадат X нули на почеток на L1 за да листите да имаат иста должина.

```
izramni_listi(L1, L2, K1, L2) :- length(L1, Len1), length(L2, Len2),
    X is Len2-Len1, dodaj_nuli(L1, X, K1), !.
```

I) СОБИРАЊЕ

3. sobiranje(L1, L2, X1, L)- во листата L го враќа резултатот од L1 + L2, а во X1 цифрата која треба да се пренесе понатака во збирот.

%Основен случај
```
sobiranje([],[],0,[]).
```

Кога се собираат две бинарни цифри можни се следните комбинации:

Реден бр.	Пренесена цифра од минатото собирање	Собирок 1	Собирок 2	Збир	Резултат	Цифра за пренесување
I	0	0	0	**0**	0	0
II	0	0	1	**1**	1	0
III	0	1	0	**1**	1	0
IV	0	1	1	**2**	0	1
V	1	0	0	**1**	1	0
VI	1	0	1	**2**	0	1
VII	1	1	0	**2**	0	1
VIII	1	1	1	**3**	1	1

I, VIII) – Цифрите кои треба да се соберат(X+X+Y) се 0, 0, 0 или 1,1,1, од табелата се гледа дека резултатот-Y и пренесеното-Y се еднакви со собироците.
```
sobiranje([X|L1], [X|L2], Y, [Y|L]) :-
    sobiranje(L1, L2, Y, L), Y is X, !.
```

II, III, V) – Цифрите кои треба да се соберат даваат збир 1, од табелата се гледа дека резултатот е 1 и се пренесува 0.
% X is X1+X2+Y-1 ова е збирот-1, се одзима 1 за променливата X да послужи како цифра што се пренесува.
```
sobiranje([X1|L1], [X2|L2], X, [1|L]) :- sobiranje(L1, L2, Y, L),
    X is X1+X2+Y-1, X is 0, !.
```

IV, VI, VII) - Цифрите кои треба да се соберат даваат збир 2, од табелата се гледа дека резултатот е 0 и се пренесува 1.
% X is X1+X2+Y-1 ова е збирот-1, се одзима 1 за променливата X да послужи како цифра што се пренесува.
```
sobiranje([X1|L1], [X2|L2], X, [0|L]) :- sobiranje(L1, L2, Y, L),
    X is X1+X2+Y-1, X is 1, !.
```

`4. sobiranje(L1, L2, L)` – во листата L го враќа резултатот од L1 + L2.
*`sobiranje(L1, L2, X1, L)` - Во листата L го враќа резултатот од L1 + L2, а во X1 цифрата која треба да се пренесе понатака во збирот.

%Ако пренесената цифра е 0, собирањето е завршено, збирот е во L.
```
sobiranje(L1, L2, L) :- izramni_listi(L1,L2,K1,K2),
    sobiranje(K1, K2, 0, L), !.
```

%Ако пренесената цифра е 1, треба да се додаде една единица на почеток на L-збирот.
```
sobiranje(L1, L2, [1|L]) :- izramni_listi(L1,L2,K1,K2),
    sobiranje(K1, K2, 1, L).
```

II) ОДЗЕМАЊЕ

`5. odzemanje(L1, L2, X1, L)` - Во L ја враќа разликата L1-L2, а во X1 пренесената цифра.

%Основен случај
```
odzemanje([],[],0,[]).
```

- Кога се одземаат две бинарни цифри можни се следните комбинации:

Реден бр.	Пренесена цифра од минатото одземање	Цифра 1	Цифра 2	Разлика (Ц1-Ц2-пренос)	Резултат	Цифра за пренесување
I	0	0	0	**0**	0	0
II	0	0	1	**-1**	1	1
III	0	1	0	**1**	1	0
IV	0	1	1	**0**	0	0
V	1	0	0	**-1**	1	1
VI	1	0	1	**-2**	0	1
VII	1	1	0	**0**	0	0
VIII	1	1	1	**-1**	1	1

I, IV, VII) Разликата е 0(X is H1-H2-Y), од табелата се гледа дека резултатот е 0 и се пренесува 0.

```
odzemanje([H1|L1], [H2|L2], X, [0|L]) :- odzemanje(L1,L2,Y,L),
    X is H1-H2-Y, X is 0, !.
```

II, V, VIII) Разликата е -1(X is H1-H2-Y), од табелата се гледа дека резултатот е 1 и се пренесува 1.

```
odzemanje([H1|L1], [H2|L2], X, [1|L]) :- odzemanje(L1,L2,Y,L),
    X is H1-H2-Y + 2, X is 1, !.
```

III) Разликата е 1, резултатот е 1 и се пренесува 0-Y. Нема потреба да се пресметува збир во променлива.

```
odzemanje([1|L1], [0|L2], Y, [1|L]) :- odzemanje(L1,L2,Y,L), Y is 0, !.
```

VI) Разликата е -2, резултатот е 0 и се пренесува 1. Нема потреба да се пресметува збир во променлива.

```
odzemanje([0|L1], [1|L2], Y, [0|L]) :- odzemanje(L1,L2,Y,L), Y is 1, !.
```

`6. odzemanje(L1,L2,L)` – Во L ја враќа разликата L1-L2.

*`odzemanje(L1, L2, X1, L)` - Во L ја враќа разликата L1-L2, а во X1 пренесената цифра.

%ако по завршување на одземањето пренесената цифра е 0, значи L1 е поголем или еднаков на L2 и резултатот е позитивен број.

```
odzemanje(L1,L2,L) :- izramni_listi(L1,L2,K1,K2),
    odzemanje(K1, K2, 0, L), !.
```

%ако по завршување на одземањето пренесената цифра е 1, значи L1 е помал од L2 и резултатот е негативен број, зчачи треба да се врати [0] во L.

```
odzemanje(L1,L2,[0]) :- izramni_listi(L1,L2,K1,K2),
    odzemanje(K1, K2, 1, L).
```

III) МНОЖЕЊЕ

`7. pomnozi_i_soberi(L1, X, L2, L)`, бројот L1 претставен со листа го множи по X и по 10(бинарно), па го додава на L2 и резултатот го враќа во L(L1*X*10 + L2).

%Ако X=1, `append(L1, [0], L3)` – дејствува како множење на L1 со 10 и резултат во L3.
%`sobiranje(L3,L2,L)` – го собира L3 со L2.

```
pomnozi_i_soberi(L1, 1, L2, L) :- append(L1, [0], L3),
    sobiranje(L3,L2,L).
```

%Ако X=0, L1*X*10 + L2 = L2, па враќа L2.

```
pomnozi_i_soberi(L1, 0, L2, L2).
```

8. mnozenje(L1, L2, L) - ги множи L1 и L2 и резултатот го враќа во L.
Множењето почнува со множење на L1 со последната цифра од L2. За тоа се одговорни основните случи.

```
%Основни случаи
mnozenje(L1, [0],[0]).
mnozenje(L1, [1], L1).

%За секоја цифра H од L2, во K се враќа производ од L1 и L2 во минатиот повик.
%Потоа K се множи со 10 и се собира со L1, за да се добие производот од L1 и L2(L) во овој повик.
mnozenje(L1, [H|L2], L) :- mnozenje(L1, L2, K),
    pomnozi_i_soberi(L1, H, K, L).
```

IV) ДЕЛЕЊЕ

9. pogolem(L1, L2) - предикат кој враќа true ако L1 е поголем или еднаков на L2.

```
%Пред да се споредуваат листите треба да се израмнат за да имаат ист број на позиции.
pogolem(L1, L2) :- izramni_listi(L1, L2, K1, K2), e_pogolem(K1, K2).
```

10. e_pogolem(L1, L2) - предикат кој враќа true ако L1 е поголем или еднаков на L2(L1 и L2 се листи со иста должина).

```
%основен случај, ако секоја цифра од листите е иста, тогаш листите се исти.
e_pogolem([], []).

%Ако во L1 се појави 1 а во L2- 0, значи L1 е поголемо, врати точно.
e_pogolem([1|L1], [0|L2]).

%Ако на иста позиција во листите има ист елемент - X, продолжи со споредбата понатаму.
e_pogolem([X|L1], [X|L2]):- e_pogolem(L1, L2), !.
```

11. delenje(L1, L2, R) – враќа делење без остаток на L1 со L2.

```
% pogolem(L1, L2) враќа true, значи L1 е поголем од L2 и може да се дели.
% odzemanje(L1, L2, Lrez) – од L1 го одзема L2 и резултатот го сместува во Lrez.
% delenje(LRez, L2, R1) – Повикува делење на Lrez со L2 и резултатот го враќа во R1.
R1 претставува колку пати L2 се содржи во Lrez.
%sobiranje(R1, [1], R) – на R1 му додава 1 за да го добие бројот на појавувања-R на L2 во L1.

delenje(L1, L2, R) :- pogolem(L1, L2),
    odzemanje(L1, L2, LRez),
    delenje(LRez, L2, R1),
    sobiranje(R1, [1], R), !.

%Основен случај, pogolem(L1, L2) вратило False, значи делењето е завршено.
delenje(_, _, [0]).
```

Задача 6. (10 поени) Да се напише предикат во PROLOG, presmetaj(M,R), којшто за дадена квадратна матрица M како резултат враќа нова матрица R која се добива како R=M*MT (MT е транспонираната матрица, односно матрицата што се добива ако колоните станат редици, а редиците колони). Секоја матрица се проследува како листа при што секој ред се дефинира како посебна подлиста.

1. pomnoziRedSoKolona(R, K, N) – го множи редот R со колоната К и резултатот го враќа во N.
Множењето се постигнува со множење на i-тиот елемент на Редот со i-тиот елемент на колоната и додавање на збирот од сите вакви претходни производи.

```
pomnoziRedSoKolona([], [], 0).
pomnoziRedSoKolona([R|Red], [K|Kolona], N) :-
    pomnoziRedSoKolona(Red, Kolona, M), N is M+R*K.
```

2. generirajRed(R, M, NR) – Го враќа редот NR, кој се добива со множење на редот R со сите колони во матрицата M(листите во M претставуваат колони).
`pomnoziRedSoKolona(Red, M, N)` – Го множи редот со колоната M.
`[N|NovRed]` – резултатот го зачувува во новиот ред.
Потоа истиот процес го прави за останатите Колони во матрицата.

```
generirajRed(_,[],[]).
generirajRed(Red, [M|Matrica], [N|NovRed]) :-
    pomnoziRedSoKolona(Red, M, N),
    generirajRed(Red, Matrica, NovRed).
```

3. presmetaj(M, R) - Во R го враќа резултатот од множење на матрицата M со транспонирана матрица M.

```
presmetaj(M, R) :- presmetaj(M, M, R).

%Елементарен случај, ако матрицата е 1x1.
presmetaj([E], [R]) :- R is E*E, !.
```

presmetaj(M, MT, R) - .Во R го враќа резултатот од множење на матрицата M со транспонираната матрица M - MT.
За секој ред на M се повикува
generirajRed(Red, MT, NovRed), во NovRed се враќа редот помножен по трансп. матрица MT, и се зачувува во резултантната матрица: [NovRed|R].
Ова се прави за секој ред од M рекурзивно.

```
presmetaj([], _, []).
presmetaj([Red|M], MT, [NovRed|R]) :-
    generirajRed(Red, MT, NovRed),
    presmetaj(M, MT, R), !.
```

Задача 7. (20 поени) Да се напише предикат во PROLOG, transform(L1,L2), кој дадена листа L1 составена од подлисти ќе ја трансформира во листа L2 во која подлистите од влезната листа се подредени според бројот на елементи по опаѓачки редослед. Може да се претпостави дека листата L1 не е празна. Доколку има две подлисти со ист број на елементи за “поголема“ се смета онаа која има поголем прв елемент. Доколку и првите елементи им се еднакви се споредуваат вторите елементи итн. се додека не се најде барем еден елемент во една од подлистите кој е поголем од елементот на соодветната позиција во втората подлиста. Доколку двете подлисти се идентични тогаш едната се отстранува од резултатот. Може да се претпостави дека нема повеќе од две подлисти со ист број елементи.

1. compare_lists(L1, L2, X) – Предикат што споредува две листи и враќа 1 во X ако L1 е „поголемата“ листа (онаа листа што треба да биде понапред во подредувањето). Инаку, ако L2 е „поголема“ враќа 2 во X. Ако се идентични, враќа 0.

```
%Прво споредува која е подолга листа
compare_lists(L1, L2, 1) :- length(L1, X1), length(L2, X2), X1>X2, !.
```

```
compare_lists(L1, L2, 2) :- length(L1, X1), length(L2, X2), X1<X2, !.
```

%Ако листите се со иста должина ги споредува соодветните елементи во листите додека не најде во која листа има поголем елемент.
Ако листите се идентични враќа 0.

```
compare_lists([], [], 0).
compare_lists([H1|L1], [H2|L2], 1) :- H1>H2, !.
compare_lists([H1|L1], [H2|L2], 2) :- H1<H2, !.
compare_lists([H1|L1], [H2|L2], T) :- compare_lists(L1, L2, T).
```

2. add_element(L1, E, L2) – во листата L1 го вметнува елементот E на соодветното место(ако E не се појавува веќе во L1), т.ш. нема да се наруши подредувањето(L1 е подредена во опаѓачки редослед) и резултатот го враќа во L2.

%Во случај кога споредбата враќа 1, значи E треба да се пропагира понатаму, не му е местото пред H.
```
add_element([H|L1], E, [H|L2]) :- compare_lists(H,E,X), X is 1,
    add_element(L1, E, L2), !.
```

%Во случај кога споредбата враќа 0, значи L ја содржи E и ништо не треба да се вметне
```
add_element([H|L1], E, [H|L1]) :- compare_lists(H,E,X), X is 0.
```

%Во случај кога споредбата враќа 2(се подразбира), значи E е поголем од првиот елемент на L и E треба да се вметне на почеток.
```
add_element(L, E, [E|L]).
```

3. transform(L, T) – Во T ја враќа листата L подредена во опаѓачки редослед.
Во последниот рекурзивен повик – кога предикатот се повикува за последниот елемент од листата(1), во T се враќа листа од само овој елемент.
При враќање наназад, **add_element(T1, H, T)** го додава вториот елемент во листата T, на соодветното место, за T да биде подредена.
Овој процес се повторува за сите елементи од L.
На крајот се довива подредена листа.

```
transform([E], [E]). (1)
transform([H|L], T) :- transform(L, T1), add_element(T1, H, T), !. (2)
```

Задача 8. (20 поени) Да се напише предикат во PROLOG, brisi_sekoe_vtoro(L,R), којшто од дадена листа L која содржи подлисти ќе го избрише секое второ појавување на некој елемент и резултатот ќе го смести во листа R. Резултантната листа треба да ја задржи структурата на оригиналната листа.

1. otstrani_element(E,L1,L2) – предикат кој го отстранува E од L1 и го враќа резултатот во L2.
Ако E не е присутен во L1 враќа неточно.

%Кога ќе го најде елементот од E во L1, како резултат ја враќа L1 без E.
```
otstrani_element(E,[E|L1],L1).
```

%Го бара E.
```
otstrani_element(E,[E1|L1],[E1|L2]) :- otstrani_element(E,L1,L2), !.
```

2. e_lista(L) – проверува дали L е листа.
```
e_lista([]).
e_lista([_|_]).
```

3. brisi_sekoe_vtoro(L, R) – од листата L, која содржи подлисти ќе го избрише секое второ појавување на некој елемент и резултатот ќе го смести во листа R.

```
brisi_sekoe_vtoro(L, R) :- brisi_sekoe_vtoro(L,R, [], NEL).
```

4.brisi_sekoe_vtoro(L, R, EL, FEL) - од листата L која содржи подлисти ќе го избрише секое второ појавување на некој елемент и резултатот ќе го врати во листа R.
Во EL се праќа листата на елементи кои до овој момент се појавиле непарен број пати и треба да се отстранат во оваа листа.
Во FEL ја враќа листата EL со промените кои настанале при обработка на листата L.

```
%Основен случај, кога ќе се испразни L.
brisi_sekoe_vtoro([], [], EL, EL).
```

%Кога елементот E во листата L е листа, треба да се повика предикатот и за тој елемент. Важно е кога обработката на E ќе заврши, во резултантната листа R да се додаде E1-резултатот од обработката на E, а листата на елементи кои се појавуваат непарен број пати-NEL да се прследи во понатамошната обработка на L.

```
brisi_sekoe_vtoro([E|L], [E1|R], EL, FEL) :- e_lista(E),
    brisi_sekoe_vtoro(E, E1, EL, NEL),
    brisi_sekoe_vtoro(L, R, NEL, FEL), !.
```

%Кога елементот E во листата L не е листа, потребно е да се провери дали тој треба да се отстрани или не.

%Ако **otstrani_element(E, EL, NEL)** врати точно, значи E се наоѓал во EL, се отстранил и резултатот е во NEL. NEL е актуелната листа на елементи кои треба да се отстранат и таа се прследува понатаму. Соодветно, во R не се додава E.

```
brisi_sekoe_vtoro([E|L], R, EL, FEL) :-
    otstrani_element(E, EL, NEL),
    brisi_sekoe_vtoro(L, R, NEL, FEL), !.
```

%Ако **otstrani_element(E, EL, NEL)** врати неточно(се подразбира), значи E не се наоѓал во EL - тој се појавува непарен пат и не треба да се отстрани.
[E|EL] – се додава во EL и се додава во R: [E|R].

```
brisi_sekoe_vtoro([E|L], [E|R], EL, FEL):-
    brisi_sekoe_vtoro(L, R, [E|EL], FEL), !.
```

*Објаснување на FEL во **brisi_sekoe_vtoro(L, R, EL, FEL).**
Оваа променлива се инстанцира кога предикатот ќе дојде до терминален случај – кога листата која се обработува ќе заврши.
(`brisi_sekoe_vtoro([], [], EL, EL).`)
Служи за да се запази листата на елементи кои се појавуваат непарен број пати, затоа што елементите на листата кои се самите листи се обработуваат рекурзивно, па при враќање наназад се губат промените кои настанале врз EL .
Затоа е потребно кога листата ќе заврши, FEL да се инстанцира со вредноста на EL, и потоа во понатамошната обработка да се прследи FEL.