

Домашна задача 1

Тијана Атанасовска (196014)

Задача 1

Палиндром претставува низа од знаци чиј редослед е ист однапред и одназад. Наједноставната имплементација е да се преврти листата и да се спореди дали е иста со влезната листа. Дополнително, барањето на задачата е да се провери дали должината е непарен број, а тоа се постигнува со проверка дали остатокот е цел број при делење на должината на листата со 2.

Се користат следните предикати:

- **непарен(X)** – Го дели бројот X со два и проверува дали резултатот е децимален број.
Влезни аргументи: број X.
Излезни аргументи: Нема. Враќа true/false.
- **Dolzina(L,X)** – Ја враќа должината на листата L како X.
Влезни аргументи: Листа L
Излезни аргументи: Број X
- **dodadi(L1,L2,L)** – Ги спојува листите L1 и L2 во резултантна листа L.
Влезни аргументи: Листи L1, L2
Излезни аргументи: Листа L
- **е_непарен(L1)** – Проверува дали низа од карактери има непарна должина.
Влезни аргументи: Листа L1
Излезни аргументи: Нема. Враќа true/false.
- **Непарен_palindrom(L1)** – Проверува дали низа од карактери е непарен палиндром. Ја зема влезната листа и ја проверува парноста на нејзината должина. Доколку овој услов не е задоволен, предикатот веднаш враќа False, бидејќи останатите услови се поврзани со AND операција. Доколку должината е непарна, се превртува влезната листа со предикатот prevrti и ново-креираната листа се сместува во променливата Z. На крај се споредува дали двете листи се исти.
Влезни аргументи: Листа L1
Излезни аргументи: Нема. Враќа true/false.

```

1 neparen(X):-Y is X/2, float(Y).
2
3 dolzina([],0).
4 dolzina([_|Opaska],X):-dolzina(Opaska,Y),X is Y+1.
5
6 dodadi([],L, L).
7 dodadi([X|O],L,[X|NL]) :- dodadi(O,L,NL).
8
9 prevrti([],[]).
10 prevrti([X|L1],L2):-prevrti(L1,L),dodadi(L,[X],L2).
11
12 e_neparen(L1):-dolzina(L1,X),neparen(X).
13
14 neparen_palindrom(L1):-e_neparen(L1),prevrti(L1,Z),Z==L1.

```

Повикување на предикатот:

```

neparen_palindrom([d,e,l,e,v,e,l,e,d])
true
?- neparen_palindrom([d,e,l,e,v,e,l,e,d])

```

```

neparen_palindrom([d,e,l,e,e,l,e,d])
false
?- neparen_palindrom([d,e,l,e,e,l,e,d])

```

Задача 2

Барањето на задачата е да се најде подниза со должина N која се појавува најмногу пати во дадена влезна низа. Крајното решение е генерирање на сите поднизи со должина N и броење колку пати се јавува секоја од нив во влезната низа. Потоа се бара максимум и се враќа онаа подниза која се појавува максимум број пати во влезната низа.

Се користат следните предикати:

- **podlista(L1,L2)** – Ги враќа сите подлисти на влезната листа. Доколку се повика со два влезни аргументи враќа true/false.
Влезни аргументи: Листа L1
Излезни аргументи: Подлиста L2

За генерирање на сите поднизи со должина N во други програмски јазици вообичаено е да се користи вгнезден for циклус. Во Пролог тоа се имплементира со два предикати кои означуваат надворешен и внатрешен циклус и притоа надворешниот го повикува внатрешниот. Овој предикат (podlista) е веќе изведен во аудиториските вежби, затоа нема да биде детално објаснет овде.

```

4 podlista(L1,L2):-nadvoresen(L1,L2).
5
6 nadvoresen([X|L1],[X|L2]):-vnatresen(L1,L2).
7 nadvoresen([X|L1],L2):-nadvoresen(L1,L2).
8
9 vnatresen(L1,[]).
10 vnatresen([X|L1],[X|L2]):-vnatresen(L1,L2).

```

- **zemi_podlista_N(Lista,Podlista,N)** – Ги враќа само подлистите со должина N. Го повикува претходниот предикат за да ги генерира сите подлисти на влезната листа L, ја проверува должината на подлистата и ја враќа подлистата само доколку нејзината должина е иста со влезниот аргумент N.

Влезни аргументи: Листа Lista, Број N

Излезни аргументи: Подлиста Podlista

```

13 zemi_podlista_N(Lista,Podlista,N):-podlista(Lista, Podlista),
14                                     dolzina(Podlista,Len), N==Len.
15

```

- **kolku_pati(Lista,Podlista,N)** – За дадена листа и подлиста, враќа број N – колку пати ја има подлистата во листата. Се користи помошниот предикат findall, кој ги собира сите одговори од повикот на предикатот на втората позиција (podlista) кога одговорот е true. Овде се користи _ бидејќи важно е само колку пати ќе се пронајде подлистата во листата. Според тоа ќе се генерираат толку мемориски локации, а потоа само ги броиме тие локации што ни ја дава информацијата колку пати се јавува подлистата во листата.

Влезни аргументи: Листа Lista, Подлиста Podlista

Излезни аргументи: Број N

```

16 kolku_pati(Lista,Podlista,N):-findall(_,podlista(Lista,Podlista),Pati),
17                                     dolzina(Pati,N).

```

- **lista_kolku_pati(L,N,Kolku)** – Предикат кој со помош на findall ги собира сите подлисти со должина N, потоа изминува низ секоја од нив и пребројува колку пати се јавува во листата. Го враќа резултатот во Kolku. Овој предикат ќе врати одговори колку што има подлисти со должина N, со кликање на Next.

Влезни аргументи: Листа L, Број N (означува барана должина на подлиста)

Излезни аргументи: Број Kolku

```

25 lista_kolku_pati(L,N,Kolku):-findall(Podlista, zemi_podlista_N(L,Podlista,N), Podlisti_N),
26                                     member(PodlistaTemp, Podlisti_N),
27                                     kolku_pati(L,PodlistaTemp, Kolku).
28

```

- **naj_podniza(L,N,Result)** – Главниот предикат кој ја враќа подлистата со должина N која се јавува најмногу пати. Со првиот повик на findall, се креира листа Collection_Kolku која содржи броеви колку пати се јавува секоја од подлистите со

должина N, кои се генерираат во lista_kolku_pati. Потоа се бара најголем број во оваа низа и се зачувува во променливата Max.

Со следниот повик на findall, ги земаме сите подлисти со должина N, итерираме низ нив со предикатот member и за секоја проверуваме колку пати се јавува во влезната листа. Доколку бројот на појавување е ист со Max, тогаш подлистата ја враќаваме. Cut операторот се користи за да се врати само првата подлиста. Ова го спречува печатењето на дупликати, бидејќи подлистите се појавуваат повеќепати. Side effect е што спречува печатење на РАЗЛИЧНИ подлисти кои имаат иста должина Max.

```
31 naj_podniza(L,N,Result):-findall(Kolku,lista_kolku_pati(L,N,Kolku),Collection_Kolku),
32   maksimum(Collection_Kolku, Max),
33   findall(Podlista,zemi_podlista_N(L,Podlista,N),Collection_Podlisti),
34   member(Result,Collection_Podlisti),
35   kolku_pati(L,Result,Pati),Pati==Max,!.
```

Повикување на предикатот:

```
L = [2]
?- naj_podniza([1,2,2,3,2,2,4,2,2,3],1,L).
L = [2, 2]
?- naj_podniza([1,2,2,3,2,2,4,2,2,3],2,L).
L = [2, 2, 3]
?- naj_podniza([1,2,2,3,2,2,4,2,2,3],3,L)
```

Задача 3

Оваа задача има 3 барања кои треба да се исполнети. За имплементација на првото се врши проверка дали должината на листата е поголема-еднаква со 2. Доколку е задоволен овој услов, се продолжува со проверка на останатите услови.

Вториот и третиот случај се опфатени со ист предикат, бидејќи вториот случај е првата проверка која треба да се изврши кога листата е подолга од 2 елементи (а тоа е третото барање).

Се користат следните предикати:

- **Dolzina(L,X)** – Ја враќа должината на листата L како X.
Влезни аргументи: Листа L
Излезни аргументи: Број X
- **vtoriot_e_pogolem(L)** – Проверува дали вториот елемент во листата е поголем од првиот
Влезни аргументи: Листа L
Излезни аргументи: Нема. Враќа true/false.

```
5 vtoriot_e_pogolem([X,Y|_]):-X<Y.
```

- **ostatok_lista** – Ја враќа листата L без првиот и вториот елемент.
Влезни аргументи: Листа L
Излезни аргументи: Листа L1

```
12 ostatok_lista([_,_|L],L).
```

- **zig** – За дадена листа L провери дали првиот елемент е помал од вториот.
Влезни аргументи: Листа L
Излезни аргументи: Нема. Враќа true/false.

```
7 zig(L):-vtoriot_e_pogolem(L).
```

- **zag** – За дадена листа L провери дали вториот елемент е поголем од третиот.
Влезни аргументи: Листа L
Излезни аргументи: Нема. Враќа true/false.

```
11 zag([_|L]):-not(vtoriot_e_pogolem(L)).
```

- **zig_zag** – Предикат кој ја имплементира главната логика на задачата.
Овој предикат ги користи помошните предикати ZIG и ZAG со кои се прават основните проверки. Истите се објаснети во претходниот дел. Потоа ги отстранува само првиот и вториот елемент од листата со предикатот **ostatok_lista**. Новодобиената листа се праќа повторно во рекурзивен повик на **zig_zag**.

Разликува три случаи: Во листата има останато еден, два или три и повеќе елементи.

Ако во листата има останато три или повеќе елементи, треба да се извршат двете проверки и потоа рекурзивно да се проверува и за другиот дел од листата.

За првиот и вториот број проверува дали вториот е поголем од првиот. За вториот и третиот од листата проверува дали третиот е помал од вториот елемент. Потоа, ги повторува проверките за останатиот дел од листата со тоа што последниот

елемент од проверките (третиот елемент) повторно е дел од листата која се праќа на рекурзивниот повик бидејќи е потребно да се провери дали е помал од четвртиот елемент, итн...

Другиот случај кој треба да се моделира е кога листата има останато само 2 елементи. Тогаш потребно е да се изврши само проверката `zig`, односно дали вториот елемент е поголем од првиот и да се врати назад во рекурзијата.

Случајот кога во листата има останато само еден елемент. Тоа означува дека условите до таму се исполнети, сам елемент нема со што да се споредува па се враќа `TRUE`.

Влезни аргументи: Листа `L`

Излезни аргументи: Нема. Враќа `true/false`.

```
21 zig_zag([_]).
22 zig_zag(L):-dolzina(L,N),N is 2,zig(L).
23 zig_zag(L):-zig(L), zag(L), ostatok_lista(L,L1), zig_zag(L1).
```

- `proveri` – Предикат кој проверува дали должината на листата е поголема/еднаква со 2. Доколку да, тогаш ги проверува следните услови кои беа наведени во задачата.

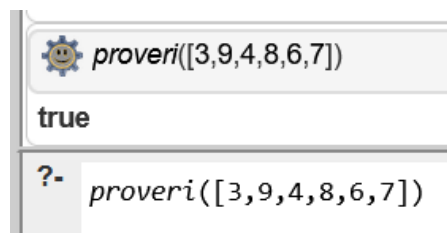
*Cut операторот е употребен за Prolog да не го враќа default-ното `false`, кое го враќа после секоја програма кога нема следни решенија.

Влезни аргументи: Листа `L`

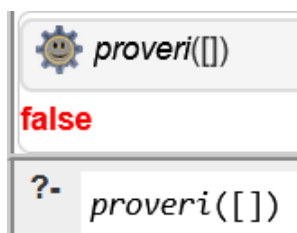
Излезни аргументи: Нема. Враќа `true/false`.

```
38 prover_i(L):-dolzina(L,X),X>=2,zig_zag(L),!.
```

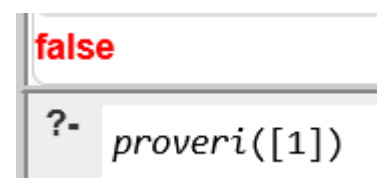
Повикување на предикатот:



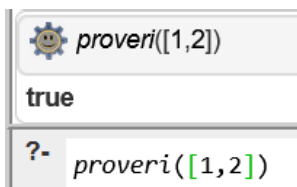
```
proveri([3,9,4,8,6,7])
true
?- prover_i([3,9,4,8,6,7])
```



```
proveri([])
false
?- prover_i([])
```



```
false
?- prover_i([1])
```



```
proveri([1,2])
true
?- prover_i([1,2])
```

Задача 4

Пермутација на низа од карактери означува да се најдат сите начини на кои низата може да се подреди.

Пермутации на елемент во дадена листа – поставување на елементот на сите можни позиции во листата.

Се користат следните предикати:

- `dodadi(L1,L2,L3)` – Предикат за спојување на L1 и L2 во излезна листа L3. Доколку предикатот се повика на поинаков начин од предвидениот дава поразлични резултати кои помагаат за да се генерира пермутација.
- `permutacija_X(Lista,Element,Permutation)` – Ги враќа сите пермутации за даден елемент во дадена листа.
Влезни аргументи: Листа Lista, Element (кој ќе се пермутира)
Излезни аргументи: Permutation – една пермутација за елементот во листата.

```
4 %permutiranje na eden element vo dadena lista -  
5 %generiranje na site mozni pozicii na toj element vo listata  
6 permutacija_X(Lista,Element,Permutation):-dodadi(X,Y,List),  
7                                         dodadi(X,[Element],L1),  
8                                         dodadi(L1,Y,Permutation).
```

- `permutacija(L, Result)` – Рекурзивен предикат за добивање на сите можни пермутации на една листа. Првин влегува во листата и ја изминува се до последниот елемент така што првиот повик кој го прави е рекурзивен. Основниот случај е единствен останат елемент. Пермутација на еден елемент е самиот елемент. Потоа на враќање – генерира пермутација за соодветниот елемент до кој стигнало изминувањето од назад и остатокот од листата – односно го става елементот на сите позиции во листата која е всушност делот ПОСЛЕ тој елемент во оригиналната листа. Така генерираната пермутација ја лепи на резултатот и се враќа назад во рекурзијата.

```
17 permutacija([X],[X]).  
18 permutacija([Head|Tail],Res):-permutacija(Tail,Res2),  
19                               permutacija_X(Res2,Head,L),  
20                               append(L,[],Res).  
21
```

- `permutacii(Input,Output)` – предикат кој за влезната листа Инпут, го повикува предикатот `permutacija` врз секој елемент и резултатите ги собира во една листа со помош на `findall`, која потоа ја враќа на излез.

Повикување на предикатот:

```
permutacii([a,b,c],L).
```

```
L = [[a, b, c], [b, a, c], [b, c, a], [a, c, b], [c, a, b], [c, b, a]]
```

```
?- permutacii([a,b,c],L).
```

Бидејќи бројот на пермутации е факториел од бројот на елементи на влезната листа, истата проверка ја правам со следниот предикат:

```
110 faktoriel(1,1):-!.
111 faktoriel(N,Out):-S is N-1,faktoriel(S,M),Out is (M*N).
112 permutacii(Input,Output):-findall(Res,permutacija(Input,Res),Output),
113     length(Input, InpLen),length(Output,OutLen),
114     faktoriel(InpLen,FaktRes),FaktRes==OutLen.
```

Задача 5

а) Собирање на бинарни броеви

Кога правиме собирање на бинарни броеви на хартија почнуваме од последните цифри, за да може да се додаваат оние што се памтат при собирање. Истата логика ја имплементирам во Пролог. За таа цел, најпрвин ги превртив двете листи во обратен редослед, за собирањето да биде при влегување во рекурзијата. Делот со ЗАПАМТИ (при собирање 1+1, запиши 0, запамти 1) го имплементирам со помошна променлива. Откако ќе се соберат двете превртени листи, резултатот е запишан во обратен редослед. Затоа на крај се превртува и резултатот пред да се врати.

Бинарното собирање на единечни цифри не може директно да се изведе во Пролог, затоа потребно е правилата да се дефинираат во базата на знаење. Тоа е изведено со предикатот `fakt`, кој е проширен така што во предвид зема дали памтиме нешто од претходно собирање, а враќа цифрата што треба да се запише и цифрата што треба да се памти.

Се користат следните предикати:

- `fakt(Sobirok1, Sobirok2, Prethodno, Zapishi, Pamti)` – предикат за дефинирање на знаењето во базата, односно собирање на единечни бинарни цифри. Опфатени се сите случаи кои може да се појават при собирање на две бинарни цифри и претходно памтење на цифра.

При тоа со променливата `Prethodno` се зема во предвид цифра која ја памтиме од претходно собирање. Пример: 11+1 =>


```
33 %Cut operator za da ne vrakja false.  
34 sobiranje(L1,L2,Result):-prevrti(L1,P1),prevrti(L2,P2),  
35                             sobiranje_helper(P1,P2,0,ResultReversed),  
36                             prevrti(ResultReversed,Result),!.  
37
```

- `sobiranje_helper(L1,L2,Prethodno,Result)` – Овој предикат прави собирање на двете влезни листи, но претходно превртени. Рекурзивниот повик ги зема првите елементи од двете листи (последни елементи во правилниот редослед), го повикува предикатот `fakt` со нив и со цифрата која ја памтиме од Претходно. Според тоа што ќе се зададе во овие три променливи ќе се повика соодветниот факт од базата на знаење и ќе се врати што треба да се Запише и што да се Запамти за следното собирање. Потоа се прави рекурзивниот повик со остатокот од листите и цифрата што ја памтиме. На резултатот кој ќе се врати од рекурзивниот повик, се лепи на почеток цифрата која требало да се запише во моменталниот повик -> `[Zapishi|SubResult]` .

```

30 sobiranje_helper([Head1|Tail1], [Head2|Tail2], Prethodno,[Zapishi|SubResult])
31     :-fakt(Head1,Head2,Prethodno,Zapishi,Pamti),
32     sobiranje_helper(Tail1,Tail2,Pamti,SubResult).
33

```

Крајните случаи на овој предикат се при испразнување на првата или втората листа, зависно која од нив е поголема. Јас ги моделирам и двата случаи. Овде треба да се земе предвид дека може да стигнеме до крајот на двете листи, но да треба да се запише цифра 1 од претходно памтење. Тоа е третиот случај на следната слика.

```

19 %Kraen slucaj koga vtorata lista e pogolema I pritoa ne pamtime 1 od prethodното собирање
20 sobiranje_helper([],L2,0,L2).
21 %Kraen slucaj koga prvata lista e pogolema I pritoa ne pamtime 1 od prethodното собирање
22 sobiranje_helper(L1,[],0,L1).
23 %Kraen slucaj koga site broevi se sobrani ama pamtime 1 od posledното собирање па потребно
24 % e da se zapise kako prv element vo listata.
25 sobiranje_helper([],[],1,[1]).

```

Останатите два случаи се кога едната од листите е празна, но со другата треба да се собере цифрата што ја памтиме од претходно. Тука, пак треба да се направи рекурзивен повик за во случај при ова собирање да има цифра која се памти, па потребно е да се запише во следниот чекор или може собирањето да продолжи додека има цифри во листата.



```

26 %Kraen slucaj koga prvata lista e pogolema I pamtime 1 od prethodnoto sobiranje,
27 %pa potrebno e da se sobere taa edinica so glavata na prvata lista.
28 %Povikot na sobiranje_helper e za vo slucaj povtorno da pamtime edinica.
29 %Primer: [1,1,0],[],1 Zapishi=0, Pamti=1; Sega potrebno e da se povika povtorno sobiranje
30 %na [1,0] so ovaa 1 sto ja pamtime. I se taka dodeka ima 1 koi se pamtat.
31 sobiranje_helper([Head1|Tail1],[],1,[Zapishi|SubResult]):-
32     fakt(Head1,1,0,Zapishi,Pamti),sobiranje_helper(Tail1,[],Pamti,SubResult).
33
34 % Kraen slucaj koga vtorata lista e pogolema I pamtime 1 od prethodnoto sobiranje.
35 % Istoto objasnuvanje e I ovde kako vo prethodniot komentar.
36 sobiranje_helper([], [Head2|Tail2],1,[Zapishi|SubResult]):-
37     fakt(Head2,1,0,Zapishi,Pamti),sobiranje_helper([],Tail2,Pamti,SubResult).
38

```

Овие крајни случаи може да се намалат доколку првин проверуваме која од листите е поголема, па секогаш повикот на `sobiranje_helper` го правиме со првата листа да е поголемата!

Повикување на предикатот:

 <code>sobiranje([1,1,0],[1,1],L).</code>	 <code>sobiranje([1,1],[1,1,1,0],L).</code>
<code>L = [1, 0, 0, 1]</code>	<code>L = [1, 0, 0, 0, 1]</code>
<code>?- sobiranje([1,1,0],[1,1],L).</code>	<code>?- sobiranje([1,1],[1,1,1,0],L).</code>

b) Одземање на бинарни броеви

Одземање на бинарни броеви може да се изведе со помош на втор комплемент и собирање.

$$A - B \Leftrightarrow A + \text{two_complement}(B) \Leftrightarrow A + (\text{inverse}(B) + 1)$$

Се користат следните предикати:

- `Inverse(C, Inverted)` – предикат со кој се внесуваат фактите за инверзна бинарна цифра.

```

52 %Fakti za implementacija na inverten binaren broj
53 inverse(1,0).
54 inverse(0,1).

```

- `invert_binary(L, InverseL)` – за дадена низа од бинарни цифри, врати ја секоја цифра заменета со инверзната. 11001 -> 00110

Првин го добива инверзниот на првиот број во низата (`Head`) со предикатот `inverse` и таквиот број го лепи на почеток на низата која ќе се генерира внатрешно во рекурзијата -> `[Inverted|Result]`. Потоа ја повикува рекурзивно истата функција за да продолжи менувањето и за останатите цифри. Кога ќе се испразни низата враќа

празна листа во која враќајќи се назад во рекурзијата ќе ги става инвертираните цифри.

```
57 invert_binary([],[]).
58 invert_binary([Head|Tail],[Inverted|Result]):-inverse(Head,Inverted),invert_binary(Tail,Result).
```

- **two_complement(L1,Result)** – Пресметува втор комплемент за влезен бинарен број (низата L1) и го враќа во Result. Го имплементира овој дел од формулата: $(inverse(B) + 1)$. Се пресметува инверзниот број на бинарниот со повик на горниот предикат `invert_binary`. За собирање со 1 го користиме предикатот `sobiranje`, така што како втор аргумент се дава [1] бидејќи аргументот мора да биде листа.

```
60 %Presmetuvanje na two complement za da se implementira odzemanje so pomos na sobiranje. Se
61 %presmetuva inverzniot broj na binariot, se sobira so 1 I se vrakja rezultatot.
62 two_complement(L1,Result):-invert_binary(L1,P1),sobiranje(P1,[1],Result).
```

- **popolni_nuli(L1,N,L2)** – Додава N нули однапред на бинарниот број даден како влезна листа L1. Овој предикат е потребен за да се пополнат нули на почеток од вториот број пред да се најде неговиот инверзен. За во случајот кога неговата должина ќе биде помала од првиот број. N – Разлика на должините меѓу првиот и вториот број.

Пример:

$1111 - 101 = 1111 + \text{complement}(0101) + 1 = 1111 + (1010 + 1) = 11010$

Ако не се додадат нули на вториот број ќе биде:

$1111 + \text{complement}(101) + 1 = 1111 + (010 + 1) = 1111 + 11 = 10010$

Предикатот е рекурзивен кој лепи N нули на резултантната листа, а на крај ја лепи листата L1 кога се стигнува до основниот случај.

```
96 popolni_nuli(L1,0,L1):-!.
97 popolni_nuli(L1,N,[0|Result]):-M is N-1,popolni_nuli(L1,M,Result).
```

- **otstrani_nuli(L, Result)** – Ги отстранува нулите на почетокот на листата L се додека не стигне до првата единица. Предикат потребен за откако ќе се отстрани првата единица од резултатот (таканаречен carriage), можно е да има неколку нули кои немаат никаква вредност (нули на почеток на бинарен број немаат вредност), па добро е да се отстранат.

Основниот случај е кога ќе стигнеме до првата единица. Тогаш во резултат го сместуваме остатокот од листата и се враќа назад.

```
101 otstrani_nuli([1|Tail],[1|Tail]).
102 otstrani_nuli([Head|Tail],Result):-Head=0,otstrani_nuli(Tail,Result).
```

- **odzemanje(L1,L2,Result)** – Одземање на два бинарни броеви дадени како влезни листи L1, L2 и враќање на нивниот резултат во излезната променлива Result. Предикатот првин ја пресметува разликата во должините на првиот и вториот број – SUBS. Гополни вториот број однапред со нули колку што е таа разлика (објаснето погоре зошто). Потоа, се пресметува втор комплемент и се сместува во InvertedL2. Се врши собирање на првата влезна листа и вториот комплемент на втората влезна

листа. Од резултатот ја остраниваме првата бројка бидејќи тоа е carriage -> [_|ResultWithZeros]. Во ResultWithZeros останува резултатот кој однапред има вишок нули, па ги остраниваме нив со `otstrani_nuli` пред да го вратиме официјалниот резултат.

```
109 odzemanje(L1,L2,Result):-length(L1,N1),length(L2,N2),Subs is N1-N2,
110      popolni_nuli(L2,Subs,L22),two_complement(L22,InvertedL2),
111      sobiranje(L1,InvertedL2,[_|ResultWithZeros]),
112      otstrani_nuli(ResultWithZeros,Result),!.
```

Повикување на предикатот:

<code>Result = [1, 0, 0]</code>	<code>L = [1, 1]</code>
<code>?- odzemanje([1,1,1,1],[1,0,1,1],Result)</code>	<code>?- odzemanje([1,1,0],[1,1],L)</code>

V) Множење на бинарни броеви

Множење на бинарни броеви се имплементира со помош на собирање.

Операциите се полесно воочливи преку следната слика.

```

1 0 1 0 ----> Multiplicand
1 0 1 1 ----> Multiplier
-----
1 0 1 0 ----> Partial product 1
1 0 1 0 x ----> Partial product 2
0 0 0 0 x x ----> Partial product 3
1 0 1 0 x x x ----> Partial product 4
-----
1 1 0 1 1 1 0 ----> Final Product

Rules of Binary Multiplication
0 x 0 = 0, 0 x 1 = 0, 1 x 0 = 0, 1 x 1 = 1
Rules of Vinary Addition
0 + 0 = 0, 0 + 1 = 1, 1 + 1 = 10, 1 + 1 + 1 = 11
```

Одејќи од последната кон првата цифра на множителот (Multiplier), ако цифрата е 1 - се собира множителот (Multiplicand), поместен во десно, со претходниот парцијален продукт, ако е 0 – се прави само поместување во десно. Имплементацијата во код бара да се преврти множителот бидејќи како што објаснив погоре, потребно е да се врши од последната цифра кон напред. Потоа се врши соодветната операција во зависност од цифрата. Кога имаме 0, на крајот од Множителот се лепи нула. Ова е еквивалентно со поместување во десно, бидејќи при следното собирање ќе имаме дополнителна цифра на крајот, а при тоа оваа цифра не смета при собирање бидејќи е 0. Кога имаме 1, го собираме Множителот со моменталниот Парцијален продукт и потоа правиме

поместување во десно за во следниот чекор. И во двата случаи мора да се прави поместување во десно!

Оваа задача ја решив со tail рекурзија, односно имплементирав акумулатор. Мора првин да се изврши собирањето со Парцијалниот продукт, пред да се продолжи множењето. Овој акумулатор на почеток е иницијализиран на нула.

Се користат следните предикати:

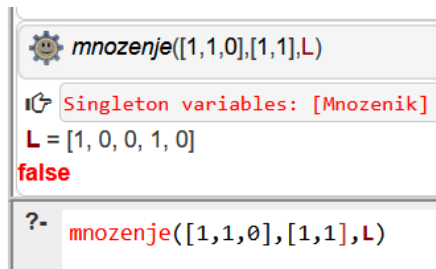
- **prevrti(L1,L)** – Преврти ја листата L1 и врати ја во L
Влезни аргументи: Листа L1
Излезни аргументи: Листа L
*Преземени од аудиториски вежби, затоа не се детално објаснети
- **sobiranje(L1,L2,Result)** – Собери ги листите L1, L2 и резултатот врати го во Result
Влезни аргументи: Листи L1, L2
Излезни аргументи: Листа Result
*Имплементирано во оваа задача, под А.
- **shift_desno(L, ShiftL)** – Додава нула на крајот од листата 0, што е исто како и поместување во десно.
Влезни аргументи: Листа L
Излезни аргументи: Листа ShiftL
- **mnozenje(Mnozenik, Mnozitel, Result)** – Помножи ги двата бинарни броеви Mnozenik и Mnozitel, дадени како листи и резултатот врати го во Result. Овој предикат врши подготовка за множењето – го превртува Множителот и потоа повикува mnozenje_helper, со акумулатор сетирани на 0, каде се извршува клучната рекурзија.
Влезни аргументи: Листа Mnozenik, Листа Mnozitel
Излезни аргументи: Листа Result

```
75 mnozenje(Mnozenik, Mnozitel, Result):-prevrti(Mnozitel, PrevrtenMnozitel),
76                                     mnozenje_helper(Mnozenik,[0],PrevrtenMnozitel,Result).
```
- **mnozenje_helper(Mnozenik,Akumulator,PrevrtenMnozitel,Result)** – Главната функција во која се врши множењето. Рекурзијата е имплементирана во зависност од моменталната цифра на Множителот, која се зема во X со [X|OstatokMnozitel]. Доколку е 0, се прави само поместување во десно со предикатот shift_desno и рекурзивно се повикува предикатот со останатиот дел од Множителот. Доколку моменталната цифра е 1, се повикува собирање на Парцијалниот продукт со Множеникот. Во овој случај повторно се прави поместување во десно. Рекурзијата секогаш се повикува со поместениот Множеник, на тој начин се имплементира поместувањето во секоја итерација како што е потребно кога множиме бинарни броеви. Основниот случај на рекурзијата е кога во Множителот нема повеќе цифри. Тогаш како последен аргумент се запишува Парцијалниот продукт

(SubResult), кој се пресметуваше се до овој момент. Назад рекурзивно само се враќа оваа вредност.

```
64 mnozenje_helper(Mnozenik, SubResult, [], SubResult).
65
66 mnozenje_helper(Mnozenik, SubResult, [X|OstatokMnozitel], Result):-
67     X==0, shift_desno(Mnozenik, ShiftMnozenik),
68     mnozenje_helper(ShiftMnozenik, SubResult, OstatokMnozitel, Result).
69
70 mnozenje_helper(Mnozenik, SubResult, [X|OstatokMnozitel], Result):-
71     X==1, shift_desno(Mnozenik, ShiftMnozenik),
72     sobiranje(Mnozenik, SubResult, PodRezultat),
73     mnozenje_helper(ShiftMnozenik, PodRezultat, OstatokMnozitel, Result).
74
```

Повикување на предикатот:



```
mnozenje([1,1,0],[1,1],L)
Singleton variables: [Mnozenik]
L = [1, 0, 0, 1, 0]
false
?- mnozenje([1,1,0],[1,1],L)
```

г) Делење на бинарни броеви

Делењето е операција која може да се изведе преку одземање, односно колку пати Делителот може да се одземе од Деленикот и остатокот да е 0.

$$8/2 \rightarrow 8-2-2-2-2 = 8-4*2 \rightarrow 8/2=4$$

Бидејќи делењето е одземање, проверувам колку пати може да се одземе Делителот од Деленикот и тој counter, кој е добиен во декаден систем, го претворам во бинарен број за да се прикаже резултатот бинарно.

Се користат следните предикати:

- `delenje_helper(L1,L2,Times)` – `L1` и `L2` се деленикот и делителот соодветно. Овој предикат ја одзема `L2` од `L1` и брои колку пати може да се изведе тоа. Првин се одзема `L2` од `L1` и се запишува во `Result`. Потоа се повикува функцијата рекурзивно со `Result` и со `L2`. Основниот случај е кога двата броеви се исти (делењето е целобројно така што мора да стигне до овој случај). Враќајќи се назад броиме колкупати била повикана рекурзијата и тој број го враќаме како резултат.

```

86 %Ova vrakja decimalna cifra, treba da se pretvore vo binaren broj!
87 delenje_helper(L1,L2,1):-L1==L2.
88 delenje_helper(L1,L2,Times):-odzemanje(L1,L2,Result),
89     delenje_helper(Result,L2,SubTimes), Times is SubTimes+1.

```

- **dekaden_vo_binaren(Dekaden, PrevrtenBinaren)** - Бројачот добиен во претходното резултатот треба да се претвори во бинарен број. Тоа се постигнува со овој предикат. За да претвориме декаден во бинарен, го делиме декадниот број со 2. Ако има остаток пишуваме 1, ако нема пишуваме 0. Пресметување на остатокот е со вградениот предикат `mod`, а делењето е изведено со `div` (иако може да се изведе и со погорниот предикат `delenje_helper([Dekaden],[2])` , така што аргументите мора да се зададат како листи!). Рекурзијата прекинува кога нема повеќе што да се дели со 2. `Cut` операторот е додаден за да не враќа фалсе. Овој предикат враќа превртен бинарен стринг бидејќи во Пролог може да лепиме цифри само од напред!

```

97 dekadnen_vo_binaren(0, []):-!.
98 dekadnen_vo_binaren(Dekaden, [0|L]):- 0 is mod(Dekaden,2),OstatokDek is div(Dekaden,2),
99     dekadnen_vo_binaren(OstatokDek,L).
100 dekadnen_vo_binaren(Dekaden, [1|L]):-1 is mod(Dekaden,2),OstatokDek is div(Dekaden,2),
101     dekadnen_vo_binaren(OstatokDek,L).
102

```

- **delenje(L1, L2, Result)** – делење на два бинарни броеви зададени како листи и враќање на резултатот. Првин се пресметува колку пати се содржи вториот број во првиот со предикатот `delenje_helper`. Добиениот број кој е во декадна форма, се префрлува во бинарен броен систем и низата се превртува за да се добие точниот редослед.

```

91 %Cut operator za da ne prodolzi nazad kon delenjeto ednas koga ke go zavrshi do kraj.
92 delenje(L1,L2,Result):-delenje_helper(L1,L2,Dekaden),!,
93     dekadnen_vo_binaren(Dekaden, PrevrtenBinaren), prevrti(PrevrtenBinaren, Result).

```

Повикување на предикатот:

```

? delenje([1,0,0,0],[1,0],L)
L = [1, 0, 0]
false
?- delenje([1,0,0,0],[1,0],L)

```


***Друго решение на задачава е претворање во декаден систем, вршење на било која од операциите, па повторно враќање во бинарен систем. Сепак, таа имплементација е помал предизвик, затоа не е прикажана овде!**

Задача 6

Множење на две матрици се сведува на множење на секоја редица од првата матрица со секоја колона од втората матрица. При тоа, потребно е да биде задоволен условот: бројот на колони на првата матрица да е еднаков со бројот на редици на втората матрица. Бројот на редици во резултантната матрица е ист со бројот на редици во првата матрица, додека бројот на колони во резултантната матрица е ист со бројот на колони во втората матрица.

$$(R1 \times K1) * (R2 \times K2) = (R1 \times K2) , K1=R2$$

Транспонирана матрица претставува матрица чии редици се колоните од оригиналната матрица.

Од погорните дефиниции може да се согледа дека множење на матрица со својата транспонирана матрица се сведува на меѓусебно множење на сите редици во оригиналната матрица.

Псевдо код за множење на две матрици:

```
Result_matrix = []
for redica in M:
    for kolona in S:
        Result_matrix.append([redica*kolona])
```

Псевдо код за множење на матрица со својата транспонирана матрица $M * M^T$:

```
Result_matrix = []
for redica1 in M:
    for redica2 in M:
        Result_matrix.append([redica1*redica2])
```

Со користење на знаењето од линеарна алгебра може да се заклучи дека нема потреба од генерирање на транспонирана матрица за да се пресмета резултатот.

Логика на имплементација на код во Пролог:

За влезната матрица M1, креираме дупликат матрица M2. Се итерира во редиците на првата матрица и редиците на дупликат матрицата. Ова е имплементација на вгнездените for циклуси од псевдо кодот погоре. Се повикува множење на соодветните редици и резултатот (единствен број) се запишува во листа. Листата изградена од сите резултантни броеви се дели на подлисти со точно R1 елементи, каде R1 е бројот на редици на првата матрица. Овде повторно е искористено знаењето од линеарна алгебра за да се олесни кодирањето, бидејќи резултантната матрица СЕКОГАШ ќе има број на редици колку и првата матрица од помножените матрици!

За да се постигне ова се користат предикатите:

- `mnozi_elementi` – множење на елементите во две листи, враќа единствен број.

```
mnozi_elementi([],[],[]).
mnozi_elementi([E1|R1],[E2|R2],Pomnozeni):- Mnozitel is E1*E2,
                                              mnozi_elementi(R1,R2,SubResult),
                                              append([Mnozitel],SubResult,Pomnozeni).
```

Рекурзивен предикат кој итерира низ елементите на две листи, ги множи и потоа го додава резултатот со листата од останатите измножени елементи која се генерира во SubResult. Основен случај на рекурзијата е празни листи и тогаш враќа празна листа во која ќе се сместуваат останатите елементи при враќање назад во повикот.

- `soberi_elementi` – собирање на елементите во дадена листа, враќа единствен број.

```
%Sobiranje na elementi vo lista
soberi_elementi([],0).
soberi_elementi([X|L],Result):-soberi_elementi(L,SubResult), Result is X+SubResult.
```

Итерира низ елементите во листата рекурзивно до последниот елементи, тогаш влегува во основниот случај и поставува вредност на Result=0. При враќање назад секој елемент се додава на претходно пресметаниот резултат. Излез е Result.

- `mnozi_matrici` – множење на две матрици, враќа листа со едно ниво каде елементите се подредени според редоследот на множење на редиците со колоните на матриците. Редоследот на овие е од суштинска важност за потоа да може само да се подели листата на подлисти.

```

mnozi_matrici(M1,M2,R):-member(RM1,M1),member(RM2,M2),mnozi_redici(RM1,RM2,R).
```

Со помошниот предикат `member` се генерираат вгнездени циклуси. Секоја редица `RM1` од првата матрица `M1`, ќе се повика со секоја редица `RM2` од втората матрица `M2`. За секоја од овие комбинации ќе се повика предикатот за множење на две редици и резултатот ќе се смести во `R`. Овој повик ќе ги враќа измножените редици кликање на `Next`, а за да ги собереме во една листа подолу ќе се искористи `findall`.

- `podeli_N_elements_helper` – помошен предикат за да се имплементира следниот предикат. Овде се земаат првите `N` елементи од листата и се враќа остатокот од листата.

```
podeli_N_elements_helper(L,0,[],L).
podeli_N_elements_helper([E|L],N,[E|SubRes],R):-
    M is N-1,
    podeli_N_elements_helper(L,M,SubRes,R).
```

Влезни аргументи: Листа `L`, број `N`.

Излезни аргументи: Листа од првите `N` елементи на листата `L`, листа која претставува остатокот од листата `L`.

Со помош на бројач кој го намалуваме во секоја итерација, го додаваме моменталниот елемент во `SubRes` и рекурзивно се повикува истиот предикат. Кога бројачот ќе стигне до 0, `SubRes` се креира како празна листа во која враќајќи се назад ќе се лепат елементите кои ги изминувавме. Во `R` се враќа остатокот од влезната листа.

- `podeli_N_elements` – делење на листата во подлисти со `N` елементи.

```
podeli_N_elements([],_,[]).
podeli_N_elements(L,N,M):-podeli_N_elements_helper(L,N,SubRes,OstatokList),
    podeli_N_elements(OstatokList,N,M1),
    append([SubRes],M1,M).
```

Влезни аргументи: Листа `L`, број `N`.

Излезни аргументи: Листа од подлисти со должина `N` (всушност матрица 😊).

Го повикува горниот предикат и листата со `N` елементи што ќе се врати ја додава на резултантна листа. Потоа рекурзивно се повикува за да се генерираат подлистите од останатите елементи.

Основен случај е празна листа, односно целата влезна листа е поделена на подлисти со големина `N`.

- `mnozenje_matrici_pipeline` – Предикат кој ги спојува сите останати предикати за да врати крајна матрица.

```
mnozenje_matrici_pipeline(M1,M2,Result):-length(M1,N),RowsFirst is N,
    findall(X,mnozi_matrici(M1,M2,X),Z),
    podeli_N_elements(Z,RowsFirst,Result).
```

Извршува:

1. Го зачувува бројот на редици на првата матрица (потребен за да се подели листата во подлисти). → RowsFirst
2. Со помош на findall креира листа Z, чии елементи (X) се резултантните броеви од множење на редица со колона.
3. Ја дели листата во подлисти со должина N.

Влезни аргументи: Матрица M1, матрица M2. (За нашиот проблем M1=M2).

Излезни аргументи: Листа од подлисти со должина N (Матрица Result).

- `presmetaj` – Главен предикат кој се повикува за да се изврши множењето. Неговата улога е да го повика предикатот `mnozenje_matrici_pipeline` со два исти аргументи (оригиналната матрица двапати). Враќа резултантна матрица од множењето.

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

Повикување на предикатот:

*Решението не е зависно од големината на матрицата.

```
presmetaj([[1,2,3],[4,5,6],[7,8,9]],R)
R = [[14, 32, 50], [32, 77, 122], [50, 122, 194]]
false

?- presmetaj([[1,2,3],[4,5,6],[7,8,9]],R)
```

```
presmetaj([[1,2],[4,5],[7,8]],R)
R = [[5, 14, 23], [14, 41, 68], [23, 68, 113]]
false

?- presmetaj([[1,2],[4,5],[7,8]],R)
```

Дополнително: Пресметување на транспонирана матрица.

Логика на имплементација во код во Пролог: За влезна матрица M, земи ги првите елементи од секоја од редиците и спој ги во нова листа. Постапката се повторува рекурзивно додека редиците имаат елементи. Ново-генерираните листи, кои се всушност транспонирани редици се враќаат споени во една листа – матрица.

Првите елементи од редиците ги земаме со предикатот `zemi_prvi`.

Влезни аргументи: Матрица M.

Излезни аргументи: Колоната од M односно новата редица во транс.матрицата, остатокот од матрицата на која треба да се изврши истата постапка.

Со првиот `append` се собираат првите елементи во една листа, а со вториот `append` се креира матрица која е под-матрица на оригиналната без првите елементи.

Пример:

$M = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$

После прв повик: NovaRedica = [1,4,7] OstatokMatrica = [[2,3], [5,6], [8,9]]

```
zemi_prv([X|OstatokRedica],X,OstatokRedica).
zemi_prvi([],[],[]).
zemi_prvi([Redica|SubMatrix],NovaRedica,OstatokMatrica):-
    zemi_prv(Redica,Prv,OstatokRedica),
    zemi_prvi(SubMatrix,Prvi,OstatokRedici),
    append([Prv],Prvi,NovaRedica),
    append([OstatokRedica],OstatokRedici,OstatokMatrica).
```

Главна функција: Земи ги првите елементи од редиците и зачувај ги во NovaRedica.

Рекурзивно најди ги сите листи од први елементи (редици во новата матрица). Враќајќи се назад залепи ја новата редица на матрицата.

Основен случај: празна редица во матрицата → [[] | _]

```
transpose([],[]).
transpose(M,T):-zemi_prvi(M,NovaRedica,OstatokMatrica),
    transpose(OstatokMatrica,T1),
    append([NovaRedica],T1,T).
```

Задача 7

Првин размислив како би ја решила задачата во другите програмски јазици. Наједноставното решение е да се користи помошна листа, која на почетокот е празна. Првиот елемент од влезната листа се става во помошната листа. Потоа за секој следен елемент од влезната листа, се споредува неговата должина со должината на елементите кои се ставени во помошната листа. Ја изминуваме помошната листа се додека

должините на елементите кои се веќе сместени во неа се поголеми од должината на моменталниот елемент. Кога овој услов повеќе нема да важи, односно елементот во помошната листа има помала должина од моменталниот елемент кој го споредуваме, тогаш го внесуваме моменталниот елемент на позицијата пред него.

За да се исполнат останатите услови на задачата, кога должините на елементите се исти потребно е да се направат уште неколку споредувања. Првин проверувам дали елементите се целосно исти. Доколку да, тогаш не го додаваме елементот во помошната листа. (Ова решение ги отстранува елементите дури и ако се појавуваат повеќе од двапати. Гарантира уникатност!)

Ако не се целосно исти, тогаш потребно е да се споредуваат сите елементи на исти позиции внатре во нив се додека не најдеме елемент од едната елемент-листа кој е поголем од елементот на истата позиција во другата елемент-листа. Ова мора да се задоволи, бидејќи ако сите елементи им беа исти, тогаш претходниот услов ќе биде задоволен и нема ниту да стигне до оваа проверка.

За имплементација би користела 3 for циклуси. Псевдо код:

```
PomosnaLista = [] , flag = 0
for ElementVlezna in VleznaLista:
    #ako flag=1, elementite se celosno isti, pa samo prodolzuvame so sporedba
    #na slednite elementi, bez da go dodavame ElementVlezna vo pomosnaLista
    if flag == 1:
        flag=0
        continue
    for ElementPomosna in PomosnaLista:
        if PomosnaLista is Empty:
            PomosnaLista.add(ElementVlezna,1)
            break

        if len(ElementVlezna)> len(ElementPomosna):
            #Dodadi go elementot od vlezna pred elementot vo pomosna
            PomosnaLista.add(ElementVlezna, pozicija(ElementPomosna))

        if len(ElementVlezna)< len(ElementPomosna):
            #Prodolzi so sporedba so dr elementi vo PomosnaLista
            Continue

        if len(ElementVlezna)== len(ElementPomosna):
            #Ako se celosno isti elementite, izlezi od ovoj for ciklus I
            #prodolzi so sporedba na drugite el od vlezna lista
            if ElementVlezna == ElementPomosna:
                flag = 1
                continue;

            #Else: dolzinite im se isti, pa treba da se sporedat site elementi
            else:
                for (T1,T2) in (ElementVlezna,ElementPomosna):
                    if T1>T2:
                        PomosnaLista.add(ElementVlezna, pozicija(ElementPomosna))
                    if T1<T2:
                        PomosnaLista.add(ElementVlezna,pozicija(ElementPomosna)+1)
                    Else:
                        #ako ovie el se isti, sporedi gi slednite
                        continue
```

Решение во Пролог:

- Повикување на првиот for циклус и иницијализација на помошната листа на празна листа.

```
3  
4 transform(L, Result):-prv_for(L,[],Result).
```

- Прв for циклус: Земи го првиот елемент од влезната листа и повикај го вториот for циклус за споредба на елементите кои се во помошната листа со тој елемент. Во NovaPomosna ќе се врати помошната листа надополнета со елементот Head на соодветната позиција, според должината. Продолжи со споредба на останатите елементи од влезната листа со повикување на рекурзивниот повик со остатокот од листата (L), проширената помошна – NovaPomosna и крајниот резултат кој уште не е генериран – Result. Основен случај на оваа рекурзија е кога ќе се испразни влезната листа па нема што повеќе да се споредува. Тогаш помошната листа која ја користев, се сместува во Result и се враќа назад.

```
6 prv_for([],Pomosna,Pomosna).  
7 %cut operator za da ne prodolzuvaa so rekurzijata ednas koga ke ja izvrsi do kraj!  
8 prv_for([Head|L], Pomosna, Result):-vtr_for(Head,Pomosna,NovaPomosna)  
9                                     ,prv_for(L,NovaPomosna,Result),!.
```

- Втор for циклус: Основен случај е кога ќе стигне до крајот на помошната листа (или кога се повикува со празна помошна листа – за првиот елемент од влезната), тогаш нема повеќе за споредба – моменталниот елемент кој го испитуваме има најмала должина од останатите, па го ставаме на крај.

Линија 12 е споредбата кога сме стигнале до елемент од помошната листа чија должина е помала од моменталниот елемент кој пробуваме да го сместиме во помошната листа. Тогаш, го лепиме овој елемент (ElementList) на остатокот од помошната листа и ја враќаме рекурзијата назад.

[ElementList, Head|CurrPomosna] -> прв е моменталниот елемент, па оној кој го извадивме за споредба и во CurrPomosna е остатокот од помошната листа.

Линија 15 е кога елементот од помошната линија има поголема должина од нашиот елемент во влезната листа, па само продолжуваме со рекурзијата додека не му го најдеме соодветното место.

Остануваат случаите кога должините се исти.

Линија 24 е кога елементите се целосно исти (копија), па затоа не го внесуваме елементот во помошната листа. Овде не мора да се повика рекурзивниот повик. Може да заврши рекурзијата, но на овој начин се отстрануваат и другите дупликати, а не само првиот кој ќе го сретнеме!

Линија 30 е последниот случај – споредба на секој од елементите со предикатот tret_for. Овој предикат ќе ги врати Поголемата и Помалата листа-елементи споредени според секој од елементите и тие ќе бидат залепени пред остатокот од помошната листа.

```

11 vtor_for(ElementList,[],[ElementList]):-!.
12 vtor_for(ElementList, [Head|CurrPomosna],[ElementList, Head|CurrPomosna]):-
13     length(ElementList, Len1),length(Head,Len2), Len1>Len2.
14
15 vtor_for(ElementList, [Head|CurrList],[Head|Res]):-length(ElementList, Len1),
16     length(Head,Len2), Len1<Len2,
17     vtor_for(ElementList,CurrList,Res).
18 % Prvo se proveruva dali listite-elementi se isti stom dolzinite im se isti.
19 % Ovde ne mora da se stave logickata sporedba bidejki samo ako ne vazat prethodnite
20 % 2 uslovi ke stigne do tuka, ama vo toj slucaj ke treba da se upotrebe cut operator za
21 % da ne prodolzuvaa rekurzijata koga ke zavrse.
22 % Ako ne se povika povtorno vtor_for nema da se izbrise i ostanatite elementi koi se isti
23 % so momentalniot.
24 vtor_for(ElementList, [Head|CurrList],Res):-length(ElementList, Len1),length(Head,Len2),
25     Len1==Len2, ElementList==Head,
26     vtor_for(ElementList,CurrList,Res).
27
28 %Ako stigne do tuka znaci deka dolzinite im se isti,
29 %no elementite razlicni, pa treba da se povika tretiot for ciklus
30 vtor_for(ElementList, [Head|CurrPomosna],Res):-tret_for(ElementList,Head,Pogolema,Pomala),
31     append([Pogolema,Pomala],CurrPomosna,Res).
32

```

- Трет for циклус:

Ако моменталниот елемент од првата листа е поголем од моменталниот елемент на втората листа, како трет аргумент врати ја првата, а како четврт врати ја втората листа. [H1|L1] -> мора да се повика на овој начин за да се залепи главата која ја извадивме за споредба.

Обратно, ако моменталниот елемент од втората листа е поголем од моменталниот елемент на првата листа, како трет аргумент(Pogolema) врати ја втората , а како четврт врати ја првата листа(Pomala).

Ако тие елементи се исти продолжи во рекурзијата со споредба на следните елементи. На враќање залепи го H1 на почетокот на резултатите за Поголемата и Помала листа. Не е важно дали ќе се залепи H1 или H2 бидејќи во овој услов влегува само ако тие се исти!

```

33 %tret_for(ElementLista1, ElementLista2, Pogolema, Pomala)
34 tret_for([H1|L1],[H2|L2],[H1|L1],[H2|L2]):-H1>H2.
35 tret_for([H1|L1],[H2|L2],[H2|L2],[H1|L1]):-H1<H2.
36 % Ne e vazno dali H1 ili H2 ke go zalepam na ostanatiot del od sporedbata bidejki
37 % se istiot element
38 tret_for([H1|L1],[H2|L2],[H1|Pogolema],[H1|Pomala]):-tret_for(L1,L2,Pogolema,Pomala).
39

```

Повикување на предикатот:


```

transform([[3,10],[2,4,6,33,1,8],[4,1,3,6],[3],[2,4,6,33,1,8],[7,12],[4,1,2,7],
[6,7,9]],L)
Singleton variables: [H2]
L = [ [2, 4, 6, 33, 1, 8],
      [4, 1, 3, 6],
      [4, 1, 2, 7],
      [6, 7, 9],
      [7, 12],
      [3, 10],
      [3]
    ]
?- transform([[3,10],[2,4,6,33,1,8],[4,1,3,6],[3],[2,4,6,33,1,8],[
7,12],[4,1,2,7],[6,7,9]],L)

```

Задача 8

Имплементираното решение користи помошна листа за да ги чува елементите за кои проверката за бришење на секој втор елемент од тој тип е помината. Логиката е да се изминува влезната листа и доколку за моменталниот елемент процесот на бришење не поминал, тогаш се користи предикат кој ќе го врши бришењето на секое второ појавување на тој Елемент во влезната Листа. Понатаму се продолжува со следниот елемент и со изминатата листа.

- `brisi_sкое_vtoro_rec(Element, Lista, FlagIn, FlagOut, R)` – рекурзивен предикат кој го врши бришењето на елементот `Element` во листата `Lista`. `FlagIn` се користи за да се памти до каде сме со бришењето. Ако неговата вредност е 0, и следниот елемент е ист со тој кој го бараме, само се продолжува со итерација низ листата, а вредноста на знаменцето се менува во 1. Ако неговата вредност е 1, и следниот елемент е ист со тој кој го бараме, тогаш го прескокнуваме тој елемент и продолжуваме понатаму.

Доколку елементот до кој сме стигнале е листа, тогаш се повикува предикатот за елементот-листа, а потоа се повикува за остатокот од листата, така што излезното знаменце од првиот повик се предава како влезно знаменце на вториот повик!

Овде е направена оптимизација со предикатот од аудиториски вежби – `clen_nivo2`, така што правиме проверка дали елементот кој го бараме е во листата. Ако не, нема потреба да се пребарува, туку само се лепи на остатокот од резултатот.

Основен случај е кога стигнуваме со празна листа. Во тој момент се враќа празна листа.

Интересно е што за отстранување на елемент од листа, го користев предикатот `paјdi_prv` кој го работевме на вежби. Овој предикат ако се повиква со недефинирана вредност на `Prv` и некоја листа, во променливата

го враќа првиот елемент од листата, а во Result остаокот од листата. Но, ако се злоупотреби, да се повика со дефинирана променлива Prv, тогаш во Result ја враќа листата со отстранет тој елемент!

```

10 %Base case e posleden bidejki ke bide ispolnet za bilo koja neprazna lista.
11 %istata funkcija ako se povika najdi_prv(1,[],[[1],2]],T), so zadadena vrednost za Prv,
12 %ja vrackja listata so izbrisan toj element, AKO E NA PRVA POZICIJA!
13 najdi_prv(Prv,[G|O],Result):-e_lista(G),najdi_prv(Prv,G,Temp),append([Temp],O,Result).
14 najdi_prv(Prv,[G|O],Result):-prazna(G),najdi_prv(Prv,O,Temp),append([G],Temp,Result).
15 najdi_prv(Prv,[Prv|O],O).

17 brisi_sekoe_vtoro_rec(Element,[],FlagIn,FlagIn,[]).
Singleton-variables: [Element]
18
19 brisi_sekoe_vtoro_rec(Element,[G|O],FlagIn,FlagOut,R):-e_lista(G), not(clen_nivo2(Element,G)).
20   brisi_sekoe_vtoro_rec(Element,O,FlagIn,FlagOut,R1),append([G],[R1],R).
21
22 brisi_sekoe_vtoro_rec(Element,[G|O],FlagIn,FlagOut,R):-e_lista(G), clen_nivo2(Element,G),
23   brisi_sekoe_vtoro_rec(Element,G,FlagIn,FlagOutG,R1),
24   brisi_sekoe_vtoro_rec(Element,O,FlagOutG,FlagOut,R2),
25   append(R1,R2,R).
26
27 brisi_sekoe_vtoro_rec(Element,[G|O],FlagIn,FlagOut,R):-not(e_lista(G)), Element==G,
28   FlagIn==1,najdi_prv(_, [G|O],Deleted),
29   brisi_sekoe_vtoro_rec(Element,Deleted,0,FlagOut,R).
30
31 brisi_sekoe_vtoro_rec(Element,[G|O],FlagIn,FlagOut,R):-not(e_lista(G)), Element==G,
32   FlagIn==0,
33   brisi_sekoe_vtoro_rec(Element,O,1,FlagOut,R1),
34   ( ( not(prazna(R1)),append([G],[R1],R));( prazna(R1),append([G],[],R))).
35
36 brisi_sekoe_vtoro_rec(Element,[G|O],FlagIn,FlagOut,R):-not(e_lista(G)), Element\=G,
37   brisi_sekoe_vtoro_rec(Element,O,FlagIn,FlagOut,R1),
38   ( ( not(prazna(R1)),append([G],R1,R));R is G).

```

Предикатот не го постигнува целосно она што го замислив, бидејќи не поставува празна листа:

```
R = [1, [2, [2], [3, 3, [[3]], [1], 1], 2, 2, 1, 3]]
```

```
?- brisi_sekoe_vtoro_rec(1, [1,[2,1,[2],[3,3,[[3]],1,[1]],1],2,1,2,1,3],0,_,R)|
```

Доколку решението беше целосно точно следниот предикат кој го креирав е за изминување на влезната листа и повикување на brisi_sekoe_vtoro_rec. Едноставен рекурзивен предикат кој за секој елемент што не е листа, повикува бришење и го додава елементот во помошната листа Изминати. Ако елементот е листа, тогаш се

повикува оваа функција за главата и за остатокот од листата, така што листата изминати од првиот повик се предава на вториот за да не се направи грешка!

```
40 helper([],L,_,L).
41
42 helper([G|O],L,Izminati,NovoIzminati,R):-not(e_lista(G)), not(member(G,Izminati)),
43     brisi_sekoe_vtoro_rec(G,L,0,_,R1),
44     append([G],Izminati,NovoIzminati),
45     helper(O,R1,NovoIzminati,R).
46
47 helper([G|O],L,Izminati,NovoIzminati,R):-e_lista(G),helper(G,L,Izminati,NovoIzminati,R1),
48     helper(O,R1,NovoIzminati,R).
```

Главниот предикат го повикува горниот предикат `helper` со ПРАЗНА иницијализирана помошна листа за изминати елементи и влезната листа од корисникот.

```
50 brisi_sekoe_vtoro(L,R):-helper(L,L,[],_,R).
```

* Друго решение кое дава точни резултати и го имплементирам е да се избришат непотребните елементи без да се зачува структурата, а потоа да се итерира низ оригиналната листа и да се отстранат тие вишок елементи. Сепак, верувам ова не е целта на задачата, па затоа истото не го прикачив.