MAX (MIN)

```
max([X], X).

max([H|T], Max) :-
    max(T, Max),
    H =< Max.

max([H|T], H) :-
    max(T, MaxT),
    H > MaxT.
```

(1.) napačen robni pogoj
max([], 0). ali karkoli...

(2.) max([X], [X]). list, namesto element!

(3.) max([X], Min). Min ≠ X!

(A.) generično: syntax error:
>= ~ =>
in
=< ~ <=

(8.) kaj, če pozabi eno izmed obeh vej?
To načeloma faila.
Ampak test cases lahko odkrijejo, da to manjka!
[1,2,3,4,5] za eno vejo
in [5,4,3,2,1] za drugo vejo ;)

Kaj pa rez? Tukaj ga še ne jemljemo, zato bi rekel takole: "kdor ga uporablja, naj ne bi potreboval namigov!" (razen kaj generičnega)

(4.) nobena izmed "vej" ne upošteva enakosti H in MaxT
(test cases) (=< or >=)

(5.) "arguments are not sufficiently instantiated" error: ← also TRIGGER

```
max([H|T], ...) :-
    H >= MaxT,
    max(T, MaxT) ...
```

↗ iz runtime err sporočila!
"MaxT še nima vrednosti, dobi jo šele kasneje po rekurziji; poskusi preurediti vrstni red ciljev"

(6.) Napačna prioriteta ("bloki") zaradi operatorja OR

```
max([H|T], Max) :-
    max(T, MaxT),
    ( H >= MaxT,
      Max = H
    ;
      H < MaxT,
      Max = MaxT).
```

( ) missing!

(7.) manjkajoč drugi pogoj (H < MaxT) ("kišiti" drugo vejo)
TRIGGER: prva rešitev pravilna, ostale ne več! (testiraj pri samih različnih elementih)

Splošna navodila / pomoč

(2.) Predikat max(List, M) drži natanko tedaj, ko je M največji element v List (ali eden izmed več največjih elementov).

Pa pomislimo...

"Če seznam L razdelim na glavo in rep, (in) vem, da je nek MaxT največji el. v repu (pazi: iskanje maksimuma v repu je manjši problem kot iskanje minimuma v celem seznamu), (in) vem, da je glava H večja od MaxT,
POTEM vem, da je H največji element v celotnem seznamu.

(ALI) pa H ni večja od MaxT, kar potem pomeni, da je največji element v repu (MaxT) tudi največji element v celotnem seznamu L.

VIZUALIZIRAJ!

prvi | Head — TAIL — tudi sem največji (MaxT)

2 možnosti:
H =< MaxT   H > MaxT (dopusti napake enakosti =< ;)

(1.) Poskusi prevesti na manjši (a enak) problem...
Recimo, da že imaš največji element v repu seznama... (recimo mu MaxElementInTail ali MaxT na kratko)
... potem ga samo še primerjaš s prvim elementom v seznamu in vidim, kdo je večji ;)
(morda tu uporabiš vizualizacijo od zgoraj?)

MEMBER

osnovna navodila

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

```
member(X, L) :-
    L = [H|T],
    H = X.
member(X, L) :-
    L = [H|T],
    member(X, T).
```

~ == namesto = (+ razlaga razlike pomena)
↓
morda damo to tudi kot zanimivost na koncu?

} poseben primer (1)

```
member(X, [H|T]) :-
    H = X
    ;
    member(X, T).
```

~ == namesto =

(1.) Manjka ali napačen robni pogoj
posebej podprimer "==" namesto "="

(2.) Manjka rekurzivni klic
~ generični hint (detektor)

(3.) Rekurzija ne "zmanjša" problema
```
member(X,L) :-
    member(X,L).
```
→ ali H namesto X

~ znamo to detektirati?
verjetno da, z len od seznama :)
(dokler smo v seznamih, je to smiselno)

(4.) member(X, []) :- fail. (ni narobe, a je odveč in ni v duhu Prologa)
* generičen hint morda

(5.) ~~member~~(X, L) :- stilističen komentar / naming
L = [H|T],
...

Loni; morda narisano kot vrsto ljudi:

↑ vidim prvega, ostali se skrivajo ~~nekje zadaj~~ v ozadju... (za prvim)

ZANIMIVOST NA KONCU:

Did you know?

Member se velikokrat (morda celo večinoma) uporablja v "obratnem" delovanju -- namesto preverjanja ali je X v seznamu L, ga lahko uporabimo kot "vrni mi nek element X, ki je v L"
→ v bistvu imamo generator elementov v L

Prav tako je, da PROLOG poskuša pogoj predikata zadostiti, torej poskuša cilj member(X,L) narediti resničen in temu ustrezno "izbere" X! (priredi)

(pogoj: robni pogoj pravilen)

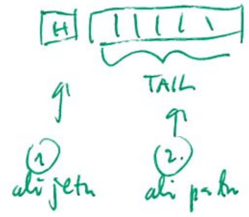
Primeri (ki se kasneje uporabljajo)

member(X, [a,b,c]).
member(X, Kvadrati[1,2,5,10,...]).
member(Operator, [+,-,*,/])
itd.

(1.) Kje se lahko skriva iskani element X?

Seznam ima dva dela, torej sta dve možnosti!

Ali je prvi element ali pa se skriva nekje v repu! (morda ga sploh ni, a v tem primeru bo PROLOG tako ali tako z veseljem rekel "NE!" — kar pomeni, da ga ni našel oz. je dokazal, da ga ni.)



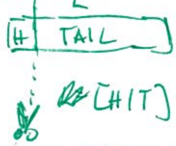
(2.) Kaj je najenostavnejša možnost? Da je ~~skriva~~ iskani element kar na prvem mestu!

seznam je kar [X,_,_,_,...] ali [X|_] oz. [X|Tail]

(3.) Kako poiščem iskani element nekje v repu?

Odstranim prvi element in iščem v preostanku

odstranim



iščem v T..., če najdem v T, je tudi v L (saj je T del L)

DELETE (INSERT je lahko samo kot zanimivost)

delete(X, [X|T], T).

delete (X, [H|T], [H|NT]):-
    delete(X, T, NT).

4. manjka robni pogoj
   4a: res manjka
   4b: je napačen
   ↑
   v tem primeru je isto, ker se ne zacikla neskončno, ampak faila.

Osnovna navodila (hints)

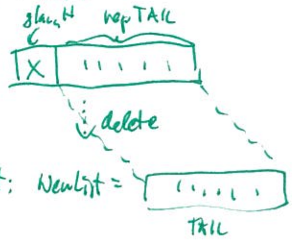
1. hint

2 možnosti za brisanje:

1. brišem glavo, če je možno
2. brišem iz repa

2. hint (če faila robni pogoj)

Kaj je najenostavnejša smiselna možnost? Brisanje prvega elementa?



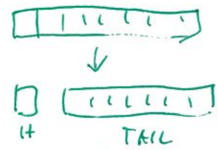
Tipične napake

1. delete (X, [], []). → "če brišem iz [], ne dobim [], ampak ne uspem – ne morem brisati"
~~ali~~
2. delete (X, [], _):- fail. → ni potrebno, ker tako deluje že sam Prolog!

*generična napaka lahko

3. hint (brisanje iz repa)
Pozor! → robni pogoj je narejen!

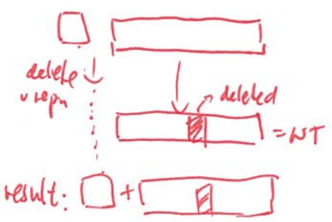
Kako brišem nekje iz repa?
Seznam razbijem na (preostali el.) glavo (prvi element) in rep.



↓
brišem od tu (uporabim rekurzijo, lahko ker je ta seznam za en element manjši!)

3. pozabljena glava oz. glave!
   delete (X, [H|T], NT):-
   ...
   → najprej opozorilo o tem
   → drugič (več..) pomoč kako to narediti.

   slikovno:



   ↳ dodajanje na začetek [Prvi|Rep]

7. == namesto =
   delete(X, L, L1):-
   L= [H|T],
   X == H, ...
   (isto kot pri member)

5. L= [H|T] ...
   delete(X, [H|T], NT1):-
      delete (X, T, NT),
      NT1 = [H|NT]. ← to ni najbolje za "vse skupaj" (se da odkriti s testnimi primeri!)

(ne čisto lepotni popravek)
↳(morda se to lepo tudi razloži?!)

6. L= [H|T], drugič   (lepotni popravek)
   delete (X, L, L1):-
      L= [H|T],   ~ ta ni nevaren, a se ga tudi ne zazna s testnimi primeri...
      ...
   *generična napaka

ZANIMIVOST (na koncu)

pokaži povezavo med delete/3 in insert/3

• pod pogojem, da naredi insert za demonstracijsko nalogo!
(kar sploh ni slab pri

CONC

```
conc([], L2, L2).
conc([H|T], L2, [H|L3]) :-
    conc(T, L2, L3).
```

generična napaka (1)

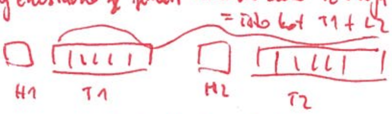
H|T, brez seznamskih oklepajev!

1. napačen robni pogoj (simple unit test case)
   → poseben primer: [] + [] = []

2. nekompatibilen robni pogoj z rekurzivnim klicem
   zmanjšuješ L1 in pogoj za prazen L2 ali obratno!

3. razbijaje/zmanjševanje obeh seznamov L1 in L2
   ↳ namig: "Bolj enostavno je jemati elemente samo iz enega seznama"

   SLIKA:



   → to je težje sestaviti, ker je veliko koščkov

   kot

4. conc([H|T], L2, ...) :-
       conc(T, [H|L2], ...).

   To se sicer ustavi, ampak vrstni red elementov ni ok
   (prvi seznam se obrne...)

   [a b c] + [1 2 3] = [c b a 1 2 3]

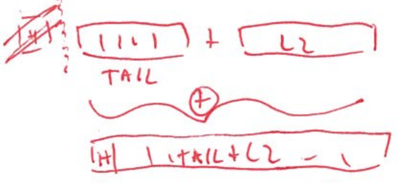
   HINT: "Dodajanje prvega/trenutnega elementa naredi šele, ko rešiš podproblem rep+L2"

   slikovno? ⇒ morda video kako rekurzija to reši!

5. conc([H|T], L2, ...) :-
       conc(T, L2, [H|L]).
                    ↑

   HINT: "Ne govori rekurziji kaj naj vrne, ona to vrne sama!"

   SLIKA:



   od kje ga pa naj dobi!?

6. ne samo statistični L=[H|T] itd.:

   conc([H|T], L2, L4) :-
       conc(T, L2, L3),
       L4 = [H|L3].

   morda video "napačnega" razvoja

   ↑ nepričakovanega, nesmisla v drugi smeri

splošna
Osnovna navodila/pomoč

pogoj: robni pogoj, če ni nanaša (testcase)

1. Začnimo z najbolj enostavnim (smiselnim) primerom...
   Kaj se zgodi, če poskusim združiti prazen seznam L1 in nek poljuben (lahko tudi prazen) seznam L2?

   ↑ kako bi vizualno to dobro predstavili (prazen seznam) eg. prazen avtobus or smth.

   <opomba: seveda bi lahko seznama L1 in L2 imela tudi drugačno vlogo, ampak recimo, da je prazen L1>

   rezultat = ? (L2)!

   * v resnici lahko tudi, če je takšno nepraktično naj conc, bi je kar append!

2. Hmm... problem bi želeli prevesti na vsaj malce manjši (a enak v osnovi/logiki) problem... torej na "manjši" concatenate

   PROCEDURALNO RAZMIŠLJANJE:

   samo en element v L1 → [ ] L1 [| | |] L2

   1. vzamem prvi (in edini) element iz L1 ([H|T] notacija)

   ~~2. dodam ta element v L2~~

   2. situacija, ki jo dobim je ... ENAKA kot prej! (robni primer)
      T = [] + L2
      rezultat = ja, L2!

      prevedel seznam manjši (za en korak) problem!

   3. dodam ~~rezultat~~ element, ki sem ga vzel iz L1 (H) na začetek rezultata konkatenacije T+L2!

   * več: kako dodam najenostavnje element na začetek seznama?
   je tukaj: [E|List] → poskusi v interpreterju (list!)

   DEKLARATIVNO RAZMIŠLJANJE
   L1 = [H|T] prvi el. + rep
   "Če je rezultat konkatenacije T (ki je seznam) + L2 nek seznam L3, in če temu seznamu (L3) na začetek dodam element H, potem je rezultat tega kar konkatenacija L1 in L2!"

   [H|L3]

LAST

1.
```
last([X], X).
last([_|T], X) :-
    last(T, X).
```

2.
```
last(L, X) :-
    conc(_, [X], X).
```

---

1. napačen robni pogoj
   last([ ], ...)

2. stilistični namig
   last(L, X) :-
   L = [H|T], ...
   * generični namig

3. namesto elementa X vrača enoelementni seznam [X]
   last([X], [X]).

4. Kaj, če bodo poskušali reševati z length?
   V smislu: robni pogoj je, ko je velja length(L,N), N is 1.

5. stilistični:
   last([ ], _) :- fail.

splošni namigi

1. Možnosti sta vsaj dve ... ali napišemo klasično rekurzijo ali pa uporabimo predikat, ki smo ga spisali pred kratkim;

a veste ka delal na ∃ (rešitev za conc/3)? →

(1a) Poskusimo z rekurzijo ...

"Če ima seznam samo en element potem smo ga že našli (zadnjega namreč) ... Sicer pa lahko seznam "razbijemo" na glavo in rep, razdelimo, zadnji element pa se seveda skriva v repu"

ALI DRUGAČE POVEDANO

(2a) "Recimo, da je X zadnji element v repu T seznama L, potem je X tudi zadnji element seznama L, logično"

(1b) Kaj pa, če poskusimo takole... katera operacija je na sliki?

seznam1 + seznam2 = večji seznam

[a|b|c] + [1|2|3|5] = [a|b|c|1|2|3| ...

ok, kaj pa naredi tole ← seznam dolžine 1

[a|b|c] + [g] = [a|b|c|g]

zadnji element je .... g ;

? (če ne rešijo s conc, pa tudi če, pravzaprav)

POUČNA ZANIMIVOST NA KONCU

conc/3 je zelo uporaben predikat! Ena izmed možnosti je ravno jemanje/iskanje ali dodajanje elementa na konec seznama!

TUDI LINK/COPY V INTERPRETER!

PRIMER1: ?- conc(_, [X], [a,b,c,d,g]).
X = g

ALI: ?- conc([a,b,c], [g], L).
L = [a,b,c,g]

DUPLICATE

```
dup([], []).
dup([H|T], [H,H|DT]) :-
    dup(T, DT).
```

(1.) nepačen robni pogoj — npr. dup([], _).

(2.) dup([X], [X,X]). ← čeprav pri tem lahko opozorimo, da naj dela tudi za prazen seznam!

```
dup([], []).
dup([H|T], DL) :-
    DH = [H,H],
    dup(T, DT),
    conc(DH, DT, DL).
```

(3.) Takoj bodo stilsko pisali [H| [H]] (3a)
(3b) in nepravilno [H|H] ~ napake!

(4.) conc/3 je "overkill".
Detekcija: pravilna rešitev + conc je del kode

(5.) conc(H, _, _)
↑ element, ne seznam

* generični namig
"Argument predikata conc/3 ni seznam!"

GENERIČNI NASVETI

(1.) dup(L, ...)
L = [H|T]

(2.) Ne bodo znali kreirati dvoelementnega seznama ali dodati dva elementa na začetek...
L1 = [H|DT],
L2 = [H|L1], ...
} Tim, se to pojavlja?

(6.) Usklajevanje rešitve rekurziji!

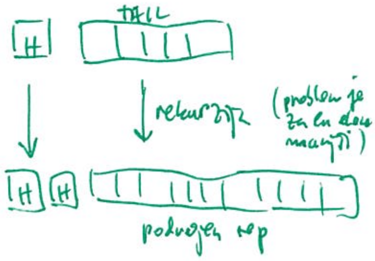
```
dup([H|T], ...) :-
    dup(T, [H,H|DT]).
```

← "kako naj to PROLOG kadarkoli naredi resnično!?"
← kaj je deklarativni pomen tega cilja? Razloži!

Splošni namigi

(1.) klasičen rekurzivni pristop...
bodi pogumni in privzemimo, da že imamo podvojen rep seznama. Potem je vse kar moramo postoriti, da podvojimo glavo (H → H,H) in to dodamo pred podvojen rep.

(2.)



(Ø.) ← Pogoj: še nima robnega primera!
Ali znam "podvojiti" prazen seznam

(3.) Primer z enim samim elementom:

[H]
↓
H = [H]
T = [] prazen
↳ rekurzija podvoji (robni primer)
← vrne [] (Prazen seznam je duplikat praznega seznama)

DupT = []
[H,H] + [] = rešitev ;)

(pogoj rešen primer (Ø))

(4.) "Če imam podvojen rep DT in pred to postavim dve glavi [H,H], potem je to skupaj ravno podvojen seznam"
ALI
"Če velja, da je DT podvojen rep potem velja, da je [H,H|DT] podvojen celoten seznam L."
→ več... kako dodam dva el. na začetek seznama

PERMUTE *

1.
```
permute([], []).
permute(L, [X|P]) :-
    delete(X, L, L1),
    permute(L1, P).
```

2.
```
permute([], []).
permute([H|T], P) :-
    permute(T, PT),
    insert(H, PT, P).
```

1. Nekateri rešijo z robnim pogojem permute([X], [X]). Ta je tudi ok, a ne pokrije posebnega primera (prazen seznam). Lahko pa štejemo obe rešitvi za pravilni! (Po moje najbolje)

2. spodaj

2. verzija z delete/3: pozabijo izbrisani element vstaviti nazaj

3. narobe povezane spremenljivke

---

1. Tukaj gre predvsem za backtracking generiranje rešitev! Ta naloga je kar težka ...

↓ zato...

2. Morda najbolje prikazati stvari z videom! [a, b, c], verzija z delete/3

to tudi backtracking razloži!

Tole je kar precej zamudno... ampak učinkovito!

---

splošni namigi

1. Videti je bolj enostavno za razložiti z insert/3, ampak ...

Razmišljajmo takole ...

predpostavimo, da že imamo rešitev (permutacijo) seznama z dvema elementoma [E2, E3], želeli pa bi rešitev za seznam s tremi elementi [E1, E2, E3].

---

z INSERT je res lažje...

[a|b|c] → [b|c] → [c] → [c]

b c
c b } insert b na poljubno mesto v [c]

⋮

Recimo delete/3 verzija in samo želeo...

[a|b]

izbrišem poljuben element → prevedem na manjši problem

aktualna rekurzija [b] permutiram in dobim [b]

vstavim na začetek prej izbrisani element

a b ← * se prične zares pojaviti v nalogah

ALI (backtrack)

izbrišem sedaj b

[a] —permute→ [a]

vstavim na začetek b:

[b|a]

s tremi bi bilo bolje pokazati...

DIVIDE

```
divide([], [], []).
divide([H], [H], []).
divide([H1,H2|T], [H1|T1], [H2|T2]) :-
    divide(T, T1, T2).
```

---

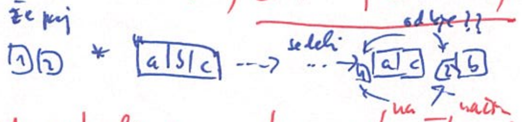
1. potrebna sta tipično dva robna pogoja!
   - naravno morajo biti za lihe dolžine

2. divide([H], [], [H]).
   → nadgje, da spremeni v ([H], [H], [])

3. divide([], _, _)
   ali divide([H], [H|_], _)

4. klasično pošiljanje rekurzije ... kaj mora ~~narediti~~ vrniti
   divide([H1,H2|T], ...) :-
   divide(T, [H1|T1], [H2|T2]).

   * ali video hint
   ali pa: * deklarativni pomen (vrstni red?)



5. Morda vstavljanje na konec (narobe ...)
   T1 = [T|H]
   * tole mora biti tudi generično

REŠITEV S PREPLETANJEM (~~kar pa na to tipično pomislijo dobri programerji~~)

* lahko pa to damo kot zanimivost na koncu ...

(relativno stara rešitev, TM?)

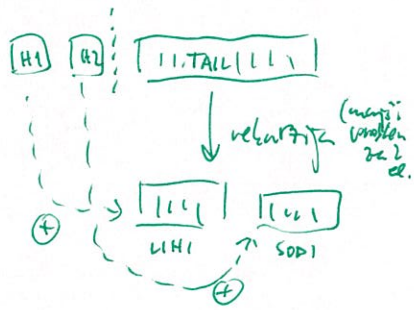
→ ta ideja se v bistvu lahko uporabi pri even/odd len ...
zato nima smisla ta kot zanimivost najprej ...

Splošni nasveti

1. pogoj: če opazimo (a lahko), da jemlje samo po en element stran:
   → več: kako vzamem dva ~~prva~~ elementa z začetka seznama?

2. Kaj, če se lotim takole ...
   Vzamem prva **dva** elementa z začetka seznama, preostanek rekurzivno razdelim rep na sode in lihe, ter potem en element dodam v en seznam, drugi pa v drug seznam.
   → več: kako vzamem dva?

3. Vizualno:



---

4. VIDEO kako rekurzija izračuna stvari
   (od zadaj naprej ... oz. kaj se dogaja ob sestopanju)

---

5. DEKLARATIVNI POMEN:
   "če sta T1 in T2 seznama z "lihimi" in "sodimi" elementi iz repa T (respectively) + in če v T1 dodamo H1 ter v T2 dodamo H2, potem je rezultat ravno razdeljen seznam L!"

ODDLEN & EVENLEN

<BREZ UPORABE ARITMETIKE>

splošni napotki

1.
```
evenlen([]).
evenlen([_|T]):-
    oddlen(T).
oddlen([_|T]):-
    evenlen(T).
```

2.
```
evenlen([]).
evenlen([_,_|T]):-
    evenlen(T).
```
<in podobno za oddlen>

---

3. len, %, mod in podobno

HINT "Prosim, reši brez aritmetike!"

1. Navsiklado (večina?) bo napisale še en robni pogoj za oddlen/1.
```
oddlen([_]).
```
↳ rešitve se bodo podvajale

HINT: "oddlen/1 ne potrebuje robnega pogoja, ker ga zanj reši evenlen"

in tudi napačen robni pogoj:
```
oddlen(_).
```
— namesto [_] !!

---

1. To nalogo si lahko razlagate kot dve ločeni, a podobni, nalogi ali pa kot eno, prepletajočo se nalogo.

L=[H|T]

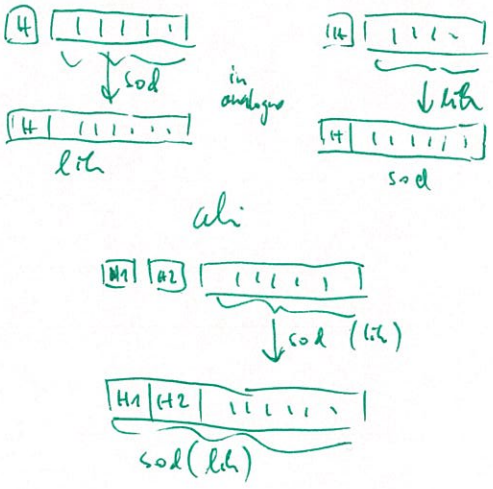
2. Naloga: "Če je rep T sode dolžine potem je L lihe dolžine" (in obratno)

Ali drugi način:

Če razmamo odvzamemo dva elementa, ostane iste dolžine (v smislu lihosti/sodosti)

→ več: Kako vzamem 2 elementa?

3. še slikovno



LENgth / SUM

```
len([], 0).

len([_|T], N) :-
    len(T, NT),
    N is NT + 1.
```
(H)

(+H za sum/2)

① len([], N) :-
N is 0.

hint: "To se da lepše in krajše napisati kot dejstvo len([], 0)."

⑥ +H namesto +1 pri len/2... posledica COPY/PASTE kode iz sum/2

② robni pogoj, kot vedno.. .. čeprav ne vem kako ga zamočiti :)

npr.: len([], _). ?

(Pri sum/2 morda lažje zafrknejo :))

HINT: "Koliko je vsota/dolžina praznega seznama?"

(Ni _ (karkoli) ali [] (prazen seznam) ali 9999.9 itd.)

③ vrstni red aritmetike... ..tukaj (prvič?) zares postane vrstni red pomemben

TRIGGER: ~ is not substituted sufficiently

④ problem z is/2 in =/2

N = NT + 1

TRIGGER: "=" in code, "is" not in code ?

HINT: objasni razliko med is/2 in =/2 (matching)

⑤ za sum (dodatno):

sum([], S) :- (uit in, glej spodaj)
S = -9999.9
or smth

(ampak to je v bistvu isto kot napačen robni pogoj, glej hint #②)

splošni nasveti

① Prazen seznam ni prav pretirano dolg, če pa ni prazen, pa ima glavo in rep. (rep je morda tudi prazen..)

②

[_|prazen]

+1, rekurzija reže: 0

1 + 0 = 1 ;)

(če to dobro razišče)

H + 0 za sum

③ "Če je rep dolg NT, potem je celoten seznam dolg NT + 1, ~~kajne~~?"

(3b) "Če je vsota repa SumT, potem je ~~celoten se~~ vsota celotnega seznama H + SumT."